TP interruptions sur microcontrôleur 8 bits

É. Carry, J.-M Friedt

29 janvier 2021

Questions sur ce TP :

1. Quel est le rôle d'une interruption matérielle? Donner un exemple d'utilisation d'une interruption *timer*.
2. Donner 3 rôles d'un compteur dans un microcontrôleur. On pourra penser à un mode de commande de vitesse de rotation de moteurs.
3. Pourquoi utiliser une interruption pour recevoir les données sur un port de communication?
4. À quoi sert un chien de garde?
5. Quelle est la conséquence d'une erreur de 20% sur la vitesse d'horloge dans une communication asynchrone?
6. Combien de signaux sont nécessaires à une liaison compatible RS232?

1 Généralités sur les interruptions : cas du GPIO

Au lieu de continuellement sonder l'état des broches du microcontrôleur pour détecter un changement d'état (au risque d'induire une latence excessive si le programme est chargé d'autres opérations de calcul entre deux lectures du port), il est souhaitable d'utiliser les interruptions. Dans ce cas, le programme principal est configuré pour exécuter séquentiellement ses opérations de calcul, et un changement d'état sur une broche de GPIO induit une *interruption* de l'éxécution séquentielle du programme pour sauter dans le gestionnaire d'interruption (Interrupt Service Routine – ISR). Le traitement du changement d'état du GPIO n'est donc pas explicitement traité dans le programme principal.

Identifier sur le schéma 1 la broche associée à INT2, l'interruption matérielle externe 2.

⚠Il est techniquement possible de démontrer l'utilisation de INT0 sur PD0 (noté D3 sur la sérigraphie) de la même façon, mais pour ce faire on pensera à alimenter le port UEXT en abaissant la tension de grille du FET au travers de PB4. En effet, PD0 fait aussi office d'horloge de communication I^2C, et omettre d'alimenter le port UEXT tire vers la masse le potentiel de cette broche si elle reste en potentiel flottant, contrant l'effet de la résistance de tirage vers l'alimentation (*pull-up*) interne au microcontrôleur.

Afin de ne pas empiler les interruptions, il est cependant judicieux d'y confiner un minimum de code – généralement définir un drapeau (*flag*) et retourner au plus vite dans le programme principal qui traitera le problème en temps voulu. Seul le code qui doit être impérativement exécuté immédiatement à la détection de l'interruption (e.g. "couper les moteurs" lorsque l'évènement "risque de collision" est détecté) se trouve dans l'ISR, et les conséquences plus longues du traitement (e.g. "trouver une trajectoire de sortie") sont prises en compte dans le programme principal.

Un exemple de gestion d'interruption sur GPIO est proposé dans le code 1.

Listing 1 – Interruption GPIO

```
1  // avr-gcc -mmcu=atmega32u4 -Os  -Wall  -o gpio_int.out gpio_int.c
2  // TODO : relier un fil entre GND et D0 (PD2) pour allumer/eteindre la diode orange
3
4  #define F_CPU 16000000UL
5  #include <avr/io.h>
6  #include <avr/interrupt.h>
7  #include <util/delay.h>  // _delay_ms
8
9  ISR(INT2_vect)
10 {PORTB^=1<<PORTB5;
11  _delay_ms(100);         // software debounce
12 }
13
14 int main(void)
15 { MCUCR &=~(1<<PUD);
16   DDRD  &= ~(1<<PORTD2); //int2 sur PD2 qui est note' D0
```

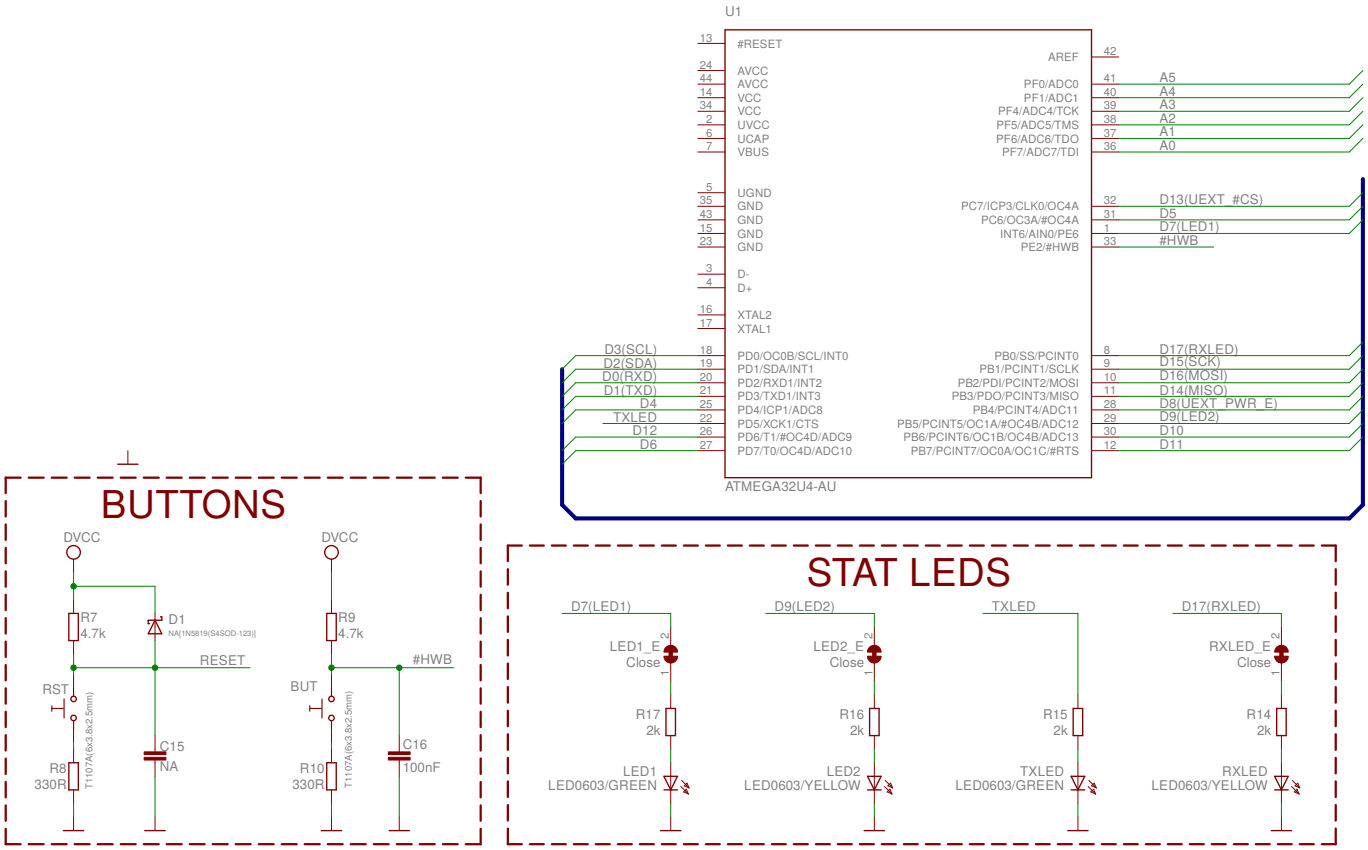

FIGURE 1 – Extrait du schéma de l'Olimexino-32U4

```
17    PORTD |= (1<<PORTD2);  //pullup on, cf doc p62
18
19    DDRB |=(1<<PORTB5);
20    DDRE |=(1<<PORTE6);
21    PORTB &= ~(1<<PORTB5);
22    PORTE &= ~(1<<PORTE6);
23
24  // vv NECESSAIRE SI ON VEUT UTILISER PD0 (sinon le FET d'uext est desactif et GPIO est en pull down)
25  // DDRB|=1<<PB4;                    // UEXT powered (FET)
26  // PORTB&=~1<<PB4;
27
28    EIMSK = 1<<INT2;   //enable int2
29    USBCON=0;          // desactive l'interruption sur USB (activ'ee par bootloader)
30    sei();    //enable interrupts
31    while(1) {PORTE^=1<<PORTE6;_delay_ms(200);}
32  }
```

Cette solution d'attendre dans l'ISR est peu élégante puisque le microcontroleur ne peut pas gérer d'autre interruption pendant ce temps. Proposer une solution déclenchant un drapeau dans l'ISR et gérant les conséquences de ce drapeau (changement d'état de la LED) dans la fonction principale (`main()`).

Pour ce faire, on se rappellera que l'interruption est appelée continuellement tant que la broche liée au signal est maintenue au niveau bas. Afin de cesser l'appel intempestif au gestionnaire d'interruption (ISR), on pourra y placer l'ordre d'ignorer les interruption (`cli();` pour *clear interrupt*) et réactiver les interruptions (`sei();`) une fois le drapeau traité dans le programme principal.

L'attribution des vecteurs d'interruption, qui définissent l'emplacement mémoire à laquelle le pointeur de programme saute pour trouver l'instruction à exécuter lors de l'interruption (saut à l'ISR), se retrouve en désassemblant le binaire compilé et lié aux fonctions d'initialisation fournies par `gcc` (Fig. 2).

1. **Valider que l'interruption INT2 a été assignée à la bonne fonction en désassemblant l'exécutable par `avr-objdump -dSt executable`. Que se passe-t-il si une interruption non-initialisée est sollicitée?**
2. **Pourquoi ne trouve-t-on pas ce résultat en observant le code assembleur généré lors de `avr-gcc -S` du code source?**

Table 9-1. Reset and Interrupt Vectors

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	Reserved	Reserved
7	$000C	Reserved	Reserved
8	$000E	INT6	External Interrupt Request 6
9	$0010	Reserved	Reserved
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	USB General	USB General Interrupt request
12	$0016	USB Endpoint	USB Endpoint Interrupt request
13	$0018	WDT	Watchdog Time-out Interrupt
14	$001A	Reserved	Reserved
15	$001C	Reserved	Reserved
16	$001E	Reserved	Reserved
17	$0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	$0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	$0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	$0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	$0028	TIMER1 OVF	Timer/Counter1 Overflow
22	$002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	$002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	$002E	TIMER0 OVF	Timer/Counter0 Overflow
25	$0030	SPI (STC)	SPI Serial Transfer Complete
26	$0032	USART1 RX	USART1 Rx Complete
27	$0034	USART1 UDRE	USART1 Data Register Empty
28	$0036	USART1TX	USART1 Tx Complete
29	$0038	ANALOG COMP	Analog Comparator

```
Disassembly of section .text:

00000000 <__vectors>:
   0: 0c 94 56 00  jmp 0xac ; 0xac <__ctors_end>
   4: 0c 94 62 00  jmp 0xc4 ; 0xc4 <__vector_1>
   8: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
   c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  10: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  14: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  18: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  1c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  20: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  24: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  28: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  2c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  30: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  34: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  38: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  3c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  40: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  44: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  48: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  4c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  50: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  54: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  58: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  5c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  60: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  64: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  68: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  6c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  70: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  74: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  78: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  7c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  80: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  84: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  88: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  8c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  90: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  94: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  98: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  9c: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  a0: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  a4: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>
  a8: 0c 94 60 00  jmp 0xc0 ; 0xc0 <__bad_interrupt>

000000ac <__ctors_end>:
  ac: 11 24        eor r1, r1
  ae: 1f be        out 0x3f, r1 ; 63
  b0: cf ef        ldi r28, 0xFF ; 255
  b2: da e0        ldi r29, 0x0A ; 10
  b4: de bf        out 0x3e, r29 ; 62
  b6: cd bf        out 0x3d, r28 ; 61
  b8: 0e 94 7f 00  call 0xfe ; 0xfe <main>
  bc: 0c 94 9f 00  jmp 0x13e ; 0x13e <_exit>

000000c0 <__bad_interrupt>:
  c0: 0c 94 00 00  jmp 0 ; 0x0 <__vectors>

000000c4 <__vector_1>:
  c4: 1f 92        push r1
  c6: 0f 92        push r0
[...]
  f8: 0f 90        pop r0
  fa: 1f 90        pop r1
  fc: 18 95        reti

000000fe <main>:
  fe: 85 b7        in r24, 0x35 ; 53
[...]
```

FIGURE 2 – Gauche : extrait de la *datasheet* explicitant les vecteurs d'interruption. Droite : code déassemblé par `avr-objdump`.

3. **Observer le code désassemblé si le lien avec le fichier de démarrage est désactivé par `-nostartfiles`.**
4. **Pour le moment le déclenchement de l'interruption se fait sur un *niveau*. Modifier le code pour déclencher l'interruption sur un *front* – transition d'état de bas vers haut par exemple.**
5. La solution de sonder périodiquement un périphérique, que ce soit le bouton poussoir pour changer la direction de défilement ou pour définir la transition d'un état à l'autre par *timer* tel que nous l'avons vu dans le TP précédent (chenillard), est peu élégante. Ces périphériques sont tous associés à des évènements déclenchés de façon asynchrone lorsqu'un évènement se déclenche en vue d'interrompre l'exécution séquentielle du programme principal, agir en conséquence de l'évènement, et reprendre l'exécution du programme principal où il avait été interrompu.
6. **En reprenant le montage du chenillard, identifier l'interruption associée à la broche à laquelle est connectée le bouton poussoir. Remplacer le test périodique de l'état du bouton poussoir par le gestionnaire d'interrpution associée.**
7. **Remplacer le test sur l'état du *timer* vu dans le TP précédent (chenillard) par le déclenchement périodique d'une interruption induisant la transition d'état.**

À l'issue de ces optimisations du programme, la boucle principale doit être vide. Elle est donc disponible pour des opérations utiles telles que calcul d'une boucle d'asservissement, loi de commande ou interaction avec l'utilisateur.

2 Interruptions *timer*

2.1 Overflow

Dans le mode *overflow*, le compteur 0 codé sur 8 bits va de 0 à 255, le compteur 1 codé sur 16 bits de 0 à 65535 (Fig. 4).

Table 13-9. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)

Table 13-9. Clock Select Bit Description (Continued)

CS02	CS01	CS00	Description
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

FIGURE 3 – Configuration de l'horloge du *timer*0[1, p.105]

Table 14-6. Clock Select Bit Description

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

FIGURE 4 – Configuration de l'horloge cadençant le *timer*1[1, p.133]

Sachant que la fréquence du quartz qui cadence le microcontrôleur est 16 MHz, quelle est la fréquence de clignotement de la LED ?

Même question avec le *timer0* : pourquoi avons nous ajouté la condition sur `c` pour commuter l'état de la diode ? Quelle condition sur `c` faudrait-il ajouter pour que la diode commute à 0,5 Hz ?

Listing 2 – Interruption timer

```
1  #define F_CPU 16000000UL
2
3  #include <avr/io.h>
4  #include <stdbool.h>
5  #include <avr/interrupt.h>
6
7  volatile int c=0;
8
9  ISR(TIMER1_OVF_vect) {PORTE^=1<<PORTE6;}
10 ISR(TIMER0_OVF_vect) {c++; if (c==5) {PORTB^=1<<PORTB5;c=0;}}
11
12 void timer_init()
13 {TCCR1B = (1 <<CS11)|(1<<CS10);
14  TIMSK1=(1<<TOIE1);                // diode verte
15
16  TCCR0B = (1 <<CS02)|(1<<CS00);
17  TIMSK0=(1<<TOIE0) ;
18  sei();
19 }
20
21 int main( void ) {
22   DDRB |=1<<PORTB5;
23   DDRE |=1<<PORTE6;
24   PORTB |= 1<<PORTB5;
25   PORTE &=~1<<PORTE6;
26
27   USBCON=0; // desactivation interrupt sur USB
28
29   timer_init();
30   while(1) {}
31 }
```

2.2 Output Compare

Nous avions vu que le *timer* peut se configurer pour ne pas atteindre sa limite de stockage mais pour s'arrêter en cours de décompte (*Output Compare*). Ce mode de fonctionnement est compatible avec les interruptions tel que illustré dans le code 3

Listing 3 – Interruption timer

```
#include <avr/io.h>
#include <avr/power.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>

#define F_CPU 16000000UL

#define delai_leds (16000*2)

#define avec_interruption

ISR (TIMER1_COMPA_vect) {PORTB^=1<<PORTB5;PORTE^=1<<PORTE6;}

void timer1_init()  // Timer1 avec prescaler=64 et CTC
{
 TCCR1A=0;
 TCCR1B= (1 << WGM12)|(1 << CS11)|(1 << CS10);
 TCNT1=0;
 OCR1A = delai_leds;      // valeur seuil <- delai
#ifdef avec_interruption
 TIMSK1= (1 << OCIE1A);   // active interruption
 sei();
#endif
}

int main(void)
{
#ifdef avec_interruption
  USBCON=0;   // https://github.com/pololu/a-star/issues/3
#endif

  DDRB |=1<<PORTB5;
  DDRE |=1<<PORTE6;
  PORTB |= 1<<PORTB5;
  PORTE &= ~(1<<PORTE6);

  timer1_init();
  while(1) {
#ifndef avec_interruption
       if (TIFR1 & (1 << OCF1A))
          {PORTB^=1<<PORTB5;PORTE^=1<<PORTE6;}
       TIFR1 |= (1 << OCF1A);
#endif
  }
}
```

Reprendre le programme du chenillard du premier TP et le modifier pour que la boucle infinie de la fonction principale soit vide, avec l'évolution du motif des LEDs calculé exclusivement dans le gestionnaire d'interruption.

2.3 *Input Capture*

Les *timers* ont jusqu'ici été exploités en sortie, *i.e.* pour générer un signal. Ils peuvent aussi être utilisés pour enregistrer l'état du compteur lors d'une transition d'état, et ainsi mesurer finement un intervalle de temps ou une fréquence. Il s'agit du mode *input capture*.

Nous profitons de l'opportunité de devoir générer un signal périodique pour tester le mode PWM (*Pulse Width Modulation*). La PWM est un mode classique de commande dans lequel la période des impulsions est constante mais la largeur des impulsions varie et encode l'information transmise. Il s'agit d'un mode couramment supporté par les *timers* puisqu'utilisés par exemple en commande de servo moteur (de modélisme notamment) ou pour générer une tension (en l'absence de DAC, un filtre passe-bas derrière une PWM fait office de sortie de tension continue commandée numériquement).

Observer à l'oscilloscope la sortie D9. Justifier la période observée par rapport à la valeur programmée dans ICR1.

Connecter D13 (Input Capture du *timer 3*) à D9 (PWM liée au *timer 1*). Observer le décompte.

Listing 4 – Interruption timer

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>  // _delay_ms

#undef rs  // affichage sur RS232 (#define) ou sur USB (#undef)

volatile unsigned short res;

#ifdef rs
#define USART_BAUDRATE 9600

void transmit_data(uint8_t data)
{while ( !( UCSR1A & (1<<UDRE1)) );
 UDR1 = data;
}

void usart_setup()              // initialisation UART
{unsigned short baud;
 baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
 DDRD  |= 0x18;
 UBRR1H = (unsigned char)(baud>>8);
 UBRR1L = (unsigned char)baud;

 UCSR1C = (1<<UCSZ11) | (1<<UCSZ10); //Set data frame format: asynchronous mode,no parity, 1 stop bit, 8 bit size
 UCSR1B = (1<<RXEN1 ) | (1<<TXEN1 ); //Enable Transmitter and Receiver
}
#else

#include "VirtualSerial.h"

extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
extern FILE USBSerialStream;

void transmit_data(uint8_t data)
{char buffer[2];
 buffer[0]=data;buffer[1]=0;
 fputs(buffer, &USBSerialStream);

}
#endif

void write_char(unsigned char c)
{if ((c>>4)>9) transmit_data((c>>4)-10+'A'); else transmit_data((c>>4)+'0');
 if ((c&0x0f)>9) transmit_data((c&0x0f)-10+'A'); else transmit_data((c&0x0f)+'0');
}

void write_short(unsigned short s)
{write_char(s>>8);
 write_char(s&0xff);
}

void myicp_setup() // initialisation timer3
{TCCR3A = 1<<WGM31 | 1<<WGM30;
 TCCR3B = 1<<ICES3 | 1<<WGM33 | 1<<CS31;
 OCR3A  = 32000;
 TIMSK3 = 1<<ICIE3;
 TIFR3  = 1<<ICF3;
}

ISR(TIMER3_CAPT_vect)
{res  = ICR3;
 TCNT3 = 0;
}
```

```
65
66 // http://blog.saikoled.com/post/43165849837/secret-konami-cheat-code-to-high-resolution-pwm-on-your
67 void mypwm_setup()  {
68   DDRB |= (1<<7)|(1<<6)|(1<<5); // B5/OC1A, B6/OC1B, B7/OC1C
69
70   // TCCR1A is the Timer/Counter 1 Control Register A.
71   // TCCR1A[7:6] = COM1A[1:0] (Mode for Channel A)
72   // TCCR1A[5:4] = COM1B[1:0] (Mode for Channel B)
73   // TCCR1A[3:2] = COM1C[1:0] (Mode for Channel C)
74   // COM1x[1:0] = 0b10 (Clear on match. Set at TOP)
75
76   // Clock Prescaler Stuff
77   // TCCR1B[2:0] = CS1[2:0] (Clock Select for Timer 1)
78   // CS1[2:0] = 0b001 (No Prescaler)
79
80   // Waveform Generation Mode Setup Stuff
81
82   // TCCR1A[1:0] = WGM1[1:0] (WGM for Timer 1 LSB)
83   // TCCR1B[4:3] = WGM1[3:2] (WGM for Timer 1 MSB)
84   // WGM[3:0] = 0b1110 (i.e. Mode 14)
85   // Mode 14 - Fast PWM, TOP from ICR1, Set OC1x at
86   // TOP, clear at OCR1x.
87
88   // ICR1 is a 16-bit register that should be written
89   // with the desired value for TOP. This is the value
90   // at which the PWM cycle restarts and defines the
91   // resolution.
92
93   // For 16-bit resolution, we need to program ICR1 with
94   // 0xFFFF.
95   ICR1 = 0x3e80;  // intervalle de repetition du crenau = 1 ms
96   TCCR1A = 0xAA;
97   TCCR1B = 0x19;
98
99   OCR1A=0x3e80/2; // largeur du crenau = 1/2 ms
100 }
101
102 int main(void)
103 {
104   DDRB |=1<<PORTB5;
105   DDRE |=1<<PORTE6;
106   PORTB |= 1<<PORTB5;
107   PORTE &= ~1<<PORTE6;
108   mypwm_setup();
109   myicp_setup();
110 //  sei();
111 #ifdef rs
112   usart_setup();
113 #else
114   USBCON=0;
115   SetupHardware();
116   CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
117   GlobalInterruptEnable();
118 #endif
119
120  while(1)
121   {if (res!=0) {write_short(res); transmit_data(0x0A); transmit_data(0x0D); res=0;
122   }
123 #ifdef rs
124 #else
125      CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
126
127      CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
128      USB_USBTask();
129 #endif
```

```
130     //_delay_ms(500);
131   }
132 }
```

Faire varier la période du signal de PWM (registre ICR1) : qu'observe-t-on ?
Faire varier la largeur du signal de PWM (registre OCR1A) : qu'observe-t-on ?

3 Le chien de garde

Un *timer* bien particulier garantit qu'un programme sur système embarqué ne doit pas pouvoir bloquer le système qu'il contrôle. Il s'agit d'un compteur qui ne peut être manipulé par logiciel pour d'autres fonctions qu'initialiser sa période et le réinitialiser. Si ce compteur atteint sa borne de répétition, il induit une réinitialisation du microcontrôleur (*reset*). Le programmeur doit prendre soin, dans sa boucle infinie, de réinitialiser le compteur pour s'assurer que cette condition n'arrive jamais (fonction `wdt_reset()`). Il s'agit donc d'une fonctionnalité très utile qui garantit le redémarrage du microcontrôleur qui se trouve dans une situation de bloquage (lecture d'un capteur défectueux par exemple). La solution de réinitialiser le microcontrôleur n'est certes pas toujours satisfaisante, mais toujours meilleure qu'attendre indéfiniment un évènement qui n'arrive pas et bloque le système que le microcontrôleur commande [2]. Le nom de ce *timer* est le chien de garde, ou *watchdog* dans la terminologie anglo-saxone.

Listing 5 – Watchdog timer

```
1  #define F_CPU 16000000UL
2  #include <avr/io.h>
3  #include <avr/wdt.h>     // voir /usr/lib/avr/include//avr/wdt.h pour les cst
4  #include <util/delay.h>  // _delay_ms
5  #include <avr/power.h>
6  #include <avr/interrupt.h>
7
8  #define USART_BAUDRATE 9600
9
10 void transmit_data(uint8_t data)
11 {while ( !( UCSR1A & (1<<UDRE1)) );
12  UDR1 = data;
13 }
14
15 void usart_setup()              // initialisation UART
16 {unsigned short baud;
17  baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
18  DDRD  |= 0x18;
19  UBRR1H = (unsigned char)(baud>>8);
20  UBRR1L = (unsigned char)baud;
21
22  UCSR1C = (1<<UCSZ11) | (1<<UCSZ10); //Set data frame format: asynchronous mode,no parity, 1 stop bit, 8 bit size
23  UCSR1B = (1<<RXEN1 ) | (1<<TXEN1 ); //Enable Transmitter and Receiver
24 }
25
26 int main (void)
27 {DDRB |=1<<PORTB5;
28  PORTB |= 1<<PORTB5;
29
30  _delay_ms(1000);
31
32  wdt_enable(WDTO_120MS);
33  usart_setup();
34  USBCON=0;
35  while (1)
36    {wdt_reset(); // pour reinitialiser le WDT
37     PORTB&=~1<<PORTB5;
38     _delay_ms(20); // 20 ms ici + USB ~ 120 ms
39     transmit_data('.');
40    }
41 }
```

Compiler le programme ci-dessus, le flasher et sonder le port de communication par `cat < /dev/ttyACM0`. Que se passe-t-il ?

Modifier la valeur de seuil du *watchdog* (fonction `wdt_enable()` avec un des intervalles de temps visible dans `/usr/lib/avr/include//avr/wdt.h` – par exemple `WDTO_60MS`. Commenter.

4 Communication – UART

La communication asynchrone UART (sur périphérique USART capable de supporter les transmissions synchrones ou asynchrones) est relativement lente car limitée car l'écart potentiel de vitesse entre les horloges des deux interlocuteurs qui supposent être cadencées au même rythme entre émission et réception. En pratique, selon la fréquence du quartz et sa capacité à générer le baudrate, un écart jusqu'à 10% de la vitesse nominale peut être observée (Fig. 5).

Table 18-12. Examples of UBRRn Settings for Commonly Used Oscillator Frequencies (Continued)

Baud Rate (bps)	f_{osc} = 16.0000 MHz				f_{osc} = 18.4320 MHz				f_{osc} = 20.0000 MHz			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%	479	0.0%	959	0.0%	520	0.0%	1041	0.0%
4800	207	0.2%	416	-0.1%	239	0.0%	479	0.0%	259	0.2%	520	0.0%
9600	103	0.2%	207	0.2%	119	0.0%	239	0.0%	129	0.2%	259	0.2%
14.4k	68	0.6%	138	-0.1%	79	0.0%	159	0.0%	86	-0.2%	173	-0.2%
19.2k	51	0.2%	103	0.2%	59	0.0%	119	0.0%	64	0.2%	129	0.2%
28.8k	34	-0.8%	68	0.6%	39	0.0%	79	0.0%	42	0.9%	86	-0.2%
38.4k	25	0.2%	51	0.2%	29	0.0%	59	0.0%	32	-1.4%	64	0.2%
57.6k	16	2.1%	34	-0.8%	19	0.0%	39	0.0%	21	-1.4%	42	0.9%
76.8k	12	0.2%	25	0.2%	14	0.0%	29	0.0%	15	1.7%	32	-1.4%
115.2k	8	-3.5%	16	2.1%	9	0.0%	19	0.0%	10	-1.4%	21	-1.4%
230.4k	3	8.5%	8	-3.5%	4	0.0%	9	0.0%	4	8.5%	10	-1.4%
250k	3	0.0%	7	0.0%	4	-7.8%	8	2.4%	4	0.0%	9	0.0%
0.5M	1	0.0%	3	0.0%	–	–	4	-7.8%	–	–	4	0.0%
1M	0	0.0%	1	0.0%	–	–	–	–	–	–	–	–
Max. (1)	1 Mbps		2 Mbps		1.152 Mbps		2.304 Mbps		1.25 Mbps		2.5 Mbps	

1. UBRR = 0, Error = 0.0%

FIGURE 5 – Débits de transferts accessibles en fonction des fréquences du quartz cadençant le microcontroleur, sachant que le facteur de division permettant de s'approcher au mieux du *baudrate* recherché est nécessairement un entier.

Quel écart maximum de vitesse est admissible lors d'une transmission au protocole 8N1 ?

4.1 Mode *polling*

Le port de communication asynchrone est probablement le périphérique le plus utile sur un microcontrôleur. Après avoir initialisé l'horloge qui définit le débit de données (*baudrate*) ainsi que la nature des transactions (nombre de bits par mot transmis, encapsulation), les caractères sont transmis dès que le registre d'émission est libre. L'universalité de ce mode de communication tient en sa simplicité : tout système numérique programmable propose ce port de communication, même sur des plateformes aussi complexes qu'un routeur wifi (`http://www.rwhitby.net/projects/wrt54gs`) ou une console de jeu (`http://nil.rpc1.org/psp/remote.html`).
Le périphérique du PC de transactions sur le port série asynchrone virtuel sur USB se nomme `/dev/ttyUSB0`. Afin d'afficher les messages transmis sur un port de communication, un logiciel dédié sonde l'état des ports de communication et affiche les messages qui y sont transmis : un terminal. Sous GNU/Linux, `minicom` est un terminal qui se lance en précisant le périphérique par `minicom -D /dev/ttyUSB0` et qui se configure par `CTRL-a` puis `o` (Options) pour définir le débit de communication (baudrate). Sous MS-Windows, `putty` propose des fonctionnalités identiques.
Qu'affiche le programme ci-dessous ? Quelle est la durée de transmission de chaque caractère ?

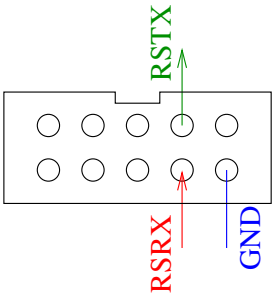

FIGURE 6: Branchement de la liaison RS232 sur connecteur UEXT (en tenant le circuit vers soi).

Listing 6 – Liaison UART en 9600 bauds

```
1 #include <avr/io.h>      // voir /usr/lib/avr/include/avr/iom32u4.h
2 #define F_CPU 16000000UL
```

```
3  #include <util/delay.h>  // _delay_ms
4
5  #define USART_BAUDRATE (57600)
6
7  // https://github.com/tomvdb/avr_arduino_leonardo/blob/master/examples/uart/main.c
8  void uart_transmit( unsigned char data )
9  {while (!(UCSR1A&(1<<UDRE1))) ;
10  UDR1 = data;
11 }
12
13 unsigned char uart_receive(void)
14 {while (!(UCSR1A&(1<<RXC1))) ;
15  return UDR1;
16 }
17
18 int main(void){
19 unsigned short baud;
20 char c=42;  // '*'
21   DDRB |=1<<PORTB5;
22   DDRE |=1<<PORTE6;
23   PORTB |= 1<<PORTB5;
24   PORTE &= ~1<<PORTE6;
25
26   UCSR1A = 0;                          // importantly U2X1 = 0
27   UCSR1B = 0;
28   UCSR1B = (1 << RXEN1)|(1 << TXEN1); // enable receiver and transmitter
29   UCSR1C = _BV(UCSZ11) | _BV(UCSZ10); // 8N1
30   // UCSR1D = 0;                          // no rtc/cts (probleme de version de libavr)
31   baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
32   UBRR1H = (unsigned char)(baud>>8);
33   UBRR1L = (unsigned char)baud;
34
35   while (1){
36     PORTB^=1<<PORTB5;PORTE^=1<<PORTE6; // LED
37     _delay_ms(10);
38
39     uart_transmit(c);                  // affiche le message
40     c++; if (c==127) c=32;
41   }
42   return 0;
43 }
```

Proposer une fonction chargée d'afficher une chaîne de caractères. Afficher la phrase "Hello World".

Proposer un programme qui reçoit des caractères sur le port série, et renvoie sur le port série le caractère suivant de l'alphabet de celui qui a été reçu.

4.2 Passage de paramètres – par valeur et par référence

Une fonction en C reçoit son argument par valeur si nous la déclarons de la forme `void fonction(int argument)` . Dans ce cas, une *copie* de la valeur de l'argument passé à l'appel de la fonction est placée sur la pile, la fonction traite l'argument, mais la valeur initiale de l'argument n'est jamais modifiée. Si la valeur de l'argument doit être modifiée par la fonction, il faut passer l'adresse mémoire à laquelle la valeur est stockée : il s'agit d'un passage par référence.

La déclaration d'une variable `short i;` se traduit par l'affectation de 2 octets contigüs en mémoire pour stocker la valeur de `i`. L'emplacement mémoire alloué pour stocker `i` est indiquée par le *pointeur* vers `i`. Le pointeur vers une variable s'obtient en préfixant le nom de la variable par `&`. Ainsi, l'adresse à laquelle est stockée `i` s'obtient par `&i`. Le contenu de cet emplacement mémoire est indiqué en préfixant le pointeur du symbole `*`. Nous avons donc l'équivalence entre `i` et `*(&i)`.

Le passage de paramètre par référence se fait donc en déclarant une fonction `void fonction(int* argument)` qui indique que la fonction attend un pointeur, et le passage de paramètre se fait en fournissant le pointeur vers l'argument. De cette façon, la pile qui s'occupe du passage de l'argument contient cette fois l'adresse de l'argument lors de l'appel à la fonction, et toute modification au contenu de l'adresse sera reproduit dans la fonction appelante.

Proposer une fonction qui reçoit en argument un caractère obtenu sur le port série, et le transforme en la majuscule si le caractère reçu est une lettre minuscule, et réciproquement. On notera que dans la table ASCII, les majuscules et minuscules sont séparées de 32.

4.3 Interruption USART

Afin de ne pas handicaper l'exécution séquentielle d'un programme par la lecture périodique du port de communication asynchrone, il est souhaitable d'interrompre l'exécution séquentielle lorsque l'évènement occasionnel de réception d'un caractère se produit. Cependant, nous ne voulons pas passer de temps inutilement dans l'interruption qui doit se réduire uniquement au code critique (dans ce cas recevoir le caractère). À ces fins, nous définissons un drapeau (*flag*) qui indique au programme principal la réception d'un caractère et qu'une action doit être prise en conséquent.

Listing 7 – Interruption timer

```
#include <avr/io.h>      // voir /usr/lib/avr/include/avr/iom32u4.h
#include <avr/interrupt.h>
#define F_CPU 16000000UL
#include <util/delay.h>  // _delay_ms

# define USART_BAUDRATE 9600

#ifdef atmega32u2
// pour simavr
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32");

const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
                { AVR_MCU_VCD_SYMBOL("PORTB"), .what = (void*)&PORTB, },
        };
// fin simavr

#include <stdio.h>
static int uart_putchar(char c, FILE *stream) {
  if (c == '\n')
    uart_putchar('\r', stream);
  //loop_until_bit_is_set(UCSR0A, UDRE0);
  //UDR0 = c;
  return 0;
}

static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL,
                                         _FDEV_SETUP_WRITE);
#endif

volatile char received,flag=0;

ISR(USART1_RX_vect) {received=UDR1;flag=1;}

void uart_transmit( unsigned char data )
{while (!(UCSR1A&(1<<UDRE1))) ;
 UDR1 = data;
}

int main(void){
unsigned short baud;
  DDRB |=1<<PORTB5;    // LEDs
  PORTB |= 1<<PORTB5;
#ifndef atmega32u2
  DDRE |=1<<PORTE6;
  PORTE &= ~1<<PORTE6;
#endif

  UCSR1A = 0;                           // importantly U2X1 = 0
  UCSR1B = (1 << RXEN1) | (1 << TXEN1); // enable receiver and transmitter
  UCSR1B|= (1 << RXCIE1);               // ACTIVATION INTERRUPTION RX Completed
  UCSR1C = _BV(UCSZ11) | _BV(UCSZ10);   // no parity, 8 data bits, 1 stop bit
// UCSR1D = 0;                           // no cts, no rts
  baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
  UBRR1H = (unsigned char)(baud>>8);
  UBRR1L = (unsigned char)baud;
```

```
57
58   USBCON=0; // desactive l'interruption USB
59   sei();
60
61 #ifndef atmega32u2
62   stdout=&mystdout;
63 #endif
64
65   while (1){
66     PORTB^=1<<PORTB5;
67 #ifndef atmega32u2
68     PORTE^=1<<PORTE6;
69 #endif
70     _delay_ms(100);  //Attente de 500ms
71
72     if (flag!=0) {flag=0;uart_transmit(received+1);}
73   }
74   return 0;
75 }
```

Justifier de la définition de variables globales pour échanger des informations entre programme principal et gestionnaire d'interruption.

Proposer une solution pour recevoir plusieurs caractères entre deux traitements du *flag* par le programme principal.

5 Conclusion

Proposer un programme qui fait clignoter une diode périodiquement, détecte une transition d'état sur un GPIO, et communique par USART, en limitant la boucle infinie dans la fonction principale `main()` **à** `while (1) {};`

Références

[1] Atmel, 8-bit Microcontroller with 16/32K Bytes of ISP Flash and USB Controller – ATmega16U4 & ATmega32U4, disponible à `https://ww1.microchip.com/downloads/en/DeviceDoc/7799S.pdf` (version 7799ES–AVR–09/2012)

[2] J. Matijevic & E. Dewell, *Anomaly Recovery and the Mars Exploration Rovers*, disponible à `https://trs.jpl.nasa.gov/handle/2014/39897`