Rapport du projet de Microcontrôleurs

Luca Jiménez Glur, Geoffroy Renaut

13 juillet 2023

1 Introduction

Dans le cadre du projet de fin de semestre du cours Microcontrôleurs, nous avons décidé de réaliser un petit jeu-vidéo se jouant à deux joueurs à l'aide de la télécommande IR-RC5, du détecteur de distance Sharp GP2Y0A21 et de l'écran LCD Hitachi44780U, avec le kit STK-300, couplé au MCU Atmega128L. L'avatar (caractère personnalisé) du premier joueur est à gauche dans le LCD et celui du second joueur est à droite, les deux pouvant se déplacer de haut en bas. Le personnage de gauche, contrôlé par le capteur de distance, doit faire de son mieux pour éviter les obstacles (« lasers ») lancés par le personnage de droite. Ce dernier est contrôlé avec la télécommande. Lorsque le joueur avec la télécommande arrive à toucher l'autre joueur avec un laser, ce dernier perd une vie. Après avoir perdu ses cinq vies, la partie s'arrête et le deuxième joueur gagne. Le score s'affiche et un écran de démarrage permettant de recommencer est proposé. Si plus de 255 points sont accumulés, la partie s'arrête et le premier joueur gagne.

Ce projet met en œuvre les éléments que nous avons vus en cours durant le semestre, notamment les interruptions en temps réel, les timers, l'utilisation de périphériques externes et leur protocole de communication.

2 Mode d'emploi

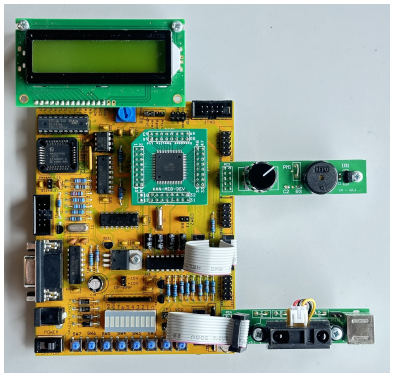

FIGURE 1 – Configuration des péripheriques

Tout d'abord, pour configurer la carte STK-300 afin de jouer au jeu, il faut brancher les périphériques suivants aux ports correspondants : l'écran LCD Hitachi44780 U sur le port LCD, le capteur de distance Sharp GP2Y0A21 (bloc M4) sur le port D et l'interface IR NEC ainsi que le buzzer (bloc M2) sur le port E. Une fois le programme chargé, une page de démarrage est affichée, elle invite à appuyer sur le bouton power de la télécommande afin de démarrer une partie. Une fois la partie commencée, les vaisseaux s'affichent et les joueurs peuvent en prendre le contrôle.

Le vaisseau à droite de l'écran est dirigé par les flèches de changement de chaîne, il peut tirer un laser en appuyant sur le bouton 5 (voir figure 2). Le second joueur doit éviter les projectiles qui lui arrivent dessus par la droite en plaçant sa main au-dessus du capteur de position pour faire que le personnage soit en haut ou en bas (voir figure 3). Son vaisseau sera par défaut sur la ligne du dessus, s'il rapproche sa main suffisamment, le vaisseau descendra sur la ligne du dessous.

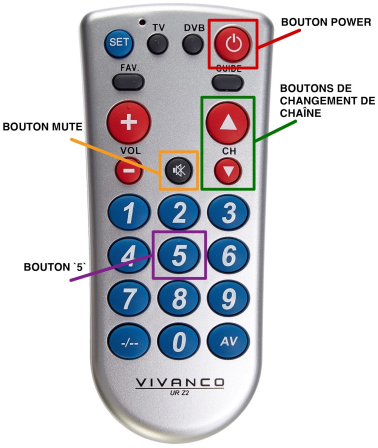

FIGURE 2 – Boutons de la télécommande

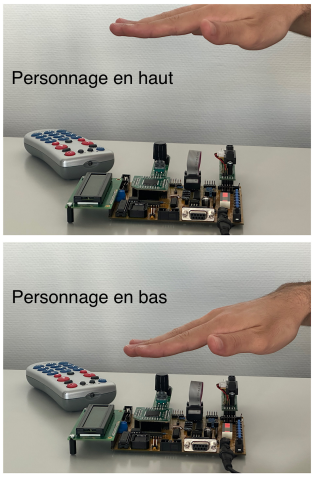

FIGURE 3 – Contrôle de la position du vaisseau

Si le joueur 1 se fait toucher par un laser, il perd une vie. Son nombre de vies restantes est affiché sur les LEDs, il en dispose initialement cinq, mais si toutes ses vies sont détruites, la partie s'arrête. Lors de la fin de la partie, un écran de *Game Over* s'affiche ainsi que le score du joueur 1. Ce score représente le nombre de décalages des projectiles sur les cases de l'écran. Après quelques secondes, cet écran de score laissera sa place à l'écran de démarrage du début de la partie, le joueur 2 pourra lancer à nouveau une partie à l'aide du bouton power.

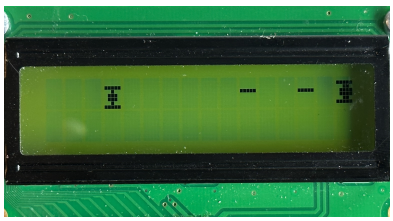

FIGURE 4 – Ecran en jeu

La musique du jeu *Tetris* peut être lancée ou arrêtée à tout moment durant la partie par le joueur avec la télécommande grâce au bouton *mute* de celle-ci. Dû à l'implémentation de la musique, les interruptions deviennent quelques millisecondes plus longues lorsque celle-ci est activée. Ceci ne cause pas de problème perceptible en termes de fluidité et de temps-réel, mais lorsque le joueur 1 veut envoyer des lasers en continu, il y a parfois des « trous », c'est-à-dire des espaces sans laser.

3 Description technique de l'application et du matériel

3.1 Interruption de la télécommande

Pour ne pas avoir à faire du *polling* qui ralentirait le programme, nous avons décidé de lire les signaux de la télécommande lors d'une interruption. Par défaut, le pin 7 du port E, est tiré au niveau haut de tension. Donc quand la télécommande commence à envoyer un signal et que le pin 7 du port E passe à l'état bas, une interruption est déclenchée. La routine d'interruption stocke bit par bit le signal dans le registre à deux octets `(b1, b0)`. Après la fin de la réception du signal, nous conservons la valeur de `b0` en la plaçant dans l'espace mémoire à l'adresse `bouton_addr` dans la RAM.

3.2 Gestion des lasers dans la mémoire

Adresse dans la RAM	Espace mémoire	Ligne du LCD
0x1003	0b00000000	Supérieure
0x1002	0b00100100	
0x1001	0b00000000	Inférieure
0x1000	0b00000000	

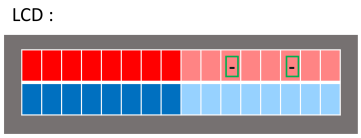

FIGURE 5 – Correspondance entre les espaces mémoire et les cases de l'écran LCD

Les lasers requièrent l'utilisation de 32 bits, soit quatre octets, pour déterminer si un obstacle est présent ou pas dans chacune des cases du LCD. Ainsi, nous avons réservé les espaces mémoires aux adresses `0x1000` à `0x1003` dans la RAM pour les obstacles. La figure 3.4 montre la relation entre les bits en mémoire et l'affichage dans l'écran LCD.

3.3 Interruption par le timer 0

Comme expliqué précédemment, nous stockons la position des obstacles dans la RAM. Ces données sont transférées vers le registre à quatre octets `a` pour être décalés vers la gauche dans la sous-routine `shift_obstacles`. Pour décaler les obstacles vers la gauche, la macro `LSL2` est utilisée une fois pour les octets de la ligne supérieure puis une autre fois pour les octets de la ligne supérieure. Cette routine est appelée lors de l'interruption d'overflow du *timer 0*. Le timer utilise comme source d'horloge le quartz horloger à 32768 Hz avec un prescaler de 32. Donc un *overflow* est censé avoir lieu tous les quarts de seconde. La routine de service d'interruption de ce timer se charge aussi de jouer une note de musique à chaque overflow. La vitesse à laquelle compte le timer fait que la musique est suffisamment fluide, même si les notes sont courtes pour ne pas rompre le fonctionnement du jeu en temps réel.

3.4 Création de caractères personnalisés

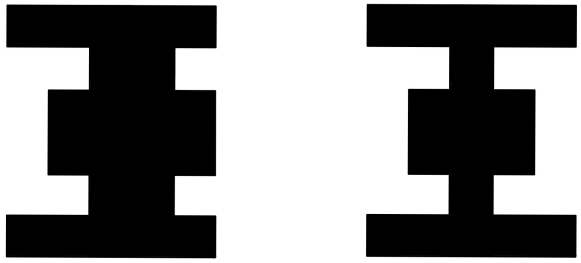

FIGURE 6 – Caractères personnalisés utilisés pour représenter les deux vaisseaux

Afin de représenter les vaisseaux de manière plus réaliste, nous avons créé deux caractères personnalisés. Pour cela, nous avons repris le code de `animation0.asm` et supprimé la partie qui s'occupe du décalage dans la lecture des octets en mémoire programme qui permettent le *scrolling* du caractère. Nous stockons les octets des caractères personnalisés dans la CGRAM grâce aux sous-routines `LCD_storeCGRAM_personnage` et `LCD_storeCGRAM_mechant`, aux adresses `0x00` et `0x01` respectivement.

4 Fonctionnement du programme

Les sections suivantes détaillent le déroulement et le fonctionnement du programme en entrant dans le `main`.

4.1 Écran d'accueil

Au début du `main`, nous entrons dans une première boucle nommée `start_screen`. Nous affichons alors un message sur le LCD invitant l'utilisateur à presser le bouton *power* de la télécommande. Nous testons en boucle le contenu à l'adresse `bouton_addr` dans la RAM, pour vérifier si l'utilisateur veut commencer une partie. Une fois le bouton *power* pressé, nous mettons le nombre de points à zéro et le nombre de vies à cinq aux adresses correspondantes dans la RAM (`points_addr` et `vies_addr`) puis nous entrons dans `game_loop` qui gère le jeu en lui-même.

4.2 La boucle `game_loop`

Lors du déroulement du jeu, une boucle d'événements est répétée indéfiniment. Nous donnons ici ses étapes générales.

4.2.1 Affichage des obstacles

L'affichage sur le LCD est actualisé à chaque passage dans la boucle game_loop du main. Le contenu de l'écran est d'abord effacé, puis trois sous-routines se chargent de l'affichage sur le LCD, il s'agit de `print_personnage`, `print_obstacles` et `print_mechant`. Comme les deux vaisseaux ne se déplacent que sur les mêmes colonnes, nous ne devons gérer que la quatrième et la dernière colonne du LCD séparément. La sous-routine `print_obstacles` ne gère donc que les cases uniquement occupables par des lasers. `print_obstacles` appelle donc la macro `AFFICHAGE_OBSTACLE` pour toutes les cases où un vaisseau ne peut pas se trouver. Cette macro vérifie le bit correspondant à la case du LCD dans les positions mémoires `0x1000` à `0x1003` pour tester si un obstacle doit être affiché. Si le bit est à 1, on appelle la sous-routine putc du fichier `LCD.asm` pour afficher le caractère « - » et s'il est à 0, on décale la position du curseur de 1.

4.2.2 Capteur de position

Au début de la boucle, nous utilisons la sous-routine `sharp` du fichier `sharp.asm` afin de stocker la position du personnage dans le registre r9. Le résultat de `sharp` est stocké dans le registre a0, mais nous copions cette valeur dans `r9` pour pouvoir la réutiliser plus tard. Quant à l'interprétation du résultat de `sharp`, nous avons décidé que pour une valeur retournée par `sharp` plus grande que 200, la position du joueur est définie comme étant à la ligne du dessous. Inversement pour une valeur plus petite que 200, le joueur est à la ligne du dessus.

4.2.3 Collisions

Nous testons les collisions grâce à la sous-routine `test_collisions`. La routine vérifie si un obstacle est présent à la position du premier joueur, déterminée par `r9`. Un obstacle est à la position du joueur si le bit 3 de l'octet à l'adresse `obstacles_addr + 3` lorsque le joueur est en haut, ou si le bit 3 de de l'octet à l'adresse `obstacles_addr + 1` est à 1 lorsque le joueur est en bas. S'il y a une collision, l'obstacle est détruit (le bit correspond en mémoire est mis à zéro) et le nombre de vies du personnage est décrémenté. Si le nombre de vies atteint zéro, on sort de la boucle `game_loop` et on arrive dans `game_over` que l'on verra plus tard.

4.2.4 Affichage du premier vaisseau

La routine `print_personnage`, quant à elle, vérifie la position de la main du joueur dans le registre `r9`. La routine appelle la macro `AFFICHAGE_PERSONNAGE` affiche un caractère ASCII personnalisé à la quatrième colonne de la bonne ligne. La bonne ligne est déterminée conformément à l'interprétation expliquée dans la sous-section 4.2.2. On teste ensuite la case où ne se situe pas le personnage pour vérifier si un obstacle s'y trouve. Dans le cas échéant, nous affichons le caractère « - ».

4.2.5 Choix de la position du second joueur et affichage de son vaisseau

La routine `modifier_position_mechant` teste la valeur contenue à l'adresse `bouton_addr`. Si celle-ci correspond au bouton de changement de chaîne vers le haut, le second vaisseau est placé à la ligne du haut. Si elle correspond au bouton de changement de chaîne vers le bas, le second vaisseau est placé à la ligne du bas. Le vaisseau est alors affiché grace à la routine `print_mechant` qui appelle la macro `AFFICHAGE_MECHANT` sur la case trouvée dans `modifier_position_mechant`.

4.2.6 Activation de la musique

La sous-routine `music_toggle` est appelée et teste si le bouton mute a été pressé (c'est-à-dire que la valeur 0x3d est contenue dans l'emplacement `bouton_addr`), si c'est le cas, la routine utilise un filtre avec la fonction logique eor pour inverser le registre r7 où est stockée le bit qui définit si la musique est activée ou non. Pour que le nouveau réglage reste activé, même si l'appui de l'utilisateur sur la touche mute est prolongé et dure plusieurs cycles de game_loop, nous stockons la valeur 0x07 dans r8, cette valeur est décrémentée à chaque nouveau passage dans la boucle jusqu'à atteindre la valeur 0x00, en attendant, il n'est pas possible de modifier l'état de la musique.

4.2.7 Test d'une victoire

Si le nombre de points dépasse le nombre de points maximal stockable dans un octet (255 points), le programme sort de la boucle et appelle la routine `game_won`.

4.3 `game_over` et `game_won`

Les sous-routines `game_over` et `game_won` sont assez similaires, la première, appelée en cas de victoire du joueur avec la télécommande, le score du second joueur est alors affiché. La seconde quant à elle affiche tout simplement le message `"Max score !"`. Après trois secondes, ces deux sous-routines sautent à l'étiquette de l'écran de démarrage et les deux joueurs peuvent relancer une partie.

5 Présentation des modules

Pour bien séparer les différentes parties du jeu dans le code, nous avons séparé le programme en plusieurs fichiers. Outre les fichiers tirés directement du livre *Microcontrôleurs*[1], nous avons 5 fichiers personnels au total (11 avec les autres fichiers). Nous les pésenterons un par un.

1. Alexandre Schmid, Raphael Holzer, "*Microcontrôleurs, Théorie et Pratique de l'AVR*," EPFL Press, February 2022.

5.1 `main.asm`

Il s'agit du module principal, il contient les routines qui gèrent les différents modes de jeu. Ces routines appellent les routines des autres modules afin de faire fonctionner le programme ainsi que les interruptions.

- `ext_int7` ;
- `overflow0` ;
- `BLINKING_MESSAGE` ;
- `main` ;
- `start_screen` ;
- `start_game` ;

5.2 `obstacles.asm`

Ce module contient toutes les sous-routines utilisées pour les obstacles, leurs déplacements et leur affichage. Les routines et macros suivantes sont placées dedans :

- `shift_obstacles` ;
- `print_obstacles` ;
- `AFFICHAGE_LIGNE_OBSTACLES` ;
- `AFFICHAGE_OBSTACLE` ;

5.3 `joueur1.asm`

Ce module regroupe toutes les sous-routines utilisées pour afficher le personnage 1, son nombre de vie, vérifier la présence d'éventuelles collisions avec des lasers et de compter son nombre de points. Voici les routines et macros présentes dans ce module :

- `AFFICHAGE_PERSONNAGE` ;
- `LCD_storeCGRAM_personnage` ;
- `print_personnage` ;
- `test_points` ;
- `affichage_vies` ;

5.4 `joueur2.asm`

Ce module est très similaire au précédent, il utilise des routines analogues à celles utilisées pour `joueur1.asm` mais pour le second joueur. Cependant, une routine pour poser des obstacles est également implémentée.

- `AFFICHAGE_MECHANT` ;
- `LCD_storeCGRAM_mechant` ;
- `modifier_position_mechant` ;
- `print_mechant` ;
- `pose_obstacle` ;

5.5 `musique.asm`

Ce module gère la possibilité d'éteindre et de rallumer la musique en jeu ainsi que le stockage des notes de la musique. Nous utilisons les deux routines suivantes :

- `music_toggle` ;
- `decrement_delay_counter` ;

5.6 Liste des autres fichiers utilisés

Notre code utilise aussi les fichiers suivants sans modification :

- `macros.asm` ;
- `definitions.asm` ;
- `sharp.asm`, qui permet une lecture du capteur de distance branché sur le port D ;

— `lcd.asm`, qui permet de configurer le LCD pour l'utiliser avec quelques sous-routines ;
— `printf.asm`, qui permet des affichages de texte formattés sur le LCD ;
— `sound.asm`, qui nous a permis de jouer des sons de durée déterminée pour faire de la musique.

6 Description de détail de l'accès aux périphériques

6.1 Télécommande et capteur infrarouge

D'abord, il faut s'assurer que la télécommande est configurée en mode RC5 et pas en un autre mode. Le capteur IR doit être branché sur le port E. Pour accéder à la télécommande avec le capteur infrarouge, nous avons largement repris le code du fichier `ir_rc5.asm`. Nous n'avons pas du modifier les *timinigs*. Nous avons toutefois utilisé le code légèrement différemment, puisqu'il est contenu dans une interruption au lieu de faire partie du `main`. Dans ce code, le signal est reconstruit en plaçant le bit lu à chaque cycle dans le fanion C, puis par rotation vers la gauche d'un registre à deux bits, les valeurs du signal sont conservées dans l'ordre de lecture. Ensuite, le complément de la valeur lue est pris pour retrouver le bon signal. Dans notre code, nous avons omis cette dernière partie, et l'avons remplacée par une addition de 48 à la valeur lue. Ceci nous avait simplifié le déverminage au début du projet, puis nous l'avons laissé comme ça pour la suite.

6.2 Capteur de distance Sharp GP2Y0A2

Le capteur de distance doit être branché sur le port D. Dans le fichier `sharp.asm`, l'obtention des données du capteur de distance se fait en suivant le même principe que celui expliqué pour la télécommande, c'est-à-dire que l'instruction `rol` permet de conserver un par un les bits du signal à travers le fanion C. Ensuite la valeur lue est interprétée de manière binaire dans notre programme, comme expliqué en section 4.2.2.

6.3 LCD Hitachi44780U

Nous utilisons les sous-routines des fichiers `printf.asm` et `lcd.asm` afin d'afficher des caractères sur l'écran LCD. Le jeu en lui-même fait appel å la sous-routine `putc` pour afficher les obstacles qui sont représentés par de simples caractères ASCII. Les écrans de démarrage et de Game over doivent afficher des chaînes de caractères, pour cela, nous préférons utiliser la routine `printf` qui est mieux indiquée dans ce genre de cas.

7 Annexes

7.1 Code source du projet

Cette section contient tout le code source du projet, sauf le code des fichiers non modifiés dont la liste est donnée en section 5.6.

```
/*
 * main.asm
 *   Authors: Luca Jimenez, Geoffroy Rrenault
 */

.include  "macros.asm"        ; include macro definitions
.include  "definitions.asm"     ; include register/constant definitions

; === interrupt vector table ===
.org    0
      rjmp  reset
.org    INT7addr
      rjmp  ext_int7
.org    OVF0addr
      rjmp  overflow0

; === definitions ===
.equ    T1              = 1778      ; bit period T1 = 1778 usec
.equ    bouton_addr         = 0x0fe0
.equ    pos_mechant_addr      = 0x0fe1
.equ    vies_addr         = 0x0fe2
.equ    points_addr         = 0x0ff0
.equ    obstacles_addr        = 0x1000
.equ    addr_caractere_personnage = 0x00
.equ    addr_caractere_mechant    = 0x01
.equ    mute_button         = 0x3d
.equ    five_button         = '5'
.equ    power_button        = 0x3c
.equ    channel_up_button     = 'P'
.equ    channel_down_button     = 'Q'
.set    timer0            = 255

; === interruption service routines ===
; purpose: receive ir signal and store result in the RAM
; arg:  string; used: r1,b0,b1,b2,b3
ext_int7:
      in    _sreg, SREG     ; store context
      PUSH4 b3,b2,b1,b0
      PUSHX

      CLR2  b1,b0       ; clear 2-byte register
      ldi   b2,14       ; load bit-counter
      WAIT_US (T1/4)        ; wait a quarter period

      loop_int7:  P2C   PINE,IR   ; move Pin to Carry (P2C)
      ROL2  b1,b0       ; roll carry into 2-byte reg
      WAIT_US (T1-4)        ; wait bit period (- compensation)
      DJNZ  b2,loop_int7    ; Decrement and Jump if Not Zero

      subi  b0, -48     ; interpretation of result
      LDIX bouton_addr
      st x, b0

      POPX
      POP4  b3,b2,b1,b0
      out   SREG, _sreg   ; restore context
      reti

; purpose: shift obstacles and play sound at regular intervals of time
```

```
; arg:  string; used: r1,r16,r17,r26,r27,r18,r0,b0
overflow0:
      in    _sreg, SREG   ; store context
      push  w
      push  _w
      PUSH4 a3,a2,a1,a0
      PUSH4 b3,b2,b1,b0
      PUSHZ
      PUSHX         ; store pointer x in the stack

      ldi   _w, -timer0+3
      out   TCNT0, _w   ; restart timer

      ; points
      LDIX  points_addr
      ld    r16,x       ; load points from RAM
      inc   r16         ; increment points
      st    x, r16        ; store points in RAM

      ; obstacles
      rcall shift_obstacles ; shift the obstacles left

      ; music
      tst   r7            ; test if music on or off
      breq  end_ovf0
      ldi   zl, low(2*tetris)   ; pointer z to begin of musical score
      ldi   zh, high(2*tetris)
      add   zl, r6
      lpm             ; load note to play
      tst   r0          ; test end of music (NUL)
      brne  PC+2
      clr   r6          ; clr r6 (notes offset) if end of music reached
      mov   a0,r0       ; move note to a0
      ldi   b0,7        ; load play duration (50*2.5ms = 125ms)
      rcall sound       ; play the sound
      inc r6

    end_ovf0:
      POPX
      POPZ
      POP4  b3,b2,b1,b0
      POP4  a3,a2,a1,a0
      pop   _w
      pop   w
      out   SREG, _sreg     ; restore context
      reti

; purpose: display a message for 500 ms and then delete it
; arg:  string; used:
.macro BLINKING_MESSAGE
      rcall LCD_home        ; place cursor to beginning of LCD
      PRINTF  LCD           ; print formatted
    .db @0,0
      WAIT_MS 500
      rcall LCD_home
      PRINTF  LCD           ; print formatted
    .db "                 ",0
      WAIT_MS 500
    .endmacro

; === reset routine ===
reset:
      LDSP  RAMEND          ; load Stack Pointer
      OUTI  DDRE, (1 << SPEAKER)  ; set PORTE IR sensor pins as inputs
                        ; and buzzer pin as output
      OUTI  DDRB, 0xff        ; set PORTB (LEDs) as output
```

```
126
127       OUTI  EIMSK, 0b10000000   ; enable int7
128       OUTI  EICRB, 0b00000000   ; make int 7 happen for low voltage level
129
130       OUTI  ASSR, (1<<AS0)      ; set clock source to quartz for timer 0
131       OUTI  TCCR0,3         ; set prescaler to to
132       OUTI  TIMSK,0         ; not enable timer interrupts for now
133
134       rcall LCD_init
135       rcall sharp_init
136
137       rcall LCD_clear
138
139       rcall LCD_storeCGRAM_personnage ; load player1 custom character to CGRAM
140       rcall LCD_storeCGRAM_mechant    ; load player2 custom character to CGRAM
141
142       clr r6
143       clr r7
144
145       sei
146       rjmp  main
147
148 ; === included files ===
149 .include  "lcd.asm"     ; include the LCD routines
150 .include  "sharp.asm"     ; include the SHARP GP2D02 distance sensor routines
151 .include  "printf.asm"    ; include formatting printing routines
152 .include  "sound.asm"     ; include sound routines
153 .include  "joueur1.asm"   ; incude player1 routines
154 .include  "joueur2.asm"   ; incude player1 routines
155 .include  "obstacles.asm"   ; incude player1 routines
156 .include  "musique.asm"   ; include music score
157
158 ; === main ===
159 main:
160       LDSP  RAMEND            ; the stack will be overwrtitten
161       rcall LCD_clear
162       rcall start_screen
163       rcall start_game
164   game_loop:
165       rcall print_obstacles
166       rcall sharp           ; get data from Sharp distance sensor
167       mov   r9, a0            ; store result from sharp in r9
168       rcall test_collision        ; test for collision between player1 and obstacles
169       rcall test_points
170       rcall print_personnage      ; display player1
171       rcall modifier_position_mechant ; modify (if needed) the position of player 2
172       rcall print_mechant
173       rcall affichage_vies        ; display lives on LEDs
174       rcall pose_obstacle         ; laying an obstacle if button '5' is pressed
175       rcall music_toggle
176     reset_button_data:
177       LDIX  bouton_addr       ;
178       clr   r16               ;
179       st    x, r16            ; clear button informatiion stored in RAM
180       rjmp  game_loop         ; continue game loop
181
182 ; === subroutines ===
183
184 ; purpose: wait for players to be ready to start the game
185 ; arg: ; used: r16, r26,r27 (x)
186 start_screen:
187       rcall LCD_home
188       PRINTF  LCD                   ; print formatted
189       .db   "Press power      ",0   ; start screen message
190       rcall LCD_lf
191       PRINTF  LCD             ; print formatted
```

```
      .db   "button to start  ",0
      LDIX  bouton_addr
    wait_for_power_button:
      ld    r16,x
      cpi   r16, power_button
      breq  PC+2
      rjmp  wait_for_power_button
      ret

; purpose: setup lives, points, obstacles and music before starting the game
; arg: ; used: r6, r8, r16, b3, b2, b1, b0, r26, r27 (x)
start_game:
      LDIX  points_addr       ; reset points
      ldi   r16, 0            ;
      st    x, r16            ;
      LDI4    b3,b1,b2,b0,0   ; reset obstacles
      LDIX  obstacles_addr    ;
      STX4  b3,b1,b2,b0       ;
      LDIX  vies_addr
      ldi   r16, 5          ; initialise number of lives to 5
      st    x, r16
      clr   r6            ; clear r6 wich gives the position
                        ; of the current musical note
      clr   r8              ; clear r8 which is a counter that allows
                        ; the user to change music if it is at 0
      OUTI  TIMSK,(1<<TOIE0)    ; activate overflow timer 0, which controls
                        ; the shifting of the obstacles,
                        ; the music and the point incrments
      ret

; purpose: display message when game is lost
; arg: ; used: b3,b2,b1,b0
game_over:
      OUTI  PORTB, 0xff       ; turn off LEDs

      LDIX  points_addr
      ld    b0, x         ; get points from RAM
      clr   b1            ;
      clr   b2            ;
      clr   b3            ;

      rcall LCD_clear
      rcall LCD_lf
      PRINTF    LCD         ; print formatted
    .db " Point(s) :  ",FDEC,b,0

      BLINKING_MESSAGE "   GAME OVER !   "
      BLINKING_MESSAGE "   GAME OVER !   "
      BLINKING_MESSAGE "   GAME OVER !   "

      OUTI  TIMSK,0         ; disable timer 0
      jmp     main

; purpose: display message when game is won
; arg: ; used:
game_won:
      OUTI  PORTB, 0x00       ; turn on LEDs
      rcall LCD_clear
      rcall LCD_lf
      PRINTF    LCD         ; print formatted
    .db " Max Points !  ",0

      BLINKING_MESSAGE "   You won !     "
      BLINKING_MESSAGE "   You won !     "
      BLINKING_MESSAGE "   You won !     "

```

```
258       OUTI  TIMSK,0          ; disable timer 0
259         jmp     main
```

Listing 1 – main

```
1  /*
2   * obstacles.asm
3   *   Authors: Luca Jimenez, Geoffroy Renaut
4   */
5
6  ; === macros ===
7
8  ; purpose: print a single obstacle at a given position on the LCD
9  ; arg: register, immediate value (from 0 to 7); used: r18 (a0)
10 .macro    AFFICHAGE_OBSTACLE    ; prints an 'o' if there is an obstacle on this spot
11                                    ; on the LCD, prints ' ' otherwise
12       ldi   a0,'-'       ; load '-' in a0
13       sbrs  @0, @1
14       ldi   a0,' '       ; load ' ' in a0 is bit @1 of @0 is clear
15       rcall lcd_putc     ; put corresponding charcter in the LCD
16       .endmacro
17
18 ; purpose: print entire line of obstacles on LCD
19 ; arg: immediate value, register, register ; used: r18 (a0)
20 .macro    AFFICHAGE_LIGNE_OBSTACLES
21       CA    LCD_POS, @0          ; set initial position of cursor
22
23       AFFICHAGE_OBSTACLE @1, 7       ; filling the rectangles of the LCD one by one
24       AFFICHAGE_OBSTACLE @1, 6
25       AFFICHAGE_OBSTACLE @1, 5
26       rcall   LCD_cursor_right     ; skip the column of the charcater
27       AFFICHAGE_OBSTACLE @1, 3
28       AFFICHAGE_OBSTACLE @1, 2
29       AFFICHAGE_OBSTACLE @1, 1
30       AFFICHAGE_OBSTACLE @1, 0
31
32       AFFICHAGE_OBSTACLE @2, 7
33       AFFICHAGE_OBSTACLE @2, 6
34       AFFICHAGE_OBSTACLE @2, 5
35       AFFICHAGE_OBSTACLE @2, 4
36       AFFICHAGE_OBSTACLE @2, 3
37       AFFICHAGE_OBSTACLE @2, 2
38       AFFICHAGE_OBSTACLE @2, 1
39       .endmacro
40
41 ; === subroutines ===
42
43 ; purpose:  shift the obstacles left in the memory
44 ; arg: ; used:  a0,a1,a2,a3
45 shift_obstacles:
46
47       LDIX  obstacles_addr
48       LDX4  a0,a1,a2,a3     ; load obstacles from RAM
49
50       LSL2  a0,a1       ; shift upper line of obstacles left
51       LSL2  a2,a3       ; shift lower line of obstacles left
52
53       LDIX  obstacles_addr
54       STX4  a0,a1,a2,a3     ; store shifted obstacles in RAM
55
56       ret
57
58 ; purpose: display the obstacles where there can't be any players
59 ; arg: ; used:  b0,b1,b2,b3
60 print_obstacles:
61       LDIX  obstacles_addr
62       LDX4  b0,b1,b2,b3
```

```
63
64      AFFICHAGE_LIGNE_OBSTACLES 0x00, b0, b1    ; print first row of obstacles
65      AFFICHAGE_LIGNE_OBSTACLES 0x40, b2, b3    ; print second row of obstacles
66
67      ret
```

Listing 2 – obstacles.asm

```
1  /*
2   * joueur1.asm
3   *   Authors: Luca Jimenez, Geoffroy Renaut
4   */
5
6    ; === custom character ===
7   personnage:
8  .db     0b00000000,0b00011111,0b00000100,0b00001110,0b00001110,0b00000100,0b00011111
       ,0b00000000
9
10 ; purpose:
11 ; arg: ; used:
12 .macro    AFFICHAGE_PERSONNAGE
13
14      CA    LCD_POS, @0
15
16      LDIX  obstacles_addr+@1
17      ld    b0, x
18      sbrc  b0, 4
19      ldi   a0,'-'
20      sbrs  b0, 4
21      ldi   a0,' '
22      rcall   lcd_putc
23
24      CA    LCD_POS, @2
25      CA    lcd_putc, 0x00
26      .endmacro
27
28 ; purpose: store cusstom character of player1 in the CGRAM
29 ; arg: ; used:
30 LCD_storeCGRAM_personnage:
31      lds   u, LCD_IR           ;read IR to check busy flag  (bit7)
32      JB1   u,7,LCD_storeCGRAM_personnage ;Jump if Bit=1 (still busy)
33      ldi   r16, 0b01000000         ;2MSBs:write into CGRAM(instruction),
34                            ; 6LSBs:address in CGRAM and in charact.
35      sts   LCD_IR, r16           ;store w in IR
36      ldi   zl,low(2*personnage)+7
37      ldi   zh,high(2*personnage)
38      mov   r23,zl
39      dec   r23
40      mov   r24,r23     ;store upper limit of character in memory
41      ldi   r18,8     ;load size of caracter in table arrow0
42      sub   zl,r22      ;subtract current value of moving offset
43
44    loop01:
45        lds   u, LCD_IR
46      JB1   u,7,loop01
47      lpm           ;load from z into r0
48      mov   r16,r0
49      adiw  zl,1
50      mov   r23,r24     ;garantee z remains in character memory
51      sub   r23,zl      ;zone, if not then restart at the begining
52      brge  _reg01      ;of character definition
53      subi  zl,8
54
55      _reg01: sts LCD_DR, r16   ;load definition of one charecter line
56      dec   r18
57      brne  loop01
58      rcall LCD_home    ;leaving CGRAM
```

```
59        ret
60
61  ; purpose:  display player1 on the LCD
62  ; arg: r9; used: r17
63  print_personnage:
64        ldi   r17, 200      ;
65        cp    r9, r17       ;
66        brlo  personnage_up   ; check if value given by distnace sensor
67                        ; is lower than 200
68      personnage_down:
69        AFFICHAGE_PERSONNAGE 0x03,3,0x43
70        rjmp    end_print_personnage
71      personnage_up:
72        AFFICHAGE_PERSONNAGE 0x43,1,0x03
73      end_print_personnage:
74        ret
75
76  ; purpose: determine if there if an obstacle has the same position as the player
77  ; arg: ; used: r9, r17, r22 (b0), (r26, r27)(x),
78  test_collision:
79        ldi   r17, 200
80        cp    r9, r17
81        brlo  up_test_collision   ; test wether the player1 is
82                        ; in the lower or upper row
83      down_test_collision:
84        LDIX  obstacles_addr+1    ; get lower left group of obstacles
85        ld    b0, x
86        sbrs  b0, 4         ;
87        rjmp  end_test_collision    ; end subroutine if there is no obstacle
88                        ; on the same spot as the player1
89        cbr   b0, (1<<4)        ; delete obstacle if collision took place
90        st    x, b0         ; store updated set of obstacles
91        rjmp  diminuer_vies
92
93      up_test_collision:
94        LDIX  obstacles_addr+3    ; get upper left group of obstacles
95        ld    b0, x         ;
96        sbrs  b0, 4         ;
97        rjmp  end_test_collision    ; end subroutine if there is no obstacle
98                        ; on the same spot as the player1
99        cbr   b0, (1<<4)        ; delete obstacle if collision took place
100       st    x, b0         ; store updated set of obstacles
101
102     diminuer_vies:
103       ldix  vies_addr
104       ld    r16, x          ; get lives from RAM
105       dec   r16              ; decrease live
106       st    x, r16          ; store updated lives in RAM
107
108     end_test_collision:
109       LDIX  vies_addr       ; get lives from RAM
110       ld    r16, x          ;
111       cpi   r16, 0x00       ; test if player has zero lives
112       brne  not_game_over
113       jmp   game_over       ; go to game_over routine if the
114                       ; player has zero lives
115     not_game_over:
116       ret
117
118 ; purpose: check if maximum number of points (255) has been reached
119 ; arg: ; used: r16
120 test_points:
121       LDIX  points_addr
122       ld    r16, x
123       cpi   r16, 0xff
124       brlo  not_won
```

```
      jmp   game_won
    not_won:
      ret

; purpose: diplay lives on LEDs
; arg: ; used: r16, r17
affichage_vies :
      clr   r17
      LDIX  vies_addr     ;
      ld    r16, x        ; get lives from RAM
      sec                 ; set carry
      rol   r17           ; rotate r17 left (carry enters r17)
      dec   r16           ;
      brne  PC-3          ; repeat set carry and rotate left for the all lives
      com   r17           ; complement r17 before sending signal to LEDs
      out   PORTB, r17        ; send signal to LEDs
      ret
```

Listing 3 – joueur1.asm

```
/*
 * joueur2.asm
 *   Author: Luca Jimenez, Geoffroy Renaut
 */

 ; === custom character ===
 mechant:
.db     0b00000000,0b00000000,0b00011111,0b00000110,0b00001111,0b00001111,0b00000110
   ,0b00011111

 ; === macros ===
; purpose: draw player 2 on the correct row
; arg: ; used:
.macro    AFFICHAGE_MECHANT
      CA    LCD_pos, @0           ; move cursor on last rectangle of the second line
      CA    LCD_putc, addr_caractere_mechant  ; put custom character
                            ; on desired rectangle
      CA    LCD_pos, @1           ; move cursor on last rectangle of the first line
      CA    LCD_putc, ' '         ; put space in the desired rectangle
      .endmacro

 ; === subroutines ===

; purpose: store player 2's custom character on the CGRAM
; arg: ; used: r16, z, r18, r23, r24
LCD_storeCGRAM_mechant:
      lds   u, LCD_IR                ;read IR to check busy flag  (bit7)
      JB1   u,7,LCD_storeCGRAM_mechant    ;Jump if Bit=1 (still busy)
      ldi   r16, 0b01001000                ;2MSBs:write into CGRAM(instruction),
                                           ;6LSBs:address in CGRAM and in charact.
      sts   LCD_IR, r16                    ;store w in IR
      ldi   zl,low(2*mechant)+9
      ldi   zh,high(2*mechant)
      mov   r23,zl
      dec   r23
      mov   r24,r23     ;store upper limit of character in memory
      ldi   r18,8     ;load size of caracter in table arrow0
      sub   zl,r22      ;subtract current value of moving offset

       loop02:
        lds   u, LCD_IR
      JB1   u,7,loop02
      lpm           ;load from z into r0
      mov   r16,r0
      adiw  zl,1
      mov   r23,r24     ;garantee z remains in character memory
      sub   r23,zl      ;zone, if not then restart at the begining
```

```
      brge  _reg02      ;of character definition
      subi  zl,8

      _reg02: sts LCD_DR, r16   ;load definition of one charecter line
      dec   r18
      brne  loop02
      rcall LCD_home    ;leaving CGRAM
      ret


; purpose:  Modify player 2's position according to the pressed button
; arg: ; used:  r16,r17,x
modifier_position_mechant :
      LDIX  bouton_addr
      ld    r16, x          ; get button info from RAM
      cpi   r16, channel_up_button
      brne  not_up
      LDIX  pos_mechant_addr
      ldi   r17, 1          ; set player2 position to 1
      st    x, r17
      rjmp  fini_pos_mechant
    not_up:
      cpi   r16, channel_down_button
      brne  fini_pos_mechant
      LDIX  pos_mechant_addr
      ldi   r17, 0          ; set player2 position to 0
      st    x, r17
    fini_pos_mechant:
      ret

; purpose:  display player 2 on the LCD
; arg: ; used: r16,r18 (a0),r26,r27
print_mechant:
      push  r16
      push  a0

      LDIX  pos_mechant_addr  ; access player2's
      ld    r16, x        ; position in the RAM
      cpi   r16,1       ; compare position to 1
      brne  mechant_bas     ; branch to mechant_bas if position is not equal to 1
  mechant_haut :
      AFFICHAGE_MECHANT 0x0f,0x4f
      rjmp  end_print_mechant
  mechant_bas :
      AFFICHAGE_MECHANT 0x4f,0x0f
  end_print_mechant:
      pop   a0
      pop   r16
      ret

; purpose: lay an obstacle at the position of the player2
; arg: ; used: r16, r26, r27
pose_obstacle:
      ldix  bouton_addr         ;
      ld    r16,  x       ; get pressed button info from RAM
      cpi   r16, five_button    ; test if button '5' is pressed
      brne  end_pose_obstacle   ;
      LDIX  pos_mechant_addr
      ld    r16, x
      cpi   r16,1         ; check where the player2 is
      brne  obstacle_bas      ; branch to obstacle bas if player2 is in the lower row
      LDIX  obstacles_addr+2    ;
      ld    r16, x          ; get upper right set of obstacles
      sbr   r16,0b00000001      ; set lsb of the set of obsatcles
      st    x,r16         ; store updated upper right set of obstacles
      rjmp  end_pose_obstacle
```

```
  obstacle_bas:
      LDIX  obstacles_addr
      ld    r16, x            ; get lower right set of obstacles
      sbr   r16,0b00000001        ; set lsb of the set of obsatcles
      st    x,r16           ; store updated lower right set of obstacles
  end_pose_obstacle:
      ret
```

Listing 4 – joueur2.asm

```
/*
 * musique.asm
 *   Authors: Luca Jimenez, Geoffroy Renaut
 */

 ; === music score ===
tetris:
.db mi4,si3,do4,re4,do4,si3,la3,la3,do4,mi4,re4,do4,si3,si3,do4,re4,mi4,do4,la3,la3
.db re4,fa4,la4,so4,fa4,mi4,do4,mi4,re4,do4,si3,si3,do4,re4,mi4,do4,la3,la3, 0, 0, 0,
    0

; purpose: enable/disable the music
; arg:  string; used:  r7,r8,r16
music_toggle:
      LDIX  bouton_addr         ;
      ld    r16,  x         ; get pressed button info from RAM
      cpi   r16, mute_button      ; test if the mute button was pressed
      brne  decrement_delay_counter
      ldi   r16, 0x00
      cp    r8, r16           ; if mute button was pressed,
                        ; test if r8 is clear
      brne    decrement_delay_counter
      ldi   r16, 0x05         ;
      mov   r8, r16           ; r8 is used as a buffer to counter
                        ; to avoid rapid toggling of the music
      ldi   r17, 0x01         ; if r8 is clear, toggle r17
      eor   r7, r17


    decrement_delay_counter:
      ldi   r16, 0x00         ;
      cp    r8, r16           ;
      breq  PC+2            ;
      dec r8                ; if r8 is not clear, decrement r8

      ret
```

Listing 5 – musique.asm