TP introduction au microcontrôleur 8 bits

É. Carry, J.-M Friedt

15 novembre 2020

Questions sur ce TP :

1. Sur quelle broche de quel port d'entrée-sortie est connectée la LED1 ?
2. Quel bit placer à quelle valeur pour passer la broche 1 du PORTB en sortie ?
3. Quelle est la conséquence de l'optimisation d'un code C par l'option `-O2` de `gcc` sur une boucle vide utilisée pour introduire une lattence dans l'exécution du code ?
4. Quelle option de `gcc` permet de fournir le code assembleur issu du code source en C ?
5. Donner un nom du premier port série virtuel sous GNU/Linux ? comment lire les informations transmises sur ce port de communication ?
6. Quelle option de `gcc` permet d'introduire les symboles de déverminage (*debug*) dans l'assembleur et le binaire issus de la compilation ?
7. Quel est le facteur de division de l'horloge cadençant un *timer* si les bits CS1 et CS0 sont placés à la valeur 1 ? Justifier.

1 Compilation et GPIO

Le schéma [1] de la Fig. 1 fournit le brochage d'un certain nombre de périphériques dont les LEDs visibles à côté de l'embase d'alimentation de la carte.

Le comportement des GPIO est décrit dans l'extrait de la Fig. 2 de la feuille descriptive (*datasheet*) de l'Atmega32U4 disponible à `ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf` (p.67) [1].

Le programme ci-dessous configure deux ports GPIO en sortie, et manipule l'état de ces sorties pour faire clignoter les diodes.

Listing 1 – Diode clignotante

```
1  #include <avr/io.h> //E/S ex PORTB
2  #define F_CPU 16000000UL
3  #include <util/delay.h>
4
5  int main(void){
6    DDRB |=1<<PORTB5;
7    DDRE |=1<<PORTE6;
8    PORTB |= 1<<PORTB5;
9    PORTE &= ~(1<<PORTE6);
10
11   while (1){
12     /* Clignotement des LEDS */
13   PORTB^=1<<PORTB5;PORTE^=1<<PORTE6;
14   _delay_ms(500);
15   }
16   return 0;
17 }
```

Analyser ce programme compte tenu du descriptif de la *datasheet* fourni en Fig. 2.

La compilation du programme (passage du code C en binaire au format ELF) s'obtient par la version de `gcc` configurée pour produire du code à destination de l'architecture AVR 8 bits par

`avr-gcc -mmcu=atmega32u4 -Os -Wall -o blink.out blink.c`

1. `https://www.olimex.com/Products/Duino/AVR/OLIMEXINO-32U4/open-source-hardware`

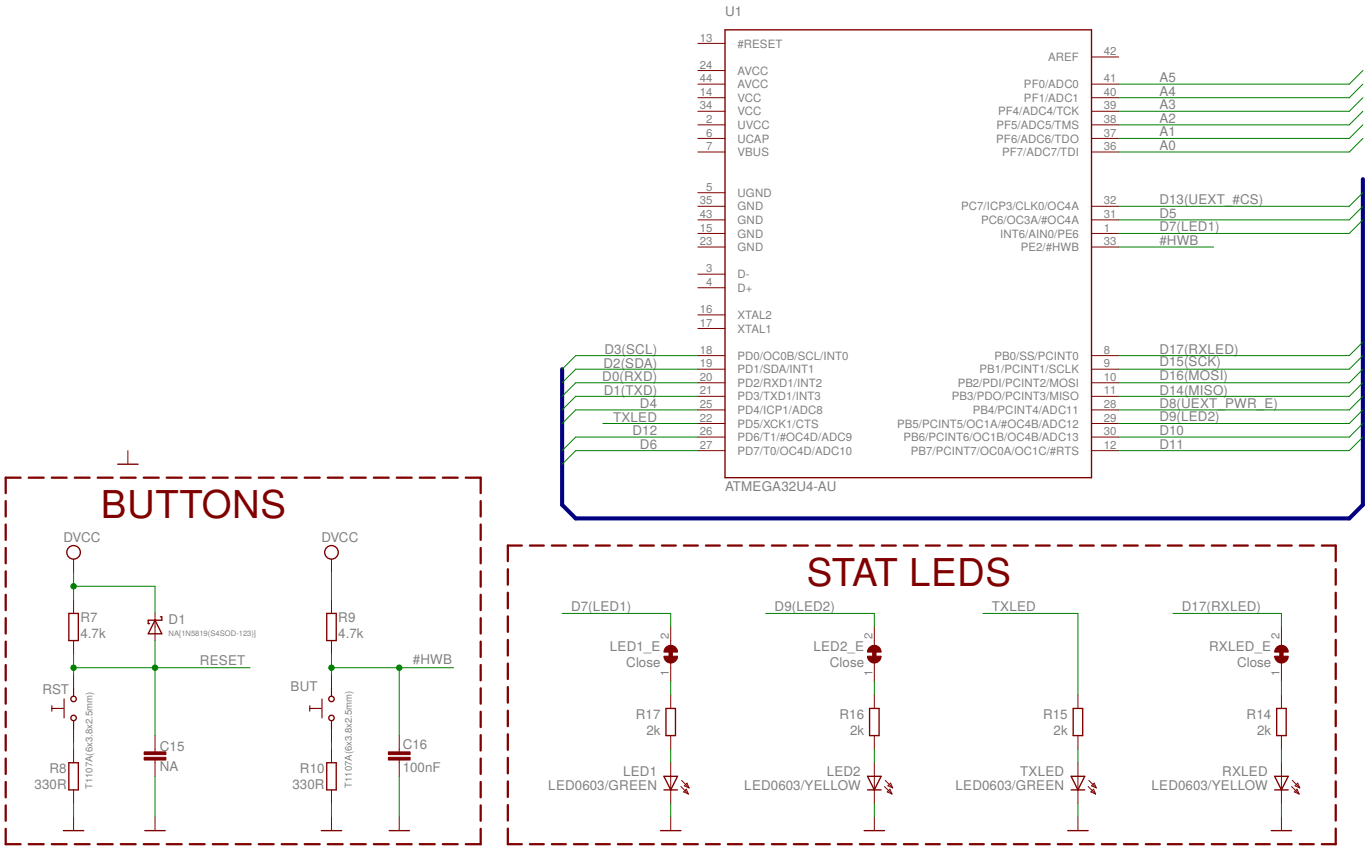

FIGURE 1 – Extrait du schéma de l'Olimexino-32U4

Table 10-1. Port Pin Configurations

DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

FIGURE 2 – Extrait du schéma de l'Olimexino-32U4

1.1 Exécution du code

À peu près tous les microcontrôleurs modernes sont équipés d'un bout de code chargé du transfert des données et de leur écriture en mémoire non-volatile (*bootloader*). Un logiciel dédié est nécessaire pour communiquer avec le bootloader : dans le cas de l'AVR, il s'agit de `avrdude` qui s'exécute *après appui du bouton RST* par

```
avrdude -c avr109 -b57600 -D -p atmega32u4 -P /dev/ttyACM0 -e -U flash:w:blink.out
```

en prenant comme arguments la déclinaison du microcontrôleur, le débit de transfert, et le port de communication (le port `ttyACM` numéro zéro, soit le premier port série virtuel).

Observer le comportement des LEDs situées à droite de l'embase d'alimentation de la carte.

1.2 Analyse du code

Convertir un code C en langage compatible d'un microprocesseur est une opération complexe qui passe par de nombreuses étapes. `gcc` permet d'arrêter la compilation à chacune de ces étapes pour en voir le résultat.

`gcc -E` arrête au précompilateur qui a simplement effectué l'analyse syntaxique du code.

`gcc -S` arrête à la génération du code assembleur.

Une fois le binaire généré, il est possible de revenir au code assembleur correspondant aux opcodes par `objdump -dSt fichier.out`.

Si les symboles de déverminage ont été inclus dans le binaire (`gcc -g`), alors les lignes de C correspondantes aux instructions assembleur sont fournies dans le listing.

Par ailleurs, `objcopy` permet de convertir des fichiers entre de nombreux formats, et en particulier le binaire ELF en images de la mémoire du microcontrôleur dans les deux formats les plus courants, Intel Hex et Motorola S19.

iHex : `avr-objcopy -O ihex blink.out blink.hex`

S19 : `avr-objcopy -O srec blink.out blink.s19`

Il est ainsi possible d'utiliser un outil de programmation ne reconnaissant que ces formats (en particulier les outils de programmation propriétaires sont souvent peu flexibles sur le format de données compris).

1. **Identifier sur le schéma les deux LEDs fournies sur le circuit Olimexino32U4 et les ports auxquelles elles sont connectées.**
2. **Sachant que les LEDs du chenillard occupent les 10 broches notées D0 à D9 sérigraphiées sur le circuit imprimé, identifier quelle broche du microcontrôleur correspond à quelle LED** (Fig.3).
3. **Sachant que la direction de chaque broche du port `x` est déterminée par le registre `DDRx` et la valeur du port par `PORTx`, proposer un programme qui allume et éteint toutes les 200 ms toutes les LEDs de la barette.** Dans cet exemple, l'attente entre la transition entre deux états de la barette de LEDs se fera au moyen de la fonction `_delay_ms()` fournie par `util/delay.h`. Cette fonction nécessite d'optimiser le programme lors de sa compilation (`-Os` ou `-O2`) pour atteindre la latence attendue.
4. **Proposer une structure de données et une fonction, qui prend en argument un entier n et un état c (allumé ou éteint, 1 ou 0) qui permette d'allumer ou éteindre la $n$ème LED de la barette**
5. **Démontrer la capacité à allumer et éteindre séquentiellement chaque LED dans une seule direction de défilement (chenillard)**
6. **Démontrer la capacité à allumer et éteindre séquentiellement chaque LED dans un motif d'aller-retour (K2000)**

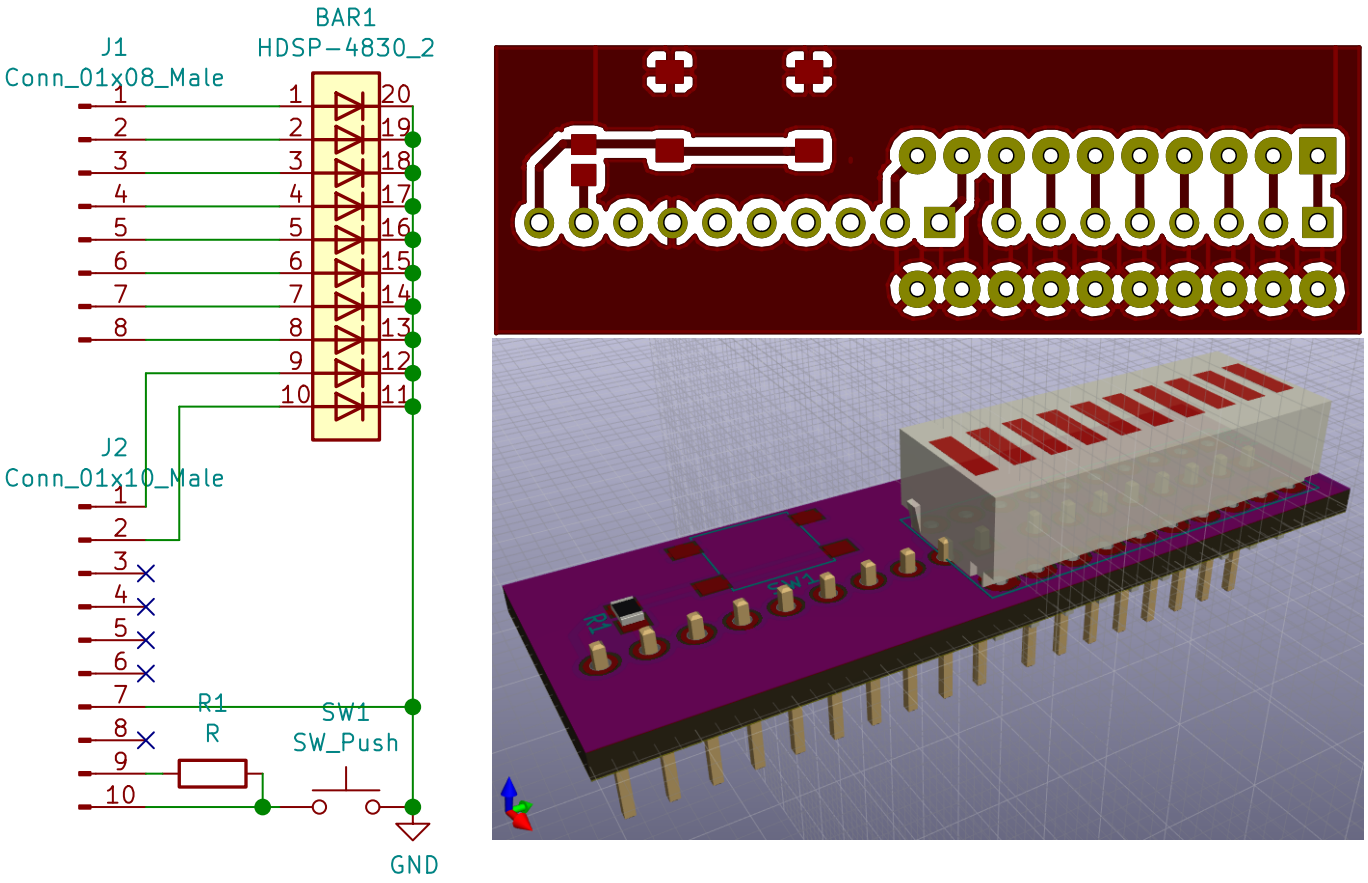

FIGURE 3 – Schéma (gauche), empreinte (haut droite) et modèle mécanique (bas droite) de la barette de LEDs connectées à l'Olimexino32U4.

2 GPIO en entrée – problème de rebonds

Pour le moment, le programme s'exécute sans action de l'utilisateur. Nous désirons que l'utilisateur puisse interagir avec le microcontrôleur pour déterminer la direction de défilement du chenillard.

1. **Sachant que le bouton poussoir se trouve sur la broche sérigraphiée D11, quelle broche de quel port sur le microcntrôleur permet d'accéder à ses fonctionnalités ?**
2. **Configurer la broche en entrée, avec une configuration de potentiel par défaut approprié et démontrer la capacité à détecter l'appui du bouton en induisant un changement de direction du chenillard chaque fois que le bouton est appuyé.**

Parfois le comportement n'est pas déterministe, et l'appui du bouton se traduit par une interruption soudaine du défilement avant sa reprise. La multitude des transisitions d'état du bouton possoir est un problème classique des transitoires sur les entrées numériques qui se résout par le *debounce* consistant à filtrer (passe bas) la connexion entre le fil et la broche. La version logicielle d'un filtre passe bas est une temporisation.

1. **Proposer un paliatif évitant les changements multiples de direction du défilement lors d'un unique appui sur le bouton poussoir.**

3 Un périphérique complexe le *timer*

La solution de séquencer l'état de l'afficheur sur une attente explicite dans la boucle infinie est peu efficace car 1/ le microcontrôleur perd son temps, et pourrait au moins se mettre en mode veille (couper l'horloge du cœur du processeur) pour économiser de l'énergie et 2/ risque de modifier les latences entre deux transitions si le microcontrôleur effectue d'autres tâches qu'attendre la prochaine transition.

La solution est d'exploiter un périphérique matériel chargé de séquencer les tâches : le *timer*.

En effet, parmi les périphériques disponibles autour du cœur de calcul du microcontrôleur, les fonctions de temporisation, ou *timer*, sont probablement les plus utiles. La fonction la plus simple est de fournir un signal au cœur lorsqu'une condition de délai a été atteinte : ceci permet d'une part de libérer le processeur pour des tâches plus intéressantes que l'attente dans une temporisation, voir de le placer en mode veille où la consommation est réduite, mais surtout de garantir de délai indépendamment de toute variabilité du logiciel.

Les codes ci-dessous (2 et 3) s'affranchissent des aléas de l'optimisation du logiciel en exploitant les fonctions matérielles du micrcontrôleur.

Le *timer* est un périphérique un peu plus compliqué à maîtriser compte tenu du nombre de modes de fonctionnement : en entrée (*input capture*), en sortie (*output compare*) ou en saturation de comptage (*overflow*).

3.1 *Overflow*

En mode dépassement, un compteur tourne librement. Lorsque le seuil de comptage est atteint, un drapeau (*flag*) est commuté. Écrire sur ce flag le remet à 0, tel qu'illustré sur le code 2. Le timer0 est sur 8 bits (*datasheet* [1] chapitre 13), le timer 1 sur 16 bits (*datasheet* chapitre 14).

Consulter l'extrait de la *datasheet* de la Fig. 4 et analyser le comportement du *timer* en mode *overflow*.

14.10.20 Timer/Counter3 Interrupt Flag Register – TIFR3

Bit	7	6	5	4	3	2	1	0	
	–	–	ICF3	–	OCF3C	OCF3B	OCF3A	TOV3	TIFR3
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – TOVn: Timer/Countern, Overflow Flag**

The setting of this flag is dependent of the WGMn3:0 bits setting. In Normal and CTC modes, the TOVn Flag is set when the timer overflows. Refer to Table 14-5 on page 131 for the TOVn Flag behavior when using another WGMn3:0 bit setting.

TOVn is automatically cleared when the Timer/Countern Overflow Interrupt Vector is executed. Alternatively, TOVn can be cleared by writing a logic one to its bit location.

FIGURE 4 – Extrait de la *datasheet* du Atmega32U4 décrivant le comportement en mode *overflow* du timer1.

Listing 2 – Timer pour temporiser le clignotement des LEDs

```
1 #include <avr/io.h>
```

```
#define F_CPU 16000000UL

void timer1_init()  // Timer1 avec prescaler=64 et mode normal
{TCCR1B |= (1 << CS11)|(1 << CS10);
}

int main(void)
{
  DDRB |=1<<PORTB5;
  DDRE |=1<<PORTE6;
  PORTB |= 1<<PORTB5;
  PORTE &= ~(1<<PORTE6);

  timer1_init();
  while(1)
    {if (TIFR1 & (1 << TOV1))
       {PORTB^=1<<PORTB5;PORTE^=1<<PORTE6;
        TIFR1 |= (1 << TOV1); // clear flag
       }
    }
}
```

Dans ce mode, le seul degré de liberté pour ajuster la temporisation est le facteur de division de l'horloge qui cadence le compteur, un mode grossier d'ajustement qui ne répond pas toujours aux exigences de la définition d'une fréquence (ou période) précise.

3.2 *CTC*

14.10.20 Timer/Counter3 Interrupt Flag Register – TIFR3

Bit	7	6	5	4	3	2	1	0	
	–	–	ICF3	–	OCF3C	OCF3B	OCF3A	TOV3	TIFR3
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCF1A: Timer/Counter1, Output Compare A Match Flag**

This flag is set in the timer clock cycle after the counter (TCNTn value matches the Output Compare Register A (OCRnA).

Note that a Forced Output Compare (FOCnA) strobe will not set the OCFnA Flag.

OCFnA is automatically cleared when the Output Compare Match A Interrupt Vector is executed. Alternatively, OCFnA can be cleared by writing a logic one to its bit location.

FIGURE 5 – Extrait de la *datasheet* du Atmega32U4 décrivant le comportement en mode *CTC* du timer1.

Clear Timer on Compare Match (CTC) Mode permet de définir un seuil avec lequel le compteur est comparé : atteindre ce seuil induit une transition sur un drapeau (*flag* – Fig. 5) tel que illustré sur le code 2.

Listing 3 – Timer pour temporiser le clignotement des LEDs

```
#include <avr/io.h>
#define F_CPU 16000000UL

#define delai_leds (16000*2)

void timer1_init()  // Timer1 avec prescaler=64 et CTC (WGM=2)
{TCCR1B |= (1 << WGM12)|(1 << CS11)|(1 << CS10);
 OCR1A = delai_leds;  // valeur seuil <- delai
}

int main(void)
{
  DDRB |=1<<PORTB5;
  DDRE |=1<<PORTE6;
  PORTB |= 1<<PORTB5;
```

```
16   PORTE &= ~(1<<PORTE6);
17
18   timer1_init();
19   while(1)
20     {if (TIFR1 & (1 << OCF1A))
21        {PORTB^=1<<PORTB5;PORTE^=1<<PORTE6;
22         TIFR1 |= (1 << OCF1A); // clear flag
23        }
24     }
25 }
```

Noter que par rapport à l'exemple de l'overflow, le drapeau (*flag*) testé est cette fois `OCF1A` au lieu de `TOV1`.

1. **Quel est l'intérêt du mode CTC par rapport au mode overflow ?**
2. **Modifier la temporisation et observer le résultat : ajuster le prescaler et la valeur de comparaison pour avoir un changement d'état de la LED à 5 Hz.**
3. **Proposer un programme dont la boucle principale infinie teste l'état du timer et séquence le défilement des LEDs sur une transition d'état de ce périphérique.**

Références

[1] Atmel, 8-bit Microcontroller with 16/32K Bytes of ISP Flash and USB Controller – ATmega16U4 & ATmega32U4, disponible à `ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf`