M2103 TD n°1 : langage C : opérateurs et expressions

Le contenu de ce polycopié peut être accompagné par un cours wikiversité : https://fr.wikiversity.org/wiki/Micro_contr%C3%B4leurs_AVR

I. Arithmétique binaire et expressions en C AVR

Pour bien comprendre la signification des expressions, il est essentiel d'avoir 2 notions en tête : la priorité et l'associativité. Nous donnons ci-après un tableau des opérateurs par priorité décroissante :

Catégorie d'opérateurs	Opérateurs	Associativité
fonction, tableau, membre de structure, pointeur sur un membre de structure	( ) [ ] . ->	Gauche -> Droite
opérateurs unaires	- ++ -- ! ~ * & sizeof (type)	Droite ->Gauche
multiplication, division, modulo	* / %	Gauche -> Droite
addition, soustraction	+ -	Gauche -> Droite
décalage	<< >>	Gauche -> Droite
opérateurs relationnels	< <= > >=	Gauche -> Droite
opérateurs de comparaison	== !=	Gauche -> Droite
et binaire	&	Gauche -> Droite
ou exclusif binaire	^	Gauche -> Droite
ou binaire	|	Gauche -> Droite
et logique	&&	Gauche -> Droite
ou logique	||	Gauche -> Droite
opérateur conditionnel	? :	Droite -> Gauche
opérateurs d'affectation	= += -= *= /= %= &= ^= |= <<= >>=	Droite -> Gauche
opérateur virgule	,	Gauche -> Droite

Exercice 1

d = a & 0xF0 | 0xFF ;
d = a | 0xFF & 0xF0 ;
d = (b & 0x0F) | 0xF0 ;
d = (c ^ 0xAA) ^ 0xAA ;
Évaluer ces expressions pour a=0XFF, b=0xAA, c=0x00

Les types du C AVR sont :

```
1       unsigned char a;//8 bits, 0 to 255
2       signed char b;  //8 bits, -128 to 127
3       unsigned int c; //16 bits, 0 to 65535
4       signed int d;   //16 bits, -32768 to 32767
5       long e;      //32 bits, -2147483648 to 2147483647
6       float f;   //32 bits
```

Exercice 2

b7	b6	b5	b4	b3	b2	b1	b0

Si une variable p1 de type signed char (8 bits signés) est déclarée écrire les expressions en C permettant de :

- mettre à 1 le bit b2
- mettre à 1 le bit b3 et b6
- mettre à 0 le bit b0
- mettre à 0 le bit b4 et b5
- inverser le bit b3 (se fait facilement avec un ou exclusif)
- mettre à 1 le bit b2 et à 0 le bit b0
- mettre à 1 les bits b0 et b7 et à 0 les bits b3 et b4

Exercice 3

Soit une variable :

```
char nb;
```

Écrire les expressions permettant de calculer les centaines, les dizaines et les unité de cette variable.

Il existe plusieurs autres méthodes pour positionner les bits (http://www.microchipc.com/HiTechCFAQ/index.php)

1° méthode pour positionner bit à bit :

```
#define bit_set(var,bitno) ((var) |= 1 << (bitno))
#define bit_clr(var,bitno) ((var) &= ~(1 << (bitno)))
unsigned char x=0b0001;
bit_set(x,3); //now x=0b1001;
bit_clr(x,0); //now x=0b1000;
```

2° méthode pour positionner plusieurs bits

```
#define bits_on(var,mask) var |= mask
#define bits_off(var,mask) var &= ~0 ^ mask
unsigned char x=0b1010;
bits_on(x,0b0001);    //now x=0b1011
bits_off(x,0b0011);   //now x=0b1000
```

II.Expression booléenne vraie

Dans la suite du cours on appellera expression booléenne (E. B.) une expression qui si elle était affectée à une variable donnerait soit la valeur 0 ou soit la valeur 1. Le C est un langage qui ne permet de fabriquer que des expressions (Expr). C'est très pratique, mais la confusion des deux peut conduire à des erreurs. Donnons un exemple :

expression (Expr)	***expression booléenne (E. B.)***

`char n=10;`
`while(n) {`
`.....;`
`n--; }`
	`char n=10;`
	`while(n!=0) {`
	`.....;`
	`n--; }`

Ces deux constructions sont permises en C car il y a une technique universelle qui permet de transformer une expression en expression booléenne.

- tout ce qui est évalué à 0 est considéré comme faux,

- tout ce qui est évalué différent de 0 est considéré comme vrai.

Si on se rappelle que dans une parenthèse d'un while on a une Expression Booléenne (E. B.). D'un côté n et de l'autre n!=0 sont donc des E. B. La différence entre les deux est que pour n elle peut prendre toutes les valeurs entre -128 et +127 (car n est de type char) alors que n!=0 ne prendra que deux valeurs 0 ou 1. La deuxième sera donc appelée E. B. et la première Expr qui sera transformée comme indiqué.

Le problème du langage C est que l'oubli d'un & ou d'un | conduit en général à une expression souvent en lieu et place d'une expression booléenne ce qui est tout à fait autorisé. Par exemple écrire a & b au lieu de a && b, ou pire encore un "=" au lieu d'un "==".

Exercice 4

Différence entre && et &

Évaluer les expressions :

a&b

a&&b

pour a= 0xF0 et b=0x0F

En déduire les valeurs booléennes correspondantes (si ces expressions étaient utilisées dans un if par exemple).

Construire des expressions booléennes sur les tests suivants

expression vraie si

le bit b6 est à 1

le bit b3 est à 0

le bit b6 est l'inverse du bit b3

le bit b2 est à 1 et le bit b4 est à 0

le bit b2 est à 1 ou le bit b7 est à 0

Les tests d'un bit particulier en C peuvent aussi être réalisés de la manière suivante (http://www.microchipc.com/HiTechCFAQ/index.php)

```
#define testbit_on(data,bitno) ((data>>bitno)&0x01)
x=0b1000; //decimal 8 or hexadecimal 0x8
if (testbit_on(x,3)) a(); else b(); //function a() gets executed
if (testbit_on(x,0)) a(); else b(); //function b() gets executed
if (!testbit_on(x,0)) b(); //function b() gets executed
```

Exercice 5

Quelle opération arithmétique est réalisée par un décalage ? Évaluer pour cela les expressions suivantes (avec a=12 et b=23) :

- a = a >> 1 (ou a >>= 1)

- a = a >> 2 (ou a >>= 2)

- b = b << 1 (ou b <<=1)

- b = b << 2 (ou b <<=2)

Généralisation.

Construire une vraie expression booléenne avec opérateur de décalage, & et ^ qui reprend le test de l'exercice précédent : le bit b6 est l'inverse du bit b3

Exercice 6

On donne le sous-programme suivant (tiré d'un ancien projet inter-semestre) :

```
void conversion(char nb,char result[8]){
  char i;
  for(i=7;i>=0;i--) if ((nb & (1<<i)) == (1<<i)) result[i]=1;
                    else result[i]=0;
}
```

1) Peut-on retirer des parenthèses dans l'expression booléenne du if ?

2) Peut-on écrire `(nb & (1<<i))` au lieu de `((nb & (1<<i)) == (1<<i))` ?

3) Construire l'expression booléenne qui permettrait l'écriture

```
for(i=7;i>=0;i--) if (E.B.?????) result[i]=0;
                  else result[i]=1;
```

en donnant toujours le même le même résultat de conversion.

4) Modifier le programme pour qu'il fasse une conversion d'entier (16 bits) vers le binaire.

5) Est-ce que l'algorithme suivant donne le même résultat de conversion :

```
 for(i=0;i<8;i++) {
  result[i]=nb%(2);
  nb = nb / 2;
 }
```

M2103 - TD n°2 Architecture des AVR ATMega

I. Architecture générale

1°) Architecture mémoire

La donnée de base de AVR (ATMegaXXX) est l'octet. L'octet comporte 8 bits. Le bit 0 est le bit de poids faible et le bit 7 est le bit de poids fort.

Mémoire programme : la mémoire programme de l'AVR (ATMega8) est organisée en mots de 16 bits et elle peut en contenir 4096 (soit 4k). Le bit 0 est aussi le bit de poids faible et le bit 15 est le bit de poids fort. La mémoire mémoire programme comporte donc 4k x 16 bits = 8 ko (d'où le nom ATMega8).

Exercice 1

On rappelle qu'une mémoire est caractérisée par un bus de donnée de largeur n (n bits de D_0 à D_{n-1}) et un bus d'adresse de largeur m (m bits de A_0 à A_{m-1}).

1°) Pour la mémoire programme de l'AVR (ATMega8) donner m et n.

2°) La mémoire programme de l'AVR (ATMega16) est 2 fois plus grande : donner les nouveaux m et n.

3°) La mémoire programme de l'AVR (ATMega328p) utilisé en TP est 4 fois plus grande qu'en 1°, donner les nouveaux m et n.

Mémoire EEPROM : l'AVR (ATMega8) dispose de 512 octets de mémoire EEPROM.

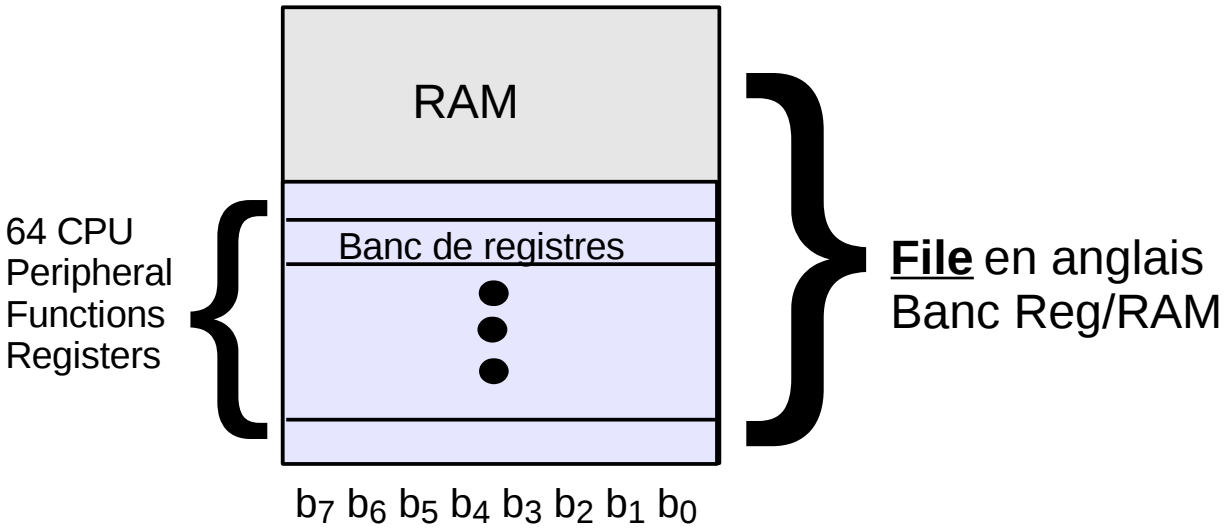
Exercice 2

Calculer m et n pour l'EEPROM de l'AVR (ATMega8).

Mémoire donnée :

Il est difficile de séparée la mémoire de données qui est une RAM statique (SRAM) des registres spécialisés.

Donnons en une description simplifiée



Description détaillée pour l'AVR (ATMega8) : la partie basse est aussi appelée Register File et commence en adresse 0x20. Puisque l'on a 64 registres on s'arrête à l'adresse 0x5F et c'est là que commence la RAM (1ko pour ATMega8 et ATMega16 et 2ko pour ATMega328p).

2°) Quelques registres spécialisés de l'AVR (ATMegaXXX)

Nous présentons quelques registres spéciaux dans un tableau :

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x3F(0x5F)	SREG	I	T	H	S	V	N	Z	C
0x32(0x52)	TCNT0	Timer0 8 bits							
0x2D(0x4D)	TCNT1H	Timer1 8 bits de poids fort							
0x2C(0x4C)	TCNT1L	Timer1 8 bits de poids faible							
0x23(0x43)	OCR2	Timer/Counter2 output compare register							
0x21(0x41)	WDTCR	-	-	-	WDCE	WDE	WDP2	WDP1	WDP0
0x20(0x40)	UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
0x18(0x38)	PORTB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
0x17(0x37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
0x16(0x36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
0x15(0x35)	PORTC	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
0x14(0x37)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
0x13(0x33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0

0x12(0x32)	PORTD	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
0x11(0x31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
0x10(0x30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
0x0C(0x2C)	UDR	Registre de données USART I/O							
0x0B(0x2B)	UCSRA	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM
0x0A(0x2A)	UCSRB	RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

Parmi les registres spéciaux, remarquez un registre appelé **SREG** (registre de statut) disposant des bits suivants :

C : retenue (Carry), Z : zéro, N : négatif, V : dépassement de capacité (overflow), S: bit de signe, H : demi-retenue, T : Bit Copy Storage et I : autorisation générale d'interruptions.

En plus de la SRAM et des 64 registres spéciaux, nous disposons de 32 registres d'usage général.

La pile est gérée en RAM par un pointeur de pile.

II. Utiliser les registres de l'AVR (ATMegaXXX) en C

L'idéal, pour une écriture facile en langage C, serait de pouvoir écrire :

```
1         // affectation d'une valeur dans un registre :
2         PORTB = val;
```

qui pourrait mettre directement la valeur contenue dans la variable val dans un registre appelé PORTB. Pour cela, il faut que le compilateur C (gcc pour nous) connaisse le mot PORTB. En général, ce sera le cas si vous mettez l'entête

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

en tout début de votre programme. Mais qu'en est-il pour les bits des registres ?

Le registre **SREG** présenté ci-dessus comporte des bits qui possèdent un nom. Ce n'est pas le seul registre a avoir cette propriété. Mais peut-on en déduire que les bits des registres possèdent tous un nom que le compilateur C connait ? La réponse est oui mais probablement pas à 100% !

1°) Exemple de fichier d'en-tête pour gcc

Pour illustrer les questions de la section précédente de manière un peu plus concrète, voici un exemple partiel de fichier d'entête :

```
1         /* Port D */
2         #define PIND  _SFR_IO8(0x10)
3         #define DDRD  _SFR_IO8(0x11)
4         #define PORTD _SFR_IO8(0x12)
5
6         /* Port C */
7         #define PINC  _SFR_IO8(0x13)
8         #define DDRC  _SFR_IO8(0x14)
9         #define PORTC _SFR_IO8(0x15)
```

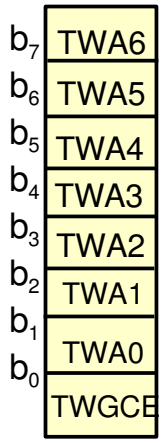
```
10
11        /* Port B */
12        #define PINB  _SFR_IO8(0x16)
13        #define DDRB  _SFR_IO8(0x17)
14        #define PORTB _SFR_IO8(0x18)
15        ....
16        /* PORTB */
17        #define PB7   7
18        #define PB6   6
19        #define PB5   5
20        #define PB4   4
21        #define PB3   3
22        #define PB2   2
23        #define PB1   1
24        #define PB0   0
25        ...
```

Essayez de distinguer la définition d'un registre dans la première partie de la définition d'un bit dans la deuxième partie !

2°) Les possibilités pour atteindre un bit particulier

Voici à travers un exemple comment accéder à un bit particulier pour le registre **TWAR** :

TWAR



Exemple en gcc

```
#define bit_set(var,bitno) ((var) |= 1 << (bitno))
#define bit_clr(var,bitno) ((var) &= ~(1 << (bitno)))
void main( void) {
 bit_set(TWAR,TWA5) ;
//*** Toujours pour set *****
  TWAR |=(1<<6);
//**** Si on connaît le nom
  TWAR |=(1<<TWA5);
//*** Toujours pour reset *****
  TWAR &= ~(1<<6);
//**** Si on connaît le nom
  TWAR &= ~(1<<TWA5);
  bit_clr(TWAR,TWA5) ;
}
```

Fichier d'entête

```
....
/* TWAR */
#define TWA6    7
#define TWA5    6
#define TWA4    5
#define TWA3    4
#define TWA2    3
#define TWA1    2
#define TWA0    1
#define TWGCE 0
...
```

Exercice 3

Pour vérifier la bonne compréhension de ce qui vient d'être dit, nous donnons à droite le fichier d'entête des bits correspondant au registre **TWCR**.

1°) Dessiner le registre correspondant

2°) En utilisant les masques du TD précédent, écrire les instructions C permettant :

Fichier d'entête

```
/* TWCR */
#define TWINT 7
#define TWEA  6
#define TWSTA 5
#define TWSTO 4
#define TWWC  3
#define TWEN  2
/* bit 1 reserved (TWI_TST?) */
#define TWIE  0
```

- mise à 1 de TWEA
- mise à 0 de TWWC.

- tester si TWINT est à 1

- tester si TWEN est à 0

3°) Refaire le même travail en utilisant les noms des bits directement.

Remarque : il existe une autre manière de faire ces mises à un et mises à 0 qui utilisent une macro « _BV » :

```
1         void main( void) {
2          ....
3          //*** Toujours pour set *****
4            //TWAR |= (1<<6) | (1<<5);
5          //**** Si on connaît le nom
6            TWAR |= _BV(TWA6) | _BV(TWA5);
7          //*** Toujours pour reset *****
8            TWAR &= ~_BV(TAW5);
9          ...
10
11         }
```

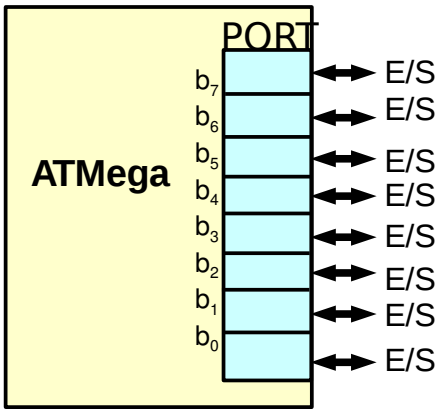
Exercice 4

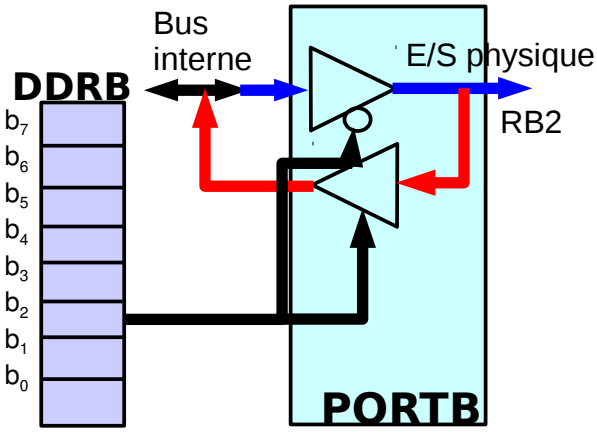
Comment peut-on utiliser la macro « _BV » pour mettre à 0 plusieurs bits à la fois ?

M2103 TD 3 Des LEDs et des boutons poussoirs sur des PORTs

I. Le PORT pour entrer et sortir des informations

1°) Le registre DDRB





Ce registre est d'un fonctionnement très simple et est lié au fonctionnement du **PORTB**.

> Chaque bit de **DDRB** positionné à 0 configure la broche correspondante en entrée. Chaque bit à 1 configure la pin en sortie

Au reset de l'AVR®, toutes les pins sont mises en entrée, afin de ne pas envoyer des signaux non désirés sur les pins. Les bits de **DDRB** seront donc mis à 0 lors de chaque reset.

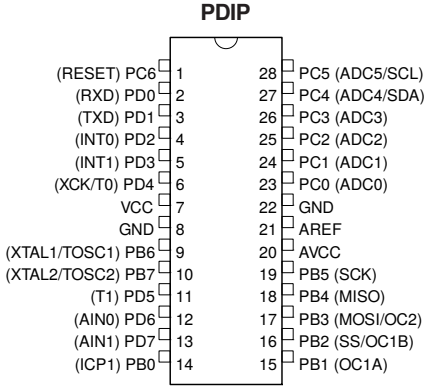
Notez que l'entrée est faite à l'aide de **PINB** et non de **PORTB** :

PORTB = val;

mais

val = PINB;

Brochage de l'ATMega8



Les entrées des **PORTX** peuvent être connectées à une résistance de rappel au +5V de manière interne, cette sélection s'effectuant par une écriture dans le registre **PORTX** (écriture d'un '1' avec bit correspondant en entrée pour pull-up et écriture d'un '0' pour désactiver le pull-up).

2°) Les registres PORTC et DDRC

Ces registres fonctionnent exactement de la même manière que **PORTB** et **DDRB**, mais concernent bien entendu les seules 7 pins PC6 ... PC0. Les seuls 7 bits utilisés sont les bits de poids faibles.

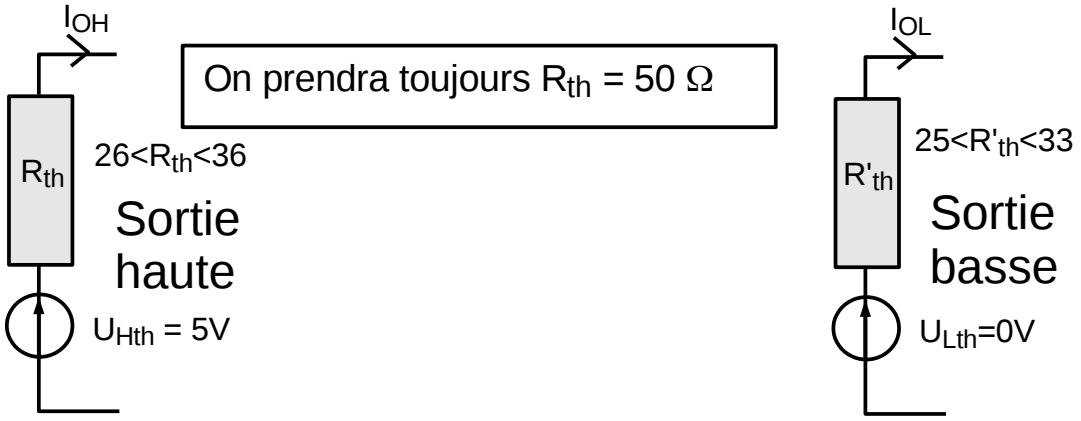
Voyons maintenant les particularités du **PORTC**.

Les 7 bits de **DDRC** sont désignés par DDC0 ... DDC6

II.Le PORT et sa modélisation électrique

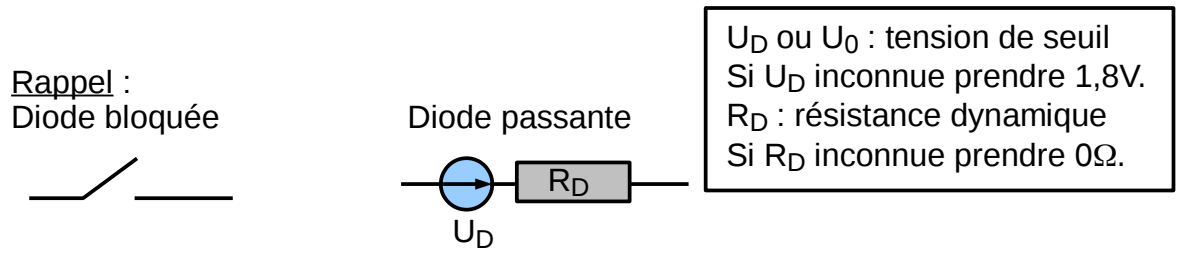
Le modèle électrique est très simple : on le modélise comme d'habitude à l'aide de Thevenin.

Il est raisonnable de prendre un Imax de 40 mA pour un ATMega8

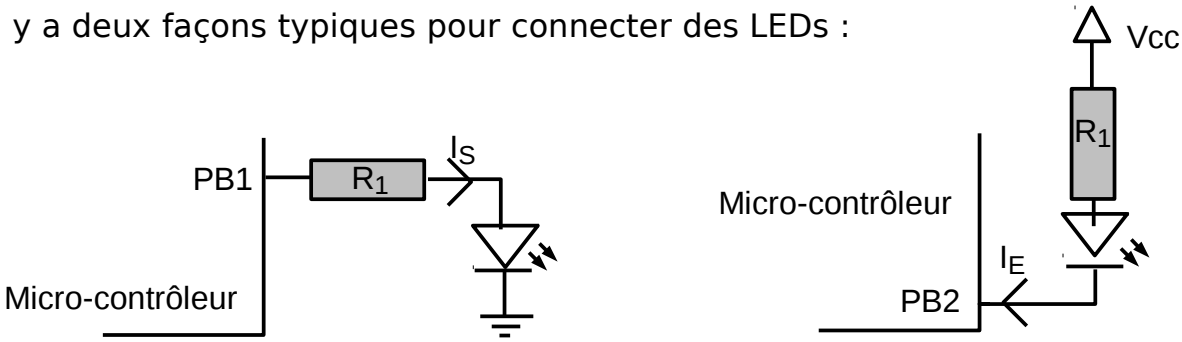


III.Connecter et dimensionner des LEDs

Une LED est une diode et par conséquent se modélise de la même manière :



Il y a deux façons typiques pour connecter des LEDs :



Micro-contrôleur comme source de courant	Micro-contrôleur comme puits de courant

Exercice 1

Nous allons essayer de dimensionner R_1 dans les montages ci-dessus.

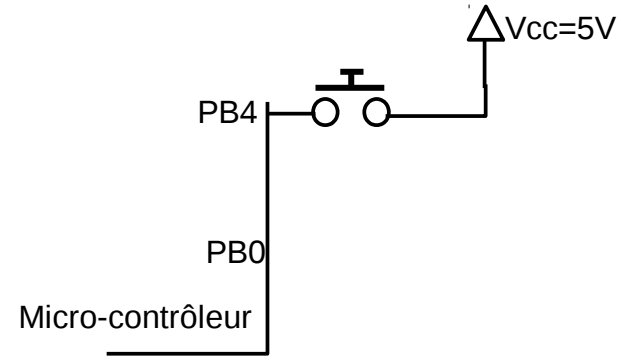
1°) Une led rouge (Kingbright λ = 627nm 15mcd à 10mA V_D = 1,95V) est montée comme à gauche sur le bit b_0 du **PORTB** d'un ATMega8. A l'aide du modèle de Thévenin du PORT, on vous demande d'évaluer la résistance R_1 à mettre pour avoir un courant ne dépassant pas 5mA.

2°) Une led verte (Kingbright λ=565nm 12mcd à 10mA V_D = 2,09V) est montée comme sur le schéma de gauche (sur le bit b_2 du **PORTB**). Calculer R_1 sans prendre l'approximation de Rth pour avoir un courant qui ne dépasse pas les 4mA.

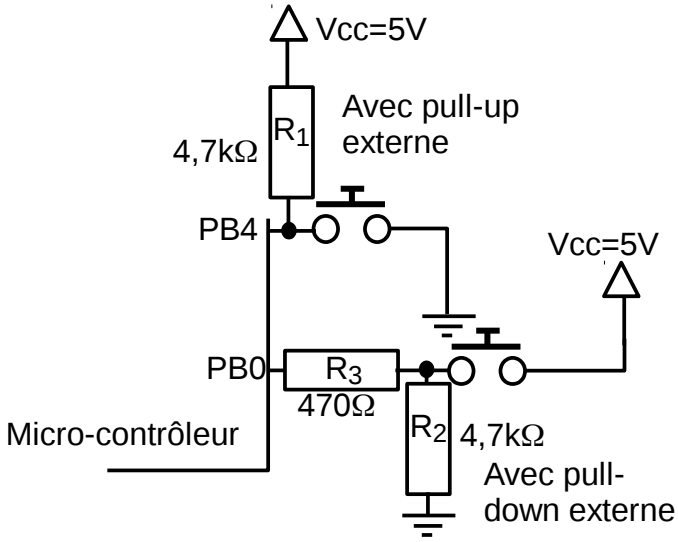
3°) On désire commander deux led rouge en série. Faire le schéma et dimensionner correctement la résistance.

IV.Connecter des boutons poussoirs

Il est naturellement possible de connecter des boutons poussoirs à un PORT et de demander au micro-contrôleur de dire si le bouton est appuyé ou relâché. (Sur un robot mobile on peut utiliser ce principe pour détecter des objets)



Version intuitive mais ne fonctionne pas toujours !



Si votre PORT ne possède pas de résistance de tirage interne, il faut en mettre une à l'extérieur comme sur la figure de droite. L'utilisation sans pull-up du tout comme à gauche est très risqué.

Remarques :

* L'idée qui consiste à croire que puisqu'on est relié à rien, alors on lit systématiquement un zéro logique est parfois une idée fausse !!!

* Le bit PB4 du schéma de droite ne fonctionne pas forcément comme on l'attend : lâché est un un logique, appuyé est un zéro logique !

* Le bit PB0 du schéma de droite fonctionne normalement : appuyé = un logique, relâché = zéro logique

Exercice 2

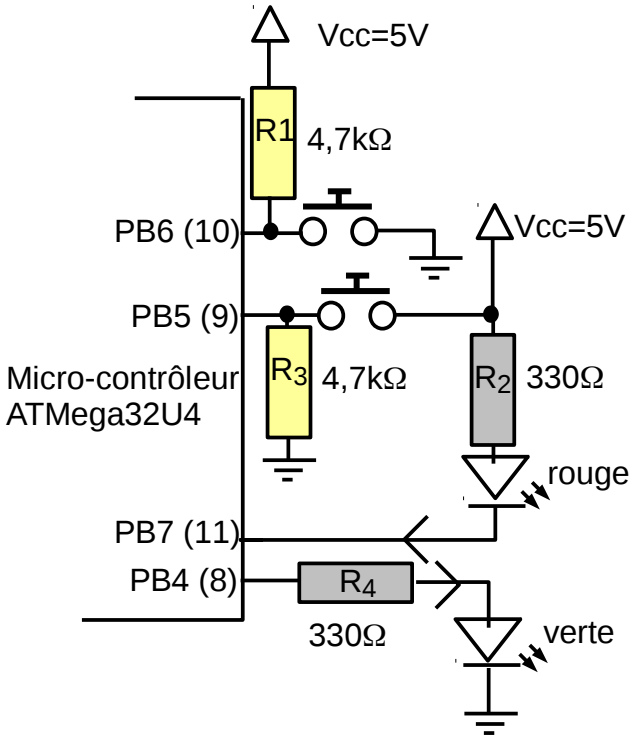
Le schéma de montage se trouve sur la page suivante : deux interrupteurs et deux LEDs.

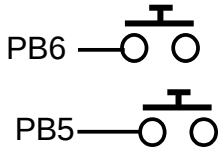
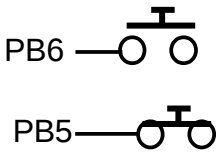
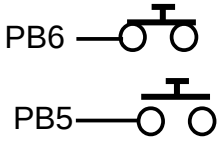
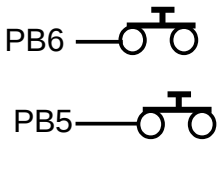
1°) On désire écrire un programme C qui ne fait rien si on n'appuie sur aucun bouton poussoir, fait clignoter la LED rouge si l'on appuie sur un bouton, fait clignoter la led verte si on appuie sur l'autre bouton, et les deux LEDs si l'on appuie sur les deux boutons.

1-a) Donner les 4 valeurs possibles de la variable interrupteurs avec l'instruction

```
interrupteurs = PINB & 0x60;
```

en remplissant en binaire et en hexadécimal ci-dessous.



			
0x	0x	0x	0x
0b	0b	0b	0b

1-b) Compléter alors le morceau de code ci-dessous pour en faire un programme :

```
#undef F_CPU
#define F_CPU 25000000UL
#include "util/delay.h"

leds = 0x00;// variable de gestion des LEDs
// boucle d'attente ATTENTION PB6=1 et PB5=0 au repos

while(interrupteurs == 0x40)
     interrupteurs = PINB & 0x60;
switch(interrupteurs) {
  case 0x60 : leds = leds^0x10;break;
  case 0x00 : leds = leds^0x80;break;
  case 0x20 : leds = leds^0x90;break; // les deux
}
PORTB = leds;
_delay_ms(1000);
```

2°) Peut-on modifier facilement le programme pour que les deux LEDs fonctionnent à deux fréquences différentes ?

Exercice 3

On désire réaliser un dé comme indiqué sur le schéma en page suivante.

1°) Calculer les sept valeurs prédéfinies pour allumer correctement le dé et les ranger dans un tableau si la première case du tableau éteint tout.

Quel est l'affichage du dé correspondant à :

```
unsigned char tableau[7]={0x00,0x10,0x44,0x54,0xAA,0xBA,0xEE};
```

2°) Écrire un programme complet qui affiche les 6 valeurs du dé avec un délai d'attente de 3s utilisant la fonction "_delay_ms()" de la librairie libc.

3°) On désire maintenant générer de manière aléatoire le score affiché sur le dé.

Pour cela on vous propose d'utiliser la fonction ci-contre.

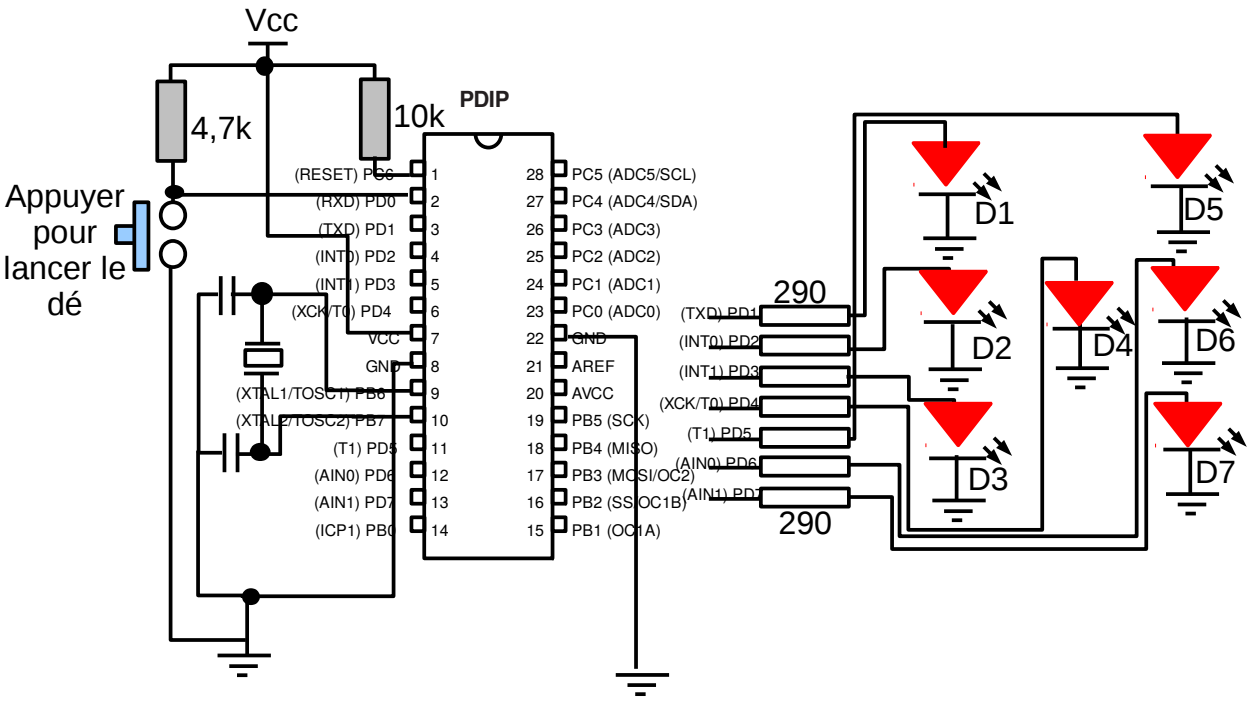
3-a) Calculer la première valeur de cette fonction si Lim=6.

3-b) Écrire le programme général qui génère les valeurs du dé de manière aléatoire dès l'appui du bouton poussoir (attente de 0,3s pour éviter les rebonds).

```
unsigned char pseudoAleat(int Lim) {
  unsigned char Result;
  static unsigned int Y=1;
  Y = (Y * 32719 + 3) % 32749;
  Result = ((Y % Lim)+1);//+1 : eviter 0
  return Result;
}
```

4°) Proposer un schéma pour réaliser deux dés en utilisant un seul PORT de sortie sur 8 bits. Écrire le programme correspondant.

L'astuce consiste à regrouper les leds qui s'allument ensembles (pour un dé) sur un même bit du PORT : il faut ainsi 4 bits par dé. Pour cela le bouton de lancer doit donc être relié à PB0.



Pour simplifier la lecture du schéma : D1 est relié à PD1, D2 à PD2, D3 à PD3, D4 à PD4, D5 à PD5, D6 à PD6 et enfin D7 à PD7.

V. Interfacer un clavier

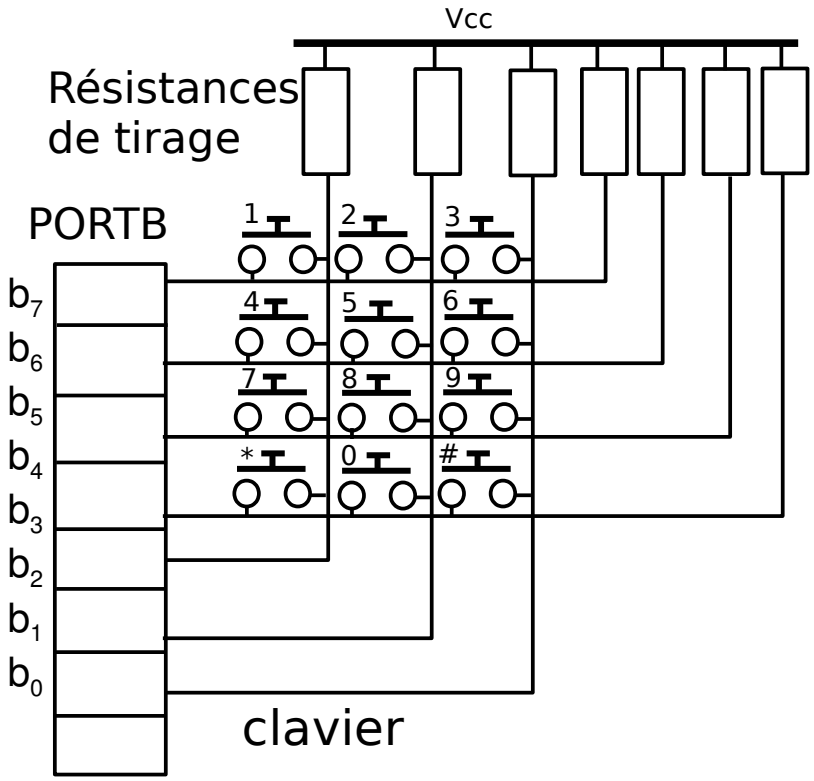
Sur un PC, le clavier est complètement décodé. C'est à dire que lorsqu'une touche est appuyée, sa position sur le clavier est envoyée sur la liaison PS2. Le fait d'envoyer la position et non le code ASCII permet de gérer les claviers en divers langues.

Pour de petites applications, on utilise un clavier à 12 touches. Il est composé de simples contacts et le décodage est réalisé par le système informatique. Avec seulement 8 touches, un PORT de 8 bits en entrée suffit. Si le clavier possède plus de 8 touches, il faut:

- soit utiliser d'avantage d'entrées,

- soit multiplexer les entrées en deux étapes.

En utilisant 4 fils pour les lignes et 4 fils pour les colonnes, on peut différencier par croisement 16 touches. On utilise donc 8 fils reliés à 8 bits d'un PORT pour 16 touches. Pour nos 12 touches on peut câbler comme indiqué ci-dessus. Il s'agit ensuite de procéder en deux phases, une pour la détection de la colonne et une autre pour la détection de ligne.



Exercice 4

Question 1 : détermination du numéro de colonne

```
#define NOTAKEY -1
// colonne 1 à gauche
char lecture_colonne(){
  char ch;
  DDRB=0xF0; // 1111 0000
  PORTB = 0x00; // B4, B5,B6 et B7  mis à 0 !!! surtout pas 0x0F =>pullup !!!
  ch = PINB & 0x0E; // on ne garde que les bits intéressants B1, B2 et B3
  switch (ch) {
  case 14 : return 0;//aucune touche
  case 6 : return 1;//a gauche
  case 10 : return 2;// au milieu
  case 12 : return 3;//a droite
// si autre cas, deux touches ou autre
  default : return NOTAKEY;
  }
}
```

Commenter en testant les bonnes valeurs du case.

Question 2 : détermination du numéro de ligne

Programmer les directions avec **DDRB** (PB7-PB4 en entrée et PB3-PB1 en sortie).

Quel est le code correspondant : sous-programme `char lecture_ligne()`

Question 3 : détermination du caractère

A partir des deux informations précédentes transformer le numéro de colonne et le numéro de ligne en caractère correspondant sur le clavier : '1' ou '2' ou ... ou '0' ou '#'

Remarque : les bits RB4-RB7 peuvent servir à déclencher une interruption (RB port change Interrupt). Si l'on veut utiliser cette interruption il faudrait câbler notre clavier autrement. Les interruptions seront abordées plus loin.

VI.Pour aller plus loin

To make port a as input with pull-ups enabled and read data from port a

```
DDRA = 0x00;       //make port a as input
PORTA = 0xFF;      //enable all pull-ups
y = PINA;          //read data from port a pins
```

- to make port b as tri stated input

```
DDRB  = 0x00;      //make port b as input
PORTB = 0x00;      //disable pull-ups and make it tri state
```

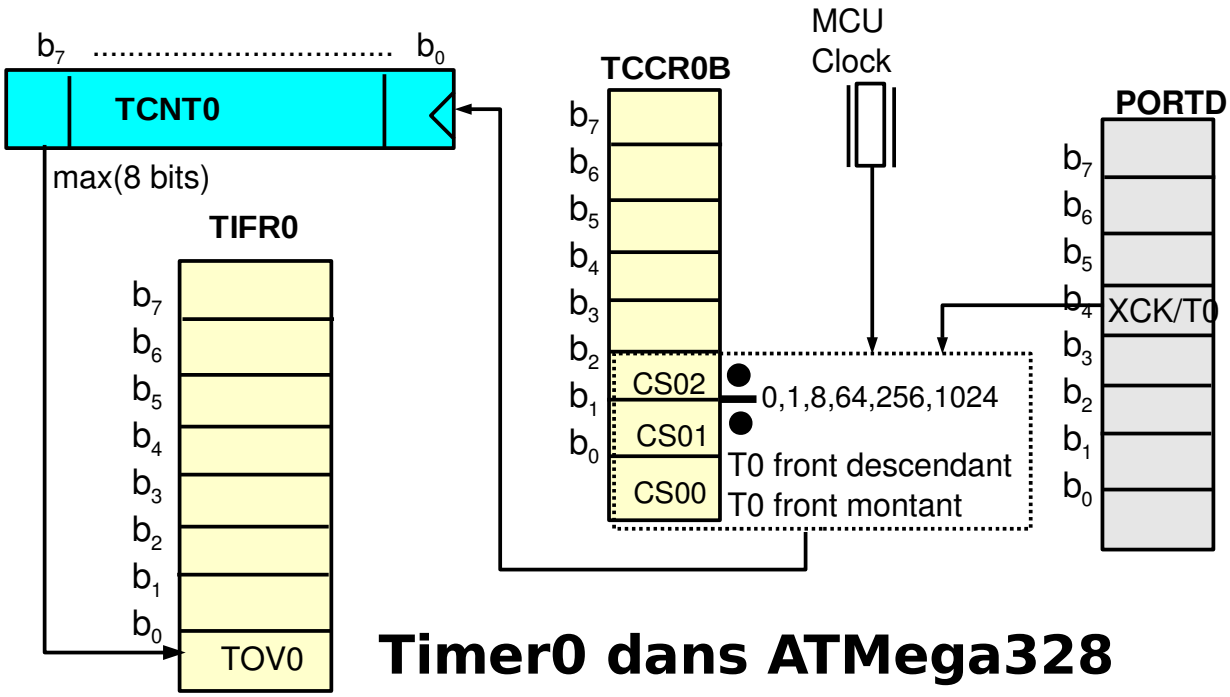
- to make lower nibble of port a as output, higher nibble as input with pull-ups enabled

```
DDRA  = 0x0F;      //lower nib> output, higher nib> input
PORTA = 0xF0;      //lower nib> set output pins to 0,
                              //higher nib> enable pull-ups
```

M2103 - TD 4 Le Timer0 des ATMega328

I. Le Timer 0

La documentation du Timer0 des ATMega328 est présentée maintenant sous forme schématique. Notez que le registre TMR0 est un registre huit bits, dans lequel on peut écrire ou lire (en une seule instruction).



$(CS02,CS01,CS00)_2$ dans l'ordre : 000=Arrêt, /1, /8, /64, /256, /1024, 110=descendant, 111=montant

La version présentée ici est la version simple trouvée dans l'architecture ATMega8 et suivante. Pour ce qui est de l'ATMega8 vous avez l'ensemble des possibilités du TIMER0 présentée (à part les interruptions), mais à partir de l'ATMega16 le TIMER0 est resté sur 8 bits mais permet de faire du PWM (voir en TD suivants).

II. Les différents modes de fonctionnement

Le timer0 est en fait un compteur. Mais qu'allez-vous compter avec ce timer ? Et bien, vous avez deux possibilités :

- En premier lieu, vous pouvez compter les impulsions reçues sur la pin XCK/T0 (RD4). Nous dirons dans ce cas que nous sommes en mode compteur
- Vous pouvez aussi décider de compter les cycles d'horloge du ATMega328® lui-même. Dans ce cas, comme l'horloge est fixe, nous compterons donc en réalité du temps. Donc, nous serons en mode « timer ».

Ses modes de fonctionnement sont choisis à l'aide des trois bits de poids faibles du registre **TCCR0B**.

TOV0 est un bit qui passe à 1 au dépassement de capacité. Il est mis à 0 par le programmeur ou automatiquement par une interruption. Son effacement par le programmeur nécessite l'écriture d'un '1' logique !!!!

III. Mesure du temps d'exécution d'un algorithme

L'optimisation d'un algorithme en vitesse (ou en taille) est très importante dans les systèmes embarqués réalisés par des micro-contrôleurs. Une recherche d'algorithmes sur Internet vous donnera des résultats qu'il vous faudra évaluer. Par exemples, le site : http://www.piclist.com/techref/language/ccpp/convertbase.htm

vous propose un algorithme de division par 10 que voici :

```
unsigned int A;
unsigned int Q; /* the quotient */
        Q = ((A >> 1) + A) >> 1; /* Q = A*0.11 */
        Q = ((Q >> 4) + Q)     ; /* Q = A*0.110011 */
        Q = ((Q >> 8) + Q) >> 3; /* Q = A*0.00011001100110011 */
        /* either Q = A/10 or Q+1 = A/10 for all A < 534,890 */
```

Exercice 1

1°) Sans chercher à comprendre l'algorithme de division, on vous demande de le transformer en une fonction `unsigned int div10(unsigned int A);`

2°) Écrire un programme complet qui mesure le temps d'exécution du sous programme de division par 10, puis modifier le programme pour qu'il puisse comparer avec une division par 10 normale.

IV. Le mode de scrutation du flag

Nous devons savoir à ce niveau, que tout débordement du timer0 (passage de 0xFF à 0x00) entraîne le positionnement du flag TOV0, bit b_0 du registre **TIFR0**. Vous pouvez donc utiliser ce flag pour déterminer si vous avez eu débordement du timer0, ou, en d'autres termes, si le temps programmé est écoulé. Cette méthode à l'inconvénient de vous faire perdre du temps inutilement dans une boucle d'attente.

Petit exemple :

```
while ((TIFR0 & 0x01) == 0); //attente passive
```

Exercice 2

Le quartz est choisi à 4MHz dans ce problème.

Question 1

On donne le programme suivant concernant le timer 0 :

```
1       int main(void){
2       // initialisation du timer   division par 8
3               TCCR0B = 0x02; // prescaler 8 , entrée sur quartz
4               TCNT0 = 0x00; //   tmr0   : début du comptage dans 2 cycles
5       // bit RB0 du PORTB en sortie
6               DDRB |= 0x01; //RB0 as output
7               while(1) {
```

```
8                       TIFR0 |= 0x01; // clr TOV0 with 1
9                       while ((TIFR0 & (1<<TOV0)) == 0);
10                      PORTB ^= 0x01; // on bascule avec ou exclusif
11              }
12      return 0;
13      }
```

Pouvez-vous donner la fréquence d'oscillation du bit b_0 du **PORTB** avec quelques explications ?

Question 2

Écrire en langage C un programme qui fait la même chose que le programme ci-dessus : initialise le timer0, efface le flag et attend à l'aide d'une boucle le positionnement de ce dernier mais 100 incrémentations seulement. On vous demande aussi d'utiliser au mieux les déclarations du fichier d'inclusion avr/io.h :

```
1       /* TIFR */
2       #define OCF2  7
3       #define TOV2  6
4       #define ICF1  5
5       #define OCF1A 4
6       #define OCF1B 3
7       #define TOV1  2
8       /* bit 1 reserved (OCF0?) */
9       #define TOV0  0
10      //***** plein de lignes ici mais cachées *****
11      /* TCCR0 */
12      /* bits 7-3 reserved */
13      #define CS02  2
14      #define CS01  1
15      #define CS00  0
```

L'utilisation des valeurs prédéfinies permet de rendre vos programmes un peu plus lisibles.

Question 3

Générer un signal de fréquence 1 KHz. Pour cela :

- calculer la valeur de la pré division
- calculer la valeur de décomptage
- Écrire le programme.

Question 4

Générer un signal de sortie de rapport cyclique 1/4 sur le même principe.

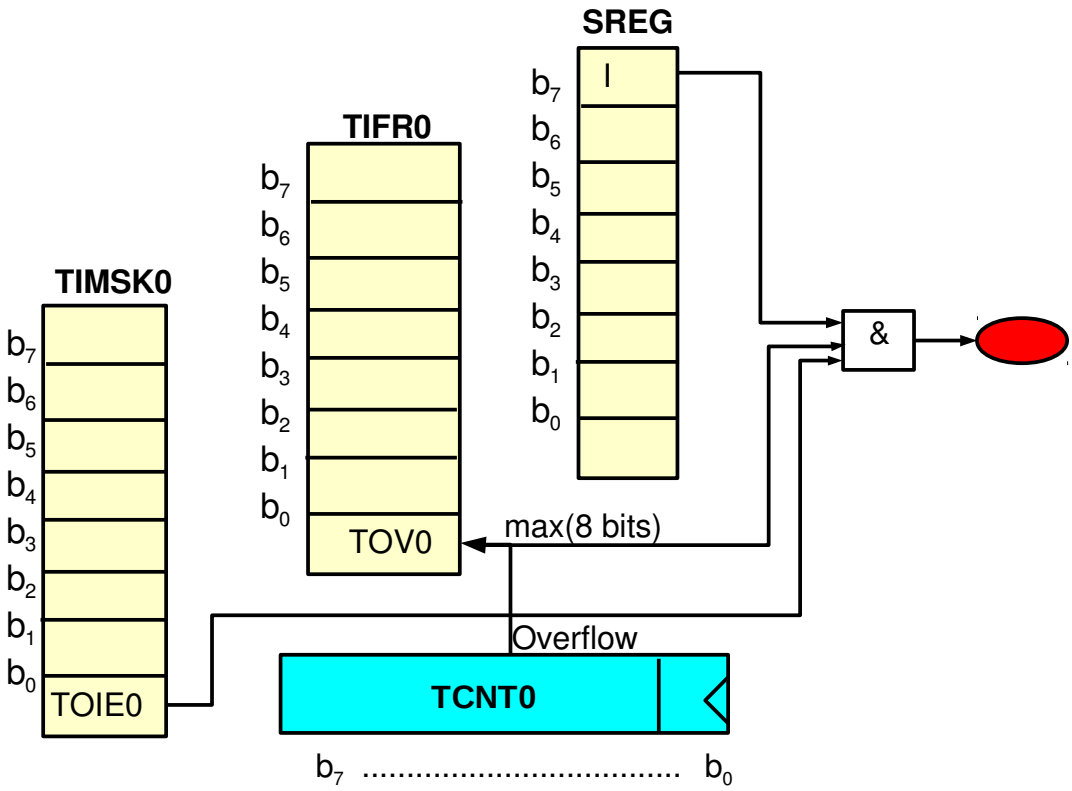
Il y a mieux à faire avec les AVRs, utiliser le module CCP (détaillé plus loin).

M2103 - TD 5 : Interruption timer0

I. Le timer 0

La documentation du Timer0 de l'AVR® ATMega328 a été présentée dans le TD4. Nous passons donc à la description de son interruption associée.

Interruption timer0 de l'ATMega328



Ce schéma de principe fonctionne de la manière suivante. Il y aura interruption si on arrive à réaliser un front montant dans l'ellipse, sachant que le matériel ne gère que l'overflow (bit TOV0 du registre **TIFR0**).

II.Les interruptions en C

Il est maintenant grand temps de parler un peu plus du bit TOV0.

- un dépassement de capacité positionne ce bit à 1
- si une interruption est déclenchée elle mettra automatiquement ce bit à 0
- si aucune interruption ce sera au programmeur de le positionner à '0' en y écrivant un '1' !!!!

La gestion des interruptions est dépendante des compilateurs.

avr-gcc

```
#include <avr/io.h>
#include <avr/interrupt.h>
// compteur
volatile unsigned char cpt=0;
// Fonction de traitement Timer 0 OverFlow
ISR(TIMER0_OVF_vect){
      cpt++;
      if(cpt==61) {
            PORTB ^=(1<<PB5);
            cpt=0;
      }
}
int main(){
// IT Timer0 Over Flow Active
 TIMSK0=(1<<TOIE0);
// Prescaler 1024 (Clock/1024)
  TCCR0B = (1<<CS02) | (1<<CS00);
//Configuration PORTB.5 en sortie
  DDRB |= (1<<DDB5);
  PORTB &= ~(1<<PB5); // PORTB.5 <-0
//activation des IT (SREG.7=1)
  sei();
  while(1);
  return 0;
}
```

Avec le compilateur gcc l'interruption est fixé à l'aide d'un paramètre déclaré dans "interrupt.h".

L'écriture d'une interruption se fait normalement en trois étapes :

- spécifier que le sous-programme n'est pas un sous-programme normal, mais un sous-programme d'interruption et commençant par ISR,

- choisir l'adresse du sous-programme d'interruption (ceci est réalisé de manière automatique)

- initialiser le mécanisme d'interruption dans le programme principal.

Pour notre compilateurs (gcc) les deux premières étapes se font simplement en donnant un nom prédéterminé "ISR" et un paramètre spécifié (ici "`TIMER0_OVF_vect`") au sous-programme d'interruption.

But what volatile keyword means in C or C++ code? This is an indicator (called qualifier) to compiler that tells that this variable may be changed during program flow even if it doesn't look like to be. This means that compiler must treat this value seriously and keep optimizer away from it.

Exercice 1

Un ATMega328 est enfoui dans un FPGA. Sa seule particularité est de fonctionner à 25 MHz contre 20 MHz de fréquence maximale d'horloge pour celui du commerce. Il exécute le programme suivant trouvé sur Internet (compilé avec le compilateur gcc) :

```
/*********************************************************************
Includes
*********************************************************************/
#include <avr/io.h>
#include <stdbool.h>
#include <avr/interrupt.h>
volatile unsigned char nb=0,vPORTB=1;
/*********************************************************************
Interrupt Routine
*********************************************************************/
// timer0 overflow
ISR(TIMER0_OVF_vect) {
  nb++;
  if (!(nb % 16))
    vPORTB = (vPORTB << 1);
  if (vPORTB == 0x00)   vPORTB = 0x01;
  PORTB = vPORTB;
}
/*********************************************************************
Main
*********************************************************************/
int main( void ) {
// Configure PORTB as output
    DDRB = 0xFF; // PORTB en sortie
    PORTB = 0xFF;
// enable timer overflow interrupt for both Timer0
    TIMSK0=(1<<TOIE0);
// set timer0 counter initial value to 0
    TCNT0=0x00;
// start timer0 with 1024 prescaler
    TCCR0B |= ((1<<CS02) | (1<<CS00));
    TCCR0B &= ~(1<<CS01); // pour être sûr
// enable general interrupts
    sei();
    while(true) { // grace a stdbool.h
    }
    return 0;
}
```

1°) Calculer si le chenillard réalisé par ce programme est visible à l'œil humain (fréquence de changement de position des LEDs inférieure à 20 Hz).

2°) Comment peut-on écrire l'instruction "`if (!(nb % 16))`" pour plus d'efficacité.

3°) Quelle est la suite des états (LEDs allumées) réalisée par ce programme.

Pour les deux questions suivantes ce ne sera pas la routine d'interruption qui sera chargée de mettre à jour le PORTB.

4°) Le programme suivant est donné et tourne dans un ATMega8 cadencé avec un quartz de 4 MHz.

```
volatile unsigned int cnt;
ISR(TIMER0_OVF_vect) {
    cnt++;                  // increment counter
    TCNT0   = 96;
  }
}
```

```
int main() {
  unsigned char  vPORTB=0x00;
  TCCR0B |= ((1<<CS01) | (1<<CS00));  // Assign prescaler to TMR0
  TCCR0B &= ~(1<<CS02);
  DDRB = 0xFF;              // PORTB is output
  PORTB = 0xFF;              // Initialize PORTB
  TCNT0  = 96;               // Timer0 initial value
  TIMSK0=(1<<TOIE0);          // Enable TMR0 interrupt
  sei();
  cnt = 0;                   // Initialize cnt
  do {
    if (cnt >= 400) {
      PORTB = ~PORTB;       // Toggle PORTB LEDs
      cnt = 0;               // Reset cnt
    }
  } while(1); return 0;
}
```

Quelle est la fréquence de clignotement des LEDs reliées au PORTB ?

5°) Modifier le programme principal pour réaliser un chenillard d'une LED se déplaçant vers les poids faibles en gardant le traitement en dehors de l'interruption.

Exercice 2

Une partie matérielle est constituée de deux afficheurs sept segments multiplexés. Les sept segments sont commandés par le **PORTC**, tandis que les commandes d'affichages sont réalisée par les bits b_0 et b_1 du **PORTB**. Un schéma de principe est donné ci-après.

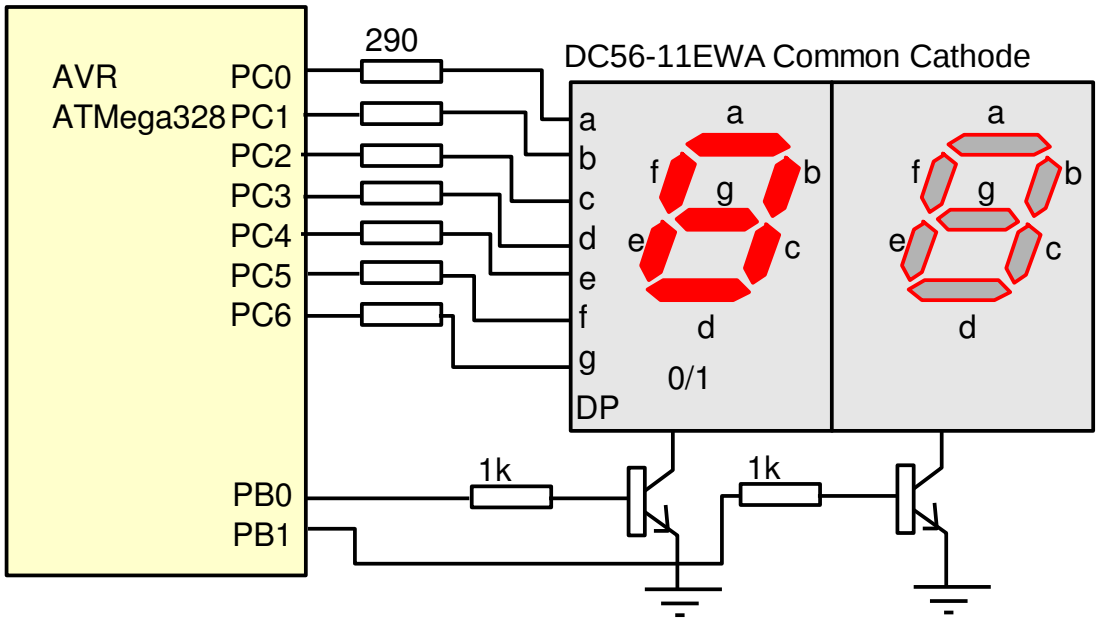
1°) A l'aide de la documentation calculer les valeurs dans un tableau "`unsigned char SEGMENT[] = {0x3F,...};`" pour un affichage des chiffres de 0 à 9.

2°) réaliser une fonction responsable du transcodage :

```
unsigned char Display(unsigned char no) {
    unsigned char Pattern;
    unsigned char SEGMENT[] = {0x3F,....
```

3°) Réaliser le programme main() responsable de l'initialisation de l'interruption qui doit avoir lieu toutes les 10ms (avec un quartz de 4MHz) et qui compte de 00 à 99 toutes les secondes environ (avec un "_delay_ms(1000);")

4°) Réaliser enfin l'interruption qui affichera tantôt les dizaines, tantôt les unités.



Indications: n'oubliez pas de mettre à jour l'entête qui permet de faire fonctionner "_delay_ms" correctement.

```
#undef F_CPU
#define F_CPU 16000000UL
#include "util/delay.h"
```

Autre idée d'exercice (http://maxembedded.com/2011/06/24/avr-timers-timer0-2/)

Exercice 3

La platine Arduino MEGA2560 est programmée en C par le programme suivant :

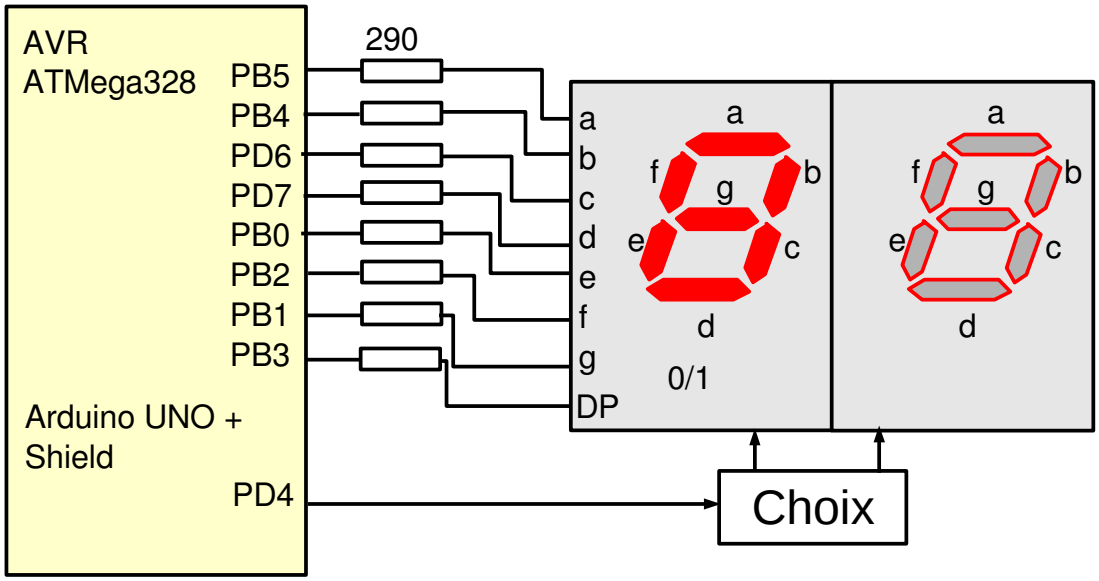
```
#include <avr/io.h>
#include <avr/interrupt.h>
// compteur
volatile unsigned char cpt=0;

// Fonction de traitement Timer 0 OverFlow
ISR(TIMER0_OVF_vect){
      cpt++;
      if(cpt==31) {
            PORTB ^=(1<<PB4);
            cpt=0;
      }
}

void main(){
// IT Timer0 Over Flow Active
 TIMSK0=(1<<TOIE0);
// Prescaler 1024 (Clock/1024)
```

```
19          TCCR0B = (1<<CS02) | (1<<CS00);
20        //Configuration PORTB.4 en sortie
21          DDRB |= (1<<DDB4);
22          PORTB &= ~(1<<PB4); // PORTB.4 <-0
23        //activation des IT (SREG.7=1)
24        //  sei();
25          SREG |= 0x80; // equivalent à sei()
26          while(1);
27          return 0;
28        }
```

1°) A quelle fréquence le bit b4 du PORTB oscille-t-il ?

2°) La documentation ci-dessus pour le processeur ATMega328p est-elle conforme à celle du processeur ATMega2560 ?

III.Shield Arduino IUT Troyes



M2103 TD 6 : Le mode "sortie par comparaison" du TIMER 0

Cette fonctionnalité n'existait pas dans l'ATMega8. Elle a été ajoutée pour gérer le PWM (Modulation de largeur d'impulsion = MLI et Pulse Width Modulation en anglais). Ce PWM sert essentiellement à commander des moteurs de Robots mais éventuellement à moduler des éclairages.

Les fonctionnalités ajoutées sont difficiles à appréhender. En effet le mode comparaison permet de gérer différents modes pour pouvoir réaliser toute une gamme de signaux du carré au PWM. Ceci complique un peu la gestion pour le programmeur. La bonne nouvelle c'est que les autres timers ont un fonctionnement un peu similaire.

I. Documentation de la comparaison

Les différents modes de comparaison sont choisis à l'aide des bits WGM02, WGM01 et WGM00. Ces choix sont conformes au tableau suivant.

Description des bits pour la génération de forme d'onde

Mode	WGM02	WGM01	WGM00	Mode de fonctionnement	Bas si	Mise à jour de OCRAx si	Drapeau TOV0 positionné si
0	0	0	0	Normal	0xFF	immédiatement	MAX
1	0	0	1	PWM phase correcte	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCR0A	immédiatement	MAX
3	0	1	1	PWM rapide	0xFF	BOTTOM	MAX
4	1	0	0	Réservé			
5	1	0	1	PWM phase correcte	OCR0A	TOP	BOTTOM
6	1	1	0	Réservé			
7	1	1	1	PWM rapide	OCR0A	BOTTOM	TOP

Remarque :

* MAX = 0xFF

* BOTTOM = 0x00

Pour chacun des modes de ce tableau ci-dessus, les bits COM0A1 et COM0A0 auront un fonctionnement différent. Ces bits sont destinés à gérer les formes d'onde du signal de sortie.

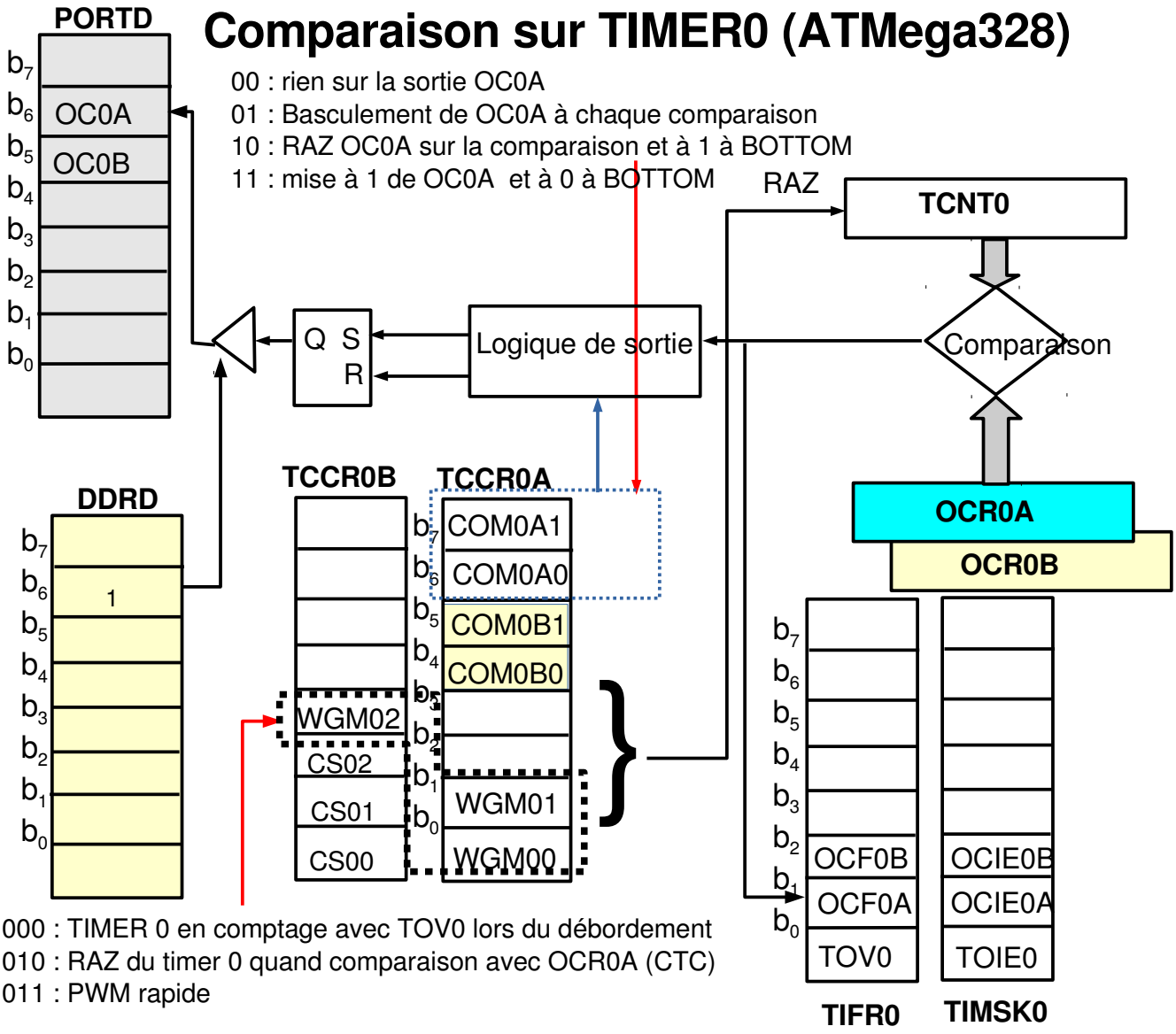
Comparaison simple

Mode non PWM pour la comparaison

COM0A1	COM0A0	Description
0	0	Opération Normale PORT, OC0A déconnecté

0	1	Bascule OC0A sur la comparaison
1	0	Mise à 0 de OC0A sur la comparaison, mise à 1 sur overflow
1	1	Mise à 1 de OC0A sur la comparaison, mise à 0 sur overflow

Et voici donc la documentation correspondante :



En résumé, le mode CTC s'utilise de la manière suivante :

- mise à un de **WGM01, WGM02** et **WGM00** sont supposés à 0
- choix de **COM0A1** et **COM0A0** pour la logique de sortie
- choix du préscaler pour le démarrage du timer 0

Sans interruption, seul le mode "basculement du bit **OC0A**" a un intérêt mais il impose d'utiliser ce bit (qui est le bit b_6 du **PORTD**).

L'interruption de comparaison peut servir à utiliser un bit de sortie quelconque. Elle n'est pas documentée mais le sera à travers un exercice (exercice 1).

Exercice 1

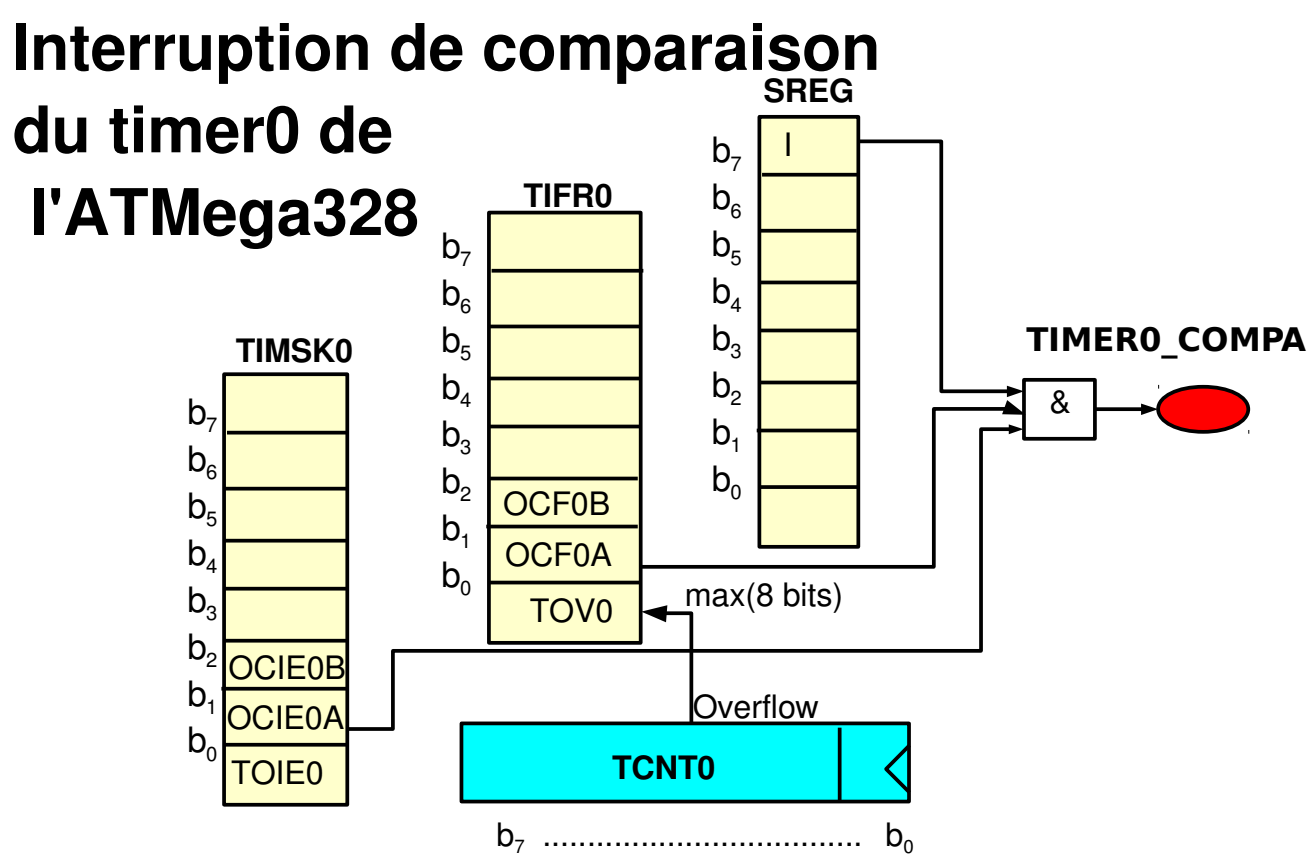
1°) Donner le squelette d'un programme qui utilise le mode CTC pour réaliser un signal de période 8 ms sur le bit **OC0A**. La fréquence du quartz sera de 16MHz comme sur la carte Arduino UNO.

2°) Dessiner sur un chronogramme le comptage du timer et le signal généré sur **OC0A**.

3°) Estimer la fréquence la plus basse que l'on peut réaliser sur le bit **OC0A** ?

4°) Estimer la fréquence la plus haute que l'on peut réaliser sur le bit **OC0A** ?

II.Comparaison et interruption



Exercice 2

Modifier l'exemple ci-dessous trouvé sur Internet pour réaliser un clignotement d'un Hertz sur une LED connectée sur le bit b4 du '''PORTB'''.

```
1         // this code sets up a timer0 for 4ms @ 16Mhz clock cycle
2         // an interrupt is triggered each time the interval occurs.
3         #include <avr/io.h>
4         #include <avr/interrupt.h>
5         int main(void){
6             // Set the Timer Mode to CTC
7             TCCR0A |= (1 << WGM01);
8             // Set the value that you want to count to
9             OCR0A = 0xF9;
10            TIMSK0 |= (1 << OCIE0A);    //Set the ISR COMPA vect
11            sei();         //enable interrupts
12            // set prescaler to 256 and start the timer
13            TCCR0B |= (1 << CS02);
```

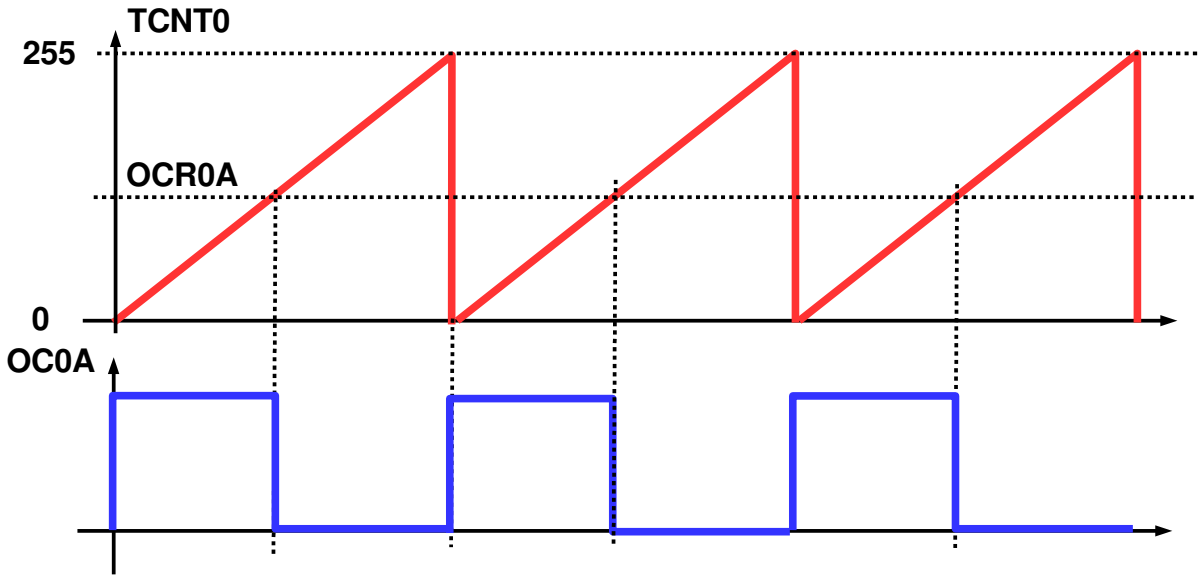
```
14              while (1) {
15                  //main loop
16              }
17              return 0;
18          }
19
20          ISR (TIMER0_COMPA_vect)  // timer0 overflow interrupt
21          {
22              //event to be exicuted every 4ms here
23          }
```

1°) Quelle est la fréquence réelle de l'exécution du code de l'interruption que vous pouvez calculer à partir des documentations fournies ? Expliquez la différence.

2°) On désire réaliser un signal d'un Hertz sur PD4. Réaliser les modifications du code ci-dessus sans changer les valeurs du préscaler et du registre **0CR0A** autrement dit en réalisant une division que l'on calculera dans l'interruption.

III.Mode PWM

Il existe deux modes de PWM : PWM rapide et PWM à phase correcte. Seul le premier mode sera utilisé.



Mode PWM rapide

Mode PWM rapide et comparaison

COM0A1	COM0A0	Description
0	0	Opération Normale PORT, OC0A déconnecté
0	1	WGM02=0 Opération Normale PORT, OC0A déconnecté
0	1	WGM02=1 Basculement de OC0A sur la comparaison
1	0	Mise à 0 de OC0A sur la comparaison et à 1 à BOTTOM
1	1	Mise à 1 de OC0A sur la comparaison et à 0 à BOTTOM

Il y a une subtile différence entre ces bits pour la comparaison et PWM mais ne nous concerne pas car on choisira le mode surligné avec WGM02 =0 et WGM01=WGM00=1

Exercice 3 : PWM rapide

On désire changer l'intensité d'éclairage d'une LED à l'aide d'une PWM rapide. La valeur du rapport cyclique varie entre 0 et 255 et sera systématiquement envoyée par la liaison série sous forme de deux caractères 0,...,9,A,...,F.

1°) Un sous-programme sera donc chargé de lire ces deux caractères d'en vérifier la syntaxe. Écrire ce sous-programme et le tester avec les afficheurs de l'exercice du TD précédent. On affichera "--" en cas d'erreur de syntaxe.

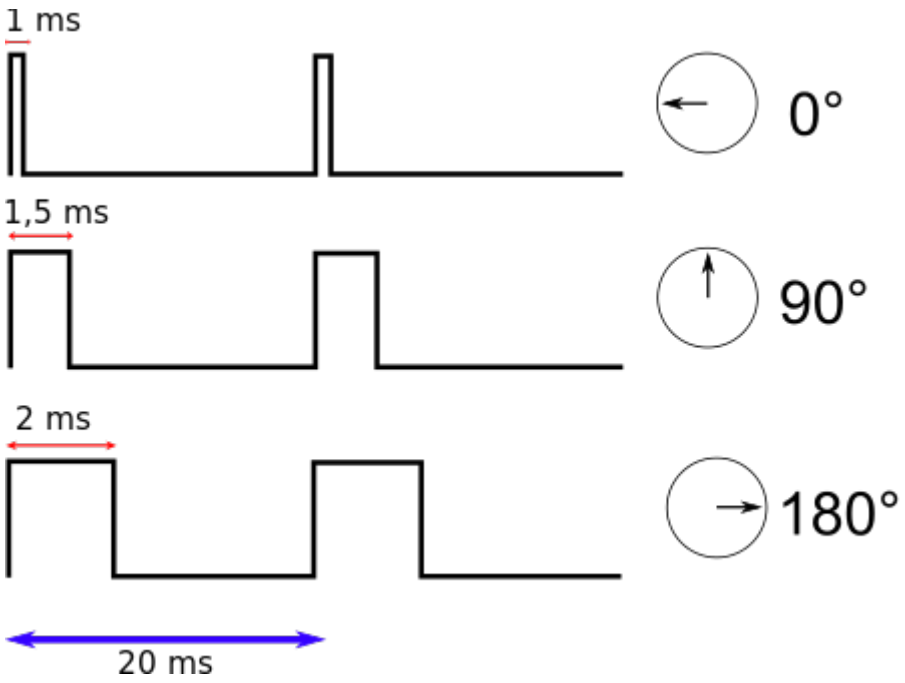
2°) Changer votre programme de test précédent pour qu'il réalise un rapport cyclique sur le bit OC0A du PORTB.

Exercice 4 : PWM rapide et servomoteur

Réaliser une commande d'un servomoteur à l'aide du timer 0.

Indications :

- La fréquence de 50 Hz (période T=20 ms) n'est pas importante pour la commande des servomoteurs.
- Le rapport cyclique par contre a besoin de varier entre 5% et 10%.
- Un calcul simple montre donc que **OCR0A** doit varier entre 255/10 et 255/20 ce qui donnera donc 26 et 13.
- On utilisera la fréquence minimale du Timer0.
- **OC0A** est le bit PD6 du PORTD. Sur une carte Arduino, il s'agit de la broche **-6**



M2103 TD 7 : La conversion analogique numérique

I. La conversion analogique numérique dans le monde Arduino

L'environnement Arduino possède une primitive simple d'utilisation : analogRead. Elle retourne un nombre sur 10 bits puisque les convertisseurs sont des convertisseurs 10 bits. Cela veut dire qu'ils sont capables de retourner une valeur entre 0 et 1023, valeur représentant une tension entre 0 et 5V. Par exemple le programme

```
1         // programme d'exemple du TP capteurs M1103
2         void setup()
3         {
4           Serial.begin(9600);
5         }
6
7         void loop()
8         {
9           Serial.print("Photocoupleur : ");
10          Serial.println(analogRead(A2),DEC);
11          delay(500);
12        }
```

enverra dans la liaison série la valeur lue sur l'entrée repérée par A2.

II.Documentation de la conversion analogique numérique

La documentation est donnée sous forme de schéma en page suivante. Une tension de référence est nécessaire pour réaliser une conversion car celle-ci est basée sur une comparaison. Elle est de 1,1V dans les versions ATMagaXX8 mais de 2,56V dans les ATMega8/16/32. AVCC peut être utilisé comme référence, c'est la tension d'alimentation.

Exercice 1

Une tension de référence de 2,56V est utilisée comme référence pour un convertisseur d'un ATMega8 sur 10 bits.

1°) Quelle est la résolution en tension de ce convertisseur ?

2°) A quelle tenson correspond le nombre 0x12F ?

3°) Une tension de 2V est présente sur l'entrée. Quelle sera la valeur de la conversion ?

4°) Un thermomètre LM35C peut mesurer une température entre -40°C et +110°C avec une précision de 1,5°C et une résolution de 10mV/°C. Quelle nombre retournera le convertisseur pour une température de 45°C ?

Il est grand temps d'expliquer le fonctionnement de la conversion.

Pour le registre **ADMUX**, le dessin est autosuffisant. Notez quand même les différentes possibilités sur les choix de la référence et de l'entrée convertie. Les

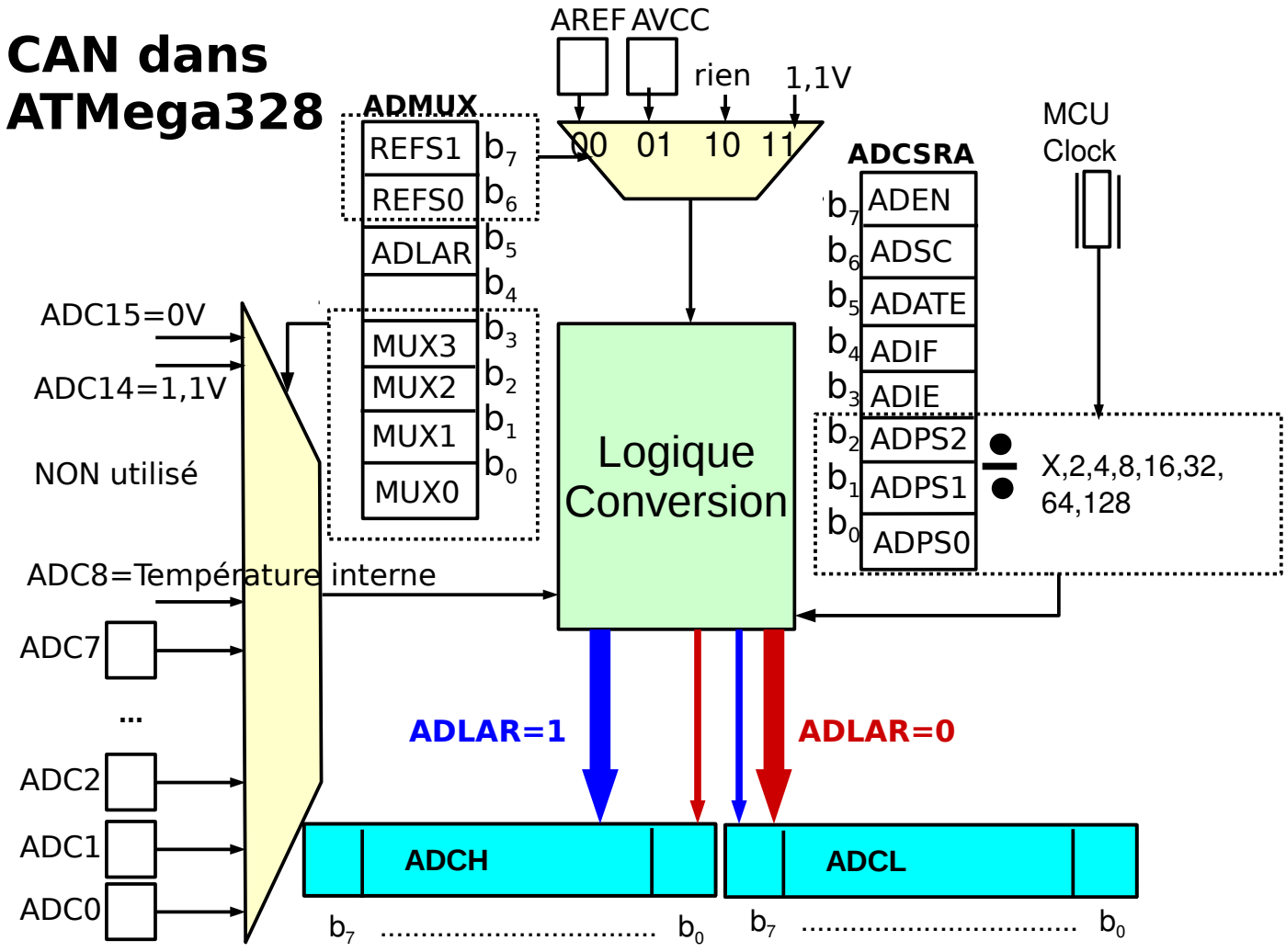
entrées AREF et AVCC existent et doivent être connectées à la masse par l'intermédiaire d'une capacité de 100nF. AVCC est en plus connectée à VCC.

Pour le registre **ADCSRA**, **ADEN** autorise la conversion tandis que **ADSC** la fait démarrer (SC=Start Conversion). Ce bit est à 1 pendant toute la durée de conversion et repasse à 0 lorsque celle-ci est terminée.

ADATE relève du déclenchement automatique. On le mettra systématiquement à 0. Il est lié au registre **ADCSRB** que nous n'étudierons pas.

ADIF est le bit qui est positionné à 1 quand la conversion est terminée. Il peut être lié à une interruption en positionnant **ADIE** à 1. L'interruption effacera le bit ADIF. Si aucune interruption n'est utilisée, il faut effacer **ADIF** en écrivant un 1 dedans.

La division d'horloge doit être choisie pour que la conversion soit réalisée à 200kHz au maximum.



La formule magique qui permet de calculer la conversion est :

$ADC = \frac{V_i}{V_{REF}} x 1023$ où ADC est un nombre entier sur 10 bits.

Exercice 2

Donner le morceau de programme qui lance la conversion et attend la fin de conversion. Deux techniques sont à explorer :

- une avec le bit ADSC

- une avec le bit ADIF (qui est un flag et nécessite une attention particulière)

Exercice 3

Pouvez-vous expliquer le result=ADCH du programme ci-dessous ainsi que chacun des commentaires. Y a-t-il erreur dans un commentaire ?

```
1         #include <avr/io.h>
2         int main() {
3                 unsigned char result;
4                 // Choose AREF pin for the comparison voltage
5                 //   (it is assumed AREF is connected to the +5V supply)
6                 // Choose channel 3 in the multiplexer
7                 // Left align the result
8                 ADMUX = (1 << REFS0) | (1 << ADLAR) | (3);
9                 // Start the ADC unit,
10                // set the conversion cycle 16 times slower than the duty cycle
11                ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADSC);
12                // Wait for the measuring process to finish
13                while (ADCSRA & (1 << ADSC)) continue;
14                // Read the 8-bit value
15                result = ADCH;
16        }
```

Conversion Analogique Numérique avec interruption

Voici un exemple d'utilisation de la conversion analogique/numérique et son interruption associée.

```
1         #include <avr/io.h>
2         #include <avr/interrupt.h>
3         ISR(ADC_vect) {
4           PORTD = ADCL;
5           PORTB = ADCH;
6           ADCSRA |= (1<<ADSC);
7         }
8         int main() {
9           unsigned char result;
10          DDRB = 0xFF;
11          DDRD = 0xFF;
12          DDRA = 0; // make port A an input for ADC
13          ADCSRA = 0x8F; //enable interrupt select clk/128
14          ADMUX = 0xC0; //2,56 Vref and ADC0
15          sei();
16          ADCSRA |= (1 << ADSC); //start conversion
17          while (1); // wait forever
18          return 0;
19        }
```

Nous rappelons que l'interruption est déclenchée seulement à la fin de la conversion.

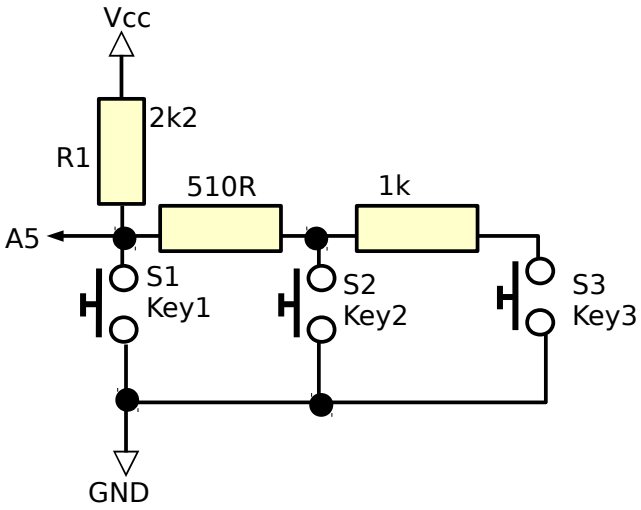
III.Applications : lecture de plusieurs interrupteurs

L'utilisation d'un bit de PORT par interrupteur devient vite très consommatrice de broches du composant. Nous allons étudier un moyen simple d'éviter cette augmentation.

Voici un premier schéma d'exemple :

Exercice 4

Le robot MiniQ 2W (1ere version) possède trois interrupteurs montés sur une entrée analogique A5 comme dans le schéma ci contre.



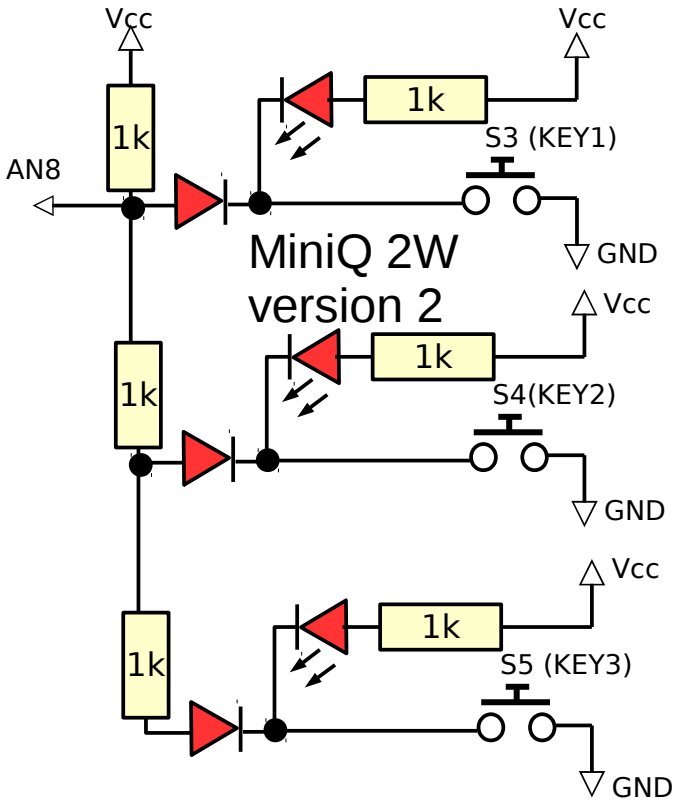
1°) Calculer toutes les tensions possibles sur A5 si Vcc=5V.

2°) Convertir ces tensions en une valeur sur 10 bits si c'est AVCC qui est choisie comme tension de référence.

3°) Écrire une boucle d'attente d'appui d'un bouton si l'on suppose qu'une interruption de fin de conversion met à jour une variable appelée « n » :

```
ISR(ADC_vect)
{
   //Interruption en fin de conversion
   uint16_t n=ADC;
   ....
}
```

4°) Faire le même travail pour le Robot miniQ 2W (version 2) avec le schéma ci-contre.

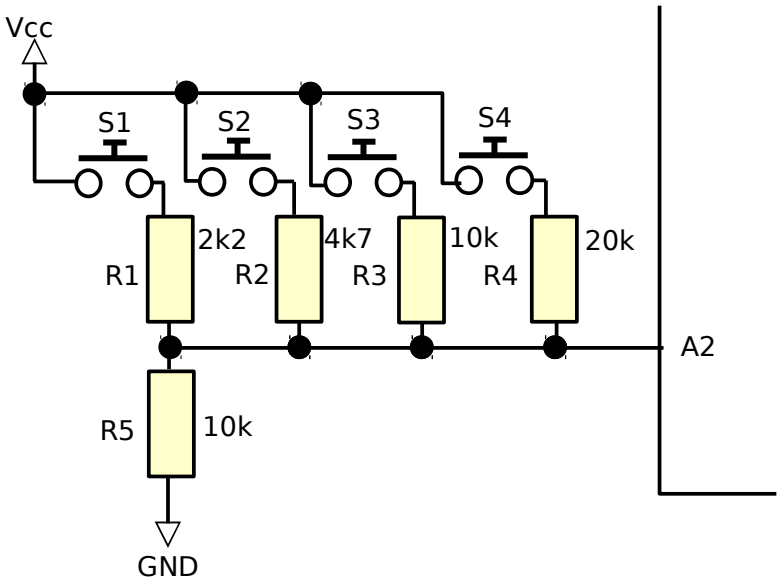


Exercice 5

1°) Trouver les valeurs des tensions en fonction de l'appui sur les boutons poussoirs. On supposera pour ce calcul que Vcc = 5V.

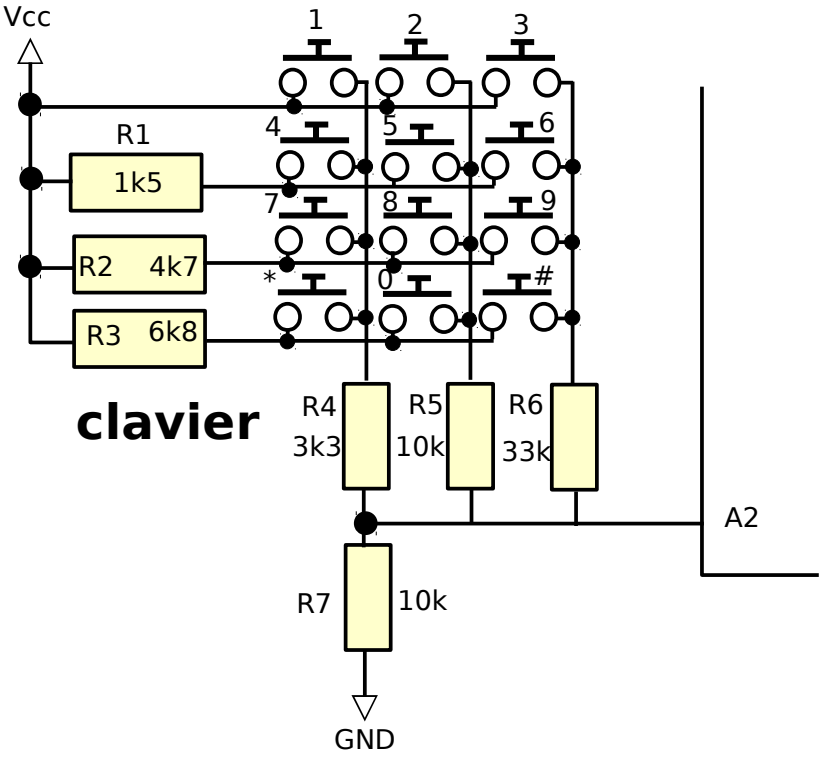
2°) Calculer les ADC correspondants sur 10 bits si V_{REF} = 5V.

3°) Écrire un programme complet capable d'afficher complètement l'état des 4 interrupteurs sous forme binaire.



Exercice 6

On peut faire encore plus que dans le cas de l'exercice 4 en décodant tout un clavier avec une seule entrée.



1°) Calculer dans un tableau les tensions et valeurs 10 bits correspondantes pour l'appui d'une touche du clavier si la tension Vcc est fixée à 5V (ainsi que V_{REF}).

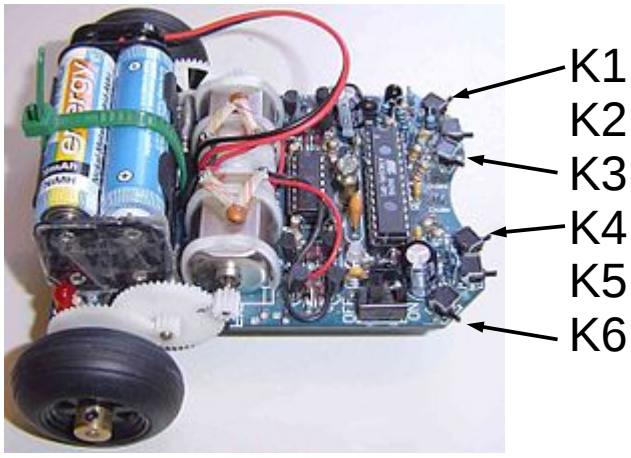
2°) Peut-on prendre en compte l'appui de plusieurs touches ?

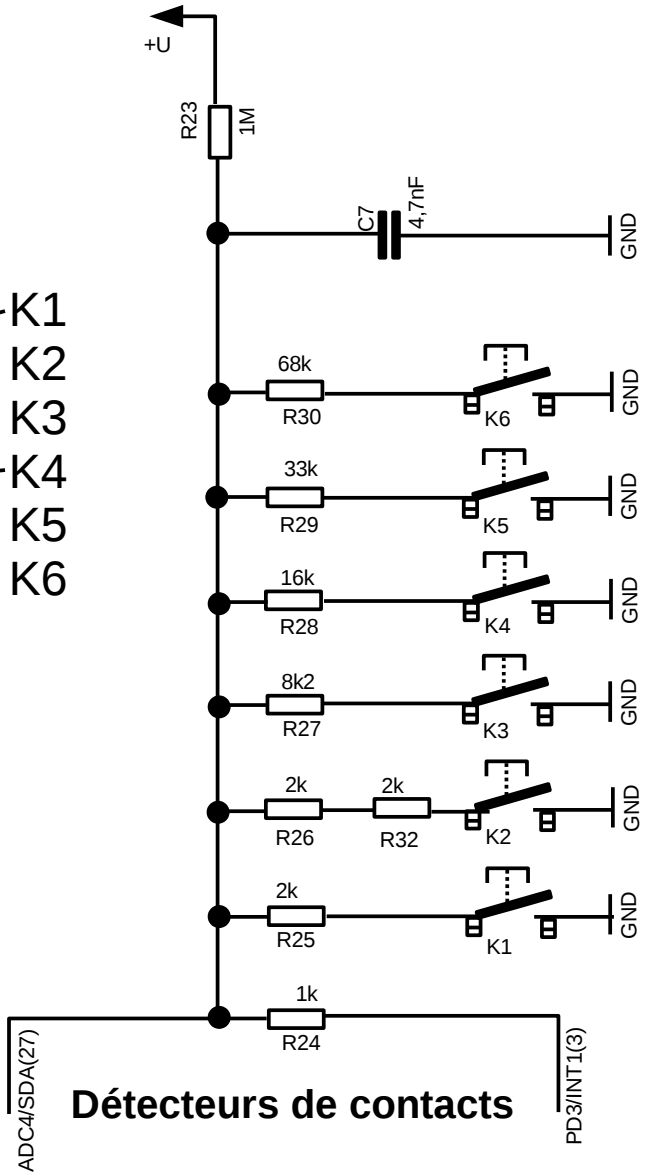
3°) Écrire un sous programme de lecture du clavier qui retourne la touche appuyée. On utilisera obligatoirement un switch.

IV.Détection avec interruption

En robotique mobile, des interrupteurs peuvent être utilisés pour détecter des obstacles. Il faut alors réagir de toute urgence. Une interruption peut être adaptée à cette situation.

Voici un exemple concernant le robot Asuro qui utilise un ATMega8. Sur l'ATMega8 une interruption liée à la conversion analogique numérique existe mais elle n'est déclenchée que lors de la terminaison de la conversion et non lors d'un changement sur une entrée. C'est pour cela que les concepteurs d'Asuro ont utilisé le schéma ci-dessous. L'appui sur un interrupteur sera aussi détecté sur l'entrée PD3 qui peut elle déclencher une interruption.





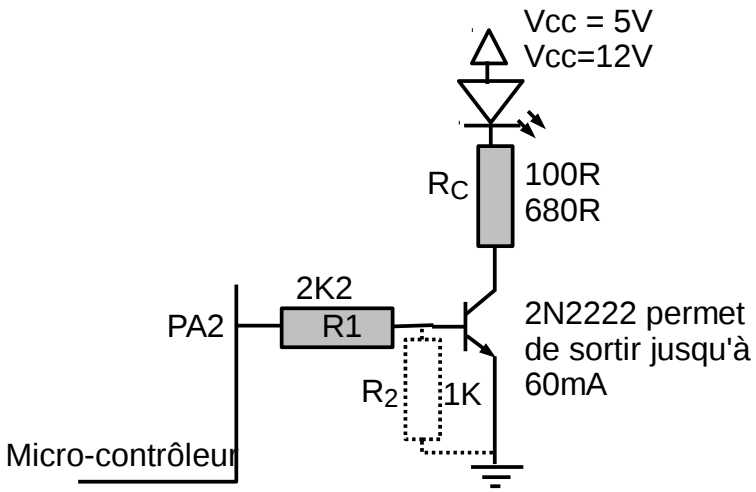
M2103 - TD8 Interfacer des éléments de puissance pour faire tourner des moteurs

Quand le courant à sortir dépasse 15 mA (par bit), il faut trouver une solution amplifiée. C'est ce que nous allons examiner dans ce chapitre.

I. Utilisation de transistors BJT

Pour commander des sorties puissantes il faut ajouter des transistors (BJT Bipolar Junction Transistor). Les sorties puissantes peuvent être :

- des relais
- des bobines
- des LEDs
- des moteurs



Exercice 1

On veut commander une LED qui nécessite 16mA. Le transistor choisi a un β de 100.

1°) Calculer I_B ?

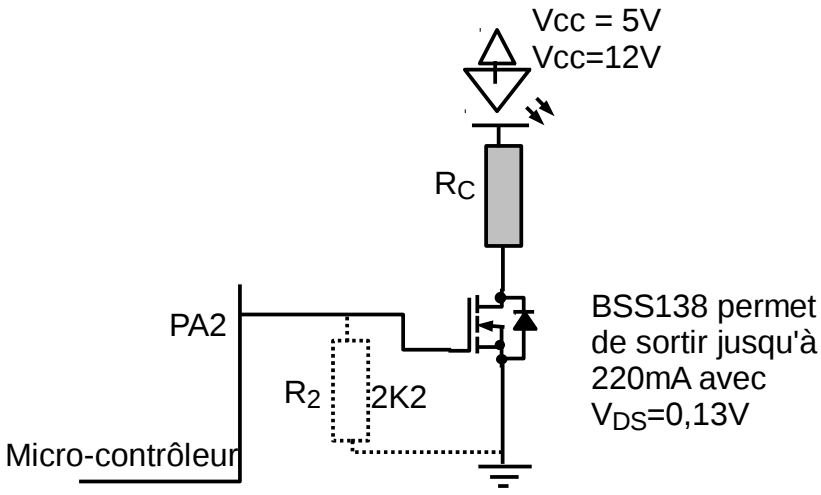
2°) En déduire R1 (si l'on prend une marge de 30% ou si l'on prend 1mA par défaut)

3°) En déduire R_C pour un V_{CEsat}=0,2V (pour une alimentation Vcc=5V) ?

4°) Une résistance R_C de 1/4W suffit-elle ?

II. Interfacer des sorties puissantes avec des FET

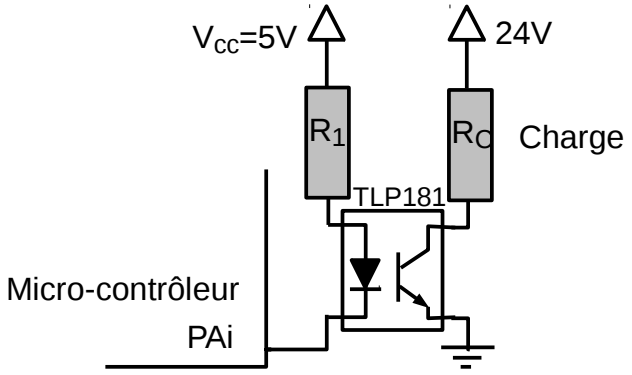
On peut utiliser des transistors à effet de champ à la place des BJT. Nous examinons le cas des MOSFET. Un paramètre important à vérifier est la tension de grille nécessaire à la commutation. En effet pour les micro-contrôleur alimenté en 3,3V cette tension peut s'avérer un peu forte pour des MOSFET de puissance.



Il existe des familles spécialement faites pour être commandées directement par des PORTs. Voir ZVN4206A (600mA) et ZVN4306A (1,1A) de chez Zetex.

III.Encore plus d'isolation électrique

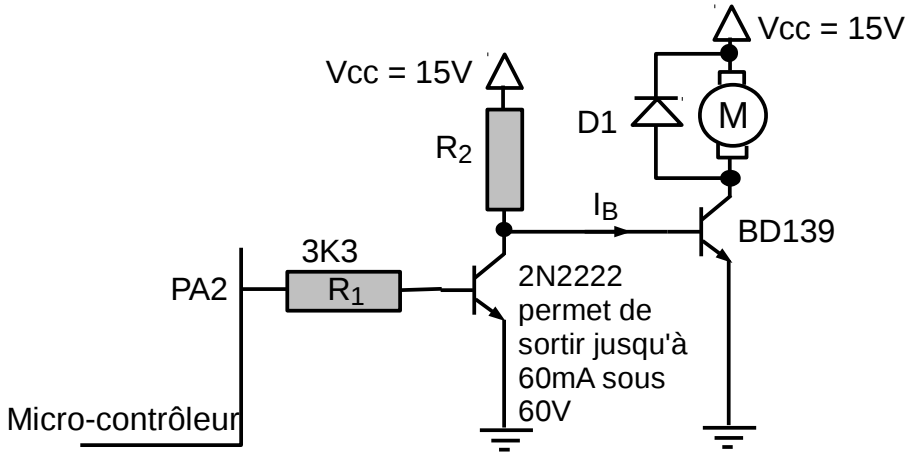
Il faut utiliser des opto-coupleurs. Il permettent aussi une isolation galvanique ce qui est important si la tension d'alimentation est très différente de celle du micro-contrôleur.



TLP181 :
16mA < I_{Dmax} < 20mA avec V_D <1,3V à 10mA
1mA < I_C <10mA
Dimensionner R_1.

IV.Commander des bobines moteurs et relais avec BJT

Il existe des transistors de puissance adaptés. Par exemple le transistor BD139 propose un gain β compris entre 25 et 250. On peut l'utiliser avec un courant I_C=1,5 A (3A en pointe). Son faible gain nécessite un courant I_B assez important, allant au-delà des possibilités des ports traditionnels (typiquement 10mA). Par exemple pour commuter 1A, il faut un I_B=40mA dans le pire des cas. Il faut donc un transistor supplémentaire.



Ce montage appelé Darlington existe en boîtier unique : par exemple le BCX38C peut commuter des courants jusqu'à 800mA.

Exercice 2

Le transistor 2N2222 est bloqué, il circule 40mA dans I_B

1°) Si R_2 est choisi à 330 Ω, quelle puissance passe dans R_2 ?

2°) Si le transistor 2N2222 est saturé (V_{CEsat}=0,2V) quelle puissance passe dans R_2?

3°) Lorsque le BD139 est saturé, il circule IC=1A et l'on a VCEsat=0,70V. Quelle puissance est dissipée dans le BD139 ?

Exercice 3 tiré du DS final de Juin 2011 (Construction d'un signal PWM pour un moteur, **adapté pour AVR**)

Dans tout cet exercice, on dispose d'un quartz générant MCU Clock à 8 MHz. On utilisera le timer 0 documenté dans un TD précédent.

C'est le Timer0 qui va être responsable de la réalisation de la période de notre signal PWM (MLI Modulation de Largeur d'Impulsion) ou à rapport cyclique variable. Cette période est choisie à 10 ms et on rappelle que notre horloge a une fréquence de 8MHz. Le fonctionnement général du programme est le suivant :

- initialise le timer

- initialiser la comparaison

Boucle infinie

- attendre une comparaison et mettre la sortie RC2 à 0

- attendre que le timer1 déclenche un overflow et mettre systématiquement la sortie RC2 à 1

fin boucle

1°) Donner la ou les instructions C permettant de réaliser une division par 256. Quelle période a-t-on alors à l'entrée du Timer0 ? Quelle est la fréquence F_{T0} correspondante ?

2°) Par combien doit-on diviser la fréquence F_{T0} pour obtenir nos T = 10ms ?

On arrondit cette division à 256. Quelle est alors la nouvelle période $T_{arrondie}$?

3°) Si la division de la question 2 est réalisée sur le timer0, avec quelle valeur doit-on initialiser le timer0 (en décimal et en hexadécimal)

4°) Quelle instruction C attend le débordement du timer0 ?

5°) Voir documentation sur la comparaison au TD précédent.

La comparaison est choisie comme : "00 : rien sur la sortie OC0A" car on préfère gérer la sortie correspondante nous même. On choisit d'autre part le mode "0000 : TIMER 1 en comptage avec TOV1 lors du débordement". Quelle instructions C permettent de choisir ce mode ?

6°) Quelles valeurs faut-il mettre dans OCR0A pour réaliser un rapport cyclique de 10% ?

7) Quelle est l'instruction C qui permet d'attendre la comparaison ?

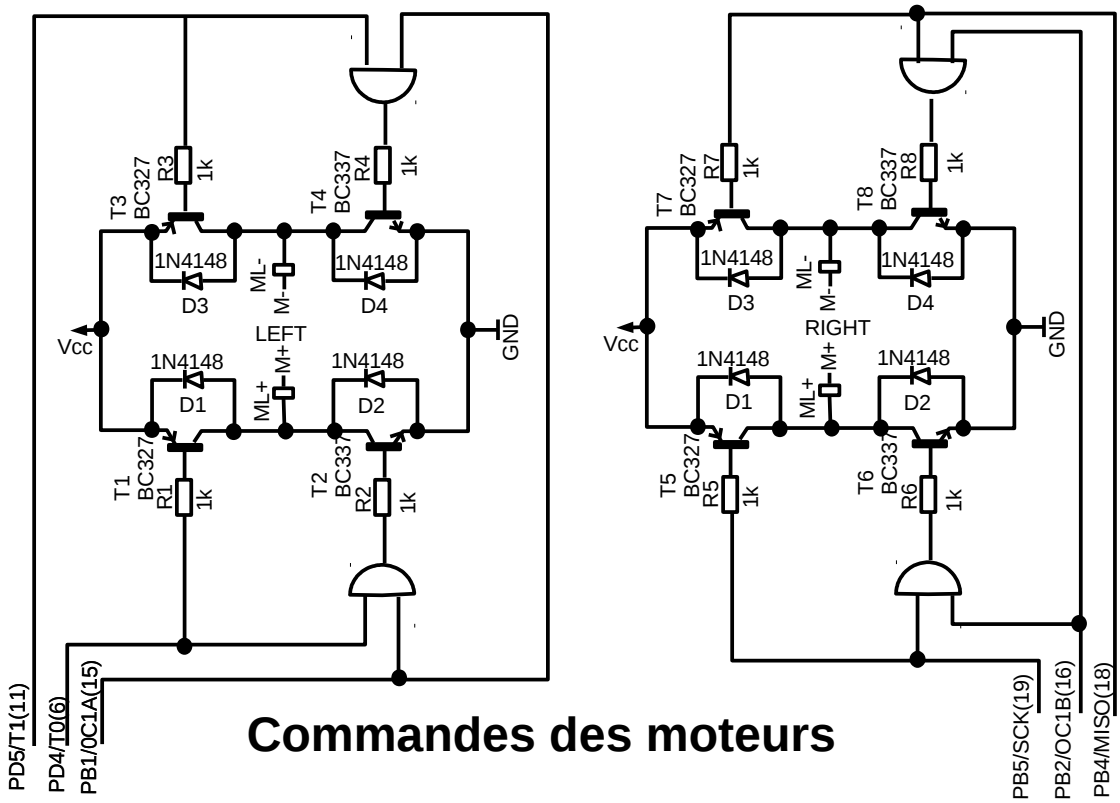
Question hors DS

8°) On utilise le schéma avec un 2N2222 et un BD139 en échangeant PA2 en RC2. Quel courant moyen passe dans le moteur si le dimensionnement des résistances est tel que le maximum de courant est de 1,5 A ?

Exercice 4 (ASURO)

ASURO est un robot mobile différentiel (2 roues propulsées par deux moteurs et une demi-balle de ping pong comme roue (point ?) libre. L'ensemble est commandé par un ATMega8.

Sa commande de propulsion est réalisée par le montage suivant :



1°) Analyse du schéma. Pour les trois entrées PD5, PD4 et PB1 pouvez-vous examiner les 8 possibilités pour la commande du moteur "LEFT" en précisant quelles combinaisons feront des courts-circuits ?

2°) Le sous-programme "MotorState est donné maintenant :

```
1         /**
2          * Motor configuration.
3          * values: FWD, RWD, BREAK, FREE
4          * @param left left motor
5          * @param right right motor
6          */
7         inline void MotorState(unsigned char left, unsigned char right)
8         {
9             PORTD = (PORTD &~ ((1 << PD4) | (1 << PD5))) | left;
10            PORTB = (PORTB &~ ((1 << PB4) | (1 << PB5))) | right;
11        }
```

En vous aidant de la question 1°) pouvez-vous expliquer ce que fait l'instruction

```
12        MotorState(FWD,FWD);
```

si la constante FWD est définie comme :

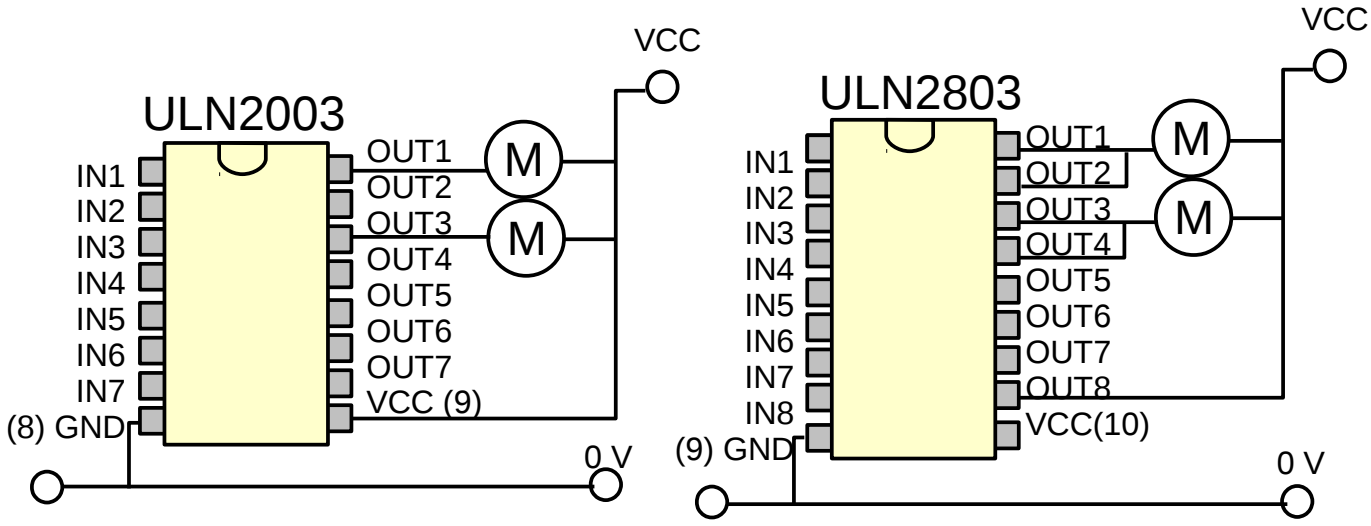
```
13      #define FWD     (1 << PB5) /* (1 << PD5) */
```

Pourquoi RWD est-il défini comme :

```
14      #define RWD     (1 << PB4) /* (1 << PD4) */
```

V. Commander plusieurs moteurs avec un circuit unique

Il existe des circuits capable de commander 7 moteurs comme le ULN2003 qui a à peu près les mêmes caractéristiques que le BCX83C (mais sept fois). Ils sont appelés Darlington driver IC. Le ULN2803 possède 16 broches et peut commander 8 moteurs avec la possibilité d'utiliser deux sorties identiques pour commander un seul moteur.



Le circuit ULN2003 est capable de délivrer 500 mA sous 50V.

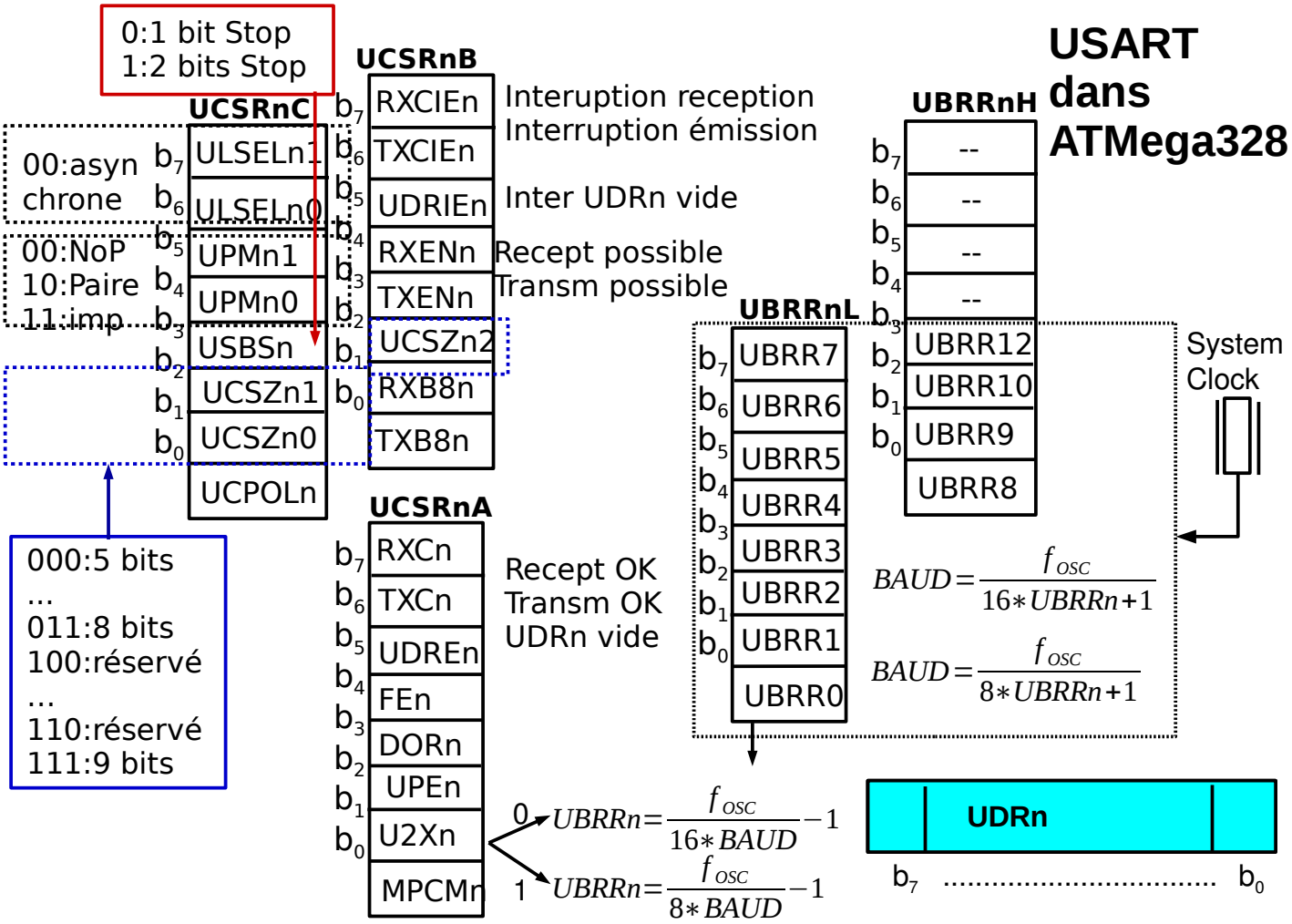
M2103 TD 9 Liaisons série, SPI et i2c

I. Liaison série asynchrone

Elle est souvent appelée USART.

Les registres

Le « n » apparaissant dans ces dessins peut prendre la valeur 0 ou 1 dans l'ATMega328.



Les bits du registre **UCSRnA** sont :

- **RXCn** : réception complète
- **TXCn** : transmission complète
- **UDREn** : USART Data Register Empty (vide)
- **FEn** : Frame Error
- **DORn** : Data OverRun
- **UPEn** : USART Parity Error
- **U2Xn** : est expliqué dans le dessin
- **MPCMn** : Multi Processor Communication Mode

Les bits du registre **UCSRnB** sont :

- **RXCIEn** : interruption quand réception complète autorisée
- **TXCIEn** : interruption quand transmission complète autorisée
- **UDRIEn** : interruption quand UDRn est vide autorisée
- **RXENn** : autorisation de la réception
- **TXENn** : autorisation de la transmission
- **UCSZn2** : taille de caractère quand combiné avec UCSZn1:0
- **RXB8n** : 9° bit de réception au cas où ce mode est choisi
- **TXB8n** : 9° bit de transmission au cas où ce mode est choisi

La relation entre le nombre de bauds et la fréquence de l'oscillateur est :

$$BAUD=\frac{f_{OSC}}{16*UBRRn+1} \quad \text{soit} \quad UBRRn=\frac{f_{OSC}}{16*BAUD}-1 \quad \text{si le bit UCXn=0}$$

$$BAUD=\frac{f_{OSC}}{8*UBRRn+1} \quad \text{soit} \quad UBRRn=\frac{f_{OSC}}{8*BAUD}-1 \quad \text{si le bit UCXn=1}$$

Exercice 1

Soit le code C suivant :

```
1         #include <avr/io.h>
2         #define F_CPU 16000000  // 16 MHz oscillator.
3         #define BaudRate 9600
4         #define MYUBRR (F_CPU / 16 / BaudRate ) - 1
5         void serialInit(void) {
6           //Serial Initialization
7                 /*Set baud rate 9600 */
8                 UBRR0H = (unsigned char)(MYUBRR>>8);
9                 UBRR0L = (unsigned char) MYUBRR;
10                /* Enable receiver and transmitter */
11                UCSR0B = (1<<RXEN0)|(1<<TXEN0);
12                /* Frame format: 8data, No parity, 1stop bit */
13                UCSR0C = (3<<UCSZ00);
14        }
```

1°) Quelle valeur décimale est mise dans **UBRR0** ?

2°) Comment modifier ce code pour une vitesse de transmission de 19200 bauds ?

3°) Comment modifier ce code pour avoir une parité paire ?

4°) Écrire deux sous-programmes, un pour l'émission, un pour la réception.

5°) Écrire un sous-programme de conversion deux caractères vers valeur :

`unsigned char usart_gets_hexa()`

II.Liaison série synchrone :SPI

1°) Description du protocole

Le protocole SPI est synchrone : une horloge synchronise l'échange de données. Dans tout échange on définit un Maître qui réalise l'horloge et un Esclave. L'esclave

(il peut y en avoir plusieurs) est choisi avec une entrée spéciale appelée SS (Slave Select).

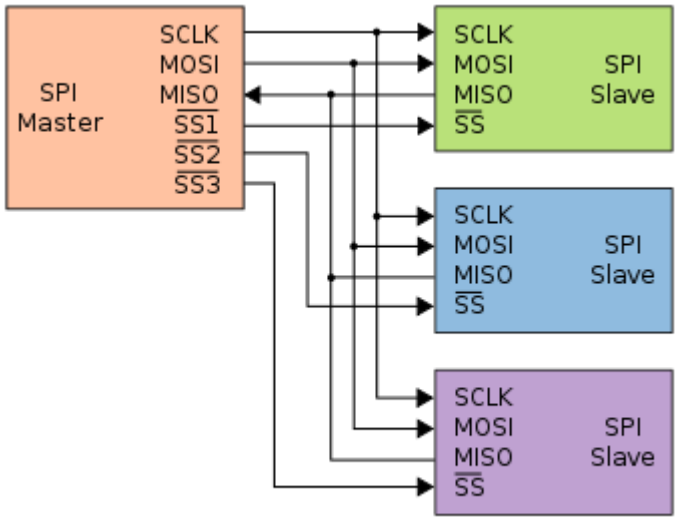
A ce point, nous connaissons deux broches du SPI : l'horloge appelée **SCK** et **SS**. Il en manque deux pour être complet :

- **MISO** : Master in Slave Out

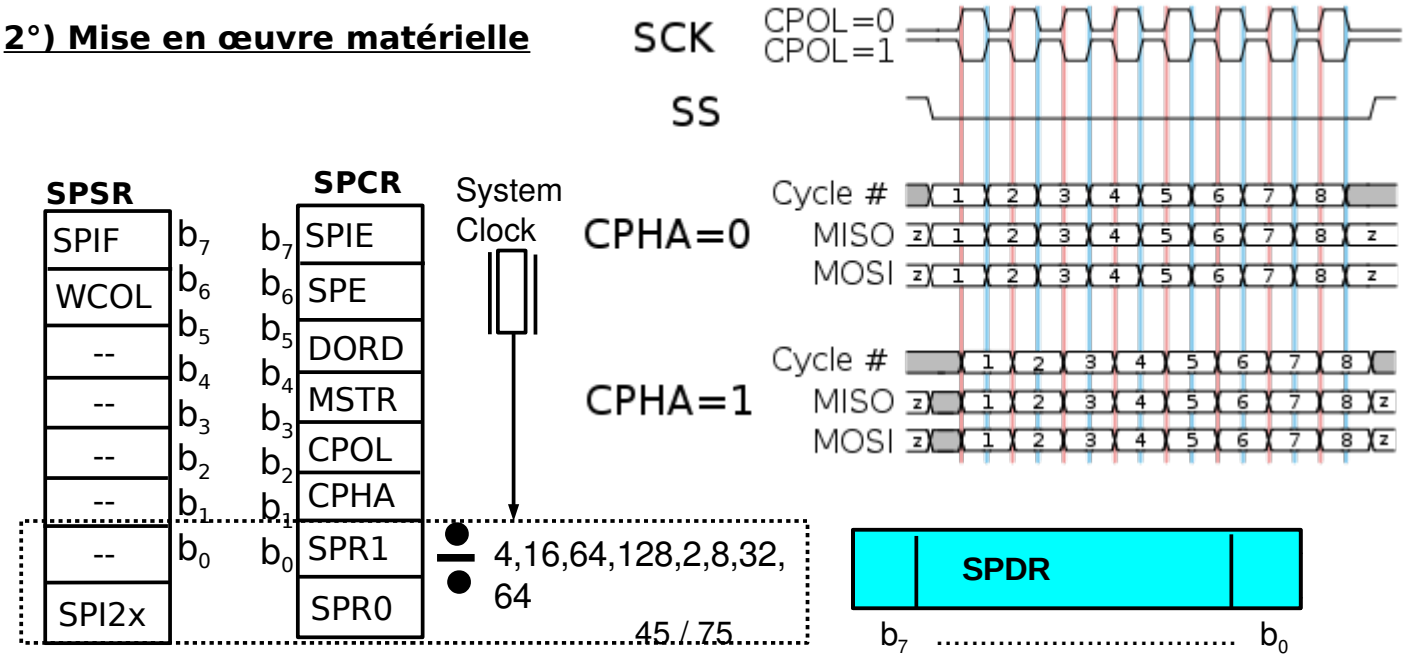
- **MOSI** : Master Out Slave In

Ces deux broches laissent entendre un échange full duplex (complet dans les deux sens).
Voici donc comment les choses se passent avec un Maître et un Esclave :

Si vous avez plusieurs esclaves une configuration de ce genre est à choisir :



2°) Mise en œuvre matérielle



En ce qui concerne le registre **SPCR** :
- **SPIE**: pour autoriser les interruptions SPI
- **SPE**: SPI Enable doit être positionné à 1 pour toute opération SPI
- **DORD**: Data Order , 1 pour une transmissin du poids faible en premier
- **MSTR**: Master/Slave Select , 1 pour le positionnement en maître. Ce bit nécessite d'être en accord avec SS (Slave Select)
- **CPOL**: Clock Polarity , 1 pour SCK haut quand idle, 0 pour SCK bas quand idle.
- **CPHA**: Clock Phase , détermine si les données sont échantillonnées sur le premier front ou le deuxième front de SCK.

Pour le registre **SPSR** :
- **SPIF**: est le drapeau d'interuption
- **WCOL**: Write COLlision Flag

Le reste est donné dans le dessin.

Exercice 2

Le brochage de quelques cartes Arduino pour le SPI est donné maintenant :

SPI	MISO	MOSI	SCK	SS
UNO	PB4 (Arduino:12)	PB3 (Arduino:11)	PB5 (Arduino:13)	PB2 (Arduino:10)
LEONARDO	PB3 (ICSP:1)	PB2 (ICSP:4)	PB1 (ICSP:3)	--aucune--
MEGA2560	PB3 (Arduino:50)	PB2 (Arduino:51)	PB1 (Arduino:52)	PB0 (Arduino:53)
Pro Micro (Sparkfun)	PB3 (Arduino:14)	PB2 (Arduino:16)	PB1 (Arduino:15)	--aucune--

1°) Pouvez-vous modifier le programme ci-dessous trouvé sur Internet pour qu'il fonctionne correctement avec un MEGA2560 avec une horloge SPI égale à la fréquence système divisée par 8.

```
1         SPIMasterInit(void) {
2         //set MOSI, SCK and SS as output
3           DDRB |= (1<<PB3)|(1<<PB5)|(1<<PB2);
4         //set SS to high
5           PORTB |= (1<<PB2);
6         //enable master SPI at clock rate Fck/16
7           SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
8         }
```

2°) Quel est le mode de fonctionnement correspondant ?

3°) Étant pointilleux sur les principes, on vous demande de modifier aussi le sous-programme de réception, lui aussi trouvé sur Internet, sachant que **SPIF** est un drapeau comme son nom l'indique.

```
//master send function
void SPIMasterSend(uint8_t data){
```

```
//select slave
  PORTB &= ~(1<<PB2);
//send data
  SPDR=data;
//wait for transmition complete
  while (!(SPSR &(1<<SPIF)));
//SS to high
  PORTB |= (1<<PB2);
}
```

III.Liaison i2C

La liaison i2c est utilisée pour une communication entre circuits électroniques. C'est une liaison assez lente mais suffisamment rapide pour un grand nombre d'applications. Elle est caractérisée par un maître (mais plusieurs sont possibles) qui est responsable de la réalisation de l'horloge (signal appelé **SCL** dans la suite). La fréquence par défaut utilisée est 100 kHz mais 400kHz sont possibles. Les données sont transmises ou reçues par un signal appelé **SDA**.

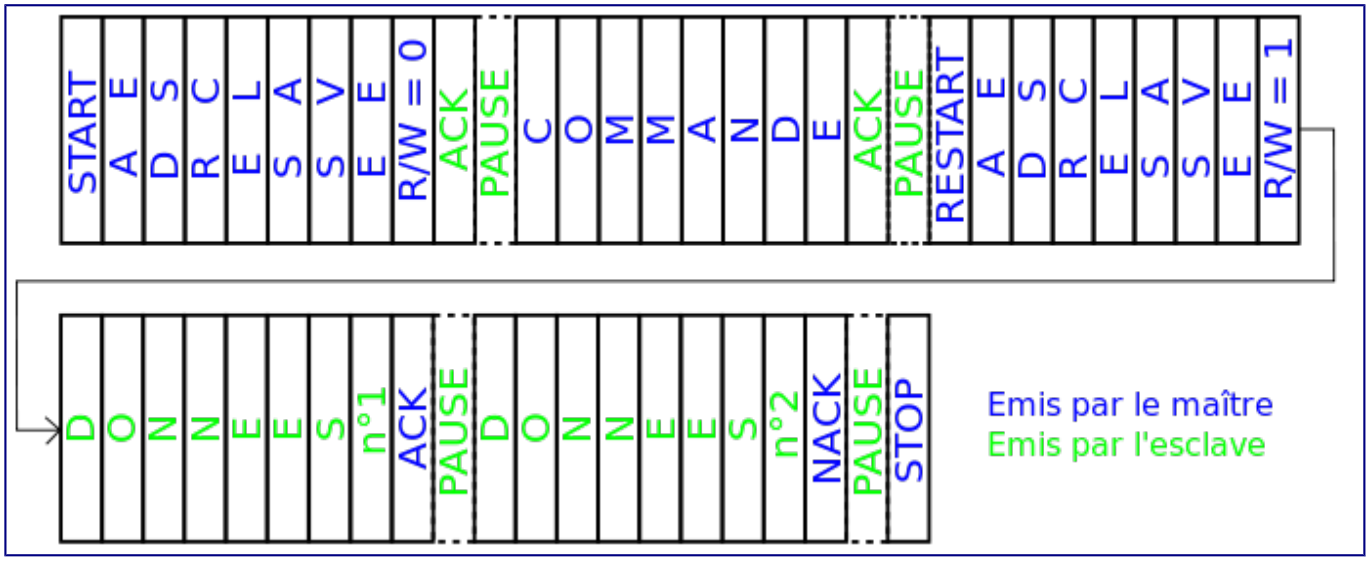
Comme il est possible de connecter plusieurs esclaves, ces derniers sont caractérisés par une adresse sur 7 bits (qui peut être étendue à 10 bits).

Échanges Maître-Esclave

Le protocole entre un maître et un esclave utilise une technique d'acquittement.

Le message peut être décomposé en 2 parties :

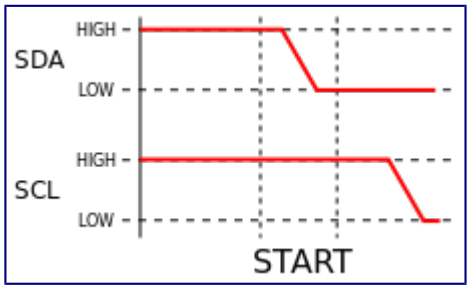
- Le maître est l'émetteur, l'esclave est le récepteur :
  - émission d'une condition de START par le maître (« S »),
  - émission de l'octet ou des octets d'adresse par le maître pour désigner un esclave, avec le bit R/W à 0 (voir la partie sur l'adressage ci-après),
  - réponse de l'esclave par un bit d'acquittement ACK (ou de non-acquittement NACK),
  - après chaque acquittement, l'esclave peut demander une pause (« PA »).
  - émission d'un octet de commande par le maître pour l'esclave,
  - réponse de l'esclave par un bit d'acquittement ACK (ou de non-acquittement NACK),
  - émission d'une condition de RESTART par le maître (« RS »),
  - émission de l'octet ou des octets d'adresse par le maître pour désigner le même esclave, avec le bit R/W à 1.
- Le maître devient récepteur, l'esclave devient émetteur :
  - émission d'un octet de données par l'esclave pour le maître,
  - réponse du maître par un bit d'acquittement ACK (ou de non-acquittement NACK),
  - (émission d'autres octets de données par l'esclave avec acquittement du maître),
  - pour le dernier octet de données attendu par le maître, il répond par un NACK pour mettre fin au dialogue,
  - émission d'une condition de STOP par le maître (« P »).



Condition de START

La condition de START est une transgression de la règle de codage des bits qui est utilisée par le maître pour signifier le début d'une trame.

Cette condition est caractérisée par le passage de la ligne SDA du niveau « HIGH » au niveau « LOW » pendant que la ligne « SCL » est maintenue au niveau « HIGH ».



Octet d'adressage

Chaque esclave doit avoir une adresse unique.

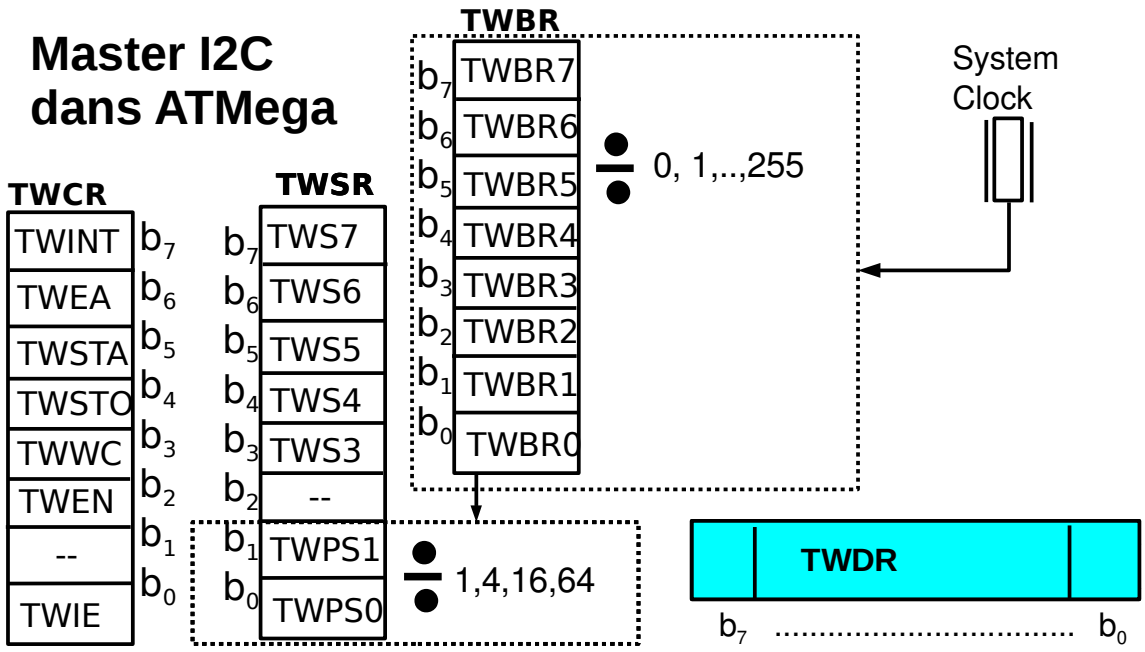
Nous allons nous intéresser à la liaison i2c prenant l'AVR comme maître

Les registres de l'AVR pour configurer l'i2c en maître

Le premier registre important est **TWBR** qui est utilisé pour régler la fréquence d'horloge de la broche SCL. Il faut ajouter au contenu de ce registre les deux bits **TWPS1** et **TWPS0** du registre **TWSR** qui sont destiné à réaliser un préscaler avec les valeurs 1, 4, 16 et 64.

Les documentations officielles donnent la formule de calcul de la fréquence :

$$f_{SCL} = \frac{1}{16 + 2 * TWBR * prescaler}$$



Comme toujours, il existe un registre de contrôle **TWCR** qui a un ensemble de bits utilisés pour :

- autoriser l'interruption TWI (**TWIE**)
- déconnecter virtuellement l'i2c (**TWEA**=0)
- autoriser TWI (**TWEN**)
- départ (**TWSTA**)
- stop (**TWSTO**)

Le bit **TWINT** a un comportement spécial qu'il nous faut mentionner (c'est probablement pour cela que les ingénieurs ne l'ont pas appelé TWIF!!!). Il se comporte comme un flag sauf pour sa mise à 0 ! Elle se fait par l'écriture d'un 1 (comme tout flag) mais une interruption ne le met pas automatiquement à 0 contrairement à flag !

Le registre d'état déjà mentionné **TWSR** contient les bits de réglage du présscaler. Mais son utilisation principale est de donner l'état du bus I2C avec les bits TWS[7:3]. **TWDR** est le registre de données qui est utilisé pour mémoriser le prochain octet à transmettre ou l'octet reçu.

Les registres **TWAR** et **TWARM** sont utilisés quand l'AVR travaille en mode esclave (pas documenté ici pour le moment).

Brochage de quelques cartes Arduino

Voici quelques brochages utiles qu'il est facile de retrouver sur Internet.

I2C	SCL	SDA
UNO	PC5 (Arduino:A5)	PC4 (Arduino:A4)
LEONARDO	PD0 (Arduino:SCL)	PD1 (Arduino:SDA)
Pro Micro (Sparkfun)	PD0 (Arduino:3)	PD1 (Arduino:2)

IV.Programmation Arduino

L'utilisation de l'i2c avec l'Arduino nécessite de commencer par inclure le fichier <wire.h> et de lancer un Wire.begin() dans le setup().

Envoi de données

Le code est très dépendant du composant i2c utilisé mais il aura une forme ressemblant au squelette ci-dessous :

```
1         Wire.beginTransmission(0x2F); // slave address is 0x2F or 0101111
2         Wire.write(0);
3         // adapter à vos besoins ici
4         Wire.endTransmission();
```

Le commentaire « adapter à vos besoins ici » sera remplacé par un ensemble de Wire.write dépendant de votre composant (voir sa documentation).

Réception de données

On commence aussi par :

```
1         Wire.beginTransmission(0x2F); // slave address is 0x2F or 0101111
2         Wire.write(0);
3         // adapter à vos besoins ici
4         Wire.endTransmission();
```

puis on demande un certain nombre d'octets à l'esclave par :

```
5         Wire.requestFrom(device_address, 3);
```

et on récupère les données :

```
6         a=Wire.read();
7         b = Wire.read();
8         c = Wire.read();
```

Notez la présence d'une fin de transmission entre les deux envois.

V. Exercices

Exercice 3

La carte cible est une Arduino UNO avec fréquence d'horloge 16 MHz.

```
1         void TWIInit(void) {
2             //set SCL to ???kHz
3             TWSR = 0x00;
4             TWBR = 0x0C;
5             //enable TWI
6             TWCR = (1<<TWEN);
7         }
```

1°) Reconstituer le commentaire de la routine ci-dessus trouvée sur internet. Comment la modifier pour avoir une horloge à 100 kHz ?

2°) Comment modifierez-vous la routine d'envoi de start pour qu'elle retourne 0 si échec et 1 si réussite (sachant que la documentation officielle donne un statut retourné de 0x08 en cas de succès).

```
1         //send start signal
2         void TWIStart(void) {
3             TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
4             while ((TWCR & (1<<TWINT)) == 0);
5         }
```

TD10 : Robot mobile MiniQ version 2 (ATMega32U4)

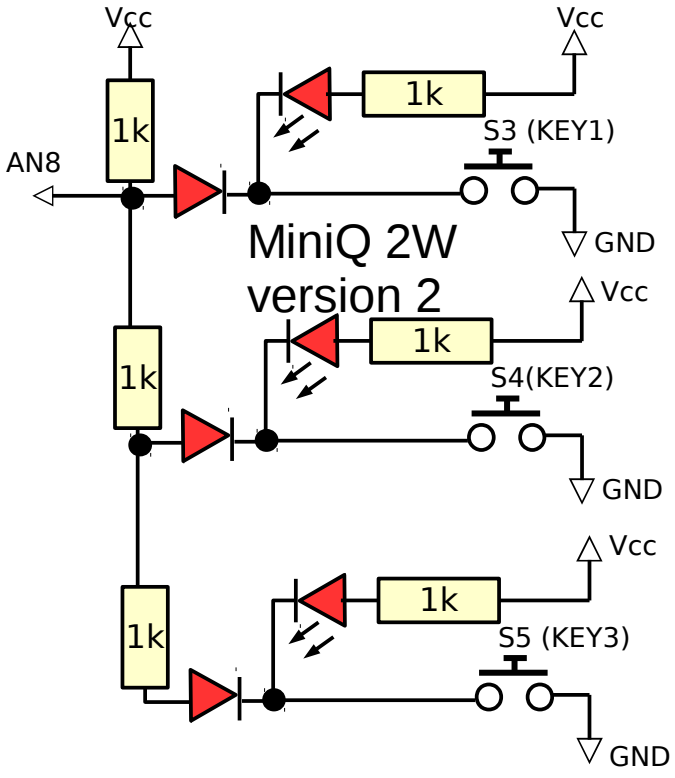
Une nouvelle version du robot MiniQ est disponible depuis 2015. La grande différence est qu'il est architecturé autour du 32U4 au lieu de l'ATMega328. Cela peut paraître anecdotique, mais les périphériques du 32U4 sont relativement différents de ceux du 328 en ce qui concerne la conversion analogique numérique et la MLI.

Voici un site commercial qui propose de la documentation] et une présentation de la version 2 (compatible Arduino Leonardo) du Robot mobile miniQ. : http://www.dfrobot.com/index.php?route=product/product&product_id=555#.VWX6T7y-cW0

I. Boutons poussoirs et Conversion Analogique Numérique

Comme déjà évoqué dans ce polycopié, il est possible d'utiliser une entrée analogique partagée avec plusieurs interrupteurs.

Voici ci-contre le schéma de connexion des Boutons Poussoirs (KEYx) à la conversion analogique numérique dans la deuxième version du MiniQ. Le point important est le numéro du convertisseur (AN8). Notez aussi le nombre d'interrupteurs : 3 (qui sont appelés de KEY1 à KEY3. KEY1 est du côté gauche (avec la convention usuelle du regard dirigé vers l'avant).



Trouver l'interrupteur appuyé

Exercice 1

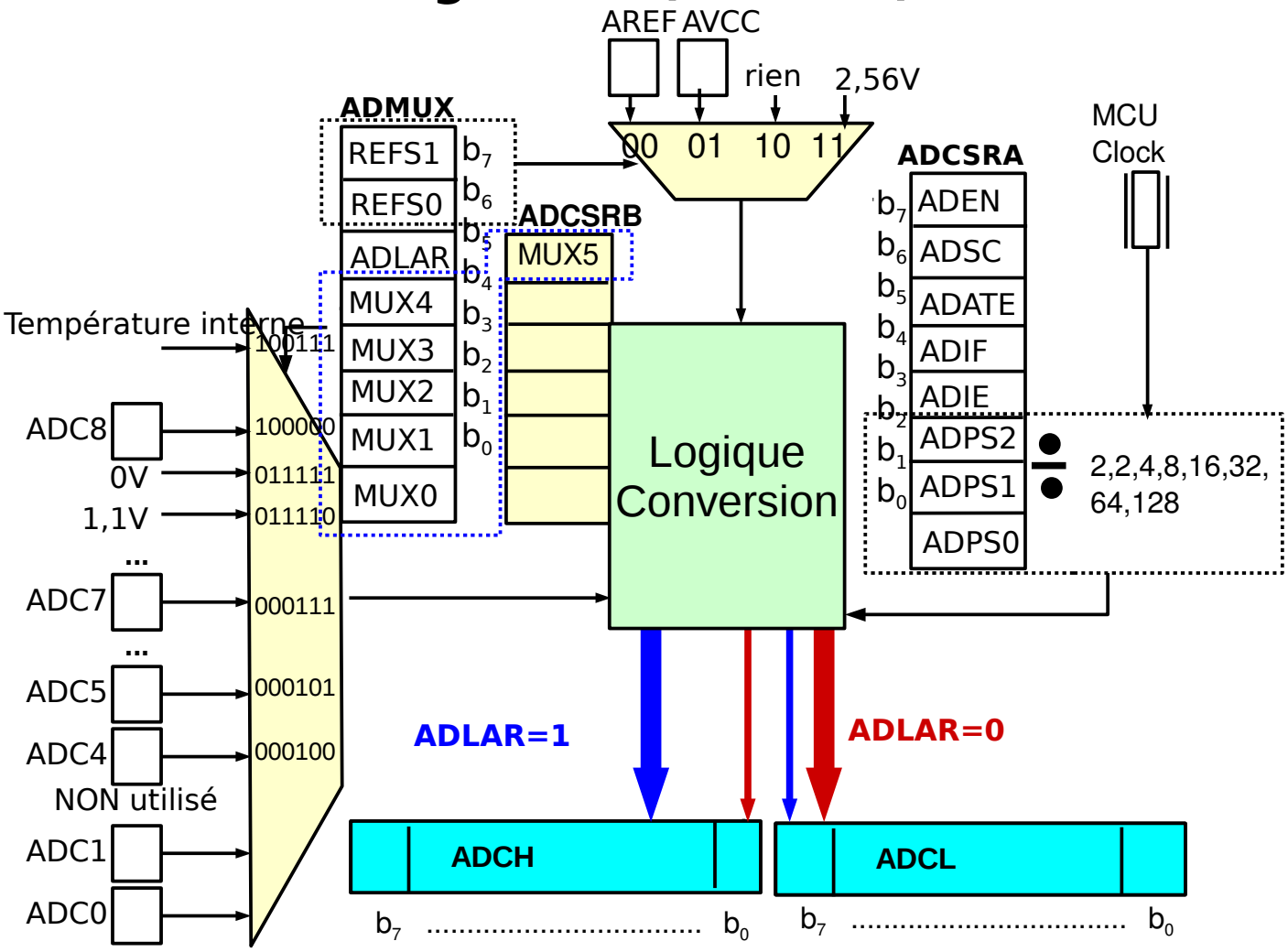
1°) Montrer que vous savez faire un programme Arduino qui utilise A6 (Conversion Analogique Numérique) et affiche dans un moniteur série les différentes valeurs suivant le/les boutons appuyés. Remplir alors le tableau ci-dessous.

Key3	*Key2*	*Key1*	*ADC8 (V et N)*
0	0	0	?
0	0	1	?

0	1	0	?
0	1	1	?
1	0	0	?
1	0	1	?
1	1	0	?
1	1	1	?

Faire un programme qui affiche correctement (en clair) la touche appuyée. Pour cela on cherchera les points milieux des valeurs trouvées dans le tableau.
2°) Écrire un sous-programme en C pur d'initialisation de la conversion analogique numérique (à partir de la documentation ci-dessous).

CAN dans ATMega32U4 (extraits)



ADEN : démarre la logique de conversion
ADSC : ADC Start Conversion

3°) Conversion en binaire

Continuer l'exercice précédent en réalisant un sous-programme qui retournera un nombre binaire représentant l'interrupteur appuyé.

b7	***b6***	***b5***	***b4***	***b3***	***b2***	***b1***	***b0***
0	0	0	0	0	key3	key2	key1

Ce tableau indique que quand key1 est appuyé on a un 1 dans b0 tandis que si c'est key3 le 1 correspondant est dans b2....

Le prototype de la fonction demandée sera :

unsigned char conversion(int can);

4°) Touches incrémentation/décrémentation
Réaliser un sous-programme capable d'incrémenter/décrémenter une variable (de type unsigned char donc variant de 0 à 255) en fonction de l'appui sur les boutons.

II. Utilisation du timer 4 pour commander les moteurs

Les moteurs du miniQ version 2 sont commandés par le timer4. Il s'agit d'un timer assez complexe, qui peut fonctionner sur 10 bits et permet la commande de transitors de puissance FET (commandes inversées avec temps morts). Heureusement, nous nous contenterons du fonctionnement simple sur 8 bits.

Un exemple pour le moteur gauche

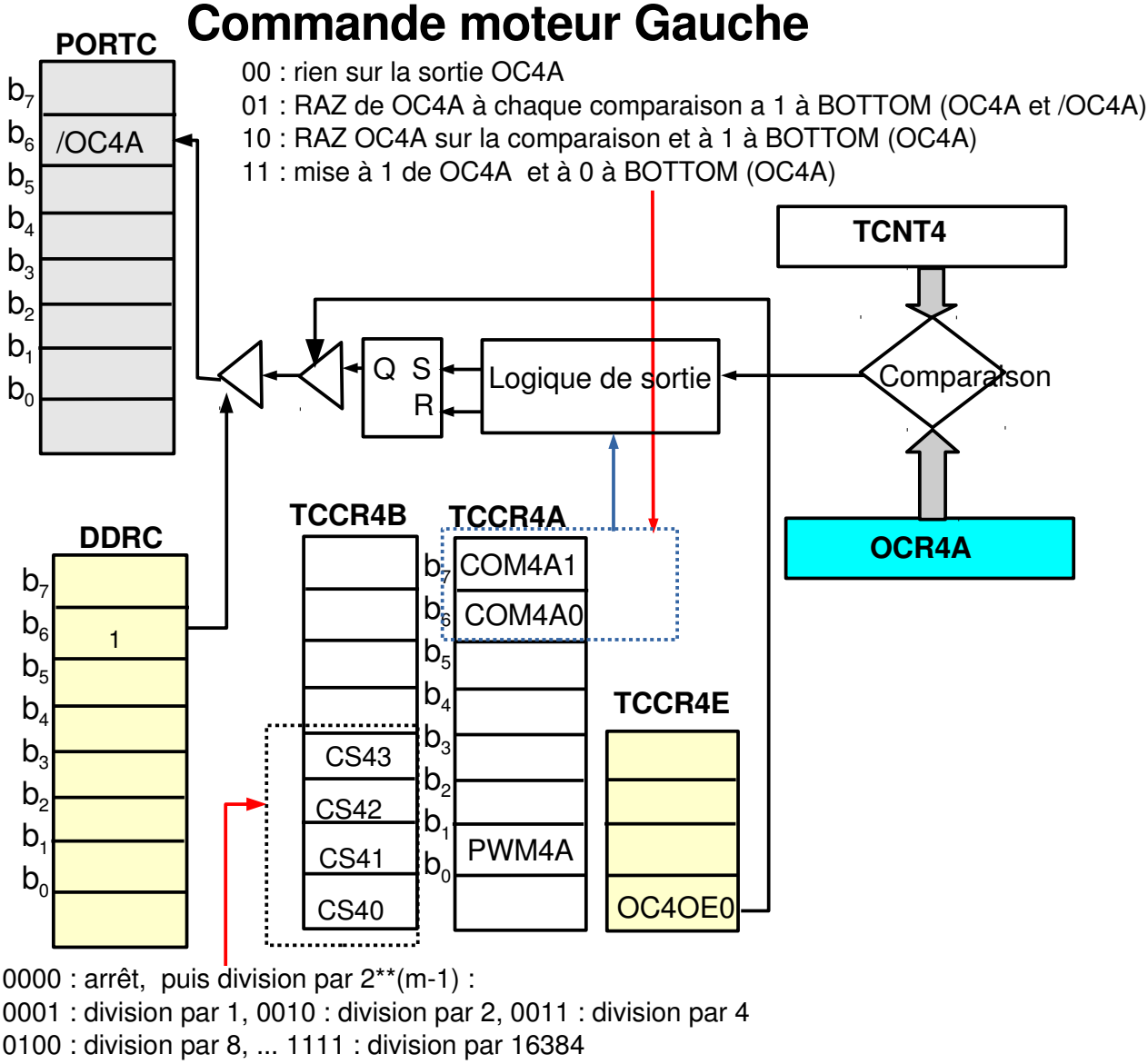
Le moteur gauche est commandé par la broche PC6(=/OC4A) du **PORTC**.

Voici un exemple de code capable d'initialiser le PWM du moteur gauche et montre un exemple d'utilisation.

```
1         void initMoteurG() {
2           DDRC |=0x40; // PC6 en sortie
3           TCCR4B |= 0x04; // division par 8
4           TCCR4A |= 0x42; // PWM en et /OC4A
5           TCCR4E |= 0x01; // 0C4OE0 pour /OC4A
6         }
7
8         void setVitesseG(unsigned char speedG) {
9           OCR4A = 255-speedG; // rapport cyclique
10        }
11
12        main() {
13          // put your setup code here, to run once:
14          initMoteurG();
15          setVitesseG(155;);
16          while(1);
17        }
```

La sélection marche avant marche arrière n'est pas réalisée. Après avoir présenté la documentation correspondante, nous passerons à l'autre moteur.

PWM avec ATMega32U4 sur TIMER4
Commande moteur Gauche



Un exemple pour le moteur droit

Le moteur droit est connecté à la sortie PD7/OC4D du **PORTD**. Voici le code permettant de gérer correctement le moteur droit.

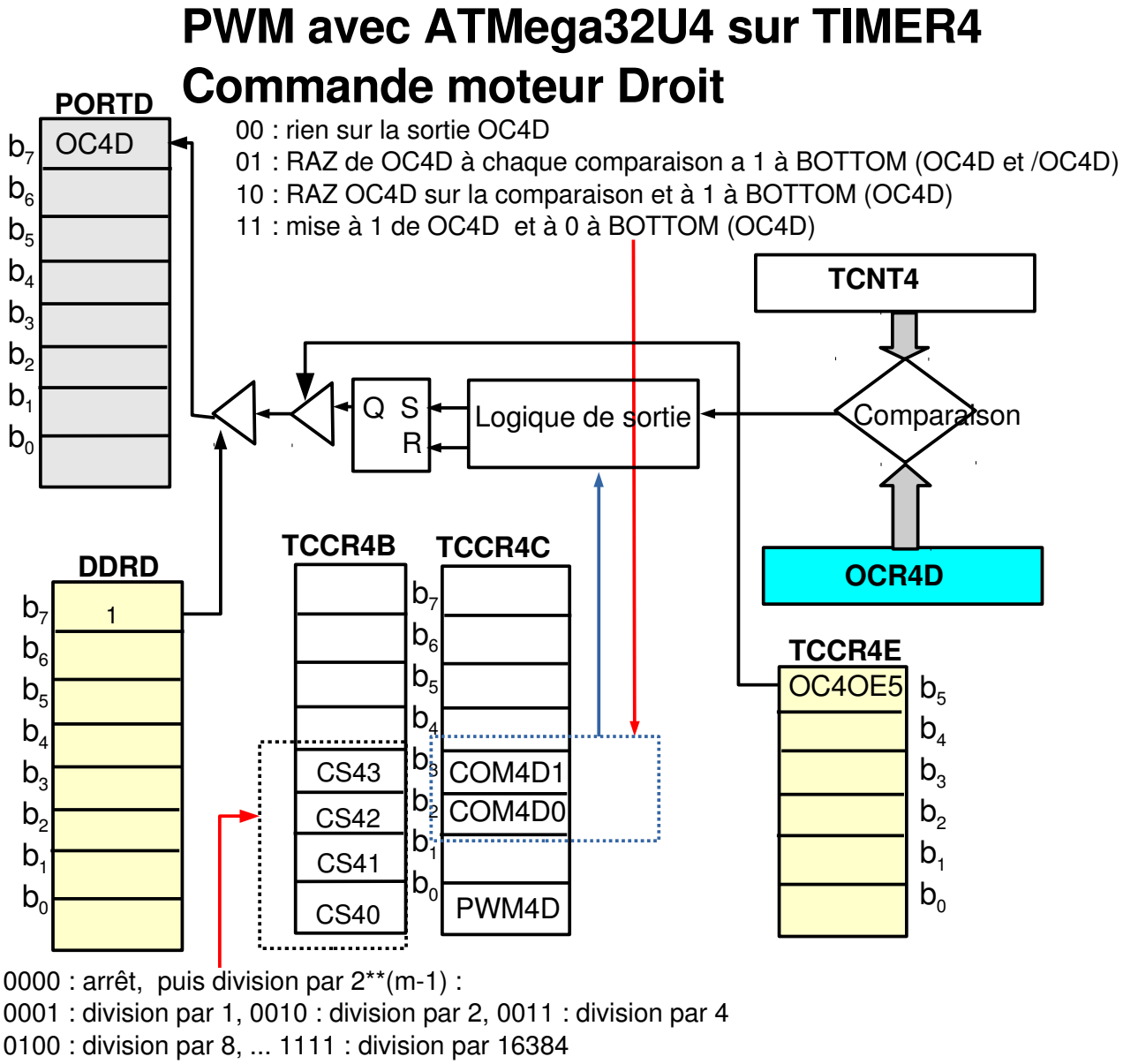
```
1         void initMoteurD() {
2           DDRD |= 0x80; // PD7 en sortie
3           TCCR4B |= 0x04; // division par 8
4           TCCR4C |= 0x09; // PWM en et OC4D et sans /OC4D
5           TCCR4E |= 0x20; // 0C4OE5 pour OC4D
6         }
7
8         void setVitesseD(unsigned char speedD) {
9           OCR4D = speedD; // rapport cyclique
10        }
```

Travail à faire

Exercice 2

Reprendre les exemples de code et refaire une initialisation commune pour les deux moteurs. On prendra alors soin de remplacer l'hexadécimal par le nom des bits.

Voici la documentation pour le moteur droit.



Exercice 3

On désire reprendre le travail de lecture des interrupteurs en C (voir plus haut dans l'exercice 1). Écrire un programme C capable de lire **ADC8** (ATTENTION c'est A6 pour l'Arduino). L'appui sur le bouton gauche KEY1 fera tourner le moteur gauche, tandis que l'appui sur le bouton droit fera tourner le moteur droit.

Exercice 4 : refaire un sous-programme qui gère les deux moteurs à la fois et le sens. On aura deux paramètres par moteur, un pour la direction (avant / arrière) et un pour le rapport cyclique.Il aura le prototype :

```
void Motor_Control(char MG_DIR,char MG_EN,char MD_DIR,char MD_EN);
```

Indication : les commandes de sens pour respectivement le moteur droit et le moteur gauche sont les broches **PE6** et **PD6**.

CORRECTIONS

M2103 TD n°1 : langage C : opérateurs et expressions

Exercice 1

```
a = 25 * 12 + b;
if (a>4 && b==18) { } // impossible d'enlever les parenthèses restantes
a>=6 && b<18 || c!=18
c = a = b+10;
```

Évaluation :

- a reçoit 318
- l'expression booléenne dans le if est vraie
- l'expression complète est vraie car c est différent de 18
- b+10 = 28 sera affecté à a puis à c

Exercice 2

```
signed char p1;
p1 = p1 | 0x04; // mettre à 1 le bit b2
p1 = p1 | 0x48; // mettre à 1 le bit b3 et b6
p1 = p1 & 0xFE; // mettre à 0 le bit b0
p1 = p1 & 0xCF; // mettre à 0 le bit b4 et b5
p1 = p1 ^ 0x08; // inverser le bit b3 (se fait facilement avec un ou exclusif)
p1 = p1 & 0xFE | 0x04 ; // mettre à 1 le bit b2 et à 0 le bit b0
p1 = p1 & 0xE7 | 0x81 ; // mettre à 1 les bits b0 et b7 et à 0 les bits b3 et b4
```

ou encore

```
signed char p1;
p1 = p1 | _BV(2); // mettre à 1 le bit b2
p1 = p1 | _BV(3) | _BV(6); // mettre à 1 le bit b3 et b6
p1 = p1 & ~_BV(0); // mettre à 0 le bit b0
p1 = p1 & ~(_BV(4)|_BV(5)); // mettre à 0 le bit b4 et b5
p1 = p1 ^ _BV(3); // inverser le bit b3 (se fait facilement avec un ou exclusif)
p1 = p1 & ~_BV(0) | _BV(2) ; // mettre à 1 le bit b2 et à 0 le bit b0
p1 = p1 & ~(_BV(3)|_BV(4)) | (_BV(0) | _BV(7)) ; // mettre à 1 les bits b0 et b7 et à 0 les bits b3 et b4
```

Exercice 3

```
unite = nb % 10;
centaine = nb % 100;
dizaine = (nb / 10) % 10;
```

Exercice 4

a= 0xF0 et b=0x0F

- a&b = 0x00
- a&&b = vrai && vrai = 0x01

expression vraie si

- le bit b6 est à 1 : (p & 0x40) == 0x40
- le bit b3 est à 0 : (p & 0x08) == 0x00
- le bit b6 est l'inverse du bit b3 :
- le bit b2 est à 1 et le bit b4 est à 0 :
- le bit b2 est à 1 ou le bit b7 est à 0 :

M2103 TD n°3

Exercice 2

1-a) Nous appelons les interrupteurs par le nom des broches auxquels ils sont connectés.

- PB6 et PB5 lâchés : 0x40
- PB6 lâché et PB5 appuyé : 0x60
- PB5 lâché et PB6 appuyé : 0x00
- PB6 et PB5 appuyés : Ox20

1-b)

```
#undef F_CPU
#define F_CPU 25000000UL
#include "util/delay.h"
int main() {
  unsigned char leds = 0x00;// variable de gestion des LEDs
// setup()
  DDRB |= 0x90; //PB7 et PB4 en sortie
// loop()
  while(1) {
    // boucle d'attente ATTENTION PB6=1 et PB5=0 au repos
    while(interrupteurs == 0x20)
       interrupteurs = PINB & 0x60;
    switch(interrupteurs) {
       case 0x60 : PORTB = leds^0x10;break;
       case 0x00: PORTB = leds^0x80;break;
       case 0x20 : PORTB = leds^0x90;break; // les deux
    }
    _delay_ms(1000);
  } // while(1)
  return 0;
} // main
```

2°) L'idée est de calculer le PGCD des deux périodes et de faire tourner la boucle infinie à période correspondante, ou plutôt à la demi-période si on utilise une technique de basculement avec un OU exclusif. On incrémentera ensuite deux compteurs, un pour chacune des leds, qui compteront jusqu'à réaliser les demi-périodes correspondantes et réaliseront alors le basculement.

Exercice 3

1°) Seuls deux et trois sont affichés bizarrement. On préférerait donc :

```
unsigned char tableau[7]={0x00,0x10,0x82,0x92,0xAA,0xBA,0xEE};
```

3-b)

```
#include <util/delay.h>

unsigned char pseudoAleat(int Lim) {
  unsigned char Result;
  static unsigned int Y=1;
  Y = (Y * 32719 + 3) % 32749;
  Result = ((Y % Lim)+1);//+1 : eviter 0
  return Result;
}

int main(){
  unsigned char tableau[7]={0x00,0x10,0x82,0x92,0xAA,0xBA,0xEE};
  unsigned char i;
  // setup()
  DDRD = 0xFE; // PD7, PD6 , ...,PD1 en sortie
  // loop()
  while(1) {
      while ((PIND & 0x01)==0x01)
        _delay_ms(300);
      i= pseudoAleat(6);
      PORTD = tableau[i];
     // _delay_ms(1000);
  }// while(1)
  return 0;
}
```

4°) L'astuce consiste à regrouper les leds qui s'allument ensembles (pour un dé) sur un même bit du PORT : il faut ainsi 4 bits par dé. Pour cela le bouton de lancer doit donc être relié à PB0.

Exercice 4

Question 2

à modifier le début car cette solution ne correspond pas au même câblage !

```
char lecture_ligne()
{
  char ch;
  DDRD |= 0x70;    // commençons par lister les sorties sur le port D
  DDRB &= 0xF0;    // puis les entrées sur le port B
  PORTD &= 0x8F;    // on place les sortie à l'état 0
  PORTB |= 0x0F;    // on active les résistances de pull-up sur les entrées
  delay(10);        // un délais est nécessaire pour l'activation des pull-ups
  ch = PINB & 0x0F; // on récupère ensuite l'état des entrées
  DDRD = 0;           // remettre toutes les pattes en entrées.
  PORTB = 0x00;
  delay(10);
  switch (ch)
  {
     case  0x0F: return 0; // aucune touche
     case  0x0E: return 1; // L1
     case  0x0D: return 2; // L2
     case  0x0B: return 3; // L3
     case  0x07: return 4; // L4
     // si autre cas, deux touches ou autre
   default : return NOTAKEY;
  }
```

```
}
```

Question 3

```
char getTouche() {
  char touches[4][3] = { // tableau de char à 2 dimensions
    {1,2,3},
    {4,5,6},
    {7,8,9},
    {10,0,11}
  };
  char line,col;
  line=lecture_ligne();
  col=lecture_colonne();
  if ((col>0) && (col<4) && (line >0) && (line < 5))
    return touches[line-1][col-1];
  else
    return NOTAKEY;
}
```

M2103 TD n°4

Exercice 1

1°)

```
unsigned int div10(unsigned int A){
        unsigned int Q; /* the quotient */
        Q = ((A >> 1) + A) >> 1; /* Q = A*0.11 */
        Q = ((Q >> 4) + Q) ; /* Q = A*0.110011 */
        Q = ((Q >> 8) + Q) >> 3; /* Q = A*0.00011001100110011 */
        /* either Q = A/10 or Q+1 = A/10 for all A < 534,890 */
        return Q; // ne pas oublier le return pour une fonction
}
```

2°)

```
void afficheLeds(unsigned char ch){
  unsigned char ch_partie;
  ch_partie = (ch << 3) & 0x78;
  PORTH = ch_partie;
  ch_partie = ch & 0xF0;
  PORTB = ch_partie;
}
```

3°)

```
//***** Compilateur avr-gcc ************
//**** OK : Arduino MEGA2560 + shield
#include <avr/io.h>
#undef F_CPU
#define F_CPU 16000000UL
#include <util/delay.h>

unsigned int div10(unsigned int A);
void afficheLeds(unsigned char ch);

int main(void){
    unsigned int res,i;
    unsigned char temps;
```

```
    DDRH = 0x78; // 4 sorties pour H
    DDRB = 0xF0; // 4 sorties pour B

// initialisation du timer avec prescaler 256 ,
    TCCR0B |= _BV(CS02);
    TCCR0B &= ~(_BV(CS01) | _BV(CS00));
    while(1) {
// départ de la mesure
// while ((TIFR & (0x01<<TOV0)) == 0);
        TCNT0 = 0x00;
// algorithme de calcul ici plusieurs fois
        for(i=0;i<55;i++) {
                res=div10(i+15558);
                PORTC = res; // autrement simplifié
        }
// fin de la mesure : lecture du timer
        temps=TCNT0;
        afficheLeds(temps);
        //afficheLeds(0x55);
        _delay_ms(500);
    } // while(1)
    return 0;
}

unsigned int div10(unsigned int A){
        unsigned int Q; /* the quotient */
        Q = ((A >> 1) + A) >> 1; /* Q = A*0.11 */
        Q = ((Q >> 4) + Q) ; /* Q = A*0.110011 */
        Q = ((Q >> 8) + Q) >> 3; /* Q = A*0.00011001100110011 */
        /* either Q = A/10 or Q+1 = A/10 for all A < 534,890 */
        return Q; // ne pas oublier le return pour une fonction
}

void afficheLeds(unsigned char ch){
  unsigned char ch_partie;
  ch_partie = (ch << 3) & 0x78;
  PORTH = ch_partie;
  ch_partie = ch & 0xF0;
  PORTB = ch_partie;
}
```

L'ajout dans la boucle de "PORTC = res;" a été essentielle pour éviter la suppression de la boucle avec juste un warning comme quoi res n'était jamais utilisé ! C'est normal c’est TCNTO qui m'intéresse pas res !

Pour comparer avec l'algorithme classique, remplacer

```
// algorithme de calcul ici plusieurs fois
        for(i=0;i<50;i++) {
                res=div10(i+15558);
                PORTC = res; // autrement simplifié
        }
```

par

```
// algorithme de calcul ici plusieurs fois
        for(i=0;i<50;i++) {
                res=(i+15558)/10; // division classique
                PORTC = res; // autrement simplifié
```

```
        }
```

Le premier algorithme avec div10 me donne 16 ou 17.

Le deuxième algorithme avec /10 me donne 45 ou 46.

Exercice 2

1°)

4 000 000) / (8 * 256 * 2) = 977 Hz environ

Le facteur 2 apparaissant dans cette formule est lié au fait que "PORTB ^= 0x01;" fait basculer le bit et qu'ainsi il faut deux basculements pour faire une période. Il est bon de retenir que toute technique de basculement nécessite la création d'une fréquence double. Ainsi pour créer 1 kHz il vous faudra 2 kHz ...

2°)

#include <avr/io.h>

```
 void main(void){
 // initialisation du timer   division par 8
         TCCR0B |= (1<<CS01); // prescaler 8 , entrée sur quartz
         TCCR0B &= ~((1<<CS02) | (1<<CS00)); // pour être sûr
         TCNT0 = 255-99; //   tmr0  : début du comptage dans 2 cycles
 // bit RB0 du PORTB en sortie
         DDRB |= 0x01; //RB0 as output
         while(1) {
                 TIFR0 |= (1<<TOV0); // clr TOV0 with 1
                 while (TIFR0 & (1<<TOV0)) == 0);
                 // ce qui est fait ici est fait tous les 256 comptages de TCNT0
                 PORTB ^= (1<<0); // on bascule avec ou exclusif
                 //TIFR &= ~(1<<TOV0); // reset the overflow flag
                 TCNT0 = 255-99; // remplace le reset de l'overflow
         }
 }
```

3°)

On va en fait générer 2kHz et changer sans arrêt un bit B0 du PORTB. 4 / 2kHz=2000. Si l'on divise par 8 on peut rester en 8 bits avec comme valeur 250 à compter donc une initialisation à 256-250=6. Ceci n'est qu'une valeur théorique qui dépend du compilateur. Il faut donc faire des essais pour trouver la valeur exacte. Une autre technique pour être quasi indépendant du compilateur c'est de remplacer :

TCNT0 = 6;

par

TCNT0 = TCNT0+6;

qui dépend moins du temps que l'on a mis pour réaliser les instructions.

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

void main(void){
 // initialisation du timer   division par 8
         TCCR0 |= (1<<CS01); // prescaler 8 , entrée sur quartz
         TCCR0 &= ~((1<<CS02) | _BV(CS00)); // pour être sûr
         TCNT0 = 6; //   tmr0  : début du comptage dans 2 cycles
 // bit RB0 du PORTB en sortie
         DDRB |= 0x01; //RB0 as output
         while(1) {
```

```
                TIFR |= (1<<TOV0); // reset the overflow flag
                while (TIFR & (1<<TOV0)) == 0);
                TCNT0 += 6; // remplace le reset de l'overflow
                // ce qui est fait ici est fait tous les 250 comptages de TCNT0
                PORTB ^= (1<<0); // on bascule avec ou exclusif

        }
 }
```

4°)

La solution de la question 3 nous a montré qu'il fallait compter 250 pour sortie à 0 et 250 pour sortie à 1. Ici il faut donc compter 125 pour la valeur 1 et 250+125=375 pour la valeur 0. J'espère que tout le monde a vu le problème ! 375 ne tient plus dans 8 bits. Il faut donc encore diviser l'horloge par 2. Mais cela n'est pas possible : on ne peut que diviser par 8 pour avoir le prescaler à 64.

Cela nous donne un comptage de 125/8 = 16 (environ) 375/8 = 47 (environ).

```
void main(void){
 // initialisation du timer   division par 64
        TCCR0 |= ((1<<CS01) | (1<<CS00)); // prescaler 64 , entrée sur quartz
        TCCR0 |= ~(1<<CS02); // pour être sûr
        //TCNT0 = 6; //   tmr0  : début du comptage dans 2 cycles
 // bit RB0 du PORTB en sortie
        DDRB |= 0x01; //RB0 as output
        PORTB |= _BV(0);// pour être sûr de la façon dont on démarre
        while(1) {
                //TIFR &= ~(1<<TOV0); // reset the overflow flag
                TCNT0 += 240; // 256-16 : remplace le reset de l'overflow
                while (TIFR & (1<<TOV0)) == 0);
                // ce qui est fait ici est fait tous les 256 comptages de TCNT0
                PORTB ^= (1<<0); // on bascule avec ou exclusif
                //TIFR &= ~(1<<TOV0); // reset the overflow flag
                TCNT0 += 209; // 256-47remplace le reset de l'overflow
                while (TIFR & (1<<TOV0)) == 0);
                PORTB ^= (1<<0); // on bascule avec ou exclusif
        }
 }
```

M2103 TD 5

Exercice 1

1°) N'oubliez pas la division par 16 qui est réalisée avec le if (!(nb % 16)) dans l'interruption.

Calcul précis : 25 / (1024*256*16) = 5,96 .

2°) "if (!(nb % 16))" est une façon pas très efficace de calculer le reste de la division par 16. J'ignore la technique utilisé par le compilateur, mais ce calcul est forcément long puisqu'il n'y a pas d'instruction de division sur l'ATMega8.

Comme la division se fait par 16 qui est une puissance de deux, on peut utiliser un masque pour faire ce calcul bien plus rapidement :

```
// langage C
```

```
if (!(nb & 0x0F)) // idem à if (!(nb % 16)) mais plus rapide
```

3°) Décalage tout simple d'une LED vers les poids forts. L'utilisation d'une variable "vPORTB" complique un peu la lecture du programme ! Ceci est lié au fait que dans le on ne peut pas lire PORTB (ce que l'on peut faire dans la vraie vie.

4°)

```
TCCR0B = (1<<CS01) | (1<<CS00);
```

met le prescaler à 64

4 000 000 / 2*64*(256-96)*400 = 0,48828125 Soit pratiquement 0,5

5°)

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

unsigned int cnt;
ISR(TIMER0_OVF_vect) {
    cnt++;                 // increment counter
    TCNT0   = 96;
  }
}
int main() {
  TCCR0B = (1<<CS01) | (1<<CS00);  // Assign prescaler to TCNT0
  DDRB = 0xFF;            // PORTB is output
  PORTB = 0x80;            // Initialize PORTB
  TCNT0  = 96;             // Timer0 initial value
  TIMSK0=(1<<TOIE0);        // Enable TMRO interrupt
  sei();
  cnt = 0;                 // Initialize cnt
  do {
    if (cnt >= 400) {
      PORTB = PORTB>>1;      // PORTB LEDs *** changé ici ****
      if (PORTB == 0x00)   PORTB = 0x80; // *** changé ici ****
      cnt = 0;             // Reset cnt
    }
  } while(1);
  return 0;
}
```

Exercice 2

1°)

```
unsigned char
transcod[16]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x77,0x7C,0x39,0x5E,0
x79,0x71};
```

2°)

```
1         void display(char nb) {
2           if (nb<16) PORTC = transcod[nb] else PORTC = 0x20 ;
3         }
```

3°)

```
int main(void){
```

```
    DDRC = 0x7F; // 7 sorties pour C
    DDRB = 0x03; // 2 sorties pour B
//** gestion du timer0
    // enable timer overflow interrupt for both Timer0
     TIMSK0 = (1<<TOIE0);
    // set timer0 counter initial value to 0
     TCNT0 = 0x00;
    // start timer0 with 1024 prescaler
     TCCR0B = (1<<CS02) | (1<<CS00);
    // enable interrupts
     sei();
    while(1) {
    // réalisation d'un compteur BCD
        cmpt ++;
        if ((cmpt & 0x0F)>9) cmpt+=6;
        if ((cmpt & 0xF0)>0x90) cmpt+=0x60;
    // on attend
        _delay_ms(1000);
    } // while(1)
    return 0;
}
```

L'addition de 6 quand on dépasse n'a rien d'intuitif mais est important à connaître.

4°) Réaliser enfin l'interruption qui affichera tantôt les dizaines, tantôt les unités.

```
// timer0 overflow

 ISR(TIMER0_OVF_vect) {
   TCNT0 = 100;
   mux++;
   if ((mux & 0x01)==0) {
     PORTB |= (1<<1) ;
     PORTB &= ~(1<<0);
     display(cmpt >> 4);
   } else {
     PORTB |= (1<<0) ;
     PORTB &= ~(1<<1);
     display(cmpt & 0X0F);
   }
 }
```

M2103 TD 6

Exercice 1

1°)

#include <avr/io.h>

```
int main(void){
  // set OC0A as output
    DDRD |= 0x40;
  // Set the Timer Mode to CTC
    TCCR0A |= (1 << WGM01);
  // Set the value that you want to count to
    OCR0A = 0xF9;
  // Set OC0A toggle mode
    TCCR0A |= (1 << COM0A0);
  // set prescaler to 256 and start the timer
    TCCR0B |= (1 << CS02);
    while (1)
```

```
    {
        //main loop
    }
    return 0;
}
```

3°) En mode basculement (mode 1) : $f_{MIN} = \frac{16.000.000}{1024 * 256 * 2} = 30{,}51\,Hz$.

Le 2 qui apparaît au dénominateur est lié au mode basculement, le 256 est lié aux 256 valeurs de TCNT0 de 0 à 255, et le 1024 est la plus grande division du préscaler.

4°) En mode CTC encore : . $f_{MAX} = \frac{16.000.000}{1 * 2 * 2} = 4\,MHz$. A vérifier quand même !!!

Le premier 2 qui apparaît au dénominateur est lié à la valeur minimale de **OCR0A** soit 1. Ainsi, pour faire une période il faut compter de 0 à 1 puis que l'on repasse à 0 (soit deux comptages). Le deuxième est lié au mode basculement.

Exercice 2

Même si la documentation de l'interruption de la comparaison n'a pas été évoquée dans ce chapitre, le code ci-dessous n'est pas difficile à comprendre. On admettant qu'effectivement la valeur de 4 ms du commentaire soit correcte, il nous faut multiplier par 125 pour avoir 500 ms de période (soit compter de 0 à 124). On calcule 500ms à cause du basculement : il faut deux basculements pour une période.

```
// this code sets up a timer0 for 4ms @ 16Mhz clock cycle
// an interrupt is triggered each time the interval occurs.
//***** testé OK sur Arduino MEGA2560 + shield
#include <avr/io.h>
#include <avr/interrupt.h>
// compteur
volatile unsigned char cpt=0;

int main(void){
    // Set the Timer Mode to CTC
    TCCR0A |= (1 << WGM01);
    // Set the value that you want to count to
    OCR0A = 0xF9;
    TIMSK0 |= (1 << OCIE0A);    //Set the ISR COMPA vect
    sei();         //enable interrupts
    // set prescaler to 256 and start the timer
    TCCR0B |= (1 << CS02);
    //Configuration PORTB.4 en sortie
    DDRB |= (1<<DDB4);
    PORTB &= ~(1<<PB4); // PORTB.4 <-0
    while (1)
    {
        //main loop
    }
    return 0;
}

ISR (TIMER0_COMPA_vect) { // timer0 overflow interrupt
    //event to be exicuted every 4ms here
    cpt++;
```

```
    if (cpt == 124) { // 500ms/4ms = 125
      cpt = 0;
      PORTB ^=(1<<PB4); // basculement
    }
}
```

Les 4 ms annoncées sont très théoriques et ne tiennent pas compte du temps que le processeur utilise pour réaliser l'interruption. Avant d'envisager de modifier ce code prenez un fréquencemètre (un oscilloscope n’est pas simple à utiliser autour du Hertz).

Exercice 3 : PWM rapide

1°)

```
#include <avr/io.h>
#undef F_CPU
#define F_CPU 16000000UL
#define BaudRate 9600
#define MYUBRR (F_CPU / 16 / BaudRate ) - 1

#include <util/delay.h>

void serialInit(void);
unsigned char serialCheckTxReady(void);
unsigned char serialCheckRxComplete(void);
void serialWrite(unsigned char DataOut);
unsigned char serialRead(void);
void usart_puts(char str[]);
void usart_puts_hexa(unsigned char val);

int main(void){
  unsigned char ch,rapport_cyclique,erreur=0;

    serialInit();
    while(1) {
       if (serialCheckRxComplete()) {
          ch = UDR0;
          rapport_cyclique = ch-'0';
          if (rapport_cyclique > 9) rapport_cyclique -= 7;
          if (rapport_cyclique > 15) {
            usart_puts("Erreur Caractère non reconnu\n");
            erreur=1;
          } else {
            rapport_cyclique <<= 4;
            ch = serialRead();
            ch -= '0';
            if (ch > 9) ch -= 7;
            if (ch > 15) {
              usart_puts("Erreur Caractère non reconnu\n");
              erreur = 1;
            } else {
              rapport_cyclique += ch;
              usart_puts_hexa(rapport_cyclique);
            } // else
          } // if (!erreur) else
       }

    // on attend
        //_delay_ms(300);
    } // while(1)
```

```
    return 0;
}
```

Un sous programme serait bien mieux pour y voir plus clair.

2°)

```
int main(void){
  unsigned char ch,rapport_cyclique,erreur=0;
    TCCR0A = 0x00;
     // Set the Timer Mode to PWM fast
    TCCR0A |= ((1 << WGM01) | (1<<WGM00));
    // clear OC0A on compare match set OC0A at TOP
    TCCR0A |= (1 << COM0A1);
    // Set the value that you want to count to
    OCR0A = 0x88; // rapport cyclique
    // set prescaler to 1024 and start the timer
    TCCR0B = 0x00;
    TCCR0B |= ((1 << CS02) | (1 << CS00));
    //Configuration PORTB.7 en sortie (PORTB.6 pour UNO)
    DDRD |= (1<<DDD6);
    TCNT0 = 0x00;
    serialInit();
    while(1) {
       if (serialCheckRxComplete()) {
          erreur=0;
          ch = UDR0;
          rapport_cyclique = ch-'0';
          if (rapport_cyclique > 9) rapport_cyclique -= 7;
          if (rapport_cyclique > 15) {
            usart_puts("Erreur Caractère non reconnu\n");
            erreur=1;
          } else {
            rapport_cyclique <<= 4;
            ch = serialRead();
            ch -= '0';
            if (ch > 9) ch -= 7;
            if (ch > 15) {
              usart_puts("Erreur Caractère non reconnu\n");
              erreur = 1;
            } else {
              rapport_cyclique += ch;
              usart_puts_hexa(rapport_cyclique);
            } // else
          } // if (!erreur) else

          // ** non ! if (PORTB & 0x80) PORTB |= 0x10; else PORTB &= 0xEF;
       }
       if (!erreur) { // gérer rapport cyclique
             OCR0A = rapport_cyclique;
        }
    // on attend
        //_delay_ms(300);
    } // while(1)
    return 0;
}
```

Exercice 4

```
#include <avr/io.h>
#include <util/delay.h>
```

```
// Frequence 16MHz

//*********** Pour platine UNO *************
int main() {
  //Configuration PORTD.6 pour UNO
  DDRD |= (1<<DDD6);
  // set prescaler to 1024 and start the timer
  TCCR0B |= (1 << CS02)|(1 << CS00);
  // Set the Timer Mode to PWM fast
  TCCR0A |= ((1 << WGM01) | (1<<WGM00));
  // clear OC0A on compare match set OC0A at TOP
  TCCR0A |= (1 << COM0A1);
  OCR0A = 26;
  TCNT0 = 0x00;
  while(1) {
    OCR0A -= 1;
    if (OCR0A == 13) OCR0A = 26;
    _delay_ms(1000);
  }
  return 0;
}
```

M2103 TD n°7 : Conversion analogique numérique

Exercice 1

1°) 2,56 / 1024 = 2,5 mV (nombre exact).

2°) 0x12F = 303 donc tension 303 * 2,5mV = 757,5 mV = 0,76 V

3°) 2 / 2,56 * 1023 = 799,21 arrondie à 799 = 0x31F

4°) Entre -40° et +110°, il y a 150 intervalles. 150 x 10mV cela fait une tension max de 1,5V. On suppose encore une tension de référence de 2,56V.

+45° donne 85 intervalles de 10mV soit 0,85V soit un nombre de 1023/2,56 * 0,85 = 339,66 arrondi à 0x154

Exercice 2

```
// on lance la conversion
ADCSRA = (1 << ADSC);
// on attend qu'elle soit finie
while ((ADCSRA & (1 << ADSC))==(1 << ADSC));
// ici le bit ADSC vient de passer à 0
```

Autre technique :

```
// on lance la conversion
ADCSRA = (1 << ADSC);
// on attend qu'elle soit finie
while (ADCSRA & (1 << ADIF) != (1 << ADIF));
// ici le bit ADIF vient de passer à 1
// c'est un flag qu'il faut remettre à 0 !!!!
ADCSRA |= (1 << ADIF);
```

Exercice 3

ADMUX = (1 << REFS0) montre que c'est VREF qui est pris comme tension de référence. Nous pensons donc qu'il faut remplacer AREF par VREF dans le commentaire puisque AREF n'est pas pris en compte.

Le result =ADCH; veut dire que l'on ne garde que les poids fort. A priori il n'y a que 2 bits de poids fort puisque **ADCL** en prend 8 ! Mais c'est sans compter sur le bit **ADLAR** qui positionné à 1 qui met les 8 bits de poids fort dans **ADCH** et deux bits de poids faible dans **ADCL**... qui sont donc perdus. En général ce n'est pas très grave de perdre les poids faibles.

Exercice 4

1°) Aucune touche appuyée : V_{AN8} = Vcc

S1 appuyée V_{AN8} = 0V

S2 appuyé $V_{AN8}=\frac{0,510}{2,710}\cdot V_{CC}=0,94\,V$

S3 appuyé $V_{AN8}=\frac{1,510}{2,710}\cdot V_{CC}=2,035\,V$

2°) Aucune touche appuyée : CAN_{AN8} = 1023

S1 appuyée CAN_{AN8} = 0

S2 appuyé $CAN_{AN8}=\frac{0,510}{2,710}\cdot 1023=193$

S3 appuyé $CAN_{AN8}=\frac{1,510}{2,710}\cdot 1023=416$

3°) while (n==1023) ; // très risqué

while (n>800) ; //certainement mieux

4°) Rien : Vcc => CAN=1023

KEY1 appuyé : V_{AN8} = 0,6V (à cause de la diode) CAN=123

KEY2 appuyé : VAN8 = 2,5+0,6=3,1V => CAN = 634

KEY3 appuyé : VAN8 = 2/3*5+0,6=3,93V => CAN = 804

Les valeurs trouvées expérimentalement sont :

- KEY1 (sw3) appuyé : CAN8 = 135
- KEY2 (sw4) appuyé : CAN8 = 575
- KEY3 (sw5) appuyé : CAN8 = 722

M2103 TD9

Exercice 1

1°)

```
4         #define F_CPU 16000000  // 16 MHz oscillator.
5         #define BaudRate 9600
6         #define MYUBRR (F_CPU / 16 / BaudRate ) - 1
```

donne 16000000/16*9600 – 1 = 103 (c'est une valeur sur 7 bits)

2°)

#define BaudRate 9600 doit être transformé en #define BaudRate 19200

3°) Comment modifier ce code pour avoir une parité paire ?

Ajouter dans serialInit() : UCSR0C |= (1<<UPM01) ;

4°)

```
// nonzero if serial data is available to read.
unsigned char serialCheckRxComplete(void) {
        return( UCSR0A & _BV(RXC0)) ;
}

// nonzero if transmit register is ready to receive new data.
unsigned char serialCheckTxReady(void) {
        return( UCSR0A & _BV(UDRE0) ) ;
}

unsigned char serialRead(void) {
        while (serialCheckRxComplete() == 0) ;// While data is NOT available to read
        return UDR0;
}

void serialWrite(unsigned char DataOut)
{
        while (serialCheckTxReady() == 0) ;   // while NOT ready to transmit
        UDR0 = DataOut;
}
```

5°)

```
// Pour comprendre la faiblesse de cette fonction donnez-lui "GG" à convertir
// elle vous retourne quelque chose alors que "GG" n'est manifestement pas un
// nombre hexadécimal
unsigned char usart_gets_hexa() {
  unsigned char val;
  char tab[3];
  tab[0] = serialRead(); //poids fort
  tab[1] = serialRead(); //poids faible
  while (serialCheckRxComplete()) serialRead(); // on vide buffer
  tab[0] -= '0';
  tab[1] -= '0';
  if (tab[0] > 9) tab[0] -= 7;
  if (tab[1] > 9) tab[1] -= 7;
  val = (tab[0] << 4) + tab[1];
  return val;
}
```

TD10 : Robot mobile MiniQ version 2 (ATMega32U4)

Exercice 1

1°)

```
1         void setup() {
2           // put your setup code here, to run once:
3           Serial.begin(9600);
4         }
5
6         void loop() {
7           // put your main code here, to run repeatedly:
8           Serial.println(analogRead(A6));
9           delay(1000);
10        }
```

Les valeurs trouvées expérimentalement sont :
* KEY1 (sw3) appuyé : CAN8 = 135
* KEY2 (sw4) appuyé : CAN8 = 575
* KEY3 (sw5) appuyé : CAN8 = 722

2°)

```
1         void initCAN8() {
2           DIDR2 |= (1<<ADC8D); // disable digital input
3           //ADMUX |= (1<<MUX2)|(1<<MUX1);//ADC6 pour KEYx NON!!!!
4           ADCSRB |= (1<<MUX5);//ADC8 pour KEYx
5           ADMUX |= (1<<REFS0); //reference à 5V
6           ADCSRA |= (1<<ADPS1)|(1<<ADPS0); // division par 8 : pas pressé
7           ADCSRA |= (1<<ADEN); // demarrage logique de conversion
8           ADCSRA |= (1<< ADSC); // start first conversion
9           while((ADCSRA & (1<< ADSC)) != 0); // attente fin conversion
10          //_delay_ms(10);
11        }
```

Exercice 2

```
1         void initpwm() {
2           DDRC |= (1<<PC6); // PC6 en sortie : moteur gauche
3           DDRD |= (1<<PD7); // PD7 en sortie : moteur droit
4           TCCR4B |= (1<<CS42); // division par 8
5           // moteur gauche
6           TCCR4A |= (1<<COM4A0)|(1<<PWM4A); // PWM en et /OC4A
7           TCCR4E |= (1<<OC4OE0); // 0C4OE0 pour /OC4A
8           // moteur droit
9           TCCR4C |= (1<<COM4D1)|(1<<PWM4D); // PWM en et OC4D
10          TCCR4E |= (1<<OC4OE5); // 0C4OE5 pour OC4D
11        }
```

Remarquez que l'on n'a pas géré les directions de rotations des deux moteurs.

Exercice 3

```
1         void setVitesseG(unsigned char speedG) {
2           OCR4A = 255-speedG; // rapport cyclique
3         }
4
```

```
5         void setVitesseD(unsigned char speedD) {
6           OCR4D = speedD; // rapport cyclique
7         }
8
9         void initpwm() {
10          DDRC |= (1<<PC6); // PC6 en sortie : moteur gauche
11          DDRD |= (1<<PD7); // PD7 en sortie : moteur droit
12          TCCR4B |= (1<<CS42); // division par 8
13          // moteur gauche
14          TCCR4A |= (1<<COM4A0)|(1<<PWM4A); // PWM en et /OC4A
15          TCCR4E |= (1<<OC4OE0); // 0C4OE0 pour /OC4A
16          // moteur droit
17          TCCR4C |= (1<<COM4D1)|(1<<PWM4D); // PWM en et OC4D
18          TCCR4E |= (1<<OC4OE5); // 0C4OE5 pour OC4D
19        }
20
21        void initCAN8() {
22          DIDR2 |= (1<<ADC8D); // disable digital input
23          //ADMUX |= (1<<MUX2)|(1<<MUX1);//ADC6 pour KEYx NON!!!!
24          ADCSRB |= (1<<MUX5);//ADC8 pour KEYx
25          ADMUX |= (1<<REFS0); //reference à 5V
26          ADCSRA |= (1<<ADPS1)|(1<<ADPS0); // division par 8 : pas pressé
27          ADCSRA |= (1<<ADEN); // demarrage logique de conversion
28          ADCSRA |= (1<< ADSC); // start first conversion
29          while((ADCSRA & (1<< ADSC)) != 0); // attente fin conversion
30          //_delay_ms(10);
31        }
32
33        main() {
34          uint16_t can8;
35          // put your setup code here, to run once:
36          initpwm();
37          initCAN8();
38          while(1){
39            ADCSRA |= (1<< ADSC); // start conversion
40            while((ADCSRA & (1<< ADSC)) != 0);// attente fin conversion
41            can8 = ADC;
42            if (can8 > 800) {
43              setVitesseG(0);
44              setVitesseD(0);
45            } else if (can8 > 600) {
46              setVitesseD(155);
47              setVitesseG(0);
48            } else if (can8 > 200) {
49              setVitesseD(0);
50              setVitesseG(0);
51            } else {
52              setVitesseD(0);
53              setVitesseG(155);
54            }
55            _delay_ms(100);
56          }
```

Exercice 4 :

```
#define FORW 1
#define BACK 0
//..........
void initpwm() {
  DDRC |= (1<<PC6); // PC6 en sortie : moteur gauche
  DDRD |= (1<<PD7); // PD7 en sortie : moteur droit
```

```
  TCCR4B |= (1<<CS42)|(1<<CS41); // division par 64
  // moteur gauche
  TCCR4A |= (1<<COM4A0)|(1<<PWM4A); // PWM en et /OC4A
  TCCR4E |= (1<<OC4OE0); // 0C4OE0 pour /OC4A
  // moteur droit
  TCCR4C |= (1<<COM4D1)|(1<<PWM4D); // PWM en et OC4D
  TCCR4E |= (1<<OC4OE5); // 0C4OE5 pour OC4D
  // pendant qu'on y est on prépare tout pour pouvoir changer de sens
  DDRE |= (1<<PE6); // sens moteur droit;
  DDRD |= (1<<PD6); // sens moteur gauche
}

void Motor_Control(char MG_DIR,char MG_EN,char MD_DIR,char MD_EN) {
  // moteur gauche
  if(MG_DIR==FORW)//si vers l'avant
    PORTD |=(1<<PD6);//IN2 passé à 1
  else
    PORTD &=~(1<<PD6);//IN2 passé à 0
  OCR4A = 255-MG_EN; // rapport cyclique ou setVitesseG(MG_EN);
  // puis moteur droit
  if(MD_DIR==FORW)//si vers l'avant
    PORTE |=(1<<PE6);//IN1 passé à 1
  else
    PORTE &=~(1<<PE6);//IN1 passé à 0
  OCR4D = MD_EN; // rapport cyclique ou setVitesseD(MD_EN) ;
}
```

Réalisé avec LibreOffice 4.1 sous Linux