Rapport de projet de fin de semestre

BÉNAND Jacques 325957, BAHUREL Benjamin 326888

30 mai 2022

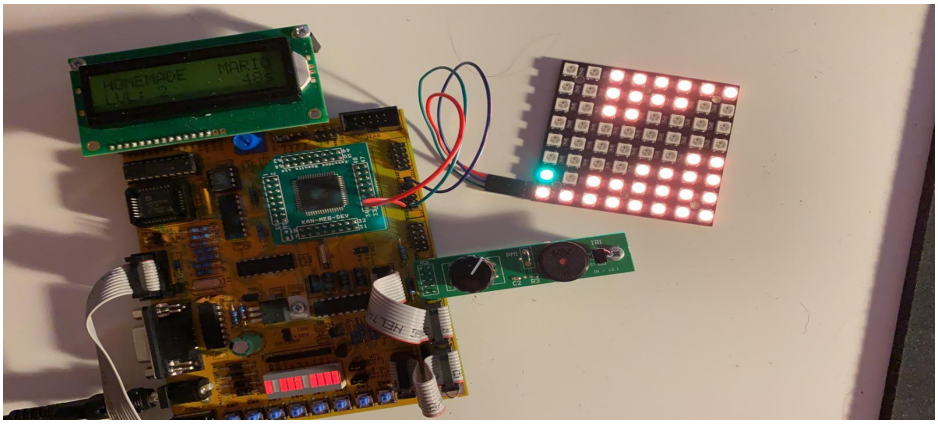

FIGURE 1 – Homemade Mario

1 Introduction

Dans le cadre du cours de Microcontrôleurs, il nous a été demandé de réaliser un travail de fin de semestre permettant d'organiser et élaborer un projet à base de microcontrôleur. Un périphérique externe nous a été imposé : une matrice de LEDs 8x8. L'application a été développée via Atmel Studio 7.0 et a été importée sur une carte ATmega128 avec une architecture AVR.

2 Description générale

Étant donné la liberté que nous avions sur le choix du projet, nous avons décidé de développer un jeu vidéo similaire à Super Mario Bros sur la matrice de LEDs.

2.1 Périphériques utilisés

Pour réaliser notre application, nous avons décidé d'utiliser les périphériques suivants :

- Matrice de LEDs 8x8 : ws2812b
- Affichage LCD 2x16 : Hitachi44780U 2x16 LCD
- Encodeur angulaire : rotary encoder
- Buzzer piezzo-électrique

2.2 Principe de l'application

Comme précisé précédemment, nous avons réalisé un jeu vidéo similaire à Super Mario Bros. Le joueur doit déplacer, à l'aide de boutons poussoirs, un personnage représenté par un pixel dans une carte remplie d'obstacles que ce dernier doit éviter afin d'arriver à la fin du niveau. La collision du personnage avec un onstacle provoque la fin du jeu et un retour à l'écran d'accueil.

3 Description détaillée de l'application

3.1 Mode d'emploi

Afin de pouvoir utiliser l'application, il faut d'abord mettre la carte sous tension puis l'allumer. Après une courte musique d'introduction, l'affichage explique la démarche à suivre pour lancer la partie. Premièrement, il faut appuyer sur le bouton 6 pour passer au choix du son. À ce moment-là, on peut choisir la vitesse des sons utilisés lors de la partie via l'encodeur angulaire qu'il faut maintenir pressé et tourner pour sélectionner. Ensuite en appuyant le bouton 7, on accède au choix du niveau qui se fait de nouveau via la rotation de l'encodeur angulaire mais cette fois-ci sans le maintenir appuyé. Presser le bouton 6 lance la partie et affiche le niveau. Le bouton 0 est utilisé pour se déplacer vers la droite et le bouton 1 sert à sauter. Le joueur a 60 secondes pour terminer le niveau. Si le personnage percute latéralement un bloc orange ou qu'il est à court de temps, la partie est terminée, sinon l'arrivée au bout du niveau se traduit par la victoire du joueur. Dans les deux cas, le joueur peut retenter sa chance après avoir resélectionné les paramètres présentés ci-dessus.

3.2 Communication avec le monde extérieur

La matrice de LEDs est connectée au **PORTE**, l'écran LCD est connecté au port **LCD**, les boutons-poussoirs sont connectés au **PORTD** et l'encodeur angulaire qui se trouve sur le module M2 (module sur lequel se trouve aussi le buzzer que nous utilisons) est connecté au **PORTB**.

3.3 Interruptions

Nous utilisons deux interruptions dans le cadre de notre application pour réaliser deux fonctions différentes. La première permet de déplacer le personnage dans le niveau et la deuxième permet d'effectuer un décompte durant lequel le joueur doit finir le niveau afin de compléter le jeu. Nous allons détailler l'implémentation de ces deux interruptions ci-dessous.

3.3.1 Interruption externe 0

Comme précisé ci-dessus, cette première interruption permet de déplacer le eprsonnage dans le niveau, plus précisément elle permet de décaler la map d'un pixel vers la gauche. Cette interruption est placée sur la ligne INT0 qui a la plus grande priorité parmi les interruptions. L'évènement externe qui déclenche l'interruption INT0 est l'utilisation du bouton poussoir 0 (bit 0 de PIND). Lorsque le bouton 0 est pressé, on saute à la sous-routine d'interruption de service appelée *shift_jump* qui décale le personnage d'une case sur la matrice si la case d'arrivée est vide. Cette sous-routine décale aussi la position du personnage dans la mémoire pour permettre un affichage cohérent.

Cette interruption est configurée de sorte à avoir comme condition d'interruption un niveau bas, pour cela les bits **ISC01** et **ISC00** du registre **EICRA** sont mis à 0.

3.3.2 Timer0 :

Cette interruption configurée en timer overflow permet de décrémenter un compteur toutes les secondes afin de créer un décompte durant lequel le joueur doit finir le niveau. L'overflow toutes les secondes est réalisé en utilisant le quartz horloger comme source d'horloge avec un prescaler de 1/128. Ce qui nous donne la période suivante :

$$T_{overflow} = \frac{256 * 128}{32768} = 1s$$

La configuration de cette interruption est faite en mettant le bit **TOIE0** du registre **TIMSK** à 1, le bit **AS0** du registre **ASSR** à 1, en chargent la valeur 0x05 dans la définition du prescaler soit en passant les bits **CS02**, **CS01** et **CS00** du registre **TCCR0** respectivement aux valeurs 1, 0 et 1.

Nous avons conscience que ces routines d'interruption sont relativement longues, en revanche cela n'impacte pas notre programme car il est toujours capable de répondre en temps réel.

3.4 Fonctionnement du programme

Nous pouvons décomposer le fonctionnement en 5 parties distinctes :

1. Reset et initialisation du programme
2. Initialisation de la matrice de LEDs et des interruptions (en fonction du niveau choisi)
3. Affichage du niveau et début du jeu
4. Victoire ou défaite, affichage correspondant et réinitialisation du programme (retour à l'étape 1)

Détaillons maintenant chaque partie.

3.4.1 Initialisation de l'application

Ceci constitue la partie *reset* dans le code. Ceci initialise l'écran LCD et l'encodeur rotatif selon les méthodes données dans le cours. Cette partie permet aussi d'initialiser le Stack Pointer au moyen de la macro **LDSP** à l'adresse RAMEND (qui est égal à l'adresse 0x10ff, ceci est défini dans le fichier *m128def.inc*). Une fois ceci réalisé, le programme charge le registre r11 avec la valeur 0x3c, ceci constituera le début du décompte. Ensuite, on affiche le menu de base sur l'écran LCD que le joueur pourra parcourir dans la suite. Une fois toutes ces étapes réalisées, le programme saute à l'étape suivante, *lancement*.

3.4.2 Initialisation de la matrice de LEDs et des interruptions

On commence par initialiser la matrice de LEDs grâce à la sous-routine *ws2812b4_ init* fournie dans le cours. Ensuite, le **PORTD** est configuré en entrée puis le **PORTB** est configuré de sorte à pouvoir utiliser le buzzer et l'encodeur angulaire. L'initialisation des interruptions est ensuite effectuée selon le protocole défini dans la section 3.3. On réalise enfin la définition des constantes que l'on utilisera au cours du programme.

3.4.3 Affichage du niveau et jeu

Après que le joueur ait sélectionné un niveau via le menu, on fait pointer le pointeur Z à l'adresse du début de la map sélectionnée au préalable. Après avoir effectué cette étape, l'affichage sur la matrice de LEDs est réalisé à l'aide des sous-routines données dans le cadre du cours. Une fois ces étapes réalisées, le jeu peut commencer, nous détaillerons les différents modules composant le programme du jeu en lui-même dans la section 3.5.1.

3.4.4 Fin du jeu et réinitialisation du programme

En fonction de si le joueur a réussi ou non le niveau, l'affichage correspondant est réalisé sur l'écran LCD et la musique correspondante est jouée par le buzzer. Ensuite, un branchement à l'étiquette *play_again* est effectué. Ceci permet de "nettoyer" la matrice de LEDs en supprimant tout affichage grâce à la macro **RESET_MAT**. Ceci effectue aussi la macro **CLEAR_ALL** qui met le contenu de tous les registres à 0, réinitialise l'état des ports d'entrée-sortie et l'état des interruptions et des timers. Tout cela permet de recommencer une nouvelle partie dans des conditions initiales satisfaisantes.

3.5 Présentation des modules

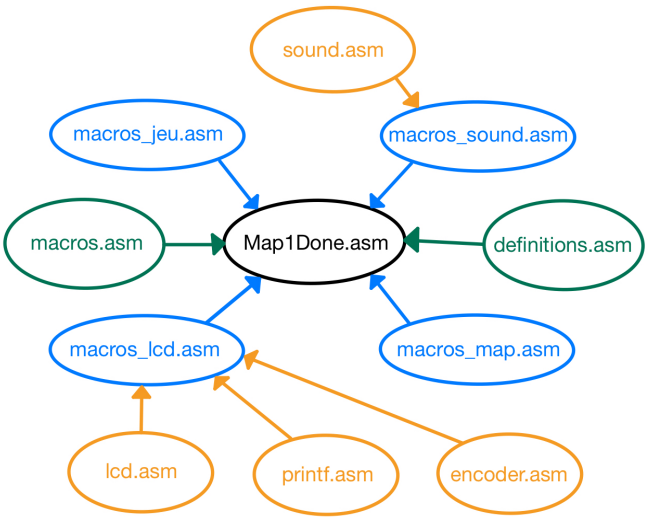

FIGURE 2 – Schéma de dépendances des différents modules

Map1Done.asm est le programme principal, il réalise toute les étapes de la section 3.4 dans l'ordre. Nous pouvons maintenant détailler les différents modules les plus intéressants.

3.5.1 macros_jeu.asm

Ce module contient toutes les fonctions relatives au contrôle du personnage par le joueur, c'est-à-dire le saut du personnage, le déplacement du personnage et l'affichage de ce dernier.

shift_mario_in_memory : Cette fonction permet de sauvegarder l'adresse de la première composante de la LED représentant le personnage dans les registres r12 et r13. Pour cela, l'adresse

pointée par Z est d'abord chargée dans r12 et r13 par les instructions mov, puis on soustrait 0x03 à la valeur de r13 et on soustrait le carry à r12 s'il y en a afin de pouvoir pointer sur l'adresse de la première composante de l'affichage sur la matrice de LEDs (les détails de cet affichage seront approfondis dans la section 4).

jump_mario : Cette fonction est constituée de trois parties. La première avec l'étiquette **jump_mario** a pour but de stocker les valeurs des registres a0, a1, b0, b1 et r0 dans la mémoire données via le pointeur Z avant de jouer le son pour le saut du personnage puis de restituer les valeurs dans les registres correspondants. La deuxième étiquette **loop_jump** contient le code qui fait monter le personnage de 3 cases au maximum si la case au-dessus de ce dernier est vide. Pour réaliser cette fonction, on place le pointeur Z à la position en mémoire du personnage puis on vérifie si la case du dessus est vide, si c'est le cas, on vide le pixel actuel du personnage et on remplit celui du dessus. Une fois ceci réalisé, les pixels sont affichés sur la matrice via la routine *affichage_matrice* puis on répète toutes ces étapes un maximum de 3 itérations grâce à la décrémentation du registre r10 qui a été initialisé auparavant à 0x03. Finalement, l'étiquette **descente** contient la partie qui fait redescendre le personnage une fois que la hauteur maximale du saut a été atteinte. Pour cela, cette partie reprend le principe de la fonction **loop_jump** avec quelques différences. Premièrement, la condition d'arrêt de cette boucle ne vient pas de la décrémentation d'un registre mais de la présence d'un pixel orange en-dessous du personnage. Secondement, l'offset appliqué à la mémoire pour placer le personnage au bon endroit est de -6 et non de +3 comme c'est le cas lors de la montée.

3.5.2 macros_map.asm

Ce module contient toutes les fonctions relatives à la génération des niveaux et à l'affichage sur la matrice de LEDs.

ORANGE et VOID : Ces deux macros permettent de placer en mémoire données des pixels de couleur orange et des pixels vide. Pour cela, la macro charge la valeur hexadécimale de G(green) dans une première case mémoire puis incrémente le pointeur pour placer la valeur hexadécimale de B(blue) dans la deuxième case mémoire puis incrémente une deuxième fois le pointeur pour placer le complosant R(red) dans la troisième case mémoire.

COLONNEX : Ces macros permettent de générer des colonnes d'obstacles en fonction du nombre de changement de couleur dans la colonne. Nous avons décidé d'utiliser des macros au lieu de sous-routines pour leur plus grande modularité qui nous permet d'éviter une multiplication des sous-routines pour chaque cas différent. Par exemple, si l'on veut réaliser un colonne avec une plateforme volante au milieu, cela nous donne 3 changement de couleurs en tout (sol(orange), vide, plateforme(orange), vide). L'exécution de ces macros remplissent 24 octets de mémoire pour réaliser une colonne de 8 pixels sur la matrice de LEDs. L'implémentation de ces macros est dans le fond assez simple : elles prennent $n+1$ arguments pour une colonne avec n changement de couleurs, le premier argument est l'emplacement mémoire du premier pixel à placer et les arguments suivants sont le nombre de cases pour chaque couleur respectivement à placer dans la colonne. Par exemple, la ligne "**COLONNE2** a0, 1, 3, 4" place un pixel orange, 3 pixels vides puis 4 pixels oranges. Cette fonction est réalisée par l'utilisation de boucle dans lesquelles une valeur est décrémentée pour donner le bon nombre de pixels.

LEVEL1 et LEVEL2 : Ces macros permettent de stocker l'entièreté du niveau choisi par le joueur en mémoire données au moyen des macros **COLONNEX** détaillées ci-dessus.

3.5.3 macros_lcd.asm

Ce module contient une grande sous-routine qui gère tout l'affichage sur l'écran LCD durant la phase d'initialisation et de choix des paramètres. C'est aussi dans ce module que l'on effectue la sauvegarde des paramètres sélectionnés par l'utilisateur.

4 Description détaillée de l'accès aux périphériques

4.0.1 Affichage LCD

Nous utilisons la macro **PRINTF** du fichier *printf.asm* pour afficher les chaînes de caractère sur l'écran LCD. Ces chaînes de caractères sont enregistrées dans la mémoire programme à la suite de l'invocation **PRINTF** correspondante. Le module **PRINTF** étant particulièrement flexible et performant, nous avons décidé d'utiliser cette macro dans le cadre de notre programme pour sa facilité d'utilisation et sa modularité. L'écran LCD est accédé à l'initialisation pour mettre en place l'interface utilisateur, pendant le jeu il est accédé lors de chaque Timer Overflow pour afficher le temps restant pour finir le niceau et à la fin du jeu, on y accède de nouveau pour y afficher un message de défaite ou de victoire en fonction de la performance du joueur.

4.0.2 Matrice de LEDs

L'affichage sur la matrice de LEDs ws2812b4 est spécial car un pixel a besoin de trois valeurs huit bits pour être affiché. Ces 3 valeurs forment un ensemble GRB qui permet de transmettre une couleur à la matrice à afficher. Le protocole de transmission se fait en envoyant 24 bits MSB en tête avec un délai de 50µs entre chaque rafraîchissement. Dans notre cas, toutes les valeurs à afficher sur la matrice de LEDs sont stockées en mémoire données et sont récupérée afin d'être transmises au module ws2812b4. Nous avons configuré le programme de sorte que les données soient envoyées à la matrice de LEDs sur le pin 1 du **PORTE**. Pour l'affichage de la map, les colonnes de 8 pixels sont placées en mémoire avec 24 octets consécutifs (3 octets par pixel), nous devons ajouter un offset de 24 (soit 0x18) dans la mémoire pour passer à la colonne d'après, c'est ce qui est réalisé lors du décalage de la map d'un pixel vers la gauche. La transmission d'un pixel de la mémoire donnée vers la matrice se fait par post-incrémentation du pointeur Z : un premier octet est envoyé à la matrice puis le pointeur est incrémenté, un second octet est envoyé puis le pointeur est incrémenté de nouveau, le troisième octet est enfin envoyé est on réincrémente à nouveau le pointeur Z pour pouvoir passer au prochain pixel. On y accède à chaque appel de la sous-routine *affiche_matrice*.

4.0.3 Encodeur angulaire

Pour communiquer avec l'encodeur angulaire, nous utilisons le fichier *encoder.asm* que nous avons modifié afin de pouvoir utiliser le **PORTB** en entrée pour l'encodeur et en sortie pour le buzzer. L'initialisation de l'encoder configure les 3 lignes d'entrées sur les pins 4, 5, et 6 du **PORTB**. La sous-routine *encoder* effectue une première lecture de l'état des trois lignes puis compare l'état actuel avec l'ancien. L'encodeur est accédé lors de la sélection de la vitesse des sons et lors de la sélection du niveau. En ce qui concerne la vitesse des sons, il faut maintenir l'encodeur appuyé (bit T de **SREG** à 1) pour incrémenter ou décrémenter le registre b0 qui sera ensuite

enregistré dans un autre registre pour être utilisé à des fins d'affichage et de branchement. Pour la sélection du niveau, l'encodeur ne doit pas être pressé (T=0) pour changer la valeur du registre a0 qui sera utilisé d'une manière analogue à b0 mais dans des objectifs différents.

4.0.4 Buzzer

Nous avons placé différentes séquences de note à des adresses définies de la mémoire programme (ce sont des Look-Up-Table) qui vont être accédées en fonction de la musique à jouer.À chaque fois qu'un son doit être produit, on invoque la macro **PLAY_SOUND** qui configure le pin 2 du **PORTB** en sortie puis joue le son et remet le pin 2 en entrée une fois que le son a été joué. Lorsqu'un son doit être joué, le pointeur Z va pointer au début de la table puis les notes vont être jouées une par une au moyen de la sous-routine *sound* du fichier *sound.asm* en incrémentant à chaque itération le pointeur Z afin de passer à la note suivante.Le buzzer est accédé à l'initialisation pour jouer la musique d'introduction, lors du saut du personnage et enfin lorsque la partie se termine (soit par une victoire soit par une défaite).

5 Annexe, code source au format pdf

5.0.1 map1Done.asm pages 8-12

5.0.2 macros_jeu.asm pages 13-15

5.0.3 macros_map.asm pages 16-23

5.0.4 macros_sound.asm page 24

5.0.5 macros_lcd.asm pages 25-26

```
; fichier     Map1Done.asm   cible ATmega128L-4MHz-STK300
;but: Jeu jouable type Mario visualisable sur la matrice de LED
;
; usage: boutons sur PORTD, ws2812 sur PORTE (bit 1)
;        presser bouton 0 pour avancer, bouton 1 pour sauter, (bouton 6 pour redémarrer)
;             Des niveaux de jeu sont stockés en mémoire puis afficher sur la matrice
;           Lorsque B0 est pressé, la matrice affiche une colonne plus loin en mémoire
;        qui se traduit par un décalage vers la droite du niveau.
;             Lorsque B1 est pressé, le personnage (pixel vert) effectue un saut. Les deux
;             actions sont combinables.
;             Le codeur angulaire permet de choisir le niveau du son de 0 à 9 (encodeur
pressé)
;             puis le choix du niveau (encodeur non pressé)
;             Le reste des boutons à actionner est indiqué sur l'écran LCD au moment voulu
;            L'écran LCD permet de communiquer avec l'utilisateur
;
; 20220315 AxS

.include "definitions.asm"
.include "macros.asm"
; === interrupt table ===

.org 0
       jmp    reset

.org INT0addr
       in _sreg, SREG      ;L'interruption 0 permet le déplacement latéral du personnage
       jmp ext_int0

.org OVF0addr
       in _sreg, SREG      ;L'interruption de timer overflow 0 permet de limiter la durée
                            d'une partie à 60 secondes
       jmp ovf_int0

.include "macros_sound.asm"
.include "macros_lcd.asm"
.include "macros_jeu.asm"
.include "macros_map.asm"

; === interrupt service routines ===
ext_int0:
       jmp shift_jump
ovf_int0:
       push b0
       push a0
       rjmp refresh_timer
;==================================
reset:
       cli
       LDSP   RAMEND       ;Le reset initialise l'écran LCD, l'encodeur et le stack pointer
       call   LCD_init     ;Le registre r11 est une constante fixée à 60 pour les 60
                            secondes à décrémenter jusqu'au game over
       call encoder_init
       _LDI r11, 60
       call   mainLCD      ;Le main LCD représente l'interface utilisateur où le joueur
                            choisit plusieurs paramètres avant de lancer la partie
       rjmp lancement
;=============================================================================================

;refresh_timer est la sous-routine d'interruption timer overflow 0
;Elle permet de limiter en temps la durée du joueur sur un niveau pour ajouter un peu de
challenge
refresh_timer:
       sbrc r7, 0    ;Lorsque le joueur a perdu, r7 est mis à 1 ou à 2 en fonction du
                      résultat du joueur
```

```
	rjmp fin_ovf  ;Cela permet de faire concorder l'affichage avec la situation
	sbrc r7, 1
	rjmp fin_ovf

	dec r11		;r11 est chargé à 60 puis décrémente, c'est le temps restant au joueur
			 pour finir le niveau
	_CPI r11, 0x00		;On décrémente le registre r11 jusqu'a ce qu'il atteigne zéro ce
				 qui implique le game over
	_BREQ inter_reset				;pour le joueur.
	mov b0, r11
	call LCD_clear
	PRINTF LCD
.db		"HOMEMADE   MARIO",0
	call LCD_lf
	mov a0, r9	;L'affichage est rafraîchi à chaque overflow du timer
	PRINTF LCD
.db "LVL:",FHEX,a,"	",FDEC,b,"s ",0
fin_ovf:
	pop a0
	pop b0
	out SREG, _sreg		;Enfin on restitue les valeurs a0, b0 ainsi que le SREG qui ont
				 pu être altérés par la sous-routine (printf notamment)
	reti
;=============================================================================================

;shift_jump est la sous-routine d'interruption externe 0
;Elle permet le décalage de la map d'une colonne à chaque fois que le bouton 0 est pressé,
de plus,
;elle décale aussi le personnage en mémoire et sur la matrice
shift_jump:
	Wait_MS 100
	mov zl, r13	;z pointe sur la LED du personnage
	mov zh, r12
	clc
	add zl, b1	;On ajoute 0x18 au byte pour arriver sur la LED à droite du personnage
	adc zh, r2

	ld r24, z	;On vérifie que la LED à droite du personnage soit orange
	sbrc r24, 1	;Si c'est le cas, le personnage est entré en collision avec une LED
			 orange, il perd la partie
	rjmp fin2	;On saute donc à l'affichage du "game over"

	clr r24
	mov    r13, zl		;On sauve la nouvelle position du personnage dans r12 et r13
	mov    r12, zh

	st z+, r8
	st z+, r2	;Ces trois instructions permettent de remplir la nouvelle LED avec les
			 couleurs du personnage
	st z+, r2

	clc
	add r14, b1	;Ici, on décale l'origine du niveau d'une colonne, pour faire "avancer"
			 le personnage
	adc r15, r2	;On ajoute donc 0x18 au byte r15:r14 pour que z pointe une colonne plus
			 loin lors de l'affichage

	call affichage_matrice
	WAIT_MS 100

	rjmp descente_int

fin:
	_LDI r10, 0x01		;Ici on charge r10 avec 0x01 pour stoper le saut du personnage si
				 l'interruption à lieu pendant un saut
```

```
	out SREG, _sreg		;On rétablit le SREG
	reti

fin2:
	_LDI r7, 0x01		;Ici on charge r7 avec 0x01 pour
	out SREG, _sreg		;On rétablit le SREG
	reti

descente_int:
	mov zl, r13	;On fait pointer z sur la nouvelle position du personnage avant qu'il
			 amorce sa descente
	mov zh, r12

	clc
	sub zl, r28	;Cette partie est similaire à la descente de la sous-routine descente
			 de jump_mario dans le fichier macros_jeu.asm
	sbc zh, r2

	ld r24, z
	sbrc r24,1
	rjmp fin

	clc
	mov zl, r13
	mov zh, r12

	st z+, r2
	st z+, r2
	st z+, r2

	clc
	sub zl, r6
	sbc zh, r2

	mov r13, zl
	mov r12, zh

	st z+, r8
	st z+, r2
	st z+, r2

	call affichage_matrice

	WAIT_MS 100
	rjmp descente_int

;=============================================================================================

;Ces deux morceaux de code servent à terminer la partie, que ce soit une victoire ou une
défaite
inter_reset:
	call LCD_clear
	PRINTF LCD
.db "   GAME  OVER   ",0	;on arrive dans cette partie du programme lorsque le joueur a
				 perdu la partie (collision ou manque de temps)
	cpi r29, 0
	breq PC+2
	PLAY_SOUND game_over, r29, 32
	WAIT_MS 2000		;un "game over" est alors affiché sur le LCD

	play_again:
		_LDI r14,0x00
		_LDI r15, 0x04
		mov zl, r14	;Ici on charge le pointeur z sur le debut du niveau pour nettoyer
				et "éteindre" la matrice de LEDs
```

```
            mov zh, r15
            RESET_MAT
            call affichage_matrice
            CLEARALL      ;Le CLEARALL permet de mettre à zéro tous les registres, ainsi
                           que les ports, interrupts et timers avant de pouvoir recommencer
                           la partie
            rjmp reset

victory:
       call LCD_clear
       PRINTF LCD    ;Cet affichage est produit lorsque le joueur parvient au bout du niveau
.db " YOU  COMPLETED ",0
       call LCD_lf
       mov a0, r9
       PRINTF LCD
.db "    LEVEL ",FHEX,a,"    ",0
       clr a0
       cpi r29, 0
       breq PC+2
       PLAY_SOUND level_complete, r29, 40
       WAIT_MS 2000
       rjmp play_again      ;A la fin de l'affichage, on saute à "play_again" pour mettre à
                             zéro comme lors de la défaite du joueur

;=============================================================================================

;Partie lancement de la partie
;Initialisation des différents ports et registres nécessaires au bon fonctionnement du jeu
lancement:
       call   ws2812b4_init       ; initialise la matrice de LEDs
       OUTI   DDRD, 0x00          ; Connecte les boutons du port D en mode entrée
       OUTI   EIMSK, 0b00000001   ;On autorise l'interruption externe 0
       OUTI ASSR, (1<<AS0)        ;On choisit le quartz horloger
       OUTI TCCR0, 5              ;ainsi qu'un prescaler à 5 pour obtenir un overflow toutes
                                   les secondes
       OUTI TIMSK, (1<<TOIE0)     ;On autorise le timer overflow interrupt 0
       _LDI r2, 0x00
       _LDI r28, 0x03             ;Constantes utilisés lors de l'exécution du programme
       _LDI r6, 0x06
       _LDI r8, 0x1c

;=============================================================================================

;Partie chargement de la map
main:
       ldi zl,low(0x0400)
       ldi zh,high(0x0400)  ;On fait pointer le pointeur z à l'adresse 0x0400 de la memoire,
                             là où sera sauvegardé le niveau à afficher
       mov r14, zl
       mov r15, zh          ;On sauve dans les registres le "début" du niveau, utilisés dans
                             d'autres parties du code

       sbrs r9, 0
       rjmp map2            ;Selon le choix fait avec l'encodeur angulaire, on place le
                             niveau 1 ou 2 en mémoire
       rjmp map1
map2:
       LEVEL2
       rjmp restart
map1:
       LEVEL1

;=============================================================================================
```

```
;Partie affichage de la map
restart:
	ldi b1, 0x18
	ldi zl,low(0x0400)
	ldi zh,high(0x0400)  ;On replace z à l'origine du niveau


	_LDI   r0,64
loop:
	ld a0, z+
	ld a1, z+     ;On charge dans a0, a1, a2 les composantes GRB de chaque pixel
	ld a2, z+

	sbrc a0,4     ;On vérifie que ici que le pixel soit vert
	call shift_mario_in_memory ;S'il est vert, on enregistre l'adresse de la
	                           première composante du pixel vert pour de futurs calculs

	cli
	call ws2812b4_byte3wr      ;Sous-routine qui va écrire des 0 et des 1 pour allumer
	                           les LEDs de la matrice
	sei

	dec r0
	brne loop
	call ws2812b4_reset        ;Reset nécessaire au bon fonctionnement de la matrice

attente_jump:
	sbis PIND, 1               ;Lorsque le bouton 1 et pressé, le personnage saute, sinon
	call jump_mario            ;on attend juste que le joueur avance ou saute dans cette
	                            boucle
	sbrc r7, 0
	jmp inter_reset
	sbrc r7, 1
	jmp victory
	rjmp attente_jump
```

```
/*
 * macros_jeu.asm
 *
 *  Created: 29/05/2022 22:42:29
 *   Author: Admin
 */
 ;shift_mario_in_memory ; sous-routine ; arg:void ; used:r12 et r13, z
;Cette sous-routine permet de sauvegarder l'adresse de la première composante de la LED
représentant le personnage dans r12 et r13
shift_mario_in_memory:
	mov r13, zl
	mov r12, zh   ;On sauve la position la position de la première composante de la LED
	               juste après celle du personnage
	clc
	_SUBI r13, 0x03     ;On retire 3 au byte pour retomber sur l'adresse de la première
	                     composante
	_SBCI r12, 0        ;de la LED du personnage, alors stocké dans r12 et r13
	ret

;=============================================================================================

;affichage_matrice ; arg:void ; used: r0,r14,r15,a0,a1,a2, z
;Cette sous-routine permet de rafraîchir l'affichage de la matrice, elle diffère du premier
remplissage de la matrice
;car elle n'enregistre pas la position du personnage à chaque invocation, cependant elle
vérifie si le personnage a atteint
;la fin du niveau.
affichage_matrice:
	mov zl, r14
	mov zh, r15   ;z pointe sur le début du niveau
	_LDI r0, 64   ;on charge r0 avec 64 pour les 64 LEDs à remplir
	boucle_affi:
		ld a0, z+
		ld a1, z+     ;Même fonctionnement que pour le premier affichage
		ld a2, z+

		cli
		call ws2812b4_byte3wr      ;Envoie les 24 bits pour allumer un pixel

		dec r0
		brne boucle_affi
		call ws2812b4_reset

		cpi zh, 0x0a        ;Ici, le bout de la map est fixé à l'adresse 0x0a30
		brsh fin_lvl
		rjmp retour
	fin_lvl:
		cpi zl, 0x30
		brsh escape         ;Lorsque z va pointer sur cette adresse, cela siginifiera
		                     que le personnage
		rjmp retour         ;a terminé le niveau, il a donc gagné la partie (l'adresse
		                     de retour du rcall devra donc être effacé de la pile)
	escape:
		_LDI r7, 0x02
		ret
retour:
ret                 ;S'il n'est pas au bout de la carte, l'affichage est juste raffraîchi

;=============================================================================================

;jump_mario ; sous-routine ; arg:void ; used: r2,r6,r8,r10,r12,r13,r24,r28, z
;Cette sous-routine permet de faire sauter le personnage, permettant de passer certains
obstacles du niveau
jump_mario:
	LDIZ 2800
```

```
st z+, a0     ;enregistrer les registres en mémoire données à l'adresse pointée par Z
st z+, a1
st z+, b0
st z+, b1
st z+, r0
cpi r29, 0
breq PC+2
PLAY_SOUND jump, r29, 2     ;joue le son du saut
LDIZ 2800
ld a0, z+     ;restitue les valeurs enregistrées en mémoire dans les registres
               correspondants
ld a1, z+
ld b0, z+
ld b1, z+
ld r0, z+
clr r24       ;r24 est utilisé comme condition pour arrêter la montée et la descente
               du personnage
_LDI r10, 3   ;r10 limite le nombre de pixels sautables à 3

loop_jump:
       mov zl, r13
       mov zh, r12   ;On place le pointeur à l'adresse du personnage
       adiw zh:zl, 3

       ld r24, z     ;On charge la première composante de la LED supérieure
       sbrc r24, 1   ;On vérifie alors si elle est orange ou pas
       rjmp descente ;Si c'est le cas, le personnage ne peut pas aller plus haut et
                      entame alors sa descente
       clr r24
                     ;Si la LED n'est pas orange on continue le programme
       mov zl, r13   ;z pointe de nouveau sur la LED du personnage
       mov zh, r12

       st z+, r2
       st z+, r2     ;Ces trois instructions permettent de remplacer les composantes
                      GRB du personnage par du vide (LED éteinte)
       st z+, r2

       mov r12, zh   ;Ces trois instructions induisent un décalage de 3 dans la
                      mémoire
       mov r13, zl   ;qui va être la nouvelle adresse de la LED du personnage

       st z+, r8
       st z+, r2     ;Ces trois instructions permettent de remplacer les composantes
                      GRB de la LED éteinte par celle du personnage (vert)
       st z+, r2

       sbrc r7, 0
       jmp inter_reset
       sbrc r7, 1
       jmp victory

       cli
       call affichage_matrice     ;On affiche alors la nouvelle matrice avec le
                                   personnage ayant effectué un saut d'une LED
       sei
       WAIT_MS 80
       dec r10
       brne loop_jump       ;Lorsque le personnage a sauté 3 LEDs, ou qu'il a
                             rencontré une LED orange, il amorce sa descente

descente:
       mov zl, r13
       mov zh, r12          ;z pointe donc sur l'adresse de la LED à laquelle le
                             personnage s'est arrêté
```

```
clc
sub zl, r28
sbc zh, r2    ;On retire 3 au byte, z pointe donc sur la LED du dessous

ld r24, z     ;On charge la première composante de la LED inférieure
sbrc r24, 1   ;On vérifie alors si elle est orange ou pas
ret           ;Si c'est le cas, le personnage ne peut pas aller plus bas, on va
               sortir de la sous-routine après rafraîchissement de l'affichage
clr r24
mov zl, r13   ;z pointe sur la LED du personnage
mov zh, r12

st z+, r2
st z+, r2     ;Ces trois instructions permettent de remplacer les composantes
               GRB du personnage par du vide (LED éteinte)
st z+, r2     ;z pointe donc actuellement sur la LED supérieure au personnage

clc
sub zl, r6    ;Ici on retire 6 à l'adresse pointée par z pr arriver à la LED
               inférieure au personnage désormais effacé
sbc zh, r2

mov r12, zh

mov r13, zl   ;on enregistre la position du personnage dans r12 et r13

st z+, r8
st z+, r2     ;Ces trois instructions permettent de remplacer les composantes
               GRB de la LED éteinte par celle du personnage (vert)
st z+, r2

sbrc r7, 0
jmp inter_reset
sbrc r7, 1
jmp victory

cli
call affichage_matrice     ;Rafraîchissement de la matrice
sei
WAIT_MS 100
rjmp descente ;Tant que le personnage ne rencontre pas de LED orange, il
               continue de descendre (Boucle TANT QUE)
```

```
/*
 * macros_map.asm
 *
 *  Created: 08/05/2022 16:27:24
 *   Author: patri
 */
;=============================================================================================

 ; WS2812b4_WR0     ; macro ; arg: void; used: void
; purpose: write an active-high zero-pulse to PD1
.macro WS2812b4_WR0
       clr u
       sbi PORTE, 1
       out PORTE, u
       nop
       nop
.endm

;=============================================================================================

; WS2812b4_WR1      ; macro ; arg: void; used: void
; purpose: write an active-high one-pulse to PD1
.macro WS2812b4_WR1
       sbi PORTE, 1
       nop
       nop
       cbi PORTE, 1
.endm
;=============================================================================================

; ORANGE      ; macro ; arg: @0=a0; used: z
; but: remplir 3 cases mémoires pour obtenir une LED orange lors de l'affichage
 .macro ORANGE
       ldi    @0, 0x06      ; pixel , orange
       st     z+,@0
       ldi @0,0x1c
       st     z+,@0
       ldi    @0, 0x00
       st z+,@0
.endmacro

;=============================================================================================

; VERT ; macro ; arg: @0=a0; used: z
; but: remplir 3 cases mémoires pour obtenir une LED verte lors de l'affichage
.macro VERT
       ldi    @0, 0x1c      ; pixel , vert
       st     z+,@0
       ldi @0,0x00
       st     z+,@0
       ldi    @0, 0x00
       st z+,@0
.endmacro

;=============================================================================================
; VOID ; macro ; arg: @0=a0; used: z
; but: remplir 3 cases mémoires pour obtenir une LED éteinte lors de l'affichage
.macro VOID
       ldi    @0, 0x00      ; pixel , vide
       st     z+,@0
       ldi @0,0x00
       st     z+,@0
       ldi    @0, 0x00
       st z+,@0
.endmacro
```

```
;================================================================================================

; COLONNE1    ; macro ; arg: @0=a0, @1 et @2 sont des constantes; used: w
; but: remplir 24 cases mémoires pour afficher une colonne sur la matrice, colonnes
modulables selon les constantes
;COLONNEx, le x représente le nombre de changements de couleur dans la colonne
.macro COLONNE1
	ldi w, @1
	loop_base:
		ORANGE @0
		dec w
		brne loop_base
	ldi w, @2
	loop_base0:
		VOID @0
		dec w
		brne loop_base0
.endmacro

;================================================================================================

; COLONNE1_START     ; macro ; arg: @0=a0, @1 et @2 sont des constantes; used: w
; but: remplir 24 cases mémoires pour afficher une colonne sur la matrice, colonnes
modulables selon les constantes
;         de plus, la colonne comporte une LED verte pour le personnage
.macro COLONNE1_START
	ldi w, @1
	loop_base:
		ORANGE @0
		dec w
		brne loop_base
	VERT @0
	ldi w, @2
	loop_base0:
		VOID @0
		dec w
		brne loop_base0
.endmacro

;================================================================================================

; COLONNE2    ; macro ; arg: @0=a0, @1 @2 et @3 sont des constantes; used: w
; but: remplir 24 cases mémoires pour afficher une colonne sur la matrice, colonnes
modulables selon les constantes
;COLONNEx, le x représente le nombre de changements de couleur dans la colonne
.macro COLONNE2
	ldi w, @1
	loop_base1:
		ORANGE @0
		dec w
		brne loop_base1
	ldi w, @2
	loop_base2:
		VOID @0
		dec w
		brne loop_base2
	ldi w, @3
	loop_base3:
		ORANGE @0
		dec w
		brne loop_base3
.endmacro

;================================================================================================
```

```
; COLONNE3    ; macro ; arg: @0=a0, @1 @2 @3 et @4 sont des constantes; used: w
; but: remplir 24 cases mémoires pour afficher une colonne sur la matrice, colonnes
modulables selon les constantes
;COLONNEx, le x représente le nombre de changements de couleur dans la colonne
.macro COLONNE3
	ldi w, @1
	loop_base4:
		ORANGE @0
		dec w
		brne loop_base4
	ldi w, @2
	loop_base5:
		VOID @0
		dec w
		brne loop_base5
	ldi w, @3
	loop_base6:
		ORANGE @0
		dec w
		brne loop_base6
	ldi w, @4
	loop_base7:
		VOID @0
		dec w
		brne loop_base7
.endmacro

;================================================================================================

; COLONNE6    ; macro ; arg: @0=a0, @1 @2 @3 @4 @5 @6 et @7 sont des constantes; used: w
; but: remplir 24 cases mémoires pour afficher une colonne sur la matrice, colonnes
modulables selon les constantes
;COLONNEx, le x représente le nombre de changements de couleur dans la colonne
.macro COLONNE6
	ldi w, @1
	loop_base8:
		ORANGE @0
		dec w
		brne loop_base8
	ldi w, @2
	loop_base9:
		VOID @0
		dec w
		brne loop_base9
	ldi w, @3
	loop_base10:
		ORANGE @0
		dec w
		brne loop_base10
	ldi w, @4
	loop_base11:
		VOID @0
		dec w
		brne loop_base11
	ldi w, @5
	loop_base12:
		ORANGE @0
		dec w
		brne loop_base12
	ldi w, @6
	loop_base13:
		VOID @0
		dec w
		brne loop_base13
	ldi w, @7
```

```
	loop_base14:
		ORANGE @0
		dec w
		brne loop_base14
.endmacro

;=============================================================================================

; COLONNE5    ; macro ; arg: @0=a0, @1 @2 @3 @4 et @5 sont des constantes; used: w
; but: remplir 24 cases mémoires pour afficher une colonne sur la matrice, colonnes
modulables selon les constantes
;COLONNEx, le x représente le nombre de changements de couleur dans la colonne
.macro COLONNE5
	ldi w, @1
	loop_base15:
		ORANGE @0
		dec w
		brne loop_base15
	ldi w, @2
	loop_base16:
		VOID @0
		dec w
		brne loop_base16
	ldi w, @3
	loop_base17:
		ORANGE @0
		dec w
		brne loop_base17
	ldi w, @4
	loop_base18:
		VOID @0
		dec w
		brne loop_base18
		ldi w, @5
	loop_base19:
		ORANGE @0
		dec w
		brne loop_base19
.endmacro

;=============================================================================================

ws2812b4_init:
	OUTI   DDRE,0x02
ret

ws2812b4_byte3wr:
	ldi w,8
ws2b3_starta0:
	sbrc a0,7
	rjmp   ws2b3w1
	WS2812b4_WR0                                    ; write a zero
	rjmp   ws2b3_nexta0
ws2b3w1:
	WS2812b4_WR1
ws2b3_nexta0:
	lsl a0
	dec    w
	brne ws2b3_starta0

	ldi w,8
ws2b3_starta1:
	sbrc a1,7
	rjmp   ws2b3w1a1
	WS2812b4_WR0                                    ; write a zero
```

```
       rjmp   ws2b3_nexta1
ws2b3w1a1:
       WS2812b4_WR1
ws2b3_nexta1:
       lsl a1
       dec    w
       brne ws2b3_starta1

       ldi w,8
ws2b3_starta2:
       sbrc a2,7
       rjmp   ws2b3w1a2
       WS2812b4_WR0                                    ; write a zero
       rjmp   ws2b3_nexta2
ws2b3w1a2:
       WS2812b4_WR1
ws2b3_nexta2:
       lsl a2
       dec    w
       brne ws2b3_starta2

ret

ws2812b4_reset:
       cbi PORTE, 1
       WAIT_US       50     ; 50 us are required, NO smaller works
ret

;=============================================================================================

; LEVEL1      ; macro ; arg:void ; used:void
; but: remplir en mémoire toutes les colonnes pour créer le niveau 1
.macro LEVEL1
       COLONNE1_START a0, 1, 6                                  ;place un pixel vert
pour définir la position du personnage dans la première colonne
       COLONNE1 a0, 1, 7
       COLONNE1 a0, 1, 7
       COLONNE1 a0, 1, 7
       COLONNE2 a0, 3, 3, 2
       COLONNE2 a0, 3, 3, 2
       COLONNE1 a0, 1, 7
       COLONNE1 a0, 1, 7
       COLONNE1 a0, 1, 7
       COLONNE1 a0, 2, 6
       COLONNE1 a0, 3, 5
       COLONNE1 a0, 4, 4
       COLONNE1 a0, 1, 7
       COLONNE1 a0, 1, 7
       COLONNE3 a0, 1, 2, 2, 3
       COLONNE3 a0, 1, 2, 2, 3
       COLONNE1 a0, 1, 7
       COLONNE1 a0, 1, 7
       COLONNE2 a0, 1, 2, 5
       COLONNE2 a0, 1, 2, 5
       COLONNE1 a0, 1, 7
       COLONNE2 a0, 3, 3, 2
       COLONNE1 a0, 1, 7
       COLONNE2 a0, 1, 2, 5
       COLONNE2 a0, 1, 5, 2
       COLONNE2 a0, 1, 6, 1
       COLONNE3 a0, 1, 2, 1, 4
       COLONNE3 a0, 1, 2, 1, 4
       COLONNE3 a0, 1, 2, 1, 4
       COLONNE3 a0, 1, 4, 1, 2
       COLONNE3 a0, 1, 4, 1, 2
```

```
	COLONNE3 a0, 1, 4, 1, 2
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 2, 6
	COLONNE1 a0, 3, 5
	COLONNE1 a0, 2, 6
	COLONNE2 a0, 1, 6, 1
	COLONNE2 a0, 1, 5, 2
	COLONNE2 a0, 1, 4, 3
	COLONNE2 a0, 1, 5, 2
	COLONNE2 a0, 1, 6, 1
	COLONNE1 a0, 2, 6
	COLONNE1 a0, 3, 5
	COLONNE1 a0, 4, 4
	COLONNE1 a0, 5, 3
	COLONNE1 a0, 5, 3
	COLONNE1 a0, 5, 3
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 4, 2, 2
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 3, 2, 3
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 2, 2, 4
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
.endmacro

;================================================================================================

; LEVEL2	; macro ; arg:void ; used:void
; but: remplir en mémoire toutes les colonnes pour créer le niveau 2
.macro LEVEL2
	COLONNE1_START a0, 1, 6
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 2, 3, 3
	COLONNE2 a0, 2, 3, 3
	COLONNE2 a0, 3, 3, 2
	COLONNE2 a0, 3, 3, 2
	COLONNE2 a0, 4, 3, 1
	COLONNE2 a0, 4, 3, 1
	COLONNE1 a0, 5, 3
	COLONNE1 a0, 5, 3
	COLONNE2 a0, 4, 2, 2
	COLONNE2 a0, 4, 2, 2
	COLONNE2 a0, 3, 2, 3
	COLONNE2 a0, 3, 2, 3
	COLONNE2 a0, 2, 2, 4
	COLONNE2 a0, 2, 2, 4
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 3, 2, 3
	COLONNE2 a0, 3, 2, 3
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
```

```
	COLONNE3 a0, 1, 1, 1, 5
	COLONNE3 a0, 1, 1, 1, 5
	COLONNE3 a0, 1, 4, 1, 2
	COLONNE3 a0, 1, 4, 1, 2
	COLONNE3 a0, 1, 2, 1, 4
	COLONNE3 a0, 1, 2, 1, 4
	COLONNE1 a0, 7, 1
	COLONNE3 a0, 2, 3, 2, 1
	COLONNE3 a0, 1, 5, 1, 1
	COLONNE3 a0, 2, 3, 2, 1
	COLONNE1 a0, 6, 2
	COLONNE1 a0, 6, 2
	COLONNE1 a0, 5, 3
	COLONNE1 a0, 5, 3
	COLONNE1 a0, 4, 4
	COLONNE1 a0, 4, 4
	COLONNE1 a0, 3, 5
	COLONNE1 a0, 3, 5
	COLONNE1 a0, 2, 6
	COLONNE1 a0, 2, 6
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 1, 2, 5
	COLONNE6 a0, 1, 2, 1, 1, 1, 1, 1
	COLONNE5 a0, 1, 2, 1, 3, 1
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 1, 2, 5
	COLONNE3 a0, 1, 4, 2, 1
	COLONNE3 a0, 1, 3, 2, 2
	COLONNE2 a0, 1, 2, 5
	COLONNE1 a0, 1, 7
	COLONNE2 a0, 1, 2, 5
	COLONNE5 a0, 1, 2, 1, 3, 1
	COLONNE3 a0, 1, 3, 3, 1
	COLONNE1 a0, 1, 7
	COLONNE5 a0, 1, 1, 1, 1, 4
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
	COLONNE1 a0, 1, 7
.endmacro

;================================================================================================

; CLEARALL    ; macro ; arg:void ; used:void
; but: Mettre à zéro les registres, ports I/O utilisés, timer/interrupts, stack
.macro CLEARALL
	clr_stack:
		pop a0
		in w, SPL
		cpi w, 0xff
		breq PC+2
		rjmp clr_stack
	clr r0
	clr r1
	clr r2
	clr r3
	clr r4
	clr r5
	clr r6
	clr r7
	clr r8
```

```
	clr r9
	clr r10
	clr r11
	clr r12
	clr r13
	clr r14
	clr r15
	clr r16
	clr r17
	clr r18
	clr r19
	clr r20
	clr r21
	clr r22
	clr r23
	clr r24
	clr r25
	clr r26
	clr r27
	clr r28
	clr r29
	clr r30
	clr r31
	OUTI   DDRD, 0x00
	OUTI   DDRE, 0x00
	OUTI   EIMSK, 0x00
	OUTI ASSR, 0x00
	OUTI TCCR0, 0x00
	OUTI TIMSK, 0x00
	OUTI SREG, 0x00
.endmacro

;=============================================================================================

; RESET_MAT   ; macro ; arg:void ; used:void
; but: remplir en mémoire l'équivalent de 64 LEDs vides pour éteindre la matrice
.macro RESET_MAT
	ldi w, 64
	loop_baseNull:
		VOID a0
		dec w
		brne loop_baseNull
.endmacro

;=============================================================================================
```

```
/*
 * macros_sound.asm
 *
 *  Created: 29/05/2022 22:43:00
 *   Author: Admin
 */
 .include "sound.asm"

  .macro PLAY_SOUND
       sbi    DDRB, SPEAKER       ;Définit le pin 2 de PORTB en sortie
       clr r21
    ldi zl, low(2*@0)             ;Fait pointer z au début de la table
    ldi zh, high(2*@0)
loop_sound:
    lpm
    adiw zl, 1
       cpi r21, @2                ;Condition de sortie, lorsque r21 atteint le troisième
                                   argument, on arrête le son
    breq end_sound
    mov a0, r0
       mov b0, @1
       inc r21
    call sound
    rjmp loop_sound
end_sound:
       clr r21
       cbi    DDRB, SPEAKER       ;Redéfinit le pin 2 de PORTB en entrée
.endmacro

.org 0x5000
mario :
       .db do2, do2, do2, do2, so, so, so, so, mi, mi, mi, mi
       .db la, la, la, si, si, si, lam, lam, la, la, la, so
       .db so, so, mi2, mi2, so2, so2, la2, la2, la2, fa2, fa2, so2, so2, so2
       .db mi2, mi2, mi2, do2, do2, re2, re2, si, si, si, si, si
       .db do2, do2, do2, do2, so, so, so, so, mi, mi, mi, mi
       .db la, la, la, si, si, si, lam, lam, la, la, la, so
       .db so, so, mi2, mi2, so2, so2, la2, la2, la2, fa2, fa2, so2, so2, so2
       .db mi2, mi2, mi2, do2, do2, re2, re2, si, si, si, si, si
       .db 0

.org 0x5040
game_over :
       .db do3, do3, do3, do3, so2, so2, so2, so2, mi2, mi2, mi2, mi2
       .db la2, la2, si2, si2, la2, la2, som2, som2, lam2, lam2, som2, som2
       .db so2, fa2, so2, so2, so2, so2, so2, so2
       .db 0

.org 0x0a80
level_complete :
       .db so, si, re2, so2, si2, re3, so3, so3, so3, re3, re3, re3
       .db som, do2, rem2, som2, do3, rem3, som3, som3, som3, rem3, rem3, rem3
       .db lam, re2, fa2, lam2, re3, fa3, lam3, lam3, lam3, lam3, lam3, lam3
       .db do4, do4, do4, do4
       .db 0

.org 0x3370
jump :
       .db si3, mi4
       .db 0
```

```
/*
 * macros_lcd.asm
 *
 *  Created: 29/05/2022 22:42:15
 *   Author: Admin
 */
 ; mainLCD    ; sous-routine ; arg:void ; used:b0,PIND,a0
; but: Offrir à l'utilisateur des menus et paramètres à régler avant de lancer la partie
visibles
;sur l'écran LCD
.include "lcd.asm"
.include "printf.asm"
.include "encoder.asm"

mainLCD:
       rcall LCD_clear
       PRINTF LCD
.db           "HOMEMADE   MARIO",0       ;Affichage au lancement du jeu
       ldi r29, 35
       PLAY_SOUND mario, r29, 100        ;Joue la musique d'introduction
       clr r29
       rcall LCD_lf
       PRINTF LCD
.db           "6:NEXT",0
       sbis PIND, 6         ;Il faut appuyer sur le bouton 6 pour accéder au paramètre
                             suivant
       rjmp select_sound
       rjmp PC-2

select_sound:
       rcall LCD_clear
       PRINTF LCD           ;Menu pour la sélection du niveau de son
.db    "SELECT SOUND:",0
       rcall LCD_lf
choix_b0:
       WAIT_MS 20
       rcall encoder        ;permet de modifier la valeur de b0
       cpi b0, 0xff
       breq mise_trois      ;Son réglable du niveau 0 (pas de son) à 3 au moyen de l'encodeur
                             angulaire
       cpi b0, 0x04
       brsh mise_zero2
       PRINTF LCD
.db "SOUND:",FHEX,b,"  7:NEXT",0
       call val_b0          ;Permet d'assigner à chaque cas une valeur numérique
       WAIT_MS 20
       sbis PIND, 7         ;Il faut appuyer sur le bouton 7 pour passer au choix du niveau
       rjmp select_lvl
       rjmp select_sound

mise_zero2:
       ldi b0, 0x00
       rjmp choix_b0
mise_trois:
       ldi b0, 0x03
       rjmp choix_b0        ;Sous-routines qui permettent de garder les valeurs voulues entre
                             deux bornes
mise_un:                    ;avec b0 qui stocke la valeur pour le son et a0 la valeur pour
                             les niveaux
       ldi a0, 0x01
       rjmp choix_a0
mise_deux:
       ldi a0, 0x02
       rjmp choix_a0
```

```
val_b0:
	cpi r22, 0
	breq case1
	cpi r22, 1
	breq case2
	cpi r22, 2
	breq case3
	ldi r29, 50
	ret
case3:
	ldi r29, 40
	ret
case2:
	ldi r29, 30
	ret
case1:
	ldi r29, 0
	ret

select_lvl:
	rcall LCD_clear
	PRINTF LCD                  ;Menu pour la sélection du niveau
.db		"SELECT LVL:",0
	rcall LCD_lf
choix_a0:
	WAIT_MS 20
	rcall encoder               ;Permet de choisir la valeur de a0
	cpi a0, 0x00
	breq mise_deux              ;Deux choix de niveau (1 et 2)
	cpi a0, 0x03
	brsh mise_un
	PRINTF LCD
.db	"LVL:",FHEX,a,"    6:PLAY",0
	WAIT_MS 20
	sbis PIND, 6                ;Il faut appuyer sur le bouton 6 pour lancer la partie sur
	                             le niveau choisi
	rjmp affi_lvl
	rjmp select_lvl

affi_lvl:
	mov r9, a0
	rcall LCD_clear
	PRINTF LCD
.db		"HOMEMADE   MARIO",0              ;Affichage du LCD lorsque la partie commence
	rcall LCD_lf
	PRINTF LCD
.db "LVL:",FHEX,a,0
	ret
```