POLYTECH NANTES

Informatique Industrielle : Partie II
Microcontrôleur 8 bits

Dept. GE - D. DELFIEU

- Troisième année -

Reproduction interdite sans autorisation de l'auteur et de l'école

UNIVERSITÉ DE NANTES

POLYTECH
Premier réseau national
des écoles d'ingénieurs
polytechniques des universités

Site de la Chantrerie
Rue Christian Pauc - BP50609
44306 Nantes cedex 3 - France
Tél. +33 (0)2 40 68 32 00
Fax. +33 (0)2 40 68 32 32

Site de Gavy
Gavy Océanis - BP152
44603 St-Nazaire cedex - France
Tél. +33 (0)2 40 90 50 30
Fax. +33 (0)2 40 90 50 24

www.polytech.univ-nantes.fr

Contents

Chapter 1

Présentation générale

1.1 Introduction

Un microcontrôleur est un microprocesseur dédié au contrôle, il contient dans un même composant une unité de calcul *CPU*, comme dans un micro-processeur, mais il a, par contre la possibilité d'adresser directement des Ports d'entrées-sorties (impossible pour un microprocesseur. Il a de plus, spécifiquement, des timers, des convertisseurs analogiques, des unités de communication et de la mémoire.

Ce cours présente l'$ATMEGA_8$, microcontrôleur de la famille Arduino (Microchip anciennement Atmel) sur lequel on développera des programmes en langage *C* dans l'environnement Arduino. On n'utilisera pas cependant le langage Arduino leur préférant un structure *C* et la manipulation de registres.

La famille des $ATMEGA_8$

Modèle	Flash	EEPROM	RAM	I0	*PWM*	Interfaces	CAN
$ATMEGA_8$	8K	512	1024	23	3	SPI-USART	10 bits
$ATMEGA_{16}$	16K	512	1024	32	4	SPI-USART	10 bits
$ATMEGA_{32}$	32K	1k	2k	32	4	SPI - USART	10 bits
$ATMEGA_{64}$	64K	2k	4k	53	8	SPI - USART(2)	10 bits
$ATMEGA_{128}$	128K	4k	4k	53	8	SPI - USART(2)	10 bits
$ATMEGA_{256}$	256K	4k	8k	53	16	SPI - USART(2)	10 bits

Dans cette famille, l'$ATMEGA_8$ et l'$ATMEGA_{16}$ sont compatibles broches à broches et il n'y a que très peu de différences au niveau du code. On pourrait sur la carte de *TP* remplacer facilement un $ATMEGA_8$ par un $ATMEGA_{16}$.

1.1.1 Les différents blocs mémoire de l'$ATMEGA_8$

La mémoire de l'$ATMEGA_8$ est constituée de $1ko$ de Mémoire vive (SRAM), de 512 octets de EEPROM et de $8ko$ de mémoire FLASH.

La mémoire de type SRAM contient les registres et la pile système. La mémoire de type Static Random Access Memory (*SRAM*) est un type de mémoire vive "volatile" utilisant des bascules pour mémoriser les données. En l'absence d'alimentation les données sont perdues. On placera ces variables dans cet espace, lorsqu'elles sont partagées par le prpgramma principal et un sous-programme d'intérruption. On peut utilise alors l'attribut *volatile*. Par exemple : *volatile int i*; Cet attribut assure que la variable sera déclarée dans la SRAM, en dehors de la zone des registres spéciaux. Les accès à la variable sont plus lent, par contre, il n'y a pas de mise en cache de la variable et donc pas de problème de synchronisation (cf annexe 4.1.4)

La mémoire FLASH permet 10.000 cycles d'écriture. Elle contient le programme et les données. C'est une mémoire non volatile. Par rapport à la SRAM, elle a un des temps d'accès moins rapide, une durée de vie est assez limitée mais une consommation faible. Elle est même nulle au repos. La FLASH utilise comme cellule de base un transistor *MOS* possédant une grille flottante enfouie au milieu de l'oxyde de grille, entre le canal et la grille. L'information est stockée grâce au piégeage d'électrons dans cette grille flottante. Cette technologie se décline sous deux principales formes : *NOR* et *NAND*, d'après le type de porte logique utilisée pour chaque cellule de stockage. Dans l'$ATMEGA_8$ on a une FLASH de type *NOR*. Les mémoires de type *NAND* sont plutôt consacrées aux mémoires de masse externes telles que les Cartes SD, disque dur,...

Les mémoires de type EEPROM sont les plus chères. Elle autorisent 100.000 cycles d'écriture. Elles ont un temps d'accès un peu plus long, et donc on y stocke des données qui n'ont pas vocation a être modifiée souvent. Une autre différence avec la FLASH classique est que l'on y écrit octet par octet.

1.1.2 Fonctionnalités

Les différentes fonctionnalités sont les timers, le convertisseur analogique (ADC), les possibilités de communication, les mémoires et les *PORTS*d'entrées/sorties. Un timer peut définir des bases de temps, faire du comptage d'événements, ou bien générer des MLI ou de gérer un watchdog [1]. L'ADC permet de convertir en valeurs numériques codées sur 10 bits des tensions entre $0v$ et $5v$. Les *PORTS* permettent d'adresser et de communiquer avec des composants externes.

1.1.3 Les modes de communication de l'$ATMEGA_8$

L'$ATMEGA_8$ dispose de manière interne d'un circuit dénommé Universal Synchronous and Asynchronous Serial Receiver (*USART*). A noter qu'on entend couramment parler d'UART, mais qu'Atmel a ajouté ici un *S* pour *Synchronous*. Ce qui veut dire que cette interface peut servir à faire aussi bien de la communication série synchrone ou asynchrone, c'est à dire avec des bits de start et de stop, mais aussi synchrone dans laquelle les bits de données sont envoyés de manière cadencée par un signal de clock, piloté par un maître du protocole de communication. Elle utilise deux fils : un pour l'émission et un pour la réception.

La communication série de type *SPI* est un bus de donnée série synchrone baptisé ainsi par Motorola, et qui opère en Full Duplex. Les circuits communiquent selon un schéma maître-esclaves, où le maître s'occupe totalement de la communication. Plusieurs esclaves peuvent co-exister sur un bus, la sélection du destinataire se fait par une ligne dédiée entre le maître et l'esclave appelée *chip select*.

Le bus Serial Peripheral Interface (*SPI*) contient 4 signaux logiques :

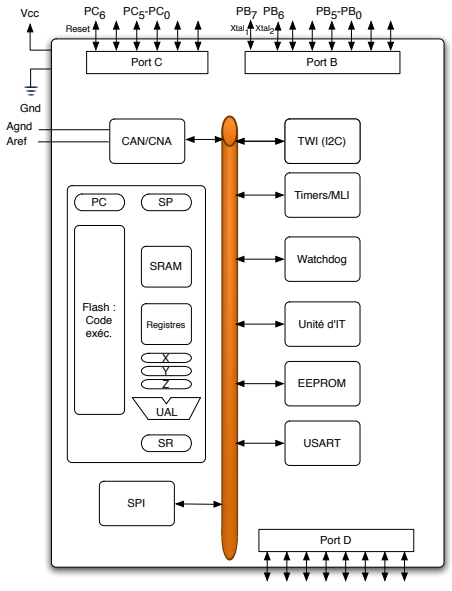
- *SCLK* : Horloge (généré par le maître)
- *MOSI* : Master Output, Slave Input (généré par le maître)
- *MISO* : Master Input, Slave Output (généré par l'esclave)
- *SS* : Slave Select, Actif à l'état bas, (généré par le maître)

La liaison *SPI* est utilisée pour la programmation de l'$ATMEGA_8$.

Grâce à une liaison série de l'$ATMEGA_8$ on peut loader un programme à exécuter dans le microcontrôleur Mais on peut aussi communiquer de façon synchrone (PI) ou asynchrone (TWI) avec des composants externes.

1.2 Architecture

L'architecture du microcontrôleur est illustrée dans la figure suivante :



[1] Système de surveillance de bon déroulement de programme : Un wachtdog est capable de détecter si un programme sort de sa boucle infinie déclenchant alors un reset qui remettra le programme dans se boucle

1.2.1 Architecture interne

Comme on l'a vu, il y a trois sortes de mémoires :

La mémoire FLASH : stocke le programme (10.000 cycles)

La mémoire SRAM (mémoire donnée) :

les 32 accus ;

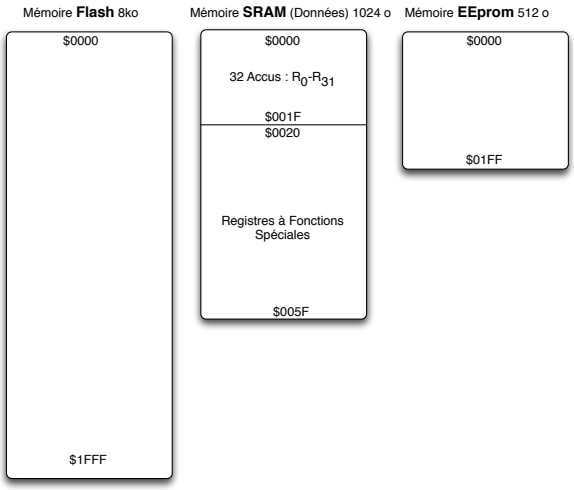
les registres à fonctions spéciales ;

la pile.

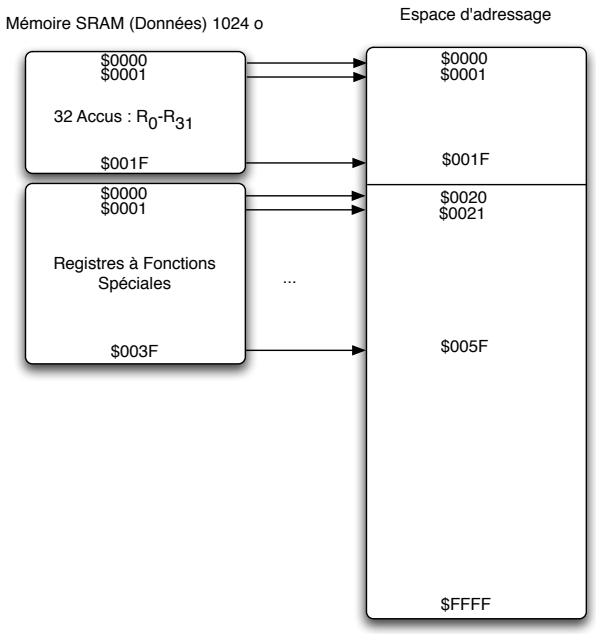
La mémoire EEPROM : on y place des données stratégiques (100 000 cycles)

Les 32 registres internes sont :

Registres	Adresse	Fonction
R_0	\$00	Accumulateur
R_1	\$01	Accumulateur
...	...	accu
R_{25}	\$19	Accumulateur
R_{26}	\$1A	X poids faible
R_{27}	\$1B	X poids Fort
R_{28}	\$1C	Y poids faible
R_{29}	\$1D	Y poids Fort
R_{30}	\$1E	Z poids faible
R_{31}	\$1F	Z poids Fort



Un mapping signifie une projection d'un plan mémoire sur l'espace d'adressage. Dans l'$ATMEGA_8$, certains plans partagent un même espace d'adressage. On distinguera alors l'accès aux variables partageant ce même espace par l'utilisation de modes d'adressage spécifiques.



1.2.2 Registres Systèmes

Ces registres agissent sur le contrôle ou indiquent l'état du processeur.

Registre *SREG*: SREG est un registre crucial dans ce micro-contrôleur. C'est lui qui surveille en permanence le microcontrôleur et positionne ces bits en fonction de la dernière opération arithmétique ou logique. Par exemple si la dernière opération donne un résultat négatif le bit N de SR passe à un.

I	T	H	S	V	N	Z	C

- C: Carry
- Z : Zero
- N : Negative
- V : oVerflow $= C_8 \oplus C_7$
- S : $V \oplus N$
- H : Half carry
- T : copy sTorage : bit tampon pour manipuler un bit
- I : autorisation générale des *interruptions* : $sei()$ – $cli()$

Registre de pile SP Ce registre permet les appels de sous-programmes et le passage de paramètres et la sauvegarde de l'état courant d'un programme. En assembleur, une pile se manipule par les instructions :

PUSH : Empile une donnée, décrémente SP

POP : Dépile une donnée, incrémente SP

Comme les piles Motorola, la pile de l'Atmel, fonctionne par adresses décroissantes. Par défaut SP contient : $0x60$ et il faudra changer cette valeur par la valeur : $0x1FF$

1.2.3 Horloges et modes des sommeil de l'$ATMEGA_8$

Les modes de sommeil : Registre *MCUCR* Ce registre définit les différents modes de sommeil dans lequel le micro peut être plongé :

SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00

Les bits de configuration des modes de sommeil SM_i:

SM2	SM1	SM0	Mode de Sommeil
0	0	0	Mode attente
0	0	1	Mode réduction de bruit pour l'ADC
0	1	0	Mode sommeil (Power down)
0	1	1	Mode économie d'énergie (Power save)
1	0	0	Réservé
1	0	1	Réservé
1	1	0	Non utilisé
1	1	1	En pause

	Domaines d'horloges actives					Oscillateurs		Sources de reveil					
Sommeil	CPU	FLA	IO	ADC	ASY	QTZ	TIM	INT	TWI	T2	EEP	ADC	IOs
Power Down								X	X				
En pause						X		X	X				
Power Save					X		X	X	X	X			
Réd. bruit				X	X	X	X	X	X	X	X	X	
Attente			X	X	X	X	X	X	X	X	X	X	X

Calibration et le contrôle de l'horloge OSCCAL

CAL7	CAL6	CAL5	CAL4	CAL3	CAL2	CAL1	CAL0

Valeur de calibrage de l'oscillateur pour la programmation de la mémoire flash ou de l'eeprom.
Suivant la configuration de certains bits, on peut utiliser les résonateurs suivants :

- Résonateur externe type céramique ou crystal
- Cristal externe basse fréquence
- Oscillateur externe ou interne de type RC
- Oscillateur calibré interne de type RC
- Horloge externe de type quelconque

La figure suivante (fig. 1.2.3) présente un montage en quartz externe basse fréquence :

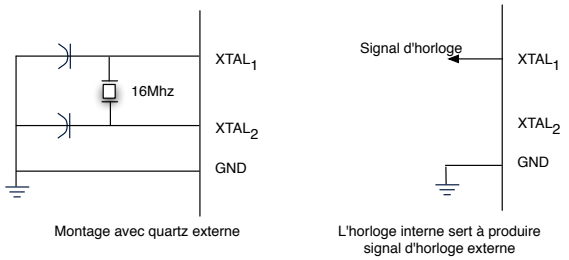


Figure 1.1: Quartz haute fréquence

On parle d'horloge temps réel lorsqu'elle permet de générer des diviseurs ou des multiples entier de la seconde. La figure suivante (fig. 1.2.3) combine le positionnement d'un quartz externe et d'une horloge temps réel :

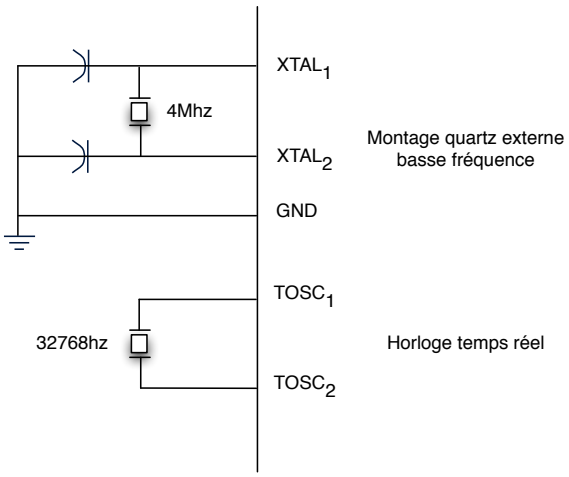


Figure 1.2: Quartz basse fréquence

La figure suivante (fig. 1.2.3) présente un montage en résonateur externe :

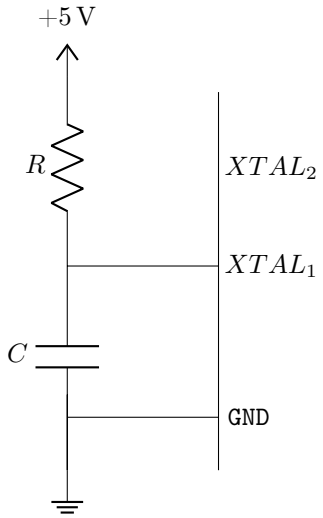


Figure 1.3: Résonateur externe

Tableau récapitulatif de tous les registres systèmes :

Addresse	Nom	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x31 (0x51)	*OSCCAL*	Registre de calibration et de contrôle de l'horloge							
0x34 (0x54)	*MCUCSR*	-	-	-	-	WDRF	BORF	EXTRF	PORF
0x35 (0x55)	*MCUCR*	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00
0x37 (0x57)	*SPMCR*	SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN
0x3D (0x5D)	*SPL*	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
0x3E (0x5E)	*SPH*	-	-	-	-	-	SP10	SP9	SP8
0x3F (0x5F)	*SREG*	I	T	H	S	V	N	Z	C

1.3 Les *interruptions* de l'$ATMEGA_8$

Un microcontrôleur fonctionne de façon normale en exécutant la boucle infinie de son programme principal. Il peut aussi être interrompu par un événement unique ou récurrent pour exécuter un sous-programme associé.

1.3.1 Le concept d'*interruption*

Une *interruption* externe correspond à la prise en compte d'un évènement dont l'occurrence est possiblement aléatoire. Lors de la survenue de cet évènement l'exécution d'un sous-programme associé est lancée automatiquement. Ces événements peuvent être par exemple :

- La fin d'un délai,
- La fin de conversion,
- Le compteur d'un timer atteint un seuil
- Un front sur une patte du microcontrôleur

Un front sur une patte peut correspondre au niveau applicatif à :

- Un arrêt d'urgence provoqué par l'appui d'un bouton ad. hoc.
- Capteur de choc ou de contact
- Dépassement de seuil sur un capteur de température, ...

D'un point de vue programmation, il serait donc peu intéressent d'utiliser des boucles d'attentes de cette alarme. Son traitement est donc réalisé en associant un sous-programme à un événement. On associe un évènement à un sous-programme par la primitive ISR.

Interruptions imbriquées : Dans la famille Arduino, une *interruption* n'est pas interruptible par défaut par une nouvelle interruption (sauf par un reset), en effet *I* de *SREG* est mis à zéro à l'entrée de l'*interruption*. Toute nouvelle interruption sera alors prise en compte lorsque l'*interruption* en cours sera terminée. Pour autoriser une nouvelle *interruption* dans l'*interruption* en cours il faut donc basculer *I* à un dans l'*interruption* en cours.

Deadlock : Il ne faut pas appeler de fonctions qui se mettent en attente d'une autre interruption. Comme l'interruption est ininterruptible par défaut, la fonction attendra indéfiniment et tout le système se bloquera. C'est ce que l'on appelle un deadlock.

Serial : Les appels aux fonctions de la bibliothèque Serial sont déconseillées dans une *interruption*.

Sources d'interruptions : Il y a de nombreuses sortes d'*interruptions* : Des *interruptions* externes, des *interruptions* internes liées aux timers, à l'ADC, ...

Plusieurs interruptions en même temps ? Les *interruptions* ont chacune une priorité. Par exemple, les interruptions externes sont plus prioritaires que les interruptions des Timers. L'Arduino exécutera les *interruptions* dans leur ordre de priorité. Dans la table ci-dessous, les priorités les plus petits numéros correspondent aux priorités les plus fortes :

Table 1.1: Tableaux des interruptions de l'$ATMEGA_8$

Priorité	Nom de l'*interruption*	Description
1	RESET	External Pin, Power-on Reset, Brown-out Reset, and Watchdog Reset
2	INT0	External Interrupt Request 0
3	INT1	External Interrupt Request 1
4	TIMER2 COMP	Timer/Counter2 Compare Match
5	TIMER2 OVF	Timer/Counter2 Overflow
6	TIMER1 CAPT	Timer/Counter1 Capture Event
7	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	TIMER1 OVF	Timer/Counter1 Overflow
10	TIMER0 OVF	Timer/Counter0 Overflow
11	SPI, STC	Serial Transfer Complete
12	USART, RXC	USART, Rx Complete
13	USART, UDRE	USART Data Register Empty
14	USART, TXC	USART, Tx Complete
15	ADC	ADC Conversion Complete
16	EE_RDY	EEPROM Ready
17	ANA_COMP	Analog Comparator
18	TWI	Two-wire Serial Interface
19	SPM_RDY	Store Program Memory Ready

1.3.2 Les *interruptions* externes INT_0 et INT_1

L'$ATMEGA_8$ offre deux *interruptions* externes sur deux pattes : PD_2 et PD_3 respectivement appelées INT_0 et INT_1. Les événements qui peuvent se produire sur ce type de patte sont soit un front montant, soit un front descendant, soit un changement de niveau.

Associer un événement à un sous-programme se fait en utilisant l'instruction ISR auquel on passe un nom d'évènement : $ISR(Nom\_evenement)$ et du code entre accolades. L'ensemble des évènements est données dans les tables de la section 4.2
Exemple :

```
ISR(INT0_vect){
  PORTD ^= 0xF0; // code associe a l'interruption
}
int main(){
  DDRD=0xF0;
  GICR |= 1<<INT0;
  MCUCR |= 1<<ISC01; // Front descendant
  sei();
  while(1){}
}
```

Si l'événement INT_0 (front sur la patte PD_2) se produit, quel que soit le comportement de la boucle infinie du programme principal (*main*), le travail est interrompu pour exécuter le sous-programme associé qui est défini entre les deux accolades du bloc *ISR*. A la fin du sous-programme d'*interruption* on revient là où l'on a été interrompu. La mise en oeuvre d'une *interruption* externe se fait grâce aux registre *GICR* et *MCUCR* :

- GICR :

INT1	INT0	-	-	-	-	IVSEL	IVCE

  - INT_0 : à un, autorise une interruption externe sur la patte PD_2
  - INT_1 : à un, autorise une interruption externe sur la patte PD_3

- MCUCR :

SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00

Considérons une *interruption* externe INT_0. Les bits ISC_{01}, ISC_{00} correspondent alors à la patte PD_2 et le niveau d'activité est défini par le tableau suivant :

- 00 : niveau bas
- 01 : front montant ou descendant
- 10 : front descendant
- 11 : front montant

Autres bits du registre *GICR*

- *IVSEL* : Interrupt Vector Select
  Quand le bit *IVSEL* est mis à zéro, les vecteurs d'interruption sont placés au début de la la mémoire flash. Quand ce bit est mis à un, les vecteurs d'interruption sont déplacés au début de la zone du boot loader de la mémoire flash. L'adresse de cette zone est modifiable par les bits "fusibles" *BOOTSZ*.
- *IVSEL* : à un, autorise le changement du bit *IVSEL*

Parmi les différents modes de fonctionnement d'un micro-contrôleur on peut citer le fonctionnement "chien de garde".

1.3.3 Le chien de garde

Un chien de garde permet de relancer/réinitialiser le programme. En effet lors d'une perturbation électromagnétique, par exemple, le déroulement du programme peut être altéré : Le compteur programme peut alors essayer d'exécuter du code dans une zone mémoire non prévue. Un chien de garde, par exemple, armé toutes les 500 millisecondes, peut alors resetter le programme et le remettre dans un déroulement normal.

Registre *MCUCSR*

-	-	-	-	WDRF	BORF	EXTRF	PORF

- *WDRF* Watchdog Reset Flag: mis à un pour activer le watchdog, raz par un reset ou par une écriture d'un 0
- *BORF* : mis à un lors d'une panne d'électricité partielle, raz par reset ou écriture de 0.
- EXTR et PORF : Détermine la source d'un reset.

1.4 Les Ports

Dans un système à base de microcontrôleur on appelle "*PORT* d'entrées-sorties", des ensembles de 8 connections entre le microcontrôleur et l'exterieur. Par ces ports, le système peut réagir à des modifications de son environnement, voire le contrôler. Elles sont parfois désignées par l'acronyme *I/O*, issu de l'anglais *Input/Output* ou encore *E/S* pour *Entrées/Sorties*. Ces *PORTS* sont programmables en entrée ou en sortie.

Des buffers sont associés aux *PORTS*, ils ont la capacité d'être à la fois source ou drain de courant ou en haute impédance Une ligne d'un *port* d'entrées est essentiellement composé d'un tampon à trois états. Ceux-ci se comportent comme des interrupteurs électroniques qui font apparaître, au moment voulu, soit deux niveaux logiques : zéro ou un et un état de haute impédance. Les niveaux logiques sont mémorisés dans un registre du processeur.

Nom	B7	B6	B5	B4	B3	B2	B1	B0
PORT D								
PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
DDRD	DDRD7	DDRD6	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
PORT C								
PINC	-	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
DDRC	-	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0
PORTC	-	PC6	PC5	PC4	PC3	PC2	PC1	PC0
PORT B								
PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
DDRB	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
PORTB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
Contrôle des Ports								
SFIOR	-	-	-	-	ACME	PUD	PSR2	PSR10

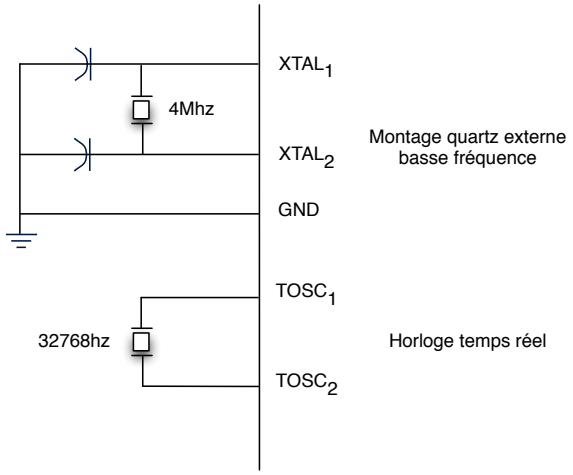


Figure 1.4: Quartz basse fréquence

1.4.1 Paramétrage et usage des *PORTS*

Pour définir le Port en Entrée ou en Sortie on utilise DDR_x $(x = B, C, D)$.

- On peux positionner des résistances de tirage sur les lignes d'un port.
- Ecriture sur le Port : Si le Port est en sortie l'écriture d'un "0" ou d'un "1" positionnera cette valeur sur le $PORT_x$. Chaque *port* de 8 bits est limité à un courant total de 200 mA.
- Lecture sur le Port : de la donnée disponible sur le Port est faite dans PIN_x . A chaque bit du registre PIN_x est associée une bascule D qui permet de de stabiliser la lecture en la synchronisant sur un front de l'horloge système. Cette stabilisation se fait au détriment d'un délai de lecture qui correspond à une période d'horloge.

1.4.2 Résistance de tirage : Pull-up resistor

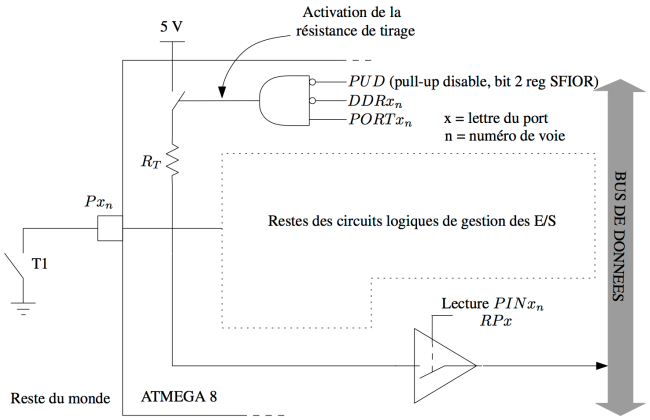


Figure 1.5: Résistances de tirage type "pull-up"

Pour poser des résistances de pull-up il est nécessaire d'avoir les trois conditions suivantes:

- Définir les lignes en entrée ;
- Ecrire un "1" sur ces lignes ;
- PUD mis à zéro dans SFIOR : $SFIOR \ \&= \ (1 << PUD)$;

Résistances de tirage : le registre $SFIOR$

ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10

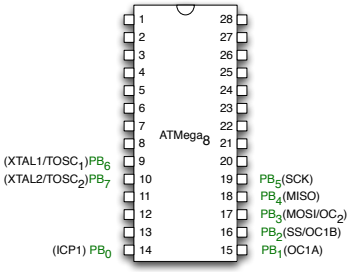
PUD="Pull Up Disable"

- 1: Les résistances de tirages sont désactivées, les entrées sont en mode 3-états.
- 0 : Les résistances de tirages sont activées.

1.4.3 Registres de manipulation des ports : $DDR_x, PIN_x et PORT_x$

- DDR_B, DDR_C, DDR_D : Permet de programmer le sens des lignes (1 : sortie, 0 entrée)
- $PORTB, PORT_C, PORT_D$: Mémorise l'état des pins, permet l'écriture ou la lecture Ou bien il permet de configurer la résistance de tirage en entrée.
- PIN_B, PIN_C, PIN_D : Permet de lire les valeurs courantes des pattes du port

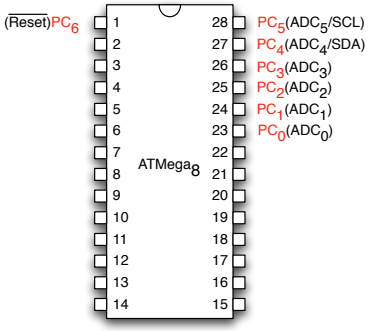
1.4.4 $PORTB$



- DDR_B : Programme les lignes du port B

- PIN_B : Lecture du port B
- $PORT_B$: Ecriture sur le port B

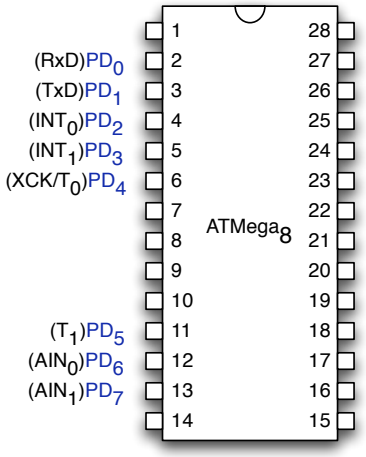
1.4.5 $PORTC$



- DDR_C : Programme les lignes du port C
- PIN_C : Lecture du port C
- $PORT_C$: Ecriture sur le port C
- $PORTC$ est un port d'entrée-sortie bidirectionnel de 7 bits avec des résistances de pulll up.
- Le buffer de sortie du $PORTC$ a la capacité d'être à la fois source ou drain de courant.
- En entrée les pattes sont tirés à la masse si les résistances de pull-up sont activées.
- Les pins de C sont en 3-états quand un reset survient.
- $PORTC_6 = \overline{RESET}$ si le fusible $RSTDISBL$ est à 1, alors la ligne est disponible en E/S

ç!oi

1.4.6 $PORTD$



- $DDRD$: Programme les lignes du port C
- $PIND$: Lecture du port D
- $PORTD$: Ecriture sur le port D

1.4.7 Lecture et écriture sur un port

Ecriture sur un Port

- Pour écrire sur un *port* on déclare une ou plusieurs ligne en sortie par un registre.
  Exemple : $DDRD = 0xFF$; Tout le *port* est déclaré en sortie
- Ensuite on réalise une écriture sur les lignes déclarées en sortie par le registre $PORTD$
  Exemple : $PORTD = 0b00000011$; On active les lignes 0 et 1

Exemple d'écriture d'un Port :

```
int main(void) {
  DDRD  = 0xFF;     // port D en sortie
  PORTD = 0x7A;      //Allume leds 1,3,4,5,6
  while(1){        //boucle infinie
    PORTD ^= 0x7A;  // ^= : ou-exclusif fait clignoter les leds
    _delay_ms(20);
  }
}
```

1.4.8 Lecture d'un Port

- Pour lire sur un *port* on déclare une ou plusieurs ligne en entrée par le registre DDR_x.
  Exemple : $DDRD = 0x00$; Tout le *port* D est déclaré en entrée
- Ensuite on réalise une lecture sur les lignes déclarées en sortie par le registre PIN_x dans une variable de type entier.
  Exemple : $int\ lu = PIND$; On active les lignes 0 et 1

Exemple structure et d'écriture d'un Port

```
int main(void) {
        DDRC = 0x00; // \pc\ en entree
  DDRD  = 0xFF;     // port D en sortie
  int lu;
  while(1){        //boucle infinie
    lu = PINC,
    PORTD = lu;  //PORTD recoit PORTC
    _delay_ms(20);
  }
}
```

1.4.9 Fonction de manipulation de bits

```
// Mise a un d'un bit sans affecter les autres bits
PORTD = PORTD | (1<<PORTD4);
PORTD |= (1<<4); // mise a un du bit 4

// Mise a un des 4 bits de poids faible sans affecter les autres bits
PORTB = PORTB | 0x0F // ou
PORTB |= 0x0F;

// Mise a zero d'un bit sans affecter les autres bits
PORTB &=~(1<<PB3);  // mise a zero du bit 3

// Mise a zero de 4 bits sans affecter les autres bits
PORTB &= 0x0F;  // mise a zero des 4 bits de poids Fort

// Commutation du bit 4 sans affecter les autres bits
PORTD = PORTD^0x10;
PORTD ^= 0x10;
```

Chapter 2

Conversion Analogique

2.1 Le Comparateur Analogique

2.1.1 Fonctionnement global

Cet élément offre la possibilité de comparer deux valeurs analogiques sur les pattes PD_6 et PD_7. $AIN_0(PD_6)$ est appelé broche positive et AIN_1 (PD_7) broche négative. Lorsque $AIN_0 > AIN_1$ alors $AC0 \leftarrow 1$, on peut alors déclencher une *interruption* (avec bit $ACIE = 1$) de comparaison.

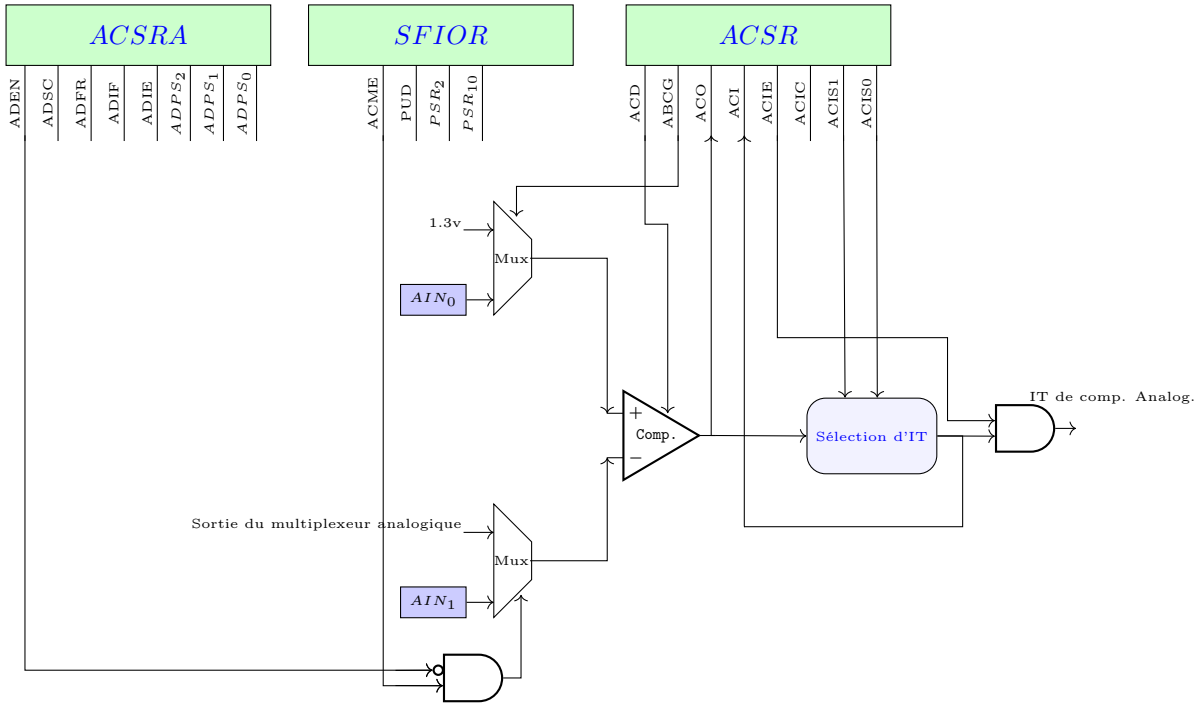


Figure 2.1: Horloge de l'ADC

On a de plus, la possibilité de changer la référence "négative" : AIN_1 par l'une des broches $ADC_0, ADC_1, ..., ADC_5$ une des sorties du multiplexeur analogique. Les bits $ACIS1, ACIS0$ peremettent à l'utilisateur de sélectionner comme évènement de déclenchement, un front montant, un front descendant ou une inversion.

En outre, toute entrée $ADC_5, ..., ADC_0$ peut jouer le rôle de AIN_1. Pour réaliser cela il faut positionner à un le bit de multiplexage $ACME$ du registre $SFIOR$ quand l'ADC est OFF ($ADEN$ à zéro) les bits $MUX2, MUX1, MUX0$ du registre $ADMUX$ permettent de modifier la source AIN_1.

Les bits MUX qui remplacent AIN_1 selon le tableau suivant :

$ACME$	$ADEN$	$MUX2:0$	$AIN0$
0	x	xxx	$AIN1$
1	1	xxx	$AIN1$
1	0	000	ADC_0
1	0	001	ADC_1
1	0	010	ADC_2
1	0	011	ADC_3
1	0	100	ADC_4
1	0	101	ADC_5
1	0	110	ADC_6
1	0	111	ADC_7

2.1.2 Les registres

Registre ACSR: Analog Comparator Status Register ACSR

ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

- ACD Analog Comparator Disable : Bit de mise en marche du comparateur analogique,

  0 : mise en marche,

  1 : arrêt

- ACO Analog Comparator Output : Contient le résultat de la comparaison : si $VAIN_0 > VAIN_1$ alors $ACO = 1$.
- ACI : Analog Comparator Interrupt : Flag Bit de demande d'interruption (bit raz ds le sp d'IT)
- Condition IT : $ACIS1$ et $ASCI0$). Ce bit est remis à 0 automatiquement après le traitement de l'interruption
- Masque d'IT : $ACIE$.
- $ACIE$ Analog Comparator Interrupt enable : Bit de validation de l'interruption ANA_COMP.
- $ACIC$: Analog Comparator Input Capture Enable : La mise à 1 de ce bit connecte la sortie du comparateur à l'entrée de capture du Timer1.
- $ACIS1$ et $ACSI0$ gère le comportement de la sortie AC_0 :

Activation de AC_0	ACIS1	ACIS0
$1 \rightarrow 0$ *ou* $0 \rightarrow 1$	0	0
non utilisé	0	1
front montant	0	0
front descendant	1	1

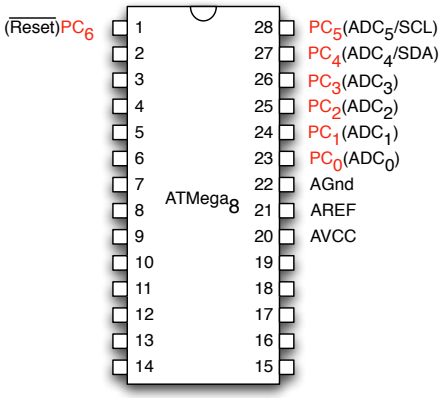
Registre SFIOR

ADTS2	ADTS1	ADTS0	-	ACME	PUD	PSR2	PSR10

$ACME$ Analog Comparator Multiplexer Enable

- Quand $ACME$= 1 et ADC est éteint ($ADEN = 0$ et $ADSC = 0$) alors le multiplexeur ADC choisit l'entrée négative du comparateur analogique.
- Quand $ACME$= 0 alors AIN_1est appliqué à l'entrée négative du comparateur analogique.

2.2 Le Convertisseur Analogique-Numérique : ADC



Le convertisseur (ADC) convertit une tension d'entrée analogique en une valeur à 10 bits digitale par approximations successives. Il possède 6 entrées simultanées avec une non-linéarité inférieure à $+/-2$ LSB avec une erreur à 0 V inférieure à 1 LSB. Le résultat de la conversion est positionné dans les registes Analog to Digital Converter High ($ADCH$) et Analog to Digital Converter Low ($ADCL$).

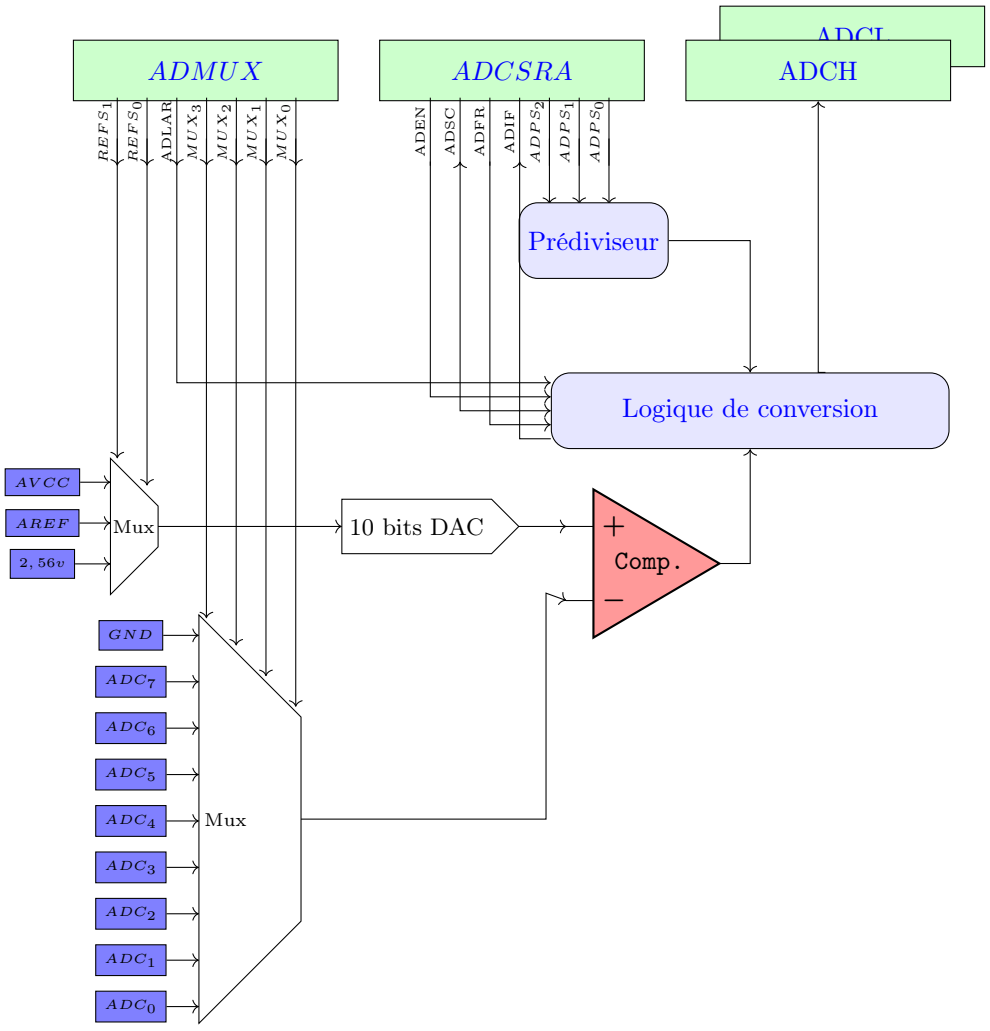


Figure 2.2: Horloge de l'ADC

Le temps de conversion prend au minimum 13 cycles d'horloge. De plus, le convertisseur a une alimentation découplée du micro : $AVcc$ tq $AVcc \in [Vcc - 0,3, Vcc + 0,3]$: Vcc est la tension de référence qui peut être externe ou interne : $AREF$ ou $AVcc$ ou $= 2,56\ v$.

2.2.1 Fonctionnement et caractéristiques

On peut le programmer en générant une interruption de fin de conversion. De plus, il est possible du faire fonctionner en limitant le bruit en mode de sommeil. Le résultat d'une conversion numérique sur 10 bits est donné par la relation :

$$\textbf{Résultat numérique} = \textbf{P. Entière}(\textbf{Tension d'entrée}/\textbf{Tension de référence} * 1024)$$

Ou la tension de référence est soit $AREF$ soit AV_{cc} soit 2,56v.

```
ADC_0= 2.5 V, AREF = 3.3 V
ADCH-ADCL = PE(2.5/3.3 x 1024) = 774
```

Réduction des bruits lors de la conversion Pour réduire au maximum la précision de la conversions on pourra :

- Mettre en sommeil l'unité centrale avant le lancement d'une conversion.
- Découpler soigneusement l'alimentation AV_{cc} avec des condensateurs.
- Règles élémentaires du routage : connexions courtes, plan de masse, ...
- Effectuer un filtre numérique des résultats (amortissement ,moyenne, ...).

La valeur minimale de conversion est GND tandis que a valeur maximale est soit la tension sur la broche $AREF$ soit le tension sur la broche AV_{cc} ou bien une tension interne de 2,56 V (cf bits $REFS_n$).

Chacune des 6 broches d'entrée ADC peuvent être choisies comme des entrées simples de l'ADC . De plus, l'ADC contient un mécanisme d'échantillonneur-bloqueur qui assure que l'entrée est tenue constante pendant 1,5 cycle d'horloge.

$ADCL$ doit être lu en premier puis $ADCH$ pour assurer la cohérence des données qui appartiennent à la même conversion. Une fois $ADCL$ lu, l'accès aux registres de commandes est bloqué afin d'empêcher une nouvelle conversion tant que $ADCH$ n'est pas lu. Quand $ADCH$ est lu, l'ADC est à nouveau opérationnel. L'ADC a sa propre interruption qui peut être déclenchée quand une conversion est achevée. La lecture de l'$ADCL$ et d'$ADCH$ interdit l'accès aux registres de commande de l'ADC, mais si une *interruption* de fin conversion se produit alors que la lecture précédente n'a pas été encore faite le résultat sera perdu. Il faut donc faire attention à lire rapidement un résultat de conversion.

Programmation

L'ensemble des registres à programmer l'ADC est $ADMUX, ADCSRA, ADCH, ADCL$, étudions d'abord $ADCSRA$:

ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

- L'ADC est activé avec le bit $ADEN$.
- La référence de tension et le choix du canal d'entrée n'entrera pas en vigueur quand $ADEN$ est mis à 1, il faut d'abord désactiver $ADEN$.
- ADC produit un résultat sur 10 bits qui est présenté dans les registres $ADCH$ et $ADCL$.
- Par défaut, le résultat est présenté ajusté à droite, mais peut facultativement être présenté ajusté à gauche en mettant le bit $ADLAR$ à un dans $ADMUX$. Pour un résultat sur 8 bits et un ajustement à gauche, la lecture du registre $ADCH$ est suffisante.

Cycle de conversion

Début d'une conversion Une conversion simple est lancée en positionnant $ADSC$ à 1.

Si l'on change le canal tandis qu'une conversion est en cours, l'ADC finira la conversion actuelle avant l'exécution du changement de canal.

Fin de conversion Dès lors, qu'une conversion est en cours, *ADSC* reste à un tant que la conversion se réalise et il redescend à 0 quand la conversion est achevée. Quand une conversion est finie, le résultat est écrit dans les registres de données *ADCH* et *ADCL* et *ADIF* est mis à 1. Le programme peut alors lancer *ADSC* de nouveau et une nouvelle conversion sera amorcée sur le premier front montant de l'horloge.

Durée d'une conversion Une conversion normale prend 13 cycles d'horloge.

2.2.2 Les différentes méthodes de programmation d'une conversion

Conversion échantillonnée

Pour réaliser une conversion échantillonée, on doit utiliser un timer qui va définir une période d'échantillonnage. On va associer à une *interruption* de débordement de ce timer dans laquelle on va déclencher la conversion analogique-numérique. Un exemple de codage est donnée dans la section suivante (cf section 2.2.3).

Free Running mode : Mode de fonctionnement libre

Dans ce mode, on n'utilise pas d'interruption et l'ADC échantillonne en permanence, sans aucune action du programmeur autre que le lancement initial, et la mise à jour les registres de données (*ADCH* et *ADCL*) est automatique. Ce mode est positionné par la valeur 1 dans $ADFR \in ADCSRA$. La première conversion doit être lancée par $ADSC \in ADCSRA$. Dans ce mode l'ADC effectue des conversions sans se préoccuper du flag d'*interruption* *ADIF*. Un exemple de codage est donnée dans la section suivante (cf section 2.2.3).

Conversion avec attente active

Pour ce mode on va définir une fonction structure qui retournera un entier image de la conversion. L'attente active est du au fait que, après avoir lancée la conversion par *ADSC*. Un exemple de codage est donnée dans la section suivante (cf section 2.2.3).

Aspects numériques Par défaut, la fréquence d'horloge d'entrée est entre 50 kHz et 200 kHz pour obtenir la résolution maximale. Si une résolution plus basse que 10 bits est nécessaire, la fréquence d'horloge d'entrée de l'ADC peut alors être plus haute que 200 kHz.

Temps de conversion Le temps de conversion qui est égal à 13 fois l'horloge système, peut, de plus, être multiplié, par un facteur de pré-division (cf bits $ADPS_{2-0}$). Le pré-diviseur produit une fréquence d'horloge acceptable pour l'ADC à partir de celle du Control Process Unit (*CPU*). La mise en marche pré-diviseur se fait en positionnant une valeur sur les bit $ADPS \in ADCSRA$ et le pré-diviseur commencera à compter dès que l'ADC est allumé en mettant le bit *ADEN* à 1 .

Les bits $ADPS2, ADPS1, ADPS0$ sélectionnent l'horloge :

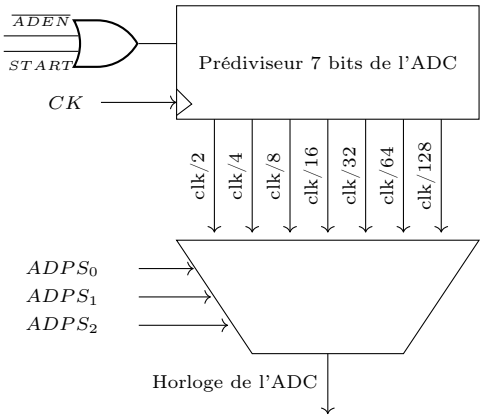


Figure 2.3: Horloge de l'ADC

2.2.3 Les Registres de l'ADC : ADMUX, ADCSRA, ADCH, ADCL

- *ADMUX*

$REFS_1$	$REFS_0$	$ADLAR$	-	MUX_3	MUX_2	MUX_1	MUX_0

  - $REFS_1, REFS_0$

$REFS1$	$REFS0$	Tension de Référence
0	0	AREF
0	1	AV_{cc} avec capacité externe sur AREF
1	0	non utilisé
1	1	$2,56\ v$

  - *ADLAR* : ADC Left Adjust Result Ajustement à gauche à 1 ou à droite à 0 du résultat dans le registre *ADCL* et *ADCH*.
  - $MUX_{3,2,1,0}$: Choix du canal ADC .

- *ADCSRA*

ADEN	ADSC	ADFR	ADIF	ADIE	$ADPS_2$	$ADPS_1$	$ADPS_0$

  - *ADEN* : (AD ENable) Mise en marche du convertisseur avec la mise à 1 du bit, l'arrêt avec la mise à 0, la conversion en cours sera terminée.
  - *ADSC* : (AD Start Conversion) Lancement de la conversion de la voie sélectionnée (retourne à 0 en fin de conversion). En mode simple conversion, il faut remettre à 1 à chaque nouvelle conversion. En mode libre, la première conversion dure 25 cycles puis les suivantes 15, il n'est pas nécessaire de remettre le bit à 1 à chaque conversion.
  - *ADFR* : (Analog Digital Free Running mode) La mise à 1 de ce bit permet de mettre en le convertisseur en mode conversion libre : mode de fonctionnement ou les conversions ont lieu en permanence sans avoir besoin dus re-lancer.
  - *ADIF* : (AD Interrupt Flag) Passe à 1 une fois la conversion terminée et déclenche l'interruption si *ADIE*= 1. Ce bit repasse automatiquement à 0 lors du traitement de la routine d'interruption.
  - *ADIE* : "AD Interrupt Enable" : Validation de l'interruption du convertisseur.
  - $ADPS_2, ..., ADPS_0$: Bits de Sélection du facteur de pré-division de l'horloge interne du convertisseur en fonction du quartz (cf figure 2.3)

- *ADCH* − *ADCL*:

  - Avec $ADLAR = 0$: On cherche un résultat sur 10 bits

-	-	-	-	-	-	ADC_9	ADC_8
ADC_7	ADC_6	ADC_5	ADC_4	ADC_3	ADC_2	ADC_1	ADC_0

```
int L   = ADCL;
int H   = ADCH;
int res = (H<<8)+L;
```

  - Avec $ADLAR = 1$: On cherche un résultat sur 8 bits en laissant tomber *ADCL*

ADC_9	ADC_8	ADC_7	ADC_6	ADC_5	ADC_4	ADC_3	ADC_2
ADC_1	ADC_0	-	-	-	-	-	-

```
int res = ADCH;
```

Programmes relatifs à l'ADC

-1- Mode scrutatif : Attente active. La led qui doit clignoter sur PB_0.

```
volatile int lu; // variable globale

void lecture_analogique_scrutative(){
  ADCSRA |= (1<<ADSC);
  while (ADCSRA & (1<<ADSC)) {};
  lu=ADCH;  // lecture sur 8 bits
}

int main(void){
  DDRB=0x1F; DDRC=0x00;  // C : entree, B : Sortie
  ADMUX=(1<<ADLAR);  //ajust Gauch, PC0 entree analog.
  ADCSRA= (1<<ADEN);  // Mise On ADC
  do {
    _delay_ms(200);
    lecture_analogique_scrutative();
  } while(1);
  return(0);
}
```

-2- Mode échantillonné : On réalise ici, un déclenchement échantillonée par le timer 1, de la conversion analogique

```
volatile int F;

ISR(ADC_vect){char L=ADCL;F=ADCH; }

ISR(TIMER1_OVF_vect) {ADCSRA |=  (1<<ADSC);PORTB^=1;}

int main(void){
  DDRB=0x01;DDRC=0x00;
  ADMUX=(1<<ADLAR);  // ajust Gauche, Lecture sur PC0
  ADCSRA= (1<<ADEN)+(1<<ADIE);  // mise ON du can, IT CAN
  TCCR1A=0;TCCR1B = (1 << CS11); // prediv du \timer\ Fcpu/8
  TIMSK = (1<<TOIE1); // Validation It de debordement */
  sei(); // toute les its autorisees
  do {
    PORTB^=1;_delay_ms(50);
  }while(1);
  return(0);
}
```

-3- Free Running : On lance l'ADC une fois puis on lit à la volée *ADCH* et *ADCL*.

```
int main(void){
    int H,L;
    DDRB=0x01; DDRC=0x00;
    ADMUX=(1<<ADLAR);
    ADCSRA= (1<<ADEN) + (1<<ADSC)+ (1<<ADFR);
     do {
        PORTB^=1;
        delai(50);
        L=ADCL; H = ADCH;
        Res = (H<<8)+L;
     } while(1);
  return(0);
}
```

-4- Mode boggué : Sans timer relance de l'ADC dans l'*interruption* de conversion.

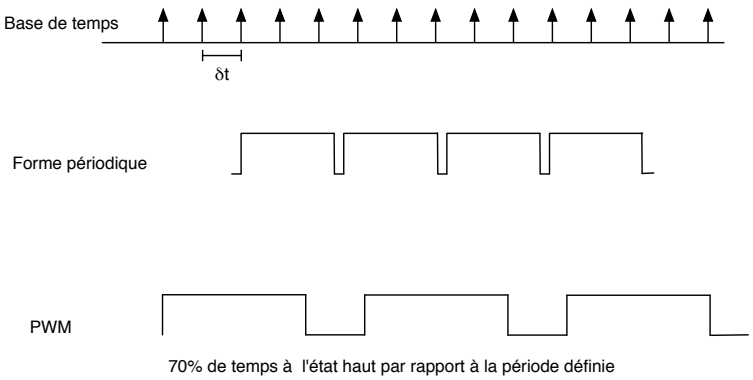
```
volatile int F;

ISR(ADC_vect){F=ADCH; ADCSRA |=  (1<<ADSC)} // Aie !!!

int main(void){
  DDRB=0x01;DDRC=0x00;
  ADMUX=(1<<ADLAR);  // ajust Gauche, Lecture sur PC0
  ADCSRA= (1<<ADEN)+(1<<ADCS)+(1<<ADIE);  // mise ON du can, IT
      CAN
  sei(); // toute les its autorisees
  do {
    PORTB^=1;_delay_ms(50);
  }while(1);
  return(0);
}
```

Chapter 3

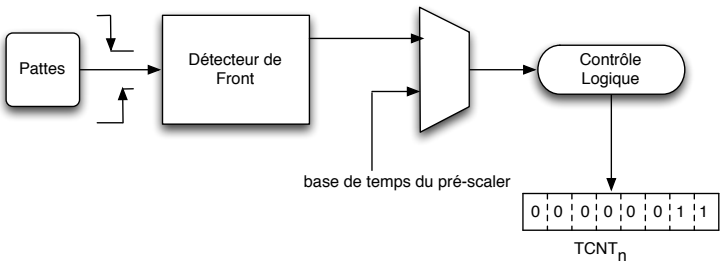
Les Timers

Les fonctions d'un timer concernant d'abord les fonctions de temporisation avec la définition de base de temps, la génération de formes (signaux carrés, *MLI*) ansi que le comptage d'événements. Un timer permet aussi de mesurer un temps entre deux événements.

Dans la figure suiva,nte nous illustrons une base de temps et la génération de formes :



Dans cette figure on décrit le comptage ou la capture d'événements :



L'$ATMEGA_8$ possède 3 timers : timer 0, timer 1 et timer 2. Les timers 0 et 2 sont des timers 8 bits tandis que les timers 1 est un timer 16 bits. Le timer 2 est plus élaboré que le timer 0, il permet en plus de faire de la *MLI* et de produire une *interruption* de comparaison.

3.1 Le timer 0

3.1.1 Caractéristiques

Ses fonctions de base

- timer à sortie unique

- Générateur de fréquences
- Comptage d'événements externes
- Pré-diviseur d'horloge 10 bits
- Le timer 0 se manipule à l'aide de seulement 3 registres :
  - $TCNT_0$
  - $TCCR_0$
  - $TIMSK$

Aperçu global du timer 0 Le timer 0 est un timer 8 bits. Un débordement provoque l'évènement $TIMER0\_OVF\_vect$ et la mise à un du drapeau TOV_0 (dans le registre $TIFR$) et possiblement une *interruption* se produit. Le masquage ou l'autorisation de cette *interruption* est réalisé par le bit $TOIE_0$ du registre $TIMSK$. Comme le montre le schéma précédent $TCNT_0$ s'incrémente suivant :

- La patte externe timer 0
- Ou le pré-diviseur d'horloge. Ce choix étant fait par les bits $CS0_{2:0} \in TCCR_0$

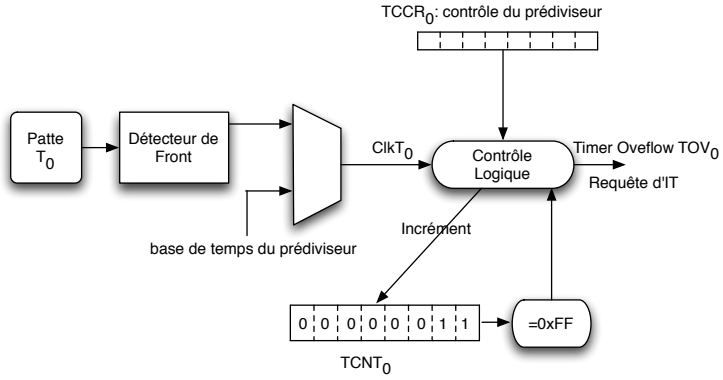


Figure 3.1: Aperçu de timer 0

- Le bloc logique ordonne alors l'incrément du registre TCNT0.
- Lorsque $TCNT_0$ atteint la valeur $0xFF$ il repasse à la valeur 0 et dans le même temps le bit TOV_0 passe à un.
- TOV_0 agit comme un $9^{\text{ième}}$ bit.
- Si une *interruption* a été mise en place alors ce bit est **raz** lors de l'exécution du sous-programme d'interruption associé.

3.1.2 Les registres associés au timer 0 : TCCR0, TCNT0

- $TCCR_0$

-	-	-	-	-	$CS0_{02}$	$CS0_{01}$	$CS0_{00}$

$CS02$	$CS01$	$CS00$	*Description*
0	0	0	T_0 en pause
0	0	1	$clk_{I/O}$: Horloge Système
0	1	0	$clk_{I/O}/8$
0	1	1	$clk_{I/O}/64$
1	0	0	$clk_{I/O}/256$
1	0	1	$clk_{I/O}/1024$
1	1	0	source externe : front descendant sur la patte $T0$
1	1	1	source externe : front montant sur la patte $T0$

Remarque 3.1.1 *Si par ailleurs, la patte est utilisée en entrée Alors un front sur cette patte affectera quand même timer 0 si les modes* 6 *ou* 7 *ont été choisis.*

- $TCNT_0$: Accessible en lecture ou écriture : Registre de comptage courant.
- $TIMSK$

$OCIE_2$	$OCIE_2$	$TICIE_1$	$OCIE_{1A}$	$OCIE_{1B}$	$TOIE_1$	-	$TOIE_0$

  La condition d'autorisation de l'*interruption* de débordement de timer 0 est $TOIE_0 = 1$ ET $I = 1 \in SR$). Sur un débordement de $TCNT_0$, on a :

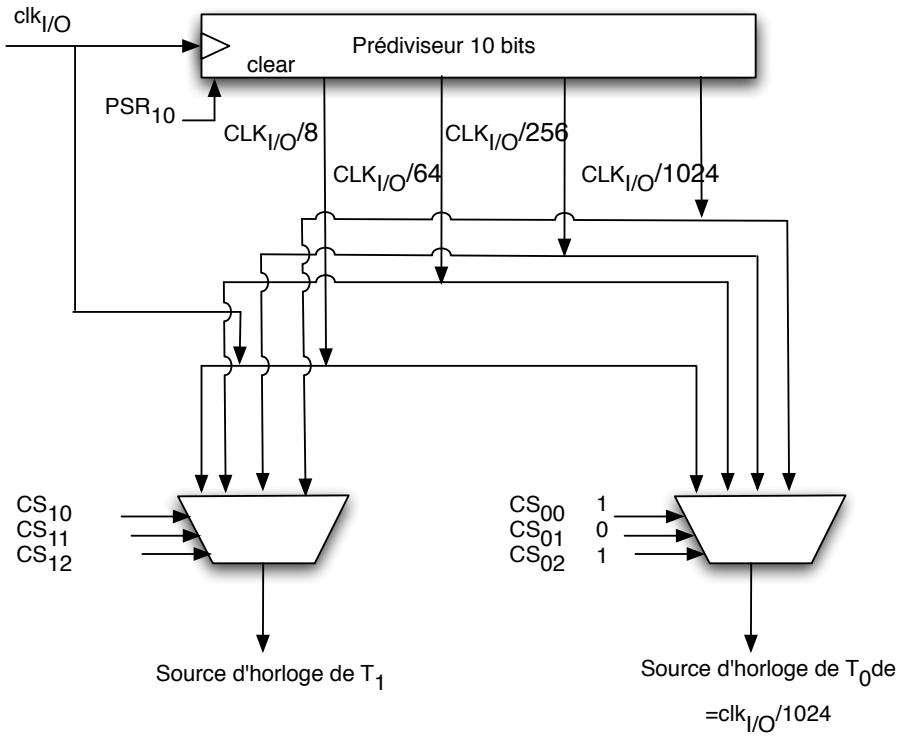
  – $TOV_0 \in TIFR$ qui passe à un, **et** le programme associé à l'*interruption* est alors exécuté.

  – TOV_0 est alors automatiquement **raz** dés le début du sous-programme d'IT.

  Sinon en mode non interruptif, c.a.d. en mode scrutatif il faut remettre a zéro ce bit **EN Y ECRIVANT UN "1"**.

3.1.3 Le prédiviseur

Les timers timer 0 et timer 1 partagent le même module de pré-division, mais ceux-ci peuvent avoir des fréquences différentes de pré-divisions par rapP à la source d'Horloge Interne. Le Timer/Compteur peut être cadencé directement par l'horloge de système (avec $CS0_{2:0} \in TCCR_0$ (timer 0) ou $CS1_{2:0} \in TCCR_1$ à 1) permettant une fréquence d'horloge maximale : $clk_{I/O}$. Autrement, on pourra choisir un facteur de pré-division suivant : $clk_{I/O}/8, clk_{I/O}/64, clk_{I/O}/256, clk_{I/O}/1024$.



Il y a la possibilité d'échantillonner la source externe à travers un détecteur de front qui induira alors un retard de 2.5 à 3.5 cycles d'horloge.

SFIOR

$ADTS_2$	$ADTS_1$	$ADTS_0$	-	ACME	PUD	PSR_2	PSR_{10}

- Quand $PSR_{10} \leftarrow 1$ le prédiviseur de timer 0 et timer 1 est **raz**.
- Attention ce prédiviseur est partagé par timer 0 et timer 1, et un **reset** du prédiviseur les affectent tous les 2.

Exemple d'utilisation du timer zéro avec la mise en place d'une interruption de débordement :

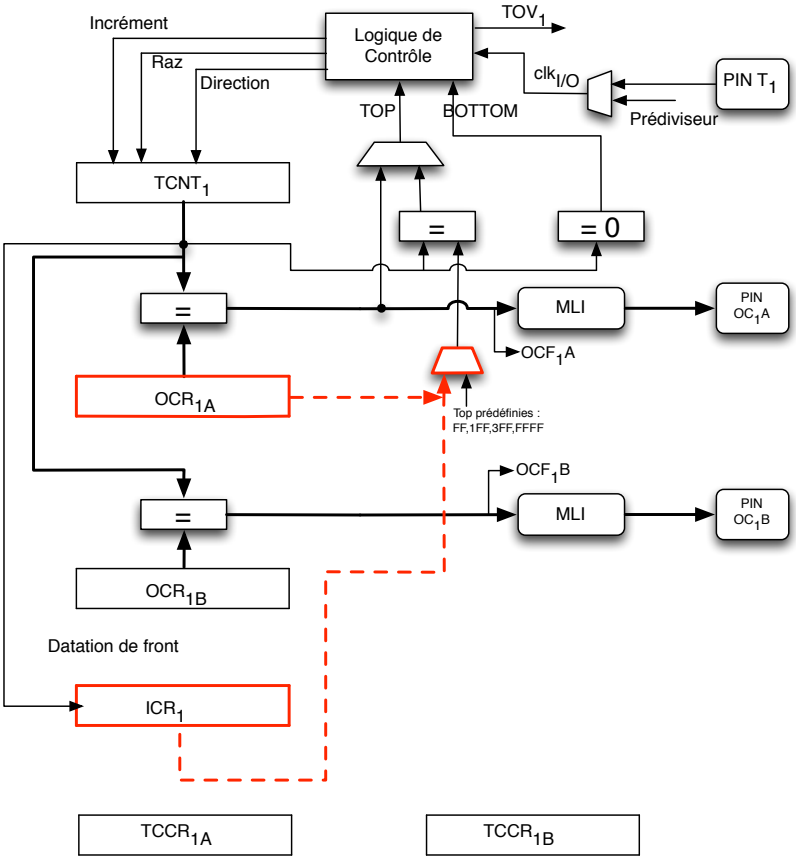
```
ISR (TIMER0_OVF_vect) {
  static int Compteur;
  if (Compteur++ == 50) {
    Compteur=0;
    PortB^=1  // permet de mesurer la periode
  }
}

void configTimer0(){
  TCCR0 = (1<<CS02) + (1<<CS00) ; // clkio/1024
  TIMSK = 1<<TOIE0; // Autorisation IT de debordement
}

int main() {
  DDRB=0xFF;
  configTimer0();
  sei();                // autorise ttes les interruptions
  while(1);
}
```

3.2 Le timer 1

3.2.1 Caractéristiques générales

Ses fonctions de base

- timer 16 bits ;
- 2 sorties indépendantes de comparaison ;
- Une entrée de capture avec réduction de bruit ;
- 16 modes de *PWM*
- Générateur de fréquence
- Compteur externe d'événements
- 4 sources d'IT : TOV_1, OCF_{1A}, OCF_{1B}, ICF_1.



Les registres 16 bits

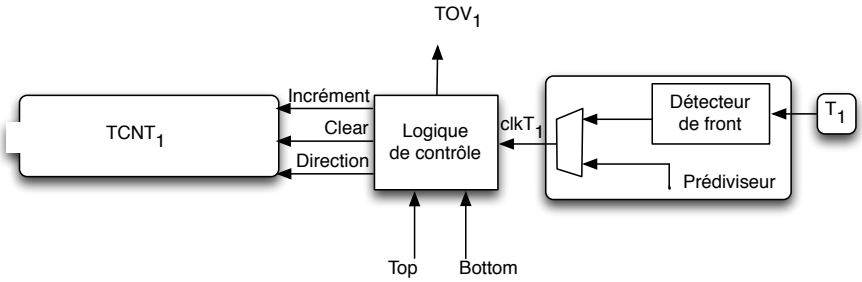
- le registre de comptage $TCNT_1$
- les registre de sortie de comparaison : OCR_{1A} - OCR_{1B}
- le registre de capture d'entrée ICR_1

les registres 8 bits

- Les registres de contrôle du timer $TCCR_{1A}$ -$TCCR_{1B}$
- le registre de sortie de comparaison : OCR_{1A} - OCR_{1B}
- le registre de capture d'entrée ICR_1

Les interruptions du timer 1 : Tous les signaux d'IT sont visibles depuis $TIFR$. Toutes les *interruptions* sont masquables individuellement dans $TIMSK$

Contrôle de $TCNT_1$: Le timer 1 peut être fréquencé de façon interne par le prédiviseur ou par une source externe patte timer 1. Le bloc logique détermine comment la source est utilisée pour incrémenter ou décrémenter $TCNT_1$. Le timer 1 est inactif quand il n'y a aucune source. La source $clkT_1$ de la figure (en haut à droite de la figure 3.2.1) est sélectionnée par $CS_{1_{2:0}} \in TCCR_{1B}$.



Comparaison de seuil : Les deux registres de comparaison OCR_{1A} - OCR_{1B} sont utilisés comme des seuils qui sont comparés avec $TCNT_1$. Le résultat ces comparaisons peut être utilisé pour générer une forme de type PWM ou bien une fréquence variable sur les pattes $OC1_B, OC1_B$. Ces comparaisons d'égalité vont positionner les flags OCF_{1A} ou OCF_{1B} qui peuvent alors être utilisé pour générer l'*interruption* correspondante.

La capture : Le registre de capture d'entrée ICR_1 peut capturer un événement (front) sur la patte ICP_1 Cette partie inclue un filtre réducteur de bruit évitant la capture de fronts parasites.

Le choix de la valeur maximum : La valeur TOP est soit le maximum du timer ($0xFFFF$) ou bien une valeur définie dans ICR_1 ou OCR_{1A}. L'utilisation de OCR_{1A} pour définir la valeur TOP, bloque la génération de PWM pour la sortie $OC1A$. Lorsque l'on utilise ICR_1 pour définir la valeur TOP les deux sorties PWM sont alors disponible (OCR_{1A} et OCR_{1B}), par contre l'entrée de capture n'est plus disponible.

$BOTTOM$	$BOTTOM = 0x0000$
MAX	$MAX = 0xFFFF$
TOP	$TCNT_1$ atteind TOP c.a.d quand il atteint soit : – $0xFF$, $0x1FF$, $0x3FF$ – $OCR1A$, ou bien ICR_1

Fonctionnement de l'Unité de Comptage

Le registre $TCNT_1$ est piloté par le bloc logique via les signaux *clear, increment ou decrement* et part la source $clkT_1$. $clkT_1$ peut être généré par une source externe ou interne selon les bits $CS12:0$. Quand $CS12:0 = 0$, le timer 1 est arrêté, la valeur de $TCNT_1$ étant toujours accessible par le CPU. Une écriture par le CPU sur $TCNT_1$ à la priorité sur toute autre opération.

La séquence de comptage peut alors déterminer une forme d'onde (waveform) sur la patte $OC1A$ selon les bits $WGM1_{3:0}$ des registres $TCCR_{1A}$ et $TCCR_{1B}$. Les formes d'onde dépendent en fait des modes de comptage (cf section 3.2.3) sur $TCNT_1$. Le bit de débordement TOV_1 dépend aussi du mode choisi sur $WGM1_{3:0}$.

Unité de capture

L'unité de capture peut capturer des événements externes en leur attachant une étiquette temporelle [1] attachée à cette occurrence. Le signal externe indiquant l'occurrence d'un ou de plusieurs évènements est disponible sur la patte ICP_1 ou par le comparateur analogique. Les étiquettes temporelles peuvent être utilisées pour calculer une fréquence, un rapport cyclique ou d'autres. Ces étiquettes temporelles sont parfois utilisées pour créer un "log" d'évènements.

Aperçu de l'Unité de Capture Quand un événement (front) se produit (cf figure 3.2.1) sur $ICP1$, ou sur ACO, ce signal passe par le réducteur de bruit et le détecteur de front. Si il passe cet étage, il déclenche l'écriture de $TCNT_1$ dans ICR_1, $TCNT_1$ constitue alors ce que l'on appelle une étiquette temporelle. Dans le même temps le drapeau ICF_1 passe à un.

[1] qui est la valeur courante de $TCNT_1$

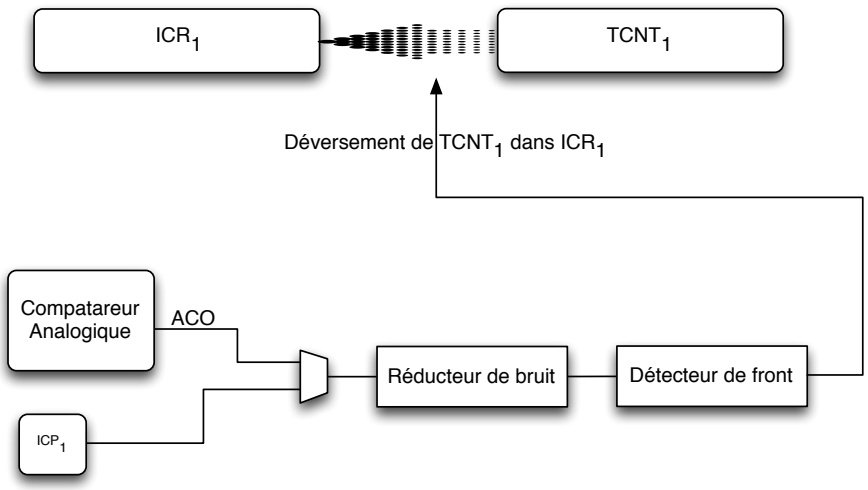


Figure 3.2: Unité de capture

En mode interruptif, Si $TICIE_1$ passe à un, une *interruption* de capture est déclenchée. ICF_1 est automatiquement remis à zéro dans le sous-programme d'IT.

Remarque 3.2.1 *ICR_1 ne peut être mis à jour que lorque l'on utilise le mode qui utilise ICR_1 fixant la valeur TOP. Dans ce cas, les bits $WGM1_{3:0}$ doivent être positionnés avant que ICR_1 soit initialisé avec la valeur TOP. On écrira dans ICR_1 d'abord l'octet de poids faible puis le poids Fort.*

La source de capture : La patte de capture est ICP1. Cette source peut aussi être connectée au comparateur analogique. Le comparateur est sélectionné avec le bit $ACIC$ du registre ACS.

Remarque 3.2.2 *Attention, changer la source peut provoquer une capture, le flag ICF doit donc être **raz** après ce changement.*

Les pattes ICP_1 et la patte de sortie ACO sont échantillonnées avec les mêmes techniques déjà évoquées : Le détecteur de front est identique. Quand le réducteur de bruit est activé, de la logique est rajoutée avant le détecteur de front ce qui ralentit le signal de 4 cycles d'horloge. Une entrée de capture peut être déclenchée par programme en contrôlant le $PORTC$ contenant la patte $ICP1$.

Réducteur de bruit C'est en fait un filtre numérique qui délivre la sortie **ssi** les 4 signaux échantillonnés consécutifs sont égaux. Le réducteur est activé par le bite $ICNC_1$ du registre $TCCR_{1B}$.

Capture des événements Tout évènement capturé écrase le précédent même si il n'a a pas été traité par le CPU. Aussi, dans un sous-programme d'*interruption* il faudra lire ICR_1 au plus tôt. On prendra garde à ne pas changer la valeur TOP lors de l'utilisation de capture. Lors de la mesure d'un signal de type PWM , la détection est changée aprés chaque capture et cela doit donc être fait aussitôt que ICR_1 a été lu.

Unité de Comparaison de sorties

Les sorties $OC1A/B$ sont controlées par les bits $COM1x1 : 0$. Un de ces bits à un provoque la sortie du générateur de forme.

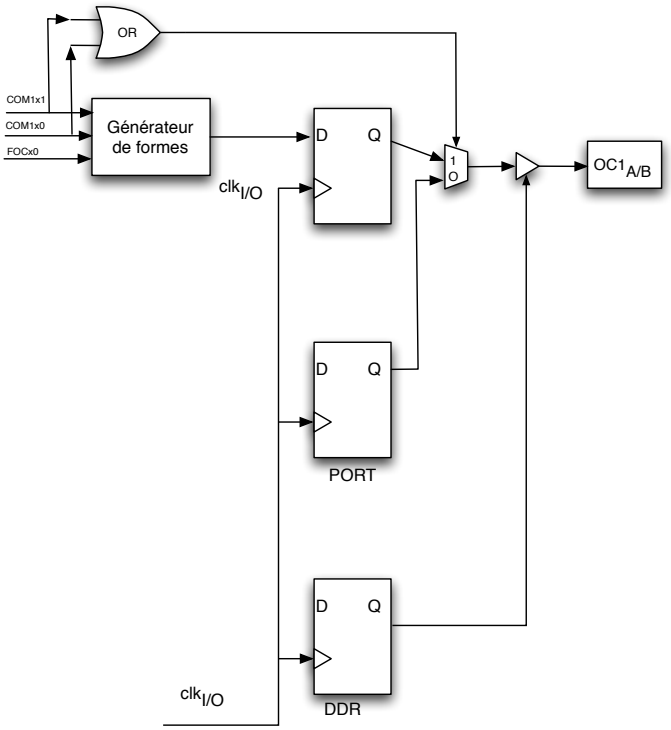


Figure 3.3: Contrôle de la sortie du comparateur

La forme produite sur les pattes $OC1A$ ou $OC1B$ dépend des bits $WGM1_{3:0}$ et des modes de comparaison défini par $COM1x1:0$, ainsi que des valeurs pour TOP et $BOTTOM$. Il faut noter que cette figure montre que les pattes $OC1A$ ou $OC1B$ doivent avoir été déclarées en sortie à l'aide du registre DDR.

3.2.2 Les 16 modes du timer 1

Le Comparateur permet de définir la valeur TOP de $TCNT_1$. Le comparateur peut fonctionner suivant différents modes.

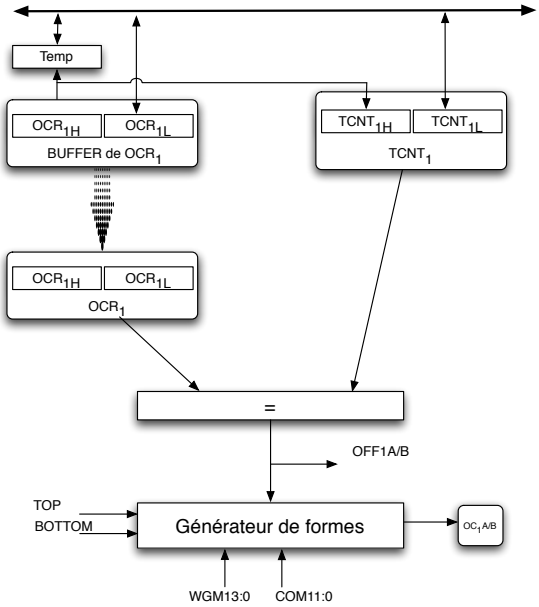


Figure 3.4: Unité de comparaison

Initialisation des registres et pattes de sorties

$TCNT_1$ **:** L'écriture dans $TCNT_1$ quel que soit le mode va bloquer les comparaisons pour un cycle d'horloge. Il y a des risques à changer $TCNT_1$ en utilisant le comparateur : Si la valeur écrite dans $TCNT_1$ est égale à OCR_{1A} ou OCR_{1B} alors le résultat de la comparaison sera perdu. Il ne faut pas écrire dans $TCNT_1$ une valeur égale à TOP en mode PWM à valeur variable. La comparaison avec TOP échouera alors et continuera jusqu'a $0xFFFF$. Même effet si l'on écrit dans $TCNT_1$ une valeur égale à $BOTTOM$ en décomptant.

Concernant les pattes OC1A et OC1B, il fait faire attention à positionner OC1A et OC1B, en sortie (PB_6, PB_7) avant de produire une MLI par l'écriture de DDR_B.

Le générateur de formes est défini par les bits $COM1x1:0$ qui permettent de définir 16 modes regroupés 5 catégories :

- Normal
- CTC
- PWM rapide
- Phase correcte PWM
- Phase et fréquence correcte PWM

En mode PWM $COM1x1:0$ définissent si la PWM est inversée ou non. Les modes sont complètement définis avec : $WGM1_{3:0}$ et $COM1x1:0$.

3.2.3 Les Modes du générateur de formes

Les 16 différents modes sont déterminés par les bits : $WGM1_{3:0}$

a) Le mode Normal : $WGM1_{3:0}$=0

La sortie OC_{1A} commute à chaque fois que $TCNT_1$ atteint la valeur TOP (si $COM1A_{1:0}$= 1)

- Dans ce mode la fréquence max est :

$$f_{OC1_A} = \frac{f_{clk_{I/O}}}{2}$$

- Sinon la fréquence est définie par :

$$f_{OC1_A} = \frac{f_{clk_{I/O}}}{2.N.(1 + OCR1_A)}$$

N représente le facteur de prédivision (1,8,64,256,1024)

Attention à ne pas changer la valeur TOP avec une valeur inférieure à la valeur courante du comparateur OCR_{1A}.

b) Le mode PWM rapide $WGM1_{3:0}$=5,6,7,14 ou 15

Cette PWM permet de générer de plus hautes fréquences. Elle est dite à simple pente et on peut ici avoir 2 commutations de la sortie sur la même pente. Ici, $TCNT_1$ compte de $BOTTOM$ à TOP et fait alors une comparaison avec un seuil (OCR_{1A}), pendant la montée puis revient à $BOTTOM$.

Sortie

- En mode non inversé, la sortie est **raz** sur égalité avec OCR_{1A} et **mise à un** sur $BOTTOM$.
- En mode inversé, la sortie est à un sur égalité et **raz** sur $BOTTOM$.
- Du fait de son fonctionnement en simple pente, on va 2 fois plus vite.

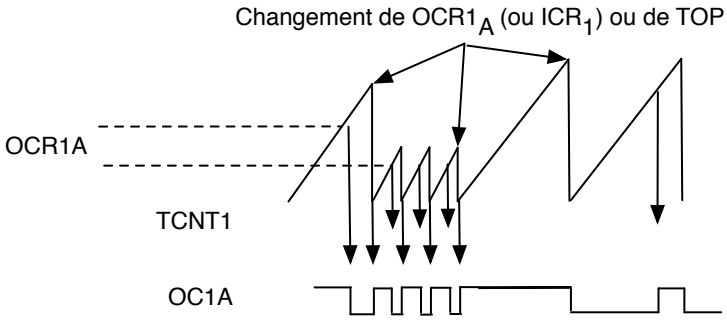


Figure 3.5: Le mode PWM rapide

Fréquence Dans ce mode la fréquence est définie par :

$$f_{OC_{1A}PWM} = \frac{f_{clk_{I/O}}}{N.(1+TOP)}$$

N représente le facteur de prédivision (1,8,64,256,1024)

Valeurs particulières Attention à ne pas changer la valeur TOP avec une valeur inférieure à la valeur courante de celle du comparateur. De plus, pour une valeur $TOP = 0$ on aura un quasi front à chaque top d'horloge.

c) Le mode PWM à phase correcte $WGM1_{3:0}$=1,2,3,10 ou 11

Ce mode permet d'obtenir une haute résolution de PWM à phase correcte. On est cette fois ci sur du double pente avec comparaison de seuil. Le compteur compte de la valeur $BOTTOM$ à TOP puis il décrémente jusqu'a $BOTTOM$ et recommence indéfiniment.

Instant de commutation de $OC1_x$ Cette commutation dépend du registre OCR_{1x} et de la valeur TOP. On voit ces instants de commutation dans la figure 3.2.3 ou deux événements se produisent lors des montées et des descentes :

- En mode non inversé, $OC1_x$ est **raz** sur l'égalité de $TCNT_1$ avec OCR_{1x} en comptant, et **mis à un** sur l'égalité de $TCNT_1$ avec OCR_{1x} en décomptant.
- En mode inversé, $OC1_x$ est **mis à un** sur l'égalité de $TCNT_1$ avec OCR_{1x} en comptant, et **raz** sur l'égalité de $TCNT_1$ avec OCR_{1x} en décomptant.

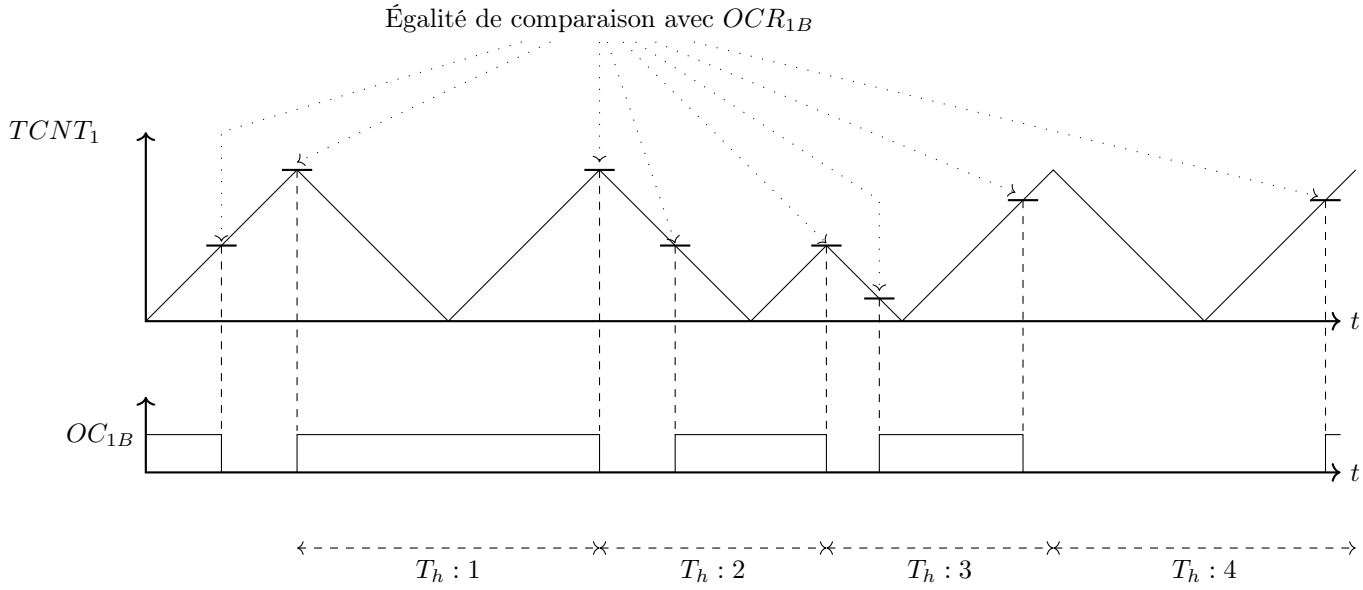
Remarque 3.2.3 *OCR_{1x} sont mis à jour sur TOP. Cela signifie que si OCR_{1x} est modifié pendant une pente, sa nouvelle valeur n'est prise en compte que sur le prochain TOP*

Ce mode à double pente à une fréquence de hachage maximale évidemment deux fois plus petite que celle à simple pente, mais son caractère symétrique le rend plus utilisé dans les applications de commande de moteurs.

Résolution La résolution varie de la valeur 2 bits à 16 bits, elle dépend de la valeur de OCR_{1A} ou ICR_1. La résolution de la PWM est donnée par l'équation :

$$R_{PCPWM} = \frac{log(TOP+1)}{log(2)}$$

Dans ce mode le compteur est incrémenté jusqu'à ce que la valeur atteigne soit une des valeurs : $0x00FF, 0x01FF, 0x03FF$ ($WGM1_{3:0} = 1, 2,$ ou 3), soit la valeur dans ICR_1 ($WGM1_{3:0}$= 10), soit la valeur dans OCR_{1A} ($WGM1_{3:0}$= 11).
Dès que $TCNT_1$ a alors atteint TOP alors $OC1A$ commute et $TCNT_1$ décompte dés lors.



Attention Là encore, lorsque l'on change la valeur TOP on doit s'assurer qu'elle est supérieure à la valeur courante de $TCNT_1$

Fréquence Dans ce mode la fréquence est définie par :

$$f_{OC_{1A}PCPWM} = \frac{f_{clk_{I/O}}}{2.N.(1+TOP)}$$

N représente le facteur de prédivision (1,8,64,256,1024)

Mode inversé ou non En positionnant les bits $COM1x1:0$ à 2 on aura du non inversé. En positionnant les bits $COM1x1:0$ à 3 on aura de l'inversé

d) Le mode PWM à phase et fréquence correcte : "MLI Centrée" $WGM1_{3:0} = 8$ ou 9

Ce mode fournit une haute résolution à phase et fréquence correcte. Ce mode comme le précédent est basé sur une double pente. Il compte indéfiniment de $BOTTOM(0x0000)$ à TOP et de TOP à $BOTTOM$

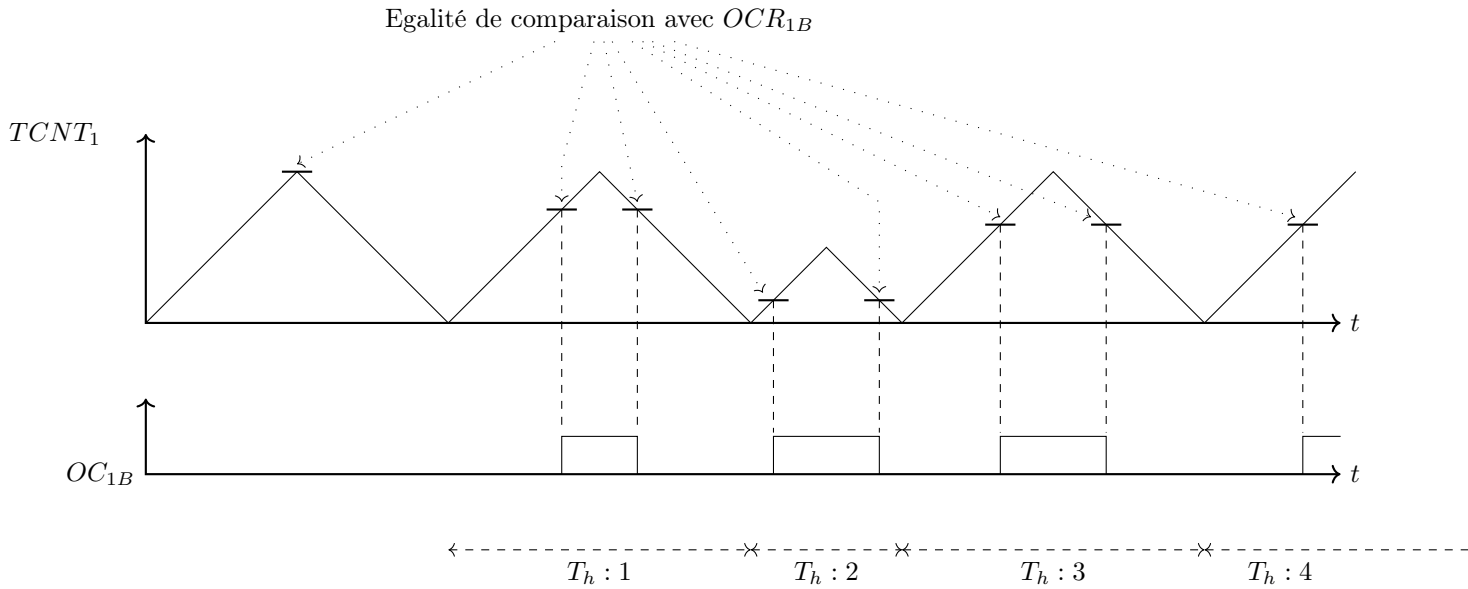
En mode non inversé (cf figure suivante) les sorties$OC1A$ et $OC1B$ sont remises à zéro sur l"égalité entre $TCNT_1$ et OCR_{1x} en comptant et **mis à un** sur égalité en décomptant. En mode inversé c'est l'inverse.

Remarque 3.2.4 *OCR_{1x} sont mis à jout sur $BOTTOM$. Cela signifie que si OCR_{1x} est modifié dans une pente alors sa nouvelle valeur n'est prise en compte que sur le prochain $BOTTOM$.*

L'opération en double pente donne une fréquence max plus basse comparée à celle en simple pente. Cependant son caractère très symétrique en fait le modu plus utilisé en terme de contrôle moteur.

Résolution La résolution de la PWM est donnée par l'équation :

$$R_{PCPWM} = \frac{log(TOP+1)}{log(2)}$$



Au contraire du mode précédent il y a une symétrie dans le signal quand on examine sa période. De plus, OC_{1X} ne sera visible sur $PORTB$ que si il a été programmé en sortie via DDR_B.

Inversé/non inversé En positionnant les bits $COM1x1:0$ à 2 on aura du non inversé. En positionnant les bits $COM1x1:0$ à 3 on aura de l'inversé.

Fréquence Dans ce mode la fréquence est définie par :

$$f_{OC_{1A}PFCPWM} = \frac{f_{clk_{I/O}}}{2.N.(1+TOP)}$$

N représente le facteur de prédivision (1,8,64,256,1024)

3.2.4 Inventaire des registres utiles au timer 1

Le registre $TCCR_{1A}$

COM_{1A1}	COM_{1A0}	COM_{1B1}	COM_{1B0}	FOC_{1A}	FOC_{1B}	WGM_{11}	WGM_{10}

Bits $COM1x1, COM1x0$ en mode *normal* et *CTC*

$COM1x1$	$COM1x0$	*Description*
0	0	P libre $OC1x$ déconnectés
0	1	changement d 'état quand $OC1x = TCNT1$
1	0	**raz** sur égalité
1	1	**mis à un** sur égalité

Bits $COM1x1$, $COM1x0$ en mode fast *PWM*

$COM1x1$	$COM1x0$	*Description*
0	0	P libre $OC1_x$ déconnectés
0	1	$WGM1_{3:0} = 15$
		$OC1_A$commute sur égalité
		$OC1_B$ déconnectés
1	0	**raz** $OC1x$ sur égalité
		mis à un sur *BOTTOM*
1	1	**mis à un** sur égalité
		raz sur *BOTTOM*

Bits $COM1x1$, $COM1x0$ en mode *PWM* à phase correcte et *PWM* à phase et fréquence correcte

$COM1A_1$ $COM1B_1$	$COM1A_0$ $COM1B_0$	*Description*
0	0	P libre $OC1_x$ déconnectés
0	1	$OC1_A$ commmute sur égalité, $OC1_B$ déconnecté
1	0	**raz** $OC1_x$ sur égalité en incrémentant
		mis à un en décrémentant sur égalité
1	1	**mis à un** sur égalité en incrémentant
		raz sur égalité en décrémentant

Bits $FOC1_x$ Ces bits $FOC1_x$ ne sont actifs que si on n'est pas en *PWM* mode. Cependant ces bits doivent être **raz** en *PWM* mode. $FOC1_x \leftarrow 1$ force à l'instant choisi, une comparaison entre $TCNT_1$ et OCR_{1A} (par ex.). Ce sont des bits d'échantillonage. En lecture ces bits valent zéro.

$WGM1_{1:0}$ **Waveform Generation Mode** Ces bits déterminent le sens du comptage, *TOP* et le type de forme et on l'a vu précédement, ils déterminent aussi le mode : normal, *CTC*, et les trois modes *PWM* :

- *PWM* rapide : Fast *PWM*
- *PWM* à phase correcte : Phase correct *PWM* .
- *PWM* à phase et fréquence correcte : Phase and frequency correct *PWM* .

Le registre $TCCR_{1B}$

$ICNC_1$	$ICES_1$	-	$WGM1_3$	$WGM1_2$	CS_{12}	CS_{11}	CS_{10}

$ICNC_1$ **: Réduction de bruit** La réduction de bruit se fait par $ICNC_1$: Input Capture Noise Canceler. Si $ICNC_1 \leftarrow 1$ cela active le réducteur de bruit de l'entrée de capture.

Quand le réducteur de bruit est activé l'entrée de capture ICP_1 est filtrée. Le filtre affecte sa sortie si on a 4 sorties successives égales ce qui induira un délai de 4 cycles d'horloges.

$ICES_1$ **: La sélection du type d'évènement** La sélection du type d'évènement se fait par $ICES_1$: Input Capture Edge Select. Ce bit sélectionne quel type de front permet de déclencher la capture. Quand $ICES_1 = 0$ c'est un front descendant, lorsque $ICES_1 = 1$ c'est un front montant.

Quand une capture est déclenchée, $TCNT_1$ est vidé dans ICR_1. $ICF1$ est positionné à un et peut générer une *interruption* . Quand c'est ICR_1 qui détermine *TOP* alors ICP_1 est déconnectée et le mécanisme de capture ne fonctionne plus.

$WGM13:2$ **Waveform Generation Mode** (cf 3.2.2)

Bit CS12:0 Clock Select Ces trois bits sélectionnent la prédivision : division de $Cllk_{io}$ par $N = (1, 8, 64, 256, 1024)$.

Les Registres $TCNT_1, OCR1_x, ICR_1$

$TCNT_1$ est accessible en lecture et en écriture. Modifier $TCNT_1$ pendant le comptage peut produire un effet indésirable si la nouvelle valeur est supérieure aux valeurs de comparaison : on manque alors une comparaison.

OCR_{1x}, les registres de comparaison contiennent une valeur 16 bits qui est continument comparée avec la valeur courante de $TCNT_1$. L'égalité est utilisée pour générer une *interruption* ou une forme sur la patte $OC1_x$.

ICR_1 est mis à jour avec la valeur du compteur $TCNT_1$ à chaque fois qu'un événement se produit sur la patte $ICP1$. Ce registre peut aussi être utilisé pour définir la valeur TOP.

Le registre $TIMSK$

$OCIE_2$	$OCIE_2$	$TICIE_1$	$OCIE_{1A}$	$OCIE_{1B}$	$TOIE_1$	-	$TOIE_0$

$TICIE_1$**: "Timer/Counter1, Input Capture Interrupt Enable"** :

Quand $TICIE_1 \leftarrow 1$, le bit I du registre d'état est mis à un, toutes les *interruptions* sont donc globalement autorisées et en particulier l'interruption de capture. L'*interruption* ad hoc est alors exécutée lorsque $ICF_1 \in TIFR$ reçoit un.

$OCIE_{1A}$**: "Timer/Counter1, Output Compare A Match Interrupt Enable" :** Quand $OCIE_{1A} \leftarrow 1$, le bit I du registre d'état est mis à un, toutes les *interruptions* sont donc globalement autorisées et en particulier l'interruption de comparaison. L'*interruption* ad hoc est alors exécutée lorsque $OCF1_A \in TIFR$ reçoit un.

$OCIE_{1B}$**: "Timer/Counter1, Output Compare B Match Interrupt Enable" :** Quand $OCIE_{1A} \leftarrow 1$, le bit I du registre d'état est mis à un, toutes les *interruptions* sont donc globalement autorisées et en particulier l'interruption de comparaison. L'*interruption* ad hoc est alors exécutée lorsque $OCF1_B \in TIFR$ reçoit un.

$TOIE_1$**: "Timer/Counter1, Overflow Interrupt Enable"** :

Quand $TOIE_1 \leftarrow 1$, le bit I du registre d'état est mis à un, toutes les *interruptions* sont donc globalement autorisées et en particulier l'interruption de comparaison. L'*interruption* ad hoc est alors exécutée lorsque $TOV_1 \in TIFR$ reçoit un.

Le registre $TIFR$

OCF_2	TOV_2	IC_1	OCF_{1A}	OCF_{1B}	TOV_1	-	TOV_0

ICF_1**: "Timer/Counter1, Input Capture Flag"** :

ICF_1 est un drapeau (flag) qui reçoit un quand la patte $ICP1$ reçoit un signal. Quand ICR_1 est utilisé pour stocker TOP avec un mode de $WGM1_3 : 0$, ICF_1 est positionné à un quand le compteur atteint TOP. ICF_1est automatiquement **raz** par le sous-programme d'*interruption* est exécuté. Sinon ICF_1 peut être **raz** en y écrivant un un.

$OCF1_A$**: "Timer/Counter1, Output Compare B Match Flag"** :

$OCF1_A$ est un drapeau (flag) qui reçoit un quand l'égalité de OCR_{1A} avec $TCNT_1$ se produit. $OCF1_A$ est automatiquement **raz** par le sous-programme d'*interruption* est exécuté. Sinon $OCF1_A$ peut être **raz** en y écrivant un un.

$OCF1_B$**: "Timer/Counter1, Output Compare B Match Flag"** :

Ce bit est un drapeau (flag) qui reçoit un quand l'égalité de $OCR1_B$ avec $TCNT_1$ se produit. $OCF1_B$ est automatiquement **raz** par le sous-programme d'*interruption* est exécuté. Sinon$OCF1_B$ peut être **raz** en y écrivant un un.

TOV_1**: "Timer/Counter1, Overflow Flag"** :

Dans les modes normal et *CTC* modes, TOV_1 est mis à un sur un dépassement de capacité TOV_1 est automatiquement **raz** par le sous-programme d'*interruption* est exécuté. Sinon TOV_1 peut être **raz** en y écrivant un un.

3.2.5 Résumé des 16 modes de *MLI* du timer 1

v	WGM_{13}	WGM_{12}	WGM_{11}	WGM_{10}	*Mode* des Timers	*TOP*	Mise à jour de $OCR1_x$	Mise à un de TOV_1
0	0	0	0	0	Normal	$0xFFFF$	*Immediate*	*MAX*
1	0	0	0	1	PWM Phase Correcte 8-bit	$0x00FF$	*TOP*	*BOTTOM*
2	0	0	1	0	PWM Phase Correcte 9-bit	$0x01FF$	*TOP*	*BOTTOM*
3	0	0	1	1	PWM Phase Correcte 10-bit	$0x03FF$	*TOP*	*BOTTOM*
4	0	1	0	0	*CTC*	$OCR1_A$	*immediate*	*MAX*
5	0	1	0	1	Fast *PWM* 8-bit	$0x0FF$	*BOTTOM*	*TOP*
6	0	1	1	0	Fast *PWM* 9-bit	$0x1FF$	*BOTTOM*	*TOP*
7	0	1	1	1	Fast *PWM* 10-bit	$0x3FF$	*BOTTOM*	*TOP*
8	1	0	0	0	PWM Phase, freq. correctes	ICR_1	*BOTTOM*	*BOTTOM*
9	1	0	0	1	PWM Phase, freq. correctes	$OCR1_A$	*BOTTOM*	*BOTTOM*
10	1	0	1	0	PWM Phase Correcte	ICR_1	*TOP*	*BOTTOM*
11	1	0	1	1	PWM Phase Correcte	$OCR1_A$	*TOP*	*BOTTOM*
12	1	1	0	0	*CTC*	ICR_1	*immediate*	*MAX*
13	1	1	0	1	Réservé	–	–	–
14	1	1	1	0	Fast *PWM*	ICR_1	*BOTTOM*	*TOP*
15	1	1	1	1	Fast *PWM*	$OCR1_A$	*BOTTOM*	*TOP*

3.2.6 Exemples d'utilisation du timer 1

-1- *MLI* centrée de 66% sur la patte $PB_2(OC1_B)$.

```
int main (void) {
  int i;
  DDRB = 0xFF;               //Port B en sortie

  TCCR1A=  (1<<COM1B1)+(1<<WGM10); //Mli centree, non inversee
  TCCR1B=(1<<WGM13)+(1<<CS10); // Prediv N=1
  OCR1A=532;    // Fhash = 15khz,  Thash = 66 micro-s
  OCR1B=361;    // Ton / Thash = 0.66
  while(1);  // Boucle infinie
}
```

-2- Interruption de débordement du timer 1:

```
#define LedToggle PortB^=1
volatile byte Compteur;

ISR (TIMER1_OVF_vect) {
   byte i;
  TCNT1 = 0;
  if (Compteur++ == 50) {    // Compteur permet de commuter la led 1 fois sur 50
    Compteur=0;
    LedToggle;  // mesure de la periode sur PB0
  }
}

int main() {
  DDRB=0xFF;

  TCCR1A = 0;           // Mode normal
  TCCR1B = 1<<CS10;    // Prediv N=1
  TIMSK = 1<<TOIE1;     // It debordement autorisee
  sei();                // autorise les interruptions
  while(1);
}
```

3.3 Le timer 2

Les fonctions du timer 2 sont la génération de fréquences, le comptage, le mode **raz** de $TCNT_2$ sur comparaison avec rechargement automatique et la fonction génération de formes (waveform). Celle-ci permet les modes suivant :

- mode sans erreur ;
- mode PWM à phase correcte ;

3.3.1 Aperçu du timer 2

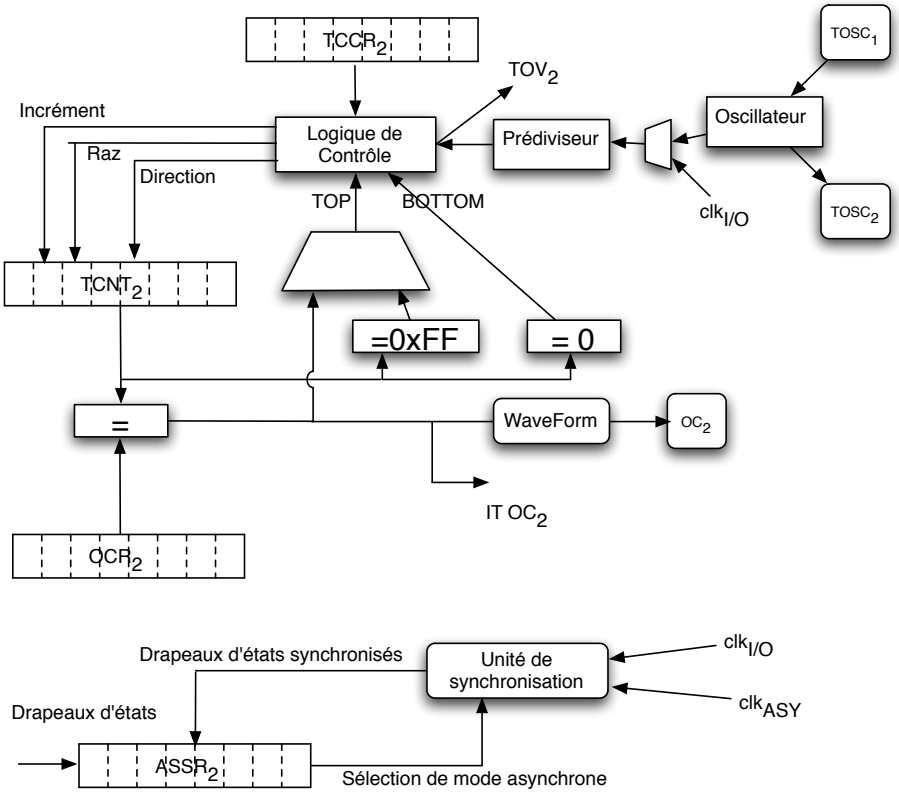


Figure 3.6: Aperçu du timer 2

Le timer 2 est un compteur 8 bits aussi, $TCNT_2$ et OCR_2 sont des registres 8-bit. Les ITs sont visibles dans $TIFR$ et masquables dans $TIMSK$. Le timer peut être clocké de façon interne par le prédiviseur ou de façon asynchrone par $TOSC_1, TOSC_2$. Cette synchronisation est contrôlée par le registre $ASSR$. Le block logique sélectionne la source d'incrémentation. OCR_2 est comparé à tout moment à $TCNT_2$ et le test de cette égalité peut être utilisé à tout moment pour générer une forme de type PWM ou pour faire varier une fréquence sur OC_2. Le débordement genère sur une *interruption* par le flag OCF_2 On a $BOTTOM = 0$, $MAX = 0xFF$, de plus TOP peut être soit MAX soit la valeur placée dans OCR_2

Contrôle du comptage

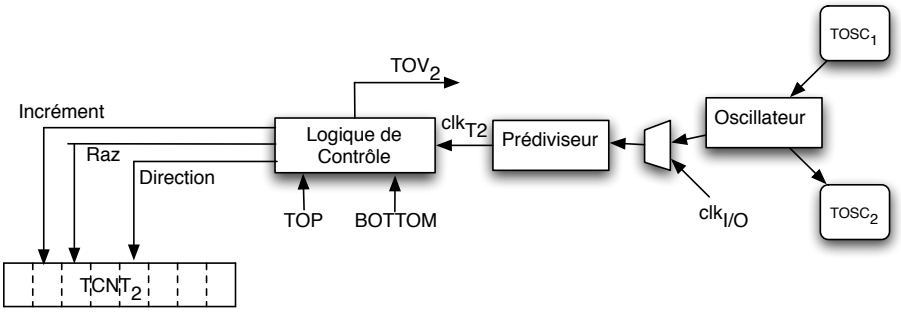


Figure 3.7: Contrôle du comptage

- Comptage : incrémentation ou décrémentation ;
- clk_{T2} horloge d'incrément
- TOP : signale que $TCNT_2$ a atteint sa valeur haute ;
- $BOTTOM$: signale que $TCNT_2$ a atteint 0 ;

Contrôle de la fréquence

- $CS22:0$ sélectionne la clock ;
- si $CS22:0 = 0$ le timer est au repos ;
- $WGM2_1$ $WGM2_1 \in TCCR_2$ permettent de déterminer la séquence de comptage.

3.3.2 Génération de forme : pulse, *PWM*,...

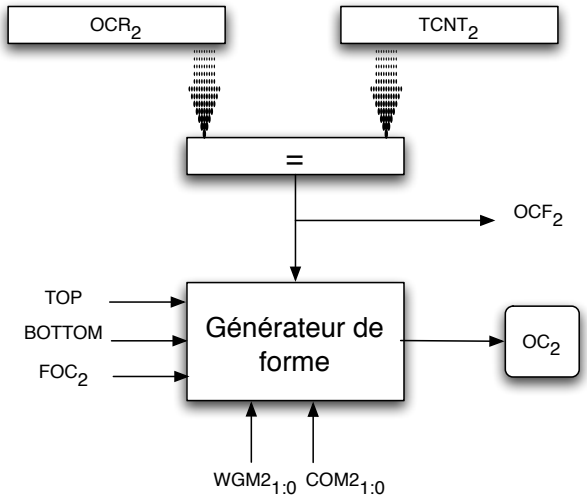


Figure 3.8: Génération de formes

OCR_2 est un registre à buffer double en mode PWM . On n'a pas la double bufferisation en mode normal et CTC. Quand on est en double buffer le CPU accède au buffer de OCR_2 , quand ce mode est désactivé le CPU accède au registre lui-même.

Mise à jour du buffer et de OCR_2 :

La mise à jour d'OCR_2 est retardée au moment ou $TCNT_2$ atteint soit TOP ou $BOTTOM$ évitant ainsi de créer des PWM non symétriques rendant la sortie "Glitch Free".

En mode non PWM , on peut forcer la comparaison par FOC_2 si il y a égalité OC_2 prend une valeur. C'est $COM2_{1:0}$ qui définissent ce qui se passe alors sur $OC2$ (1, 0 ou commutation). Toute écriture du CPU bloque toute comparaison avec $TCNT_2$ ce qui permet de positionner une même valeur d'initialisation sur OCR_2 et $TCNT_2$. Par contre, si on met une valeur dans OCR_2 égale à $TCNT_2$ le test d'égalité va être perdu.

Unité de comparaison

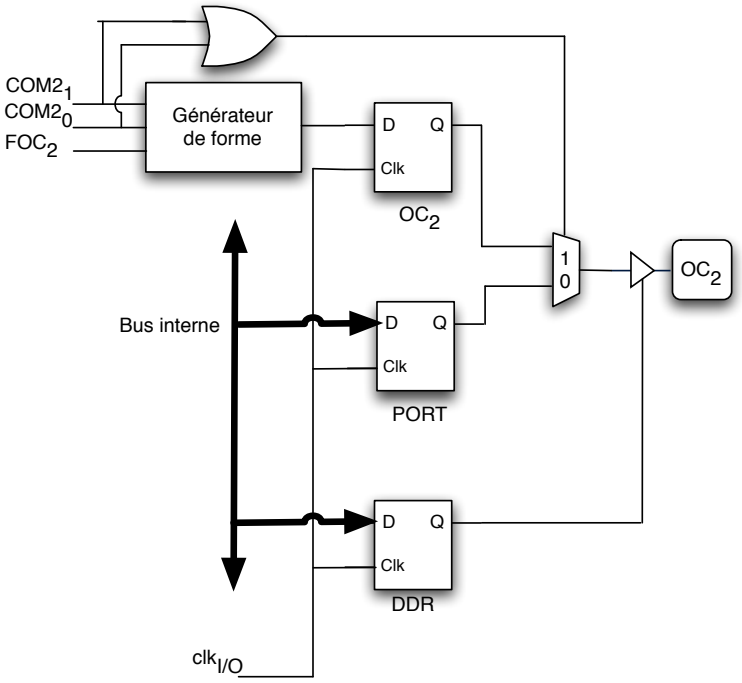


Figure 3.9: Unité de comparaison

- $COM2_{1:0}$ contrôle 2 fonctions :
  - La sortie de comparaison du générateur de formes
  - Ce que reçoit OC_2 dans le cas ou l'un des deux bits est à un.
- Cette patte reçoit soit :
  - La sortie du générateur de formes
  - soit un bit d'un *port* , suivant la configuration du Port fixé par DDR

3.3.3 Les modes du timer 2

Les modes sont programmés par les bits $WGM2_{1:0}$ et $COM2_{1:0}$. Les bits $COM2_{1:0}$ contrôlent si la PWM est inversée ou non. Pour les modes non PWM (tel que le mode comparaison), les bits $COM2_{1:0}$ contrôlent aussi, si OCF_2 doit être **raz**,**mis à un**, ou commuté (toggle). Enfin, $WGM2_{1:0}$ contrôlent les modes.

Mode normal $WGM2_{1:0}= 0$

Dans ce mode $TCNT_2$ s'incrémente jusqu'a $0XFF$ puis recommence depuis 0. TOV_2 passe à un quand $TCNT_2$ retombe à zero. TOV_2 sert alors de $9^{\text{ième}}$ bits, il sert aussi à déclencher une *interruption* . L'*interruption* est **raz** automatiquement par l'exécution du sous-programme d'*interruption* et en mode non interruptif il faut faire une **raz** par une **mise à un**.

Mode Clear timer on Compare *CTC* : $WGM2_{1:0} = 2$

Dans ce mode OCR_2 définit la résolution. $TCNT_2$ est **raz** quand $TCNT2$ devient égal à OCR_2 qui définit la valeur TOP (cf figure 3.3.3).

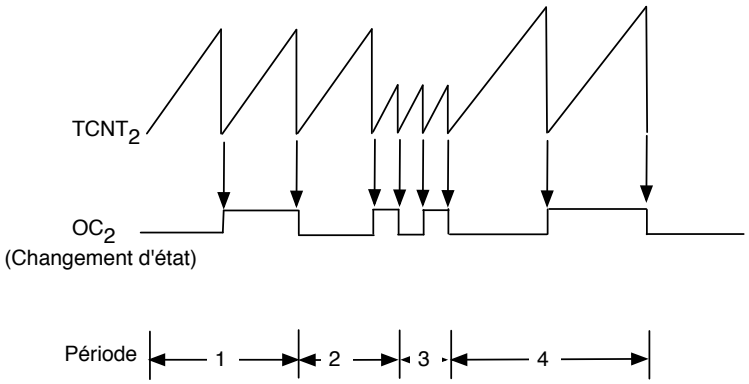


Figure 3.10: le timer 2 : Mode Clear timer on Compare

Interruption: Une *interruption* peut être générée à chaque TOP. Dans le sous-programme d'*interruption* on peut changer le TOP. Attention : ne pas changer TOP avec une valeur à $TCNT_2$

Fréquence

$$f_{OC_2} = \frac{f_{clk_{I/O}}}{2.N.(1 + OCR_2)}$$

Mode *PWM* a fréquence rapide avec $WGM2_{1:0} = 3$

C'est un mode à simple front. TOV_2 est **mis à un** quand $TCNT_2$ atteint MAXn l'*interruption* peut mettre à jour OCR_2 . $TCNT_2$ compte de $BOTTOM$ à MAX et repart de $BOTTOM$; En mode non inversé, $OC2$ est **raz** sur égalité de $TCNT_2$ et OCR_2 , et **mis à un** sur $BOTTOM$

Fréquence

$$f_{OC_2PWM} = \frac{f_{clk_{I/O}}}{N.(1 + OCR_2)}$$

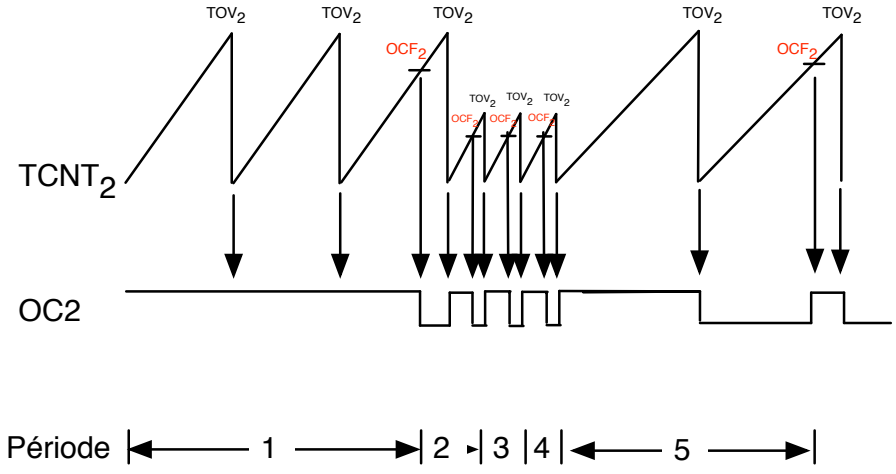


Figure 3.11: le timer 2 : Mode PWM rapide

Chronogramme de la *PWM* rapide

Le mode *PWM* à phase correcte

Dans ce mode $WGM2_{1:0}$= 1, on a une PWM à phase correcte. Elle est basée sur un mode double front. TOV_2 est **mis à un** quand $TCNT_2$ atteint $BOTTOM$ et l'*interruption* peut mettre à jour OCR_2 $TCNT_2$ compte de $BOTTOM$ à MAX et repart de MAX jusqu'a $BOTTOM$;

inversé/non inversé En mode non inversé, en comptant $OC2$ est **raz** sur égalité de $TCNT_2$ et OCR_2 , tandis que en décomptant, OC_2 est **mis à un** sur sur égalité de $TCNT_2$ et OCR_2 . On a un mode ici symétrique : "phase correcte".

Fréquence

$$f_{OC_2PWM} = \frac{f_{clk_{I/O}}}{2.N.(1 + OCR_2)}$$

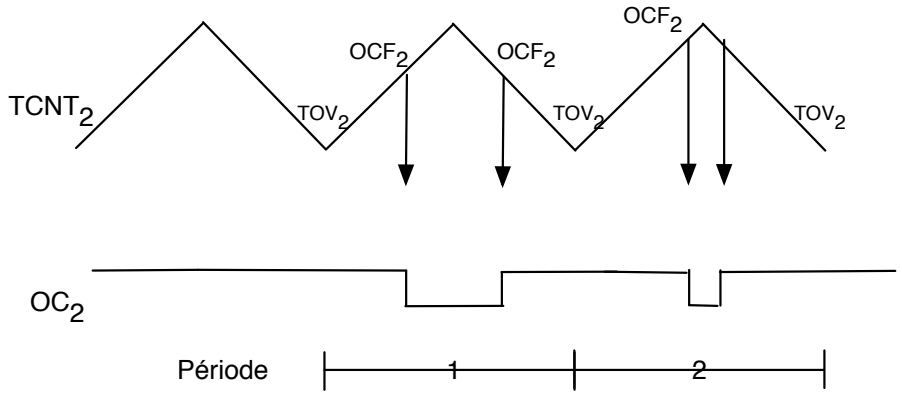


Figure 3.12: le timer 2 : Mode PWM phase correcte

Chronogramme de la *PWM* à phase correcte

3.3.4 Les registres utiles au timer 2

Le registre $TCCR_2$

Le registre $TCCR_2$

FOC_2	WGM_2	$COM21$	$COM2_0$	$CS2_2$	$CS2_1$	$CS2_0$	$TCCR_2$

Bit 7 : FOC_2: Force Output Compare FOC_2 est actif quand on est pas en mode PWM . Cependant ce bit doit êtr **raz** quand $TCCR_2$ est mis à jour en mode PWM . Quand FOC_2← 1 une comparaison immédiate est forcée selon la valeur des bits de $COM2_{1:0}$ OC_2 sera mis à jour. FOC_2 est un bit provoquant un échantillonnage.

Bit 6,3 : $WGM2_{1:0}$: Waveform Generation Mode Ces Bits contrôlent la séquence de comptage, le maximum et le type de forme :

v	$WGM2_1$	$WGM2_0$	$Mode$	TOP	maj OCR_2	TOV_2 ← 1?
0	0	0	Normal	$0xFF$	Immédiatement	MAX
1	0	1	PWM phase correcte	$0xFF$	TOP	$BOTTOM$
2	1	0	CTC	OCR_2	Immédiatement	MAX
3	1	1	PWM rapide	$0XFF$	$BOTTOM$	MAX

Bit $COM2_{1:0}$: Compare Match Output Mode En mode de comparaison

$COM2_1$	$COM2_0$	*description*
0	0	OC_2 est deconnectée du P
0	1	OC_2 est commutée sur égalité
1	0	OC_2 est **raz** sur égalité
1	1	OC_2 est **mise à un** sur égalité

Bit $COM2_{1:0}$: Compare Match Output Mode En mode de comparaison et *PWM* rapide

$COM2_1$	$COM2_0$	*description*
0	0	OC_2 est deconnectée du P
0	1	réservé
1	0	OC_2 est **raz** sur égalité, **mise à un** sur $BOTTOM$
1	1	OC_2 est **mise à un** sur égalité, **raz** sur $BOTTOM$

Bit 5:4 : $COM2_{1:0}$: Compare Match Output Mode En mode de comparaison et *PWM* à phase correcte

$COM2_1$	$COM2_0$	*description*
0	0	OC_2 est deconnectée du P
0	1	réservé
1	0	OC_2 est **raz** sur égalité en comptant, **mise à un** sur égalité en décomptant
1	1	OC_2 est **mise à un** sur égalité en comptant, **raz** sur égalité en décomptant

Bit 2:0 : $CS2_{2:0}$: Clock Select

$CS2_2$	$CS2_1$	$CS2_0$	*description*
0	0	0	pas d'horloge, le timer 2 est stoppé
0	0	1	$clk_{I/O}$
0	1	0	$clk_{I/O}/8$
0	1	1	$clk_{I/O}/32$
1	0	0	$clk_{I/O}/64$
1	0	1	$clk_{I/O}/128$
1	1	0	$clk_{I/O}/256$
1	1	1	$clk_{I/O}/1024$

ASSR

Le registre *ASSR*

-	-	-	-	AS_2	$TCN2U_B$	$OCR2U_B$	$TCR2U_B$

Bit AS_2: Asynchronous Timer/Counter2 Quand AS_2= 0, le timer 2 est cadencé par $clk_{I/O}$ Quand AS_2= 1, le timer 2 est cadencé par le quartz connecté à $TOSC_1$

Bit $TCN2U_B$: Timer/Counter2 Update Busy Quand le timer 2 fonctionne en mode asynchrone, et que $TCNT_2$ est **mis à jour** alors $TCN2U_B \leftarrow 1$. Un niveau bas sur ce bit indique que $TCNT_2$ est prêt à être mis à jour avec une nouvelle valeur.

Bit 1 : $OCR2U_B$: Output Compare Register2 Update Busy Quand le timer 2 fonctionne en mode asynchrone, et que OCR_2 est **mis à jour** alors $OCR2U_B \leftarrow 1$. Un niveau bas sur ce bit indique que OCR_2 est prêt à être mis à jour avec une nouvelle valeur.

Bit 0 : $TCR2UB$: Timer/Counter Control Register2 Update Busy Quand le timer 2 fonctionne en mode asynchrone, et que $TCCR_2$ est **mis à jour** alors $TCCR2U_B \leftarrow 1$. Un niveau bads sur ce bit indique que $TCCR_2$ est prêt à être mis à jour avec une nouvelle valeur.

Definition 3.3.1 *Accès aux registres "busy" Si une écriture est réalisée sur un des trois registres : $TCNT_2$, OCR_2 ou $TCCR_2$ alors que les flags correspondants sont busy alors on peut corrompre les calculs ou provoquer une interruption non désirée.*

Précautions sur le passage en mode asynchrone Lors des changements synchrones-asynchrones les valeurs des registres $TCNT_2$, OCR_2 ou $TCCR_2$ peuvent être erronées. La procédure de change vers le mode asynchrone est la suivante :

- Désactiver les *interruptions* relatives à le timer 2 par **raz** de $OCIE_2$ et $OCIE_2$
- Sélectionner la source avec AS_2
- Ecrire les nouvelles valeurs dans $TCNT_2$, OCR_2 ou $TCCR_2$
- Passer en mode Asynchrone : Attendre que les bits $TCN2U_B$, $OCR2U_B$ ou $TCR2U_B$ passe de "busy" à "libre" (valeur 0).
- **raz** les flags d'*interruption* relatifs à le timer 2
- Ré-autoriser les ITs
- L'oscillateur est optimisé pour utiliser un quartz de $32,768\ khz$.
- Attention la fréquence du quartz principal doit être au moins 4 fois supérieure à celle du quartz que l'on utliserait en mode asynchrone connecté à $TOSC_1$.
- Lors d'une écriture dans $TCNT_2$, OCR_2 ou $TCCR_2$, la nouvelle valeur passe par $TEMP$ et n'est accessible que après deux fronts positifs sur $TOSC_1$. Il faudra à l'utilisateur scruter les bits "busy" déjà évoqués avant d'écrire de nouvelles valeurs.

3.3.5 Les *interruptions* du timer 2

3.3.6 $TIMSK$

- Bit 7 : $OCIE_2$: Timer/Counter2 Output Compare Match Interrupt Enable
  Quand $OCIE_2 \leftarrow 1$ et que $I = 1 (\in SREG)$ alors l'*interruption* de comparaison du timer 2 est active. Si une égalité entre $TCNT_2$ et OCR_2 se produit alors $OCF_2 \leftarrow 1 (\in TIFR)$
- Bit 6 : $OCIE_2$: Timer/Counter2 Overflow Interrupt Enable
  Quand $TOIE_2 \leftarrow 1$ et que $I = 1 (\in SREG)$ l'*interruption* de débordement est active et $TOV_2 \leftarrow 1 (\in TIRF)$

3.3.7 TIFR

- Bit 7 : OCF_2: Output Compare Flag 2
  $OCF_2 \leftarrow 1$ quand une égalité de comparaison se produit entre $TCNT_2$ et OCR_2 .
  - $OCF_2 \leftarrow 1$ par le hard dans le sous-programme d'*interruption*.
  - Sinon, $OCF_2 \leftarrow 0$ en écrivant un **un**.
  - C'est quand on a $I = 1 (\in SREG)$ et $TOCIE_2 = 1$ et $OCF_2 = 1$ que le sous-programme d'*interruption* est exécuté.
- Bit 6 :TOV_2: Timer/Counter2 Overflow Flag
  - $TOV_2 \leftarrow 1$ quand un overflow se produit.
  - $TOV_2 \leftarrow 0$ par le hard dans le sous-programme d'IT.
  - Sinon, $TOV_2 \leftarrow 0$ en écrivant un **un**.
- C'est quand on a $I = 1 (\in SREG)$ et $TOIE_2 = 1$ et $TOV_2 = 1$ que le sous-programme d'IT est exécuté.

- En mode *PWM* TOV_2 reçoit un quand il y a un changement de direction à $0x00$

Exemple d'utilisation du timer 2 avec la mise en place d'une interruption de débordement :

```
#define LedToggle PortB^=1
volatile byte Compteur;

ISR (TIMER2_OVF_vect)
{ byte i;
  TCNT2 = 0;
  if (Compteur++ == 50) {
    Compteur=0;
    LedToggle;  // mesurer la periode
  }
}

void configTimer2(){
  TCCR2A = 0;           // Mode normal
  TCCR2B = (1<<CS22) + (1<<CS21) ; // clkio/256 est incremente toutes les 16uS
  TIMSK2 = 1<<TOIE2; // TOIE2
}
int main() {
  DDRB=0xFF;
  configTimer2();
  sei();                // autorise les interruptions
  while(1);
}
```

Chapter 4

La programmation en langage C

4.1 Structure de programme

Un programme C est composé de trois parties. Les entêtes, les fonctions nécessaires et le programme principal (main).

```
//----------------ENTETE--------------------------
#include <avr/io.h>
#define F_CPU 12E6 // reglage fclkIO sur fquartz
#include <util/delay.h> //utilisation de _delay_ms
#include <avr/interrupt.h>
#define Max_del 262
//----------------FONCTIONS-----------------------
void sleep(int dzs){
  int i=0;
  for(i=0;i<10*dzs;i++) _delay_ms(10);
}
//----------------MAIN----------------------------
int main(void)
{
  DDRC = 0x00;   // \pc\ en entree
   DDRB = 0x01;   // PB0 en sortie
   DDRD = 0b1111011;   // Port D en sortie, PORTD2 en entree
   do {
     sleep(10);
   } while(1);
   return(1);
}
```

4.1.1 Entête

Sous l'IDE Arduino, il n'y a pas besoin d'inclure de bibliothèques, et par contre, on peut y définir des constantes non modifiables ou des fichiers à inclure.

```
#define Max_del 262
#include "mesfonctions.c" // Inclusions de fichier c
#include "interface.h"  // Inclusion de bibliotheque utilisateur
```

4.1.2 Main

Un main commence par des initialisations et se termine généralement par une boucle infinie :

```
void fonction1(){...}
```

```
int fonction1(){int a; ... ; return a}
int main(void)
{
   DDRC = 0x00;    // PORTC en entree
   DDRB = 0x01;    // PB0 en sortie
   DDRD = 0b1111011;    // PORTD en sortie, PD2 en entree
   int j;
   do {
     sleep(10);
     fonction1();
     j=fonction2();
   } while(1);
   return(1);
}
```

4.1.3 Fonctions

Elles apparaissent de préférence (sinon on mettra les prototypes) avant le main :

```
void sleep(int dzs){
  int i=0;
  for(i=0;i<10*dzs;i++) _delay_ms(10);
}

void entier(int nbtours, int sens){  // passage de variables
int i,j;
int Tentier[4]={1,2,4,8};

  for(j=0;j<nbtours*12;j++) {
    for(i=3;i>=0;i--){
      delai2();
  PB=Tentier[i];
}
  }
}
```

4.1.4 Déclaration de variable globales

Pour les variables globales, on aura parfois besoin d'utiliser l'attribut volatile qui est une directive qui indique au compilateur de déclarer une variable dans la RAM et non dans la zone de registres. On utilisera cet attribut quand une variable dans deux cas :

- Quand la variable peut être, à la fois, modifiée dans un programme d'interruption et à la fois lue dans le programme principal. On évite alors des problèmes de mise en cache et de synchronisation entre l'interruption et la boucle principale qui font que la variable a peut être un valeur qui n'est pas à jour.
- Détection d'un changement d'état : Si vous utilisez une interruption pour détecter un changement d'état sur une broche (par exemple, un bouton-poussoir), vous pouvez déclarer la variable de détection comme "volatile" pour vous assurer qu'elle sera mise à jour instantanément, elle n'est pas mise en cache.
- Attention, cet attribut dégrade par contre le temps d'accès à la variable.

```
volatile int F;

void delai(){
  int i,deltaT=F;
```

```
  for(i=0;i<=deltaT;i++) _delay_ms(1);
}

int main(void){
  F=10;
  DDRB=0x1F;
  DDRC=0x00;

  do {
    PB^=1;
    delai();
  }while(1);
  return(0);
}
```

4.1.5 Fonction de manipulation de bits

- Mise à un par masque avec opérateur “ou” |:
- Mise à zéro par masque avec opérateur “et” &:
- Inversion avec opérateur “xor” ∧:

```
// Mise a zero des 4 bits de poids faible
PORTB=PORTB & 0xF0;  // ou bien
PORTB &= 0xF0;

// Mise a un des 4 bits de poids faible d'un port
PORTB = PORTB | 0x0F // ou bien
PORTB |= 0xF0;

// Le Pipe : On utilise un pipe quant on veux modifier un registre
// Si l'on veux simplement initialiser un registre on utilise l'affectation  '='
// Mise a un du bit 4 de PORTD et seulement lui !
PORTD |= 1<<PORTD4;

//Mise a zero du bit PORTD4 et seulement lui !
PORTD &= ~(1<<PORTD4);

// Commutation du bit PORTD4 et seulement lui !
 PORTD ^= (1<<PORTD4);
```

4.2 Mise en oeuvre des interruptions

Une interruption se mete en oeuvre sont :

- L'autorisation de la source d'IT : Par exemple pour le timer $TOIE_i$ pour le timer i, ou le $ACIE$ pour l'ADC.
- L'autorisation de toutes les ITs : sei();
- La déclaration de la routine d'interruption : Directive ISR

La directive ISR associe une source d'interruption à une routine :

```
ISR(TIMER0_OVF_vect){ // routine d'interruption
  PORTD = PORTD^0x10;   // Commutation du but num 4
  TCNT0=0;
}
int main(void)
{
   DDRD = 0b1111011;   // Port D en sortie, PORTD2 en entree
   TCNT0= 0;  // valeur initiale du compteur
   TCCR0 = 5; // facteur de predivision de clk/IO
   sei(); // Toutes les ITs sont possibles
   TIMSK |= (1<<TOIE0); // Validation de l'It de debordement
   do {} while(1);
   return(1);
}
```

Description des sources d'Interruptions de l'$ATMEGA_8$

Nom de l'interruption	Description
ADC_vect	ADC Conversion Complete
ANALOG_COMP_0_vect	Analog Comparator 0
ANALOG_COMP_1_vect	Analog Comparator 1
ANALOG_COMP_2_vect	Analog Comparator 2
ANALOG_COMP_vect	Analog Comparator
ANA_COMP_vect Analog	Comparator
EE_RDY_vect	EEPROM Ready
EE_READY_vect	EEPROM Ready
EXT_INT0_vect	External Interrupt Request 0
INT0_vect	External Interrupt 0
INT1_vect	External Interrupt Request 1
IO_PINS_vect	External Interrupt Request 0
LCD_vect	LCD Start of Frame
LOWLEVEL_IO_PINS_vect	Low-level Input on Port B
OVRIT_vect CAN	timer Overrun
PCINT0_vect	Pin Change Interrupt Request 0
PCINT1_vect	Pin Change Interrupt Request 1
PSC0_CAPT_vect	PSC0 Capture Event
PSC0_EC_vect	PSC0 End Cycle
PSC1_CAPT_vect	PSC1 Capture Event
PSC1_EC_vect	PSC1 End Cycle
PSC2_CAPT_vect	PSC2 Capture Event
PSC2_EC_vect	PSC2 End Cycle
SPI_STC_vect	Serial Transfer Complete
SPM_RDY_vect	Store Program Memory Ready
SPM_READY_vect	Store Program Memory Read
TIM0_COMPA_vect	Timer/Counter Compare Match A
TIM0_COMPB_vect	Timer/Counter Compare Match B
TIM0_OVF_vect	Timer/Counter0 Overflow
TIMER0_CAPT_vect	ADC Conversion Complete
TIMER0_COMPA_vect	TimerCounter0 Compare Match A
TIMER0_COMPB_vect	timer Counter 0 Compare Match B
TIMER0_COMP_A_vect	Timer/Counter0 Compare Match A
TIMER0_COMP_vect	Timer/Counter0 Compare Match
TIMER0_OVF0_vect	Timer/Counter0 Overflow
TIMER0_OVF_vect	Timer/Counter0 Overflow
TIM1_CAPT_vect Timer/Counter1	Capture Event
TIM1_COMPA_vect	Timer/Counter1 Compare Match A
TIM1_COMPB_vect	Timer/Counter1 Compare Match B
TIM1_OVF_vect	Timer/Counter1 Overflow
TIMER1_CAPT1_vect	Timer/Counter1 Capture Event
TIMER1_CAPT_vect	Timer/Counter Capture Event
TIMER1_CMPA_vect	Timer/Counter1 Compare Match 1A
TIMER1_CMPB_vect	Timer/Counter1 Compare Match 1B
TIMER1_COMP1_vect	Timer/Counter1 Compare Match
TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
TIMER1_COMPB_vect	Timer/Counter1 Compare MatchB
TIMER1_COMPC_vect	Timer/Counter1 Compare Match C
TIMER1_COMPD_vect	Timer/Counter1 Compare Match D
TIMER1_COMP_vect	Timer/Counter1 Compare Match
TIMER1_OVF1_vect	Timer/Counter1 Overflow
TIMER1_OVF_vect	Timer/Counter1 Overflow

Nom de l'interruption	Description
TWI_vect	2-wire Serial Interface
TXDONE_vect	Transmission Done, Bit timer Flag 2 Interrupt
TXEMPTY_vect	Transmit Buffer Empty, Bit Itmer Flag 0 Interrupt
UART0_RX_vect	UART0, Rx Complete
UART0_TX_vect	UART0, Tx Complete
UART0_UDRE_vect	UART0 Data Register Empty
UART1_RX_vect	UART1, Rx Complete
UART1_TX_vect	UART1, Tx Complete
UART1_UDRE_vect	UART1 Data Register Empty
UART_RX_vect	UART, Rx Complete
UART_TX_vect	UART, Tx Complete
UART_UDRE_vect	UART Data Register Empty
USART0_RXC_vect	USART0, Rx Complete
USART0_RX_vect	USART0, Rx Complete
USART0_TXC_vect	USART0, Tx Complete
USART0_TX_vect	USART0, Tx Complete
USART0_UDRE_vect	USART0 Data Register Empty
USART1_RXC_vect	USART1, Rx Complete
USART1_RX_vect	USART1, Rx Complete
USART1_TXC_vect	USART1, Tx Complete
USART1_TX_vect	USART1, Tx Complete
USART1_UDRE_vect	USART1, Data Register Empty
USART2_RX_vect	USART2, Rx Complete
USART2_TX_vect	USART2, Tx Complete
USART2_UDRE_vect	USART2 Data register Empty
USART3_RX_vect	USART3, Rx Complete
USART3_TX_vect	USART3, Tx Complete
USART3_UDRE_vect	USART3 Data register Empty
USART_RXC_vect	USART, Rx Complete
USART_RX_vect	USART, Rx Complete
USART_TXC_vect	USART, Tx Complete
USART_TX_vect	USART, Tx Complete
USART_UDRE_vect	USART Data Register Empty
USI_OVERFLOW_vect	USI Overflow
USI_OVF_vect	USI Overflow
USI_START_vect	USI Start Condition
USI_STRT_vect	USI Start
USI_STR_vect	USI START
WATCHDOG_vect	Watchdog Time-out
WDT_OVERFLOW_vect	Watchdog timer Overflow
WDT_vect	Watchdog Timeout Interrupt

Travaux dirigés micro 8 bits, GE3

David Delfieu

November 15, 2023

1 *PORTS* d'entrées-sorties

On souhaite avec les *PORTS* d'E/S d'un μC de type $ATMEGA_8$ pour lequel on considère les connections suivantes :

- 3 leds sont connectées sur les voies PB_7, PB_6, PB_5,
- 5 boutons de commande sont connectés sur les voies $PB_4, PB_3, PB_2, PB_1, PB_0$.
- 8 autres led sont connectées sur $PORT_C$

On donne le cahier des charges suivant : À l'appui d'un seul des 5 boutons de commande, un chenillard démarre en partant de PD_0 jusqu'a PD_7. La led connectée à la PC_0 va s'allumer d'abord pendant 500 ms puis celle de PC_1,... jusqu'à PC_5, après PC_5, c'est la led connectée sur PC_0 qui s'allume et le cycle repart. Ce chenillard se déplace avec une fréquence de 2 Hz pendant 10 s.

Dans le même temps on affichera sur PB_7, PB_6, PB_5 le numéro du bouton de commande qui aura été activé. Par exemple, si le bouton 3 a été activé on aura sur ces bits la valeur 011.

1.1 Algorithme

Donner l'algorithme de premier niveau du code à réaliser.

1.2 Initialisation

On s'intéresse dans un premier temps à l'initialisation des *PORTS* :

1. Coder en décimal, hexadécimal et binaire la configuration des *PORTS* B et D
2. On souhaite améliorer la sûreté de fonctionnement en activant les résistances de tirages de pull-down sur les boutons de commande. Donner l'intérêt des résistances de tirage, puis le code pour les activer.

1.3 Affichage

On cherche ensuite à gérer les différents affichages. Pour simplifier l'écriture on définie les masques suivant $M_1 = 0x1F$ et $M_2 = \overline{M_1} = 0xE0$. Considérons une variable de type short X initialisée à la valeur 177.

1. Donner le résultat du calcul binaire $X\&M_1$ et $X|M_2$.
2. A l'aide de l'annexe, trouver une écriture alternative à ces masques.
3. Donner le code complet correspondant à l'algorithme.

1.4 Lecture de Port

Le banc de leds est maintenant connecté à $PORT_C$. $PORT_B$ est utilisé en entrée. Considérons le programme suivant :

```
#define ALLUME 0xFF;
#define ETEINT 0x00;
#define TOUTES 0xFF;
#define AUCUNE 0x00;
#define PREMIER 0b00000001;

int main(void){
  int test,i;

  DDRB  = AUCUNE;
  SFIOR &=~(1<<PUD);
  PORTB = PREMIER; //Resistance de tirage sur PB0

  DDRC  = TOUTES;
  PORTC = ETEINT;

  while(1){
    test=PINB & PREMIER;
    if (test==0) PORTC =  ALLUME;
    else PORTC = ETEINT;
    for(i=0;i<500;i++) _delay_ms(10); //on attend 5 secondes
  }
}
```

- Que fait ce programme ?
- Comment définiriez-vous le temps de cycle de ce programme ?
- Si l'entrée PB_0 passe à 0, le $PORT_C$ est-il immédiatement commuté ?
- Quel élément du μC pourriez-vous utiliser pour obtenir l'exécution d'une tâche **aussitôt que** PB_0 passe à zéro ?

Annexe

- Mise à un par masque avec opérateur "ou" |:
- Mise à zéro par masque avec opérateur "et" &:
- Inversion avec opérateur "xor" $\wedge$:

```
// Initialisation ou modification ?
//  Utiliser |= ou &= quant on veux modifier partiellement un registre
//  Pour initialiser simplement un registre on utilise l'affectation =
PORTD  = 0b0000 0011;
PORTD |= 1<<PORTD4;   // PORTD = Ob0001 0011;
PORTD  = 0b0000 0011;
PORTD = 1<<PORTD4;    // PORTD = Ob0001 0000;

// SET 4 bits de poids faible d'un port SANS MODIFIER les autres bits
```

```
PORTB = PORTB | 0x0F; // ou bien
PORTB |= 0xF0;

// RAZ des 4 bits de poids faible SANS MODIFIER les autres bits
PORTB=PORTB & 0xF0;  // ou bien
PORTB &= 0xF0;

// RAZ du bit 4 du PORT D SANS MODIFIER les autres bits
PORTD &= ~(1<<PORTD4);

// Commutation du bit 4 du PORT D SANS MODIFIER les autres bits
 PORTD ^= (1<<PORTD4);
```

2 Interruptions externes

L'$ATMEGA_8$ a deux pattes d'interruption externe INT_0 et INT_1 . Toutes les pattes de l'$ATMEGA_8$ (sauf alimentation) ont plusieurs fonctions. Ainsi, la patte PD_2 correspond à l'interruption externe INT_0 comme la patte PD_3 pour l'interruption INT_1. Afin d'utiliser ces interruptions externes, on peut distinguer trois phases de programmation :

1. Donner les instructions de masque sur les registres $SREG, GICR$ *et* $MCUCR$ qui permettent de configurer les bits qui autorisent le déclenchement d'une interruption lorsqu'un front montant intervient sur la patte INT_0.
2. Compléter le programme présenté en section 1.4 de manière à ce que $PORT_C$ soit incrémenté à chaque front montant sur la patte INT_0.

MCUCR : $SE\ SM_2\ SM_1\ SM_0\ ISC_{11}\ ISC_{10}\ ISC_{01}\ ISC_{00}$
GICR : $INT_1\ INT_0 - - - - IVSEL\ IVCE$
SREG : $I\ T\ H\ S\ V\ N\ Z\ C$

Table 1: ISCx1 et ISCx0

ISCx1	ISCx0	Trigger
0	0	un niveau bas génère une *Interruption*
0	1	Front montant ou front descendant
1	0	Front descendant
1	1	Front montant

3 Chenillard et *Interruption Externe*

Faire un chenillard par défaut qui tournera sur $PORT_C$ dans un sens que l'on appellera sens positif.

- Pour un front montant sur PD_3 faire tourner dans le sens négatif.
- Pour un front descendant sur PD_2 faire tourner dans le sens positif.

4 Chenillard et *Convertisseur Analogique*

Faire un chenillard sur $PORT_C$. Mettre en place une conversion analogique du bit PC_3 qui règlera la vitesse du chenillard.

- On fonctionnera on mode lecture analogique scrutative.
- Puis on fonctionnera en mode free-running.

5 Leds et *Timer* 2

Faire clignoter $PORT_C$ avec une temporisation réalisée grâce à *Timer* 2. L'interruption de débordement du timer doit provoquer le clignotement.

6 *Timer* 1 et *Convertisseur Analogique*

- A l'aide du timer 1, générer une PWM centrée avec un rapport cyclique constant de 60% sur PB_2 et une fréquence de hachage de 12khz. Observer les signaux à l'oscilloscope.
- Faire varier la fréquence avec une conversion analogique sur PC_2 selon un mode échantilloné : L'interruption de débordement du timer 1 doit lancer la conversion.

TP1 : La station de fabrication de béton

Introduction

Tout automatisme industriel doit être évidement réalisé avec un automate proprement dit (Schneider, Siemens,...). Cependant, la conception d'un automate étant réalisée à partie de μC, il est intéressant, d'un point de vue pédagogique, d'étudier le codage d'un automate en langage C et c'est l'objet de ces Travaux Pratiques.

La maquette de Travaux Pratique modélise la fabrication de béton, par un dosage de sable et de ciment. Ce processus est modélisé par le Grafcet de la figure 1.

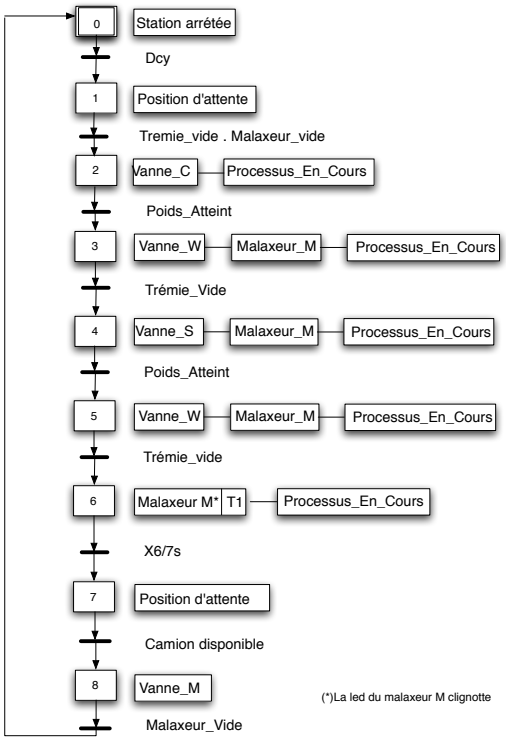


Figure 1: Grafcet

- La maquette comporte 6 sorties, des entrées vu du coté micro : $PA_0, PA_1, PA_2, PA_3, PA_4, ALARM$
  Elles correspondent aux interrupteurs de la carte. Ces derniers permettent de valider les différentes transitions du Grafcet. On utilisera des puissances de 2 dans les étiquettes car si l'on avait choisi un nombre. Par exemple considérons que l'on lit 5 sur nos entrées. 5 peut se décomposer en $4+1$ ou $3+2$, il y a ambiguïté sur les entrées (du micro) possibles. Soit on considère que les entrées 4 et 1 sont actives soit ce sont les entrées 3 et 2.

- La maquette comporte 8 entrées, des sorties vues du coté micro : $PB_0, PB_1, PB_2, PB_3, PB_4, PB_5, PB_6, PB_7$
  Elles permettent d'activer des actions sur les étapes du Grafcet. Elles serviront aussi à commander les leds simulant le processus de fabrication de béton. Les étiquettes attachées à ces entrées correspondront à des nombres croissants.

1 Initialisations

Nous allons utiliser l'environnement Arduino [1]

1. Positionner les options dans le menu "outils" :

   - Type de carte : NG or older
   - Processeur : Processeur $ATMEGA_8$
   - Port : choisissez le port "COM 1"
   - Programmateur : $USBASP$

   Pour compiler vous utiliserez dans le menu "croquis" ou "fichier" : Téléverser avec un programmeur

2. Dans votre fichier C principal, éditer les étiquettes suivantes :

```
#define DEPART_CYCLE   1     // entrees PORTC
#define TREMIE_VIDE    2  // entrees PORTC
#define MALAXEUR_VIDE  4  // entrees PORTC
#define POIDS_ATTEINT  8  // entrees PORTC
#define CAMION_DISPO   16  // entrees PORTC
#define TEMPO          32  // entrees PORTC

#define POSITION_ATTENTE     0  //  Attention  PORTB !
#define VANNE_C              0  //  Sorties PORTD
#define VANNE_S              1  //  Sorties PORTD
#define MALAXEUR_M           3  //  Sorties PORTD
#define VANNE_W              4  //  Sorties PORTD
#define VANNE_M              5  //  Sorties PORTD
#define PROCESSUS_COURS      6  //  Sorties PORTD
#define STATION_ARRETEE      7  //  Sorties PORTD
```

3. Effectuez les câblages des sorties de la maquette sur les ports $PORTC$ et $PORTD$ de la carte μC :

   - PA_0 : Départ cycle sur PC_0 ;
   - PA_1 : Trémie vide sur PC_1 ;
   - PA_2 : Malaxeur vide sur PC_2 ;
   - PA_3 : Poids atteint sur PC_3 ;
   - PA_4 : Camion disponible sur PC_4 ;
   - $ALARM$ sur PD_2.

4. Effectuez les câblages des entrées de la maquette sur les ports $PORTD$ et $PORTB$ de la carte μC :

   - PB_0 : Position d'attente sur PB_0 ;
   - PB_1 : Vanne C sur PD_0 ;
   - PB_2 : Vanne S sur PD_1 ;
   - PB_3 : Malaxeur M sur PD_3 ;

[1] Deux versions logicielles d'Arduino cohabitent sur les PC et nous allons utiliser la 1.6.4 (Icone ArduinoGE3) pour tous les Travaux Pratique de première année.

- PB_4 : Vanne W sur PD_4 ;
- PB_5 : Vanne M sur PD_5 ;
- PB_6 : Processus en cours sur PD_6 ;
- PB_7 : Station arrêtée sur PD_7.

5. Relier la masse de la carte micro (chercher la patte GND) à celle de l'alimentation.

6. Fonction *transition* : Ecrire la fonction *etape1()*, compiler et téléverser et tester pour différentes valeurs de conditions.

```
void transition(int condition) {
  int capteur=0;
  // if (condition == TEMPO) {??}  else
  do {
    capteur = PINC;
    capteur = capteur & condition;
  } while(capteur != condition);
}

void etape0(void){PORTD=(1<<STATION_ARRETEE);}

int main(void)
{
   DDRC = 0x00;          // port C en entree
   DDRB = 0x01;          // port B : PB0 en sortie
   DDRD = 0b11111011;    // port D en sortie sauf  PD2
                         // PD2 : entree d'interruption externe
  do {
    etape0();
    transition(DEPART_CYCLE+CAMION_DISPO);
    etape1();
    transition(MALAXEUR_VIDE);
  } while(1);            // le G7 est en boucle infinie
}
```

2 Le Grafcet

- Compléter la fonction *transition* pour aussi prendre en compte la temporisation entre les étapes X_6 et X_7 à la place d'une condition logique. Le clignotement de la led $MALAXEUR\_M$ de l'étape X_6 doit être réalisé à avec une fréquence de $1hz$. La temporisation pourra être gérée par la fonction *_delay_ms()*.

- Compléter le programme principal pour mettre en oeuvre le Grafcet intégralement (voir la figure 1).

3 Arrêt d'urgence

Modifier le programme de façon à pouvoir traiter un arrêt d'urgence. On considèrera le signal d'alarme comme un front descendant. L'évènement lié à l'interruption se nomme $INT0\_vect$. Le type d'événement susceptible de déclencher une interruption externe se programme à l'aide des bits ISC_{01}, ISC_{00} du registre $MCUCR$. Tandis que cette interruption se valide à l'aide du bit INT_0 du registre $GICR$.

Le sous-programme d'interruption lié à l'arrêt d'urgence devra sauvegarder l'état des leds puis il devra faire clignoter l'ensemble des leds commandables pendant 5 secondes, puis reprendre le Grafcet là, où il a été interrompu avec les leds qui seront restaurées. Le délai de clignotement sera gérée par la fonction $\_delay\_ms()$.

4 Processus En cours

Nous allons modifier la prise en compte de la led PROCESSUS_EN_COURS. Celle-ci va dorénavant, clignoter tout au long du déroulement des étapes X_2, X_3, X_4, X_5, X_6. Pour le délai, que vous fixerez avec une valeur quelconque, du clignotement vous utiliserez l'interruption de débordement du timer 0. En plus de l'usage du pré-diviseur de fréquence, vous devrez re-diviser la fréquence de clignotement à l'aide d'une variable statique dans le sous-programme d'*interruption*. Vous activerez cette interruption, à l'aide de son bit de masquage, à l'étape 2 et vous la désactiverez à l'étape 7.

Rappels :

- Le bit de masquage se nomme $TOIE_0$ et appartient au registre $TIMSK$.
- La valeur initiale de comptage du timer 0 est faite dans le registre $TCNT_0$.
- $TCCR_0$ contrôle le pré-diviseur de fréquence.
- L'interruption d'Overflow du timer zéro, se nomme $TIMER0\_OVF\_vect$.

5 Compte rendu

- Une introduction (0,5 page max) décrit le but du TP.
- Chaque question du TP doit donner lieu à un texte décrivant le test et les résultats d'exécution accompagnés quand cela est necessaire par un algorithme ou un organigramme. Vous préciserez comment vous avez testé votre code (résultats visuels, éventuellement mesures à l'oscilloscope) et vous donnerez quand cela est possible des extraits quantitatifs ou visuels (photos ou gifs animés) des résultats d'exécution.
- La conclusion (1 page max) se définit comme une recherche d'une application industrielle des notions abordées en TP. Cette conclusion doit faire clairement apparaître une recherche documentaire.
- En annexe, donnez votre code commenté : Pour chaque sous-programme ou routine d'interruption vous donnerez les informations en commentaire suivant :

```
Fonction  : Nom de la fonction et des paramètres avec leur types ;
    * Entrées : Paramètres en entrée de la fonction ;
    * Sorties : Paramètres modifiés par la fonction (si il y en a) ;
Variables globales utilisées et sous-programmes appelés ;
Fonctionnement : description rapide/sommaire du fonctionnement.
```

TP2 : Commande d'un moteur pas a pas

Introduction

Ce moteur comporte 48 pas correspondant à des secteurs angulaires identifiés sur une corolle. Le secteur 1 contient un ajourement rendant ainsi détectable la position zéro du moteur. La maquette supportant le moteur pas à pas comprend :

- 4 bits de commande ;
- Une borne COMPT qui passe à *un* dès que l'optocoupleur détecte la position 1 du moteur pas à pas. Cet état est mémorisé grâce à une bascule D (74HCT74).
- La borne Reset (a coté de compt) permettant de remettre à zéro la bascule D ;
- Un interrupteur marche/arrêt permettant de couper l'alimentation du moteur.
- Un potard permet de délivre une tension variable $V \in [0, 5]$ entre la borne se trouvant près de l'optocoupleur et la masse. On trouve aussi un connecteur d'alimentation et de masse.

Dans ce sujet on abordera la commande du moteur pas à pas.

1 Initialisations

1. Créer un projet ;
2. Effectuer les câblages suivant :
   - PB_0 sur la borne 0 ;
   - PB_1 sur la borne 1 ;
   - PB_2 sur la borne 2 ;
   - PB_3 sur la borne 3 ;
   - PD_2 sur la borne $COMPT$;
   - PB_4 sur la borne RST ;
3. Relier la masse de la carte à celle de l'alimentation.
4. Relier une sonde d'oscilloscope sur une des commandes de bobines du moteur

2 Commande moteur

2.1 mondelai

Ecrire une fonction *mondelai* qui réalise un délai qui est un multiple entier de 100 microsecondes. Pour cela vous appellerez la fonction $delayMicroseconds(100)$.

2.2 Mode 1 : Pas entier, une bobine

En utilisant *mondelai*, écrire une fonction appellée dans le *main* pour réaliser une fonction permettant de faire tourner le moteur infiniment, en pas entier à vitesse constante dans le sens horaire de 48 pas. Tous les 4 pas commuter le bit PD_0. Faire varier le délai et observer l'image de la vitesse sur bit PD_0 pour obtenir la vitesse maximale à l'oscilloscope.

2.3 Mode 2 : Pas entier, deux bobines

En utilisant *mondelai*, écrire une fonction appellée dans le *main* pour réaliser une fonction permettant de faire tourner le moteur infiniment, en pas entier, deux bobines par deux bobines, à vitesse constante dans le sens horaire de 48 pas. Tous les 4 pas commuter le bit PD_0. Faire varier le délai et observer l'image de la vitesse sur bit PD_0 pour obtenir la vitesse maximale à l'oscilloscope.

2.4 Mode 3 : demipas

En utilisant *mondelai*, écrire une fonction appellée dans le *main* pour réaliser une fonction permettant de faire tourner le moteur de infiniment, en demi-pas dans le sens trigonométrique de 48 pas. Tous les 8 pas commuter le bit PD_0. Faire varier le délai et observer l'image de la vitesse sur bit PD_0 pour obtenir la vitesse maximale à l'oscilloscope.

2.5 Modes et vitesse maximale

- Quel est le mode qui permet d'atteindre la plus grande vitesse maximale ?
- Trouver une explication physique de la vélocité d'un mode par rapport aux autres ?

2.6 changement de mode

A l'aide du bit PB_5 défini en entrée, ré-écrire vos fonctions pour permettre de changer de mode : Pas entier sens horaire vers le mode demi-pas sens trigonométrique (et vice-versa). Pour que la prise en compte du changement de sens puisse se faire entre deux pas, on va réaliser un test de PB_5 dans la boucle de commande des pas et on sortira en case de test positif (c.a.d. si PB_5 indique qu'il faut changer de sens) de cette boucle par l'instruction break.

2.7 Convertisseur Analogique Numérique

Relier la borne du potard à PC_0.

- Initialiser le CAN en mode en 8 bits et utiliser le mode free-running sans timer ni interruption.
- Utiliser la valeur de conversion pour déterminer le délai qui réglera la vitesse du moteur.

3 Compte rendu

- Une introduction (0,5 page max) décrit le but du Travaux Pratique.
- Chaque question du Travaux Pratique doit donner lieu à un texte décrivant le test et les résultats d'exécution accompagnés quand cela est necessaire par un algorithme ou un organigramme. Vous préciserez comment vous avez testé votre code (résultats visuels, éventuellement mesures à l'oscilloscope) et vous donnerez quand cela est possible des extraits quantitatifs ou visuels (photos, gifs) des résultats d'exécution.

- La conclusion (1 page max) se définit comme une recherche d'une application industrielle des notions abordées en Travaux Pratique. Cette conclusion doit faire clairement apparaître une recherche documentaire ou de type internet.

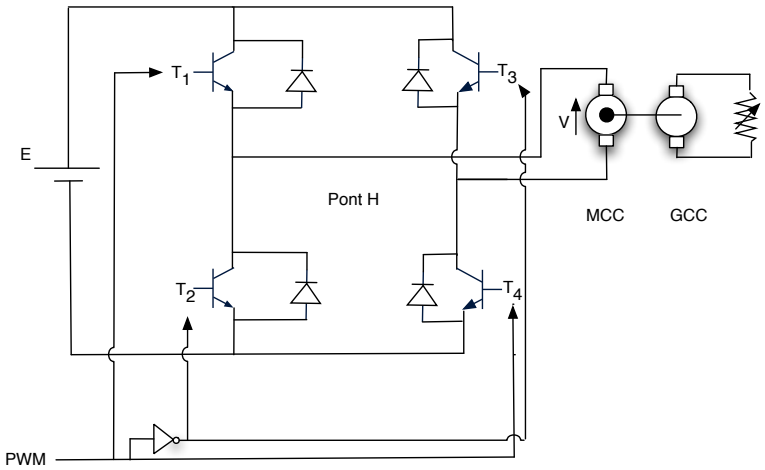
- En annexe, donnez votre code commenté : Pour chaque sous-programme ou routine d'interruption vous donnerez les informations en commentaires suivant :

```
Fonction  : Nom de la fonction et des paramètres avec leur types ;
    * Entrées : Paramètres en entrée de la fonction ;
    * Sorties : Paramètres modifiés par la fonction (si il y en a) ;
Variables globales utilisées et sous-programmes appelés ;
Fonctionnement : description rapide/sommaire du fonctionnement.
```

TP3: Commande d'une Machine courant continu

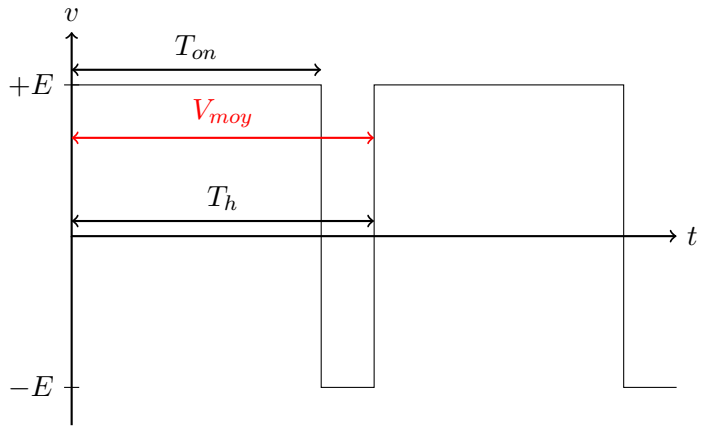
Introduction

La figure 1 décrit une Machine à Courant Continu(MCC) commandée par le biais d'un hacheur. Une génératrice est couplée au moteur, elle se comporte comme un frein grâce à un ensemble de résistances variables. Le schéma de principe du hacheur est donné dans la figure suivante :



On a 4 transistors qui sont appairés en 2 couples (T_1, T_4) et (T_2, T_3). Ces deux couples sont reliés à une même ligne de commande PWM, mais le second couple à travers une porte inverseuse (74LS04). Donc suivant le niveau logique sur cette ligne on a soit (T_1, T_4) passant soit (T_2, T_3) passant. Si (T_1, T_4) sont passant alors (T_2, T_3) sont bloqués et on a une tension de $+E$ aux bornes du moteur, sinon on a $-E$.

L'objet de ces Travaux Pratique est de réaliser une commande en boucle ouverte de la MCC. Soit T_h une période appelée période de hachage, le rapport cyclique $\frac{T_{ON}}{T_h}$ permet la commande de la MCC.
En hachant la tension de commande E, la PWM permet de produire une tension moyenne V_{moy}, que l'on peut calculer en faisant la somme des aires, comme le montre la figure suivante :



La somme des aires pour une période T_h produit le calcul de l'intégrale suivante :

$$V_{moy} = \frac{1}{T_h}\left(\int_0^{T_{on}} E.dt + \int_{T_{on}}^{T_h} -E.dt\right)$$

Prédétermination

- Retrouver les expressions littérales de T_{on} et de $\frac{T_{on}}{T_h}$ en fonction de V_{moy}, E, T_h et de V_{moy} en fonction de T_{on}, T_h, E.
- Déterminer la valeur numérique de $\frac{T_{on}}{T_h}$ pour obtenir une tension moyenne de 30 v sachant que $E = 35v$.
- Etablir dans une feuille de calcul (libreOffice) la formule de cours, qui lie OCR_1A, F_{hash}, N et F_{clkio}. N ne prenant que cinq valeurs $(1, 8, 64, 256, 1024)$, trouver tous les couples (N,OCR_1A) valides pour établir une fréquence de hachage à 15 khz. Quel est le meilleur couple et pourquoi ?

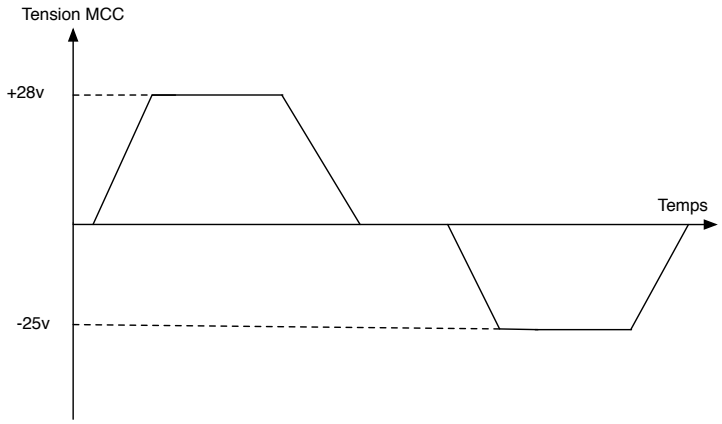
Branchements

- Brancher PB_2 sur l'entrée numérique de la maquette ;
- Brancher la masse de la carte $ATMEGA_8$ à une masse de la maquette.
- Placer le commutateur sur entrée numérique (maquette éteinte).

1 Commande de la MCC

1.1 Profils de tensions

On veut dans un premier temps commander la MCC avec une commande avec le profil de tensions suivant :



- Cette courbe de tensions permet de modéliser une accélération dont vous fixerez la pente par la fonction _delay_ms, suivie d'une vitesse constante suivie d'une décélération et un petit palier à vitesse constante. Un cycle en tensions négatives termine le cycle. Pour cela vous utiliserez le $TIMER_1$ en mode "phase et fréquence correcte", on utilisera la patte $OC1_B$ pour générer la PWM. On appliquera une fréquence de hachage de $15khz$. Sachant deux tensions correspondant aux vitesses "plateaux" (28 v et -25v). Calculer les valeurs initiales des différents registres d'initialisations de la PWM.
- Visualisez la vitesse et le courant à l'oscilloscope. Appliquez une charge et déterminer comment varient la vitesse et le courant.

1.2 Commande par un potard analogique

On va commander maintenant la MCC à l'aide du seul potard de la carte μC et à l'aide du module de conversion analogique. Le potard devra commander le moteur entre les tensions $[-35, +35]$. Ecrivez trois versions de la conversion en utilisant les modes suivants :

1. Utiliser une conversion sur 10 bits en mode interruptif échantillonné : Attention que la relance de la conversion soit bien faite dans l'IT de débordement du Timer et non pas dans l'IT du Can (on ne change pas la fréquence $f_{ech} = 10khz$ ni le mode). On réalise donc 2 interruptions : L'interruption de débordement du Timer 1 pour lancer la conversion et l'interruption de fin de conversion qui permettra de lire le résultat de la conversion. La valeur lue servira à définir la variation de vitesse du moteur.

2. Utiliser une conversion sur 8 bits, sans interruption, en mode attente active : Boucle d'attente scrutative du bit $ADCS$ (appartenant au registre $ADCSRA$) dans le programme principal.

Pour chacun des 3 modes on écrira une fonction $Init\_CAN$ dédiée.

1.3 Compte rendu

Une introduction (1 page max) décrit le but du TP. La conclusion (2 pages max) se définit comme une recherche d'une application industrielle des notions abordées en TP. Faites dans votre conclusion une recherche sur les différents types de moteurs électriques. De plus, dans votre code, pour chaque sous-programme ou routine d'interruption vous donnerez les informations en commentaires suivant :

```
/*-------------------------------------------------------------
Fonction  : nom de la fonction et des paramètres avec leur types ;
Entrées : Paramètres en entrée de la fonction ;
Sorties : paramètres modifiés ou de sortie ;
Variables globales utilisées ;
Sous-programmes appelés ;
Fonctionnement : description rapide/sommaire du fonctionnement.
-------------------------------------------------------------*/
```

Dans votre compte-rendu vous préciserez comment vous avez testé votre code. Vous donnerez des résultats visuels des courbes obtenues à l'oscilloscope (mode run) pour différentes charges.

Contrôle 2019 de Micro 8 bits GE_3 - 1h30 Polycopié autorisé - D. Delfieu

1 Exposé du problème

On veux implémenter un grafcet de commande d'un poste d'usinage. Ce poste décrit sur la figure 2 comporte un plateau qui porte une pièce à usiner. Ce plateau peut se mouvoir horizontalement, à l'aide d'un moteur pas à pas MPP_1 depuis son Début de Course détecté par le capteur DC_1 vers sa Fin de Course détectée par le capteur $FC1$. Une fraise est reliée un moteur de positionnement MPP_2 (moteur pas à pas) vertical. La rotation de la fraise est entrainée par une machine à courant continu par l'actionneur MCC. Le déplacement vertical de la fraise est contrôlé par des capteurs de Début de Course DC_2 et de Fin de Course FC_2.

- Initialement, on attend que le plateau et la fraise se trouvent bien en début de course DC_1 et DC_2 .
- Ensuite, si il n'y pas de pièce détectée alors une pièce est sortie du magasin par l'actionneur $Placer\_piece$.
- Si la pièce se positionne bien, c'est à dire qu'elle est détectée par le capteur $Piece\_detectee$ alors les deux moteurs pas à pas sont actionnés, par l'unique actionneur $MPP++$, pour avancer le plateau portant la pièce vers sa fin de course FC_1 et descendre la fraise jusqu'à sa fin de course FC_2. De plus, la fraise est mise en rotation par l'actionneur MCC.
- Lorsque les deux fins de course sont atteintes, la fraise est maintenue en rotation par l'actionneur MCC. La pièce etant fixée sous la fraise et l'usinage se produit.
- L'utilisateur, lorsqu'il juge que la pièce est suffisement usinée, appuie sur le bouton (capteur) $Fin\_usinage$ qui arrête la fraise par l'actionneur $Stop\_mcc$ et repositionne le plateau et la fraise dans leurs positions initiales (positions définies par les débuts de course DC_1 et DC_2) par l'actionneur $MPP--$.
- La pièce est alors éjectée, par l'actionneur $Ejecter\_piece$ et dès que le capteur $Piece\_detectee$ indiquera que la pièce a effectivement éte éjecté alors le cycle recommencera.

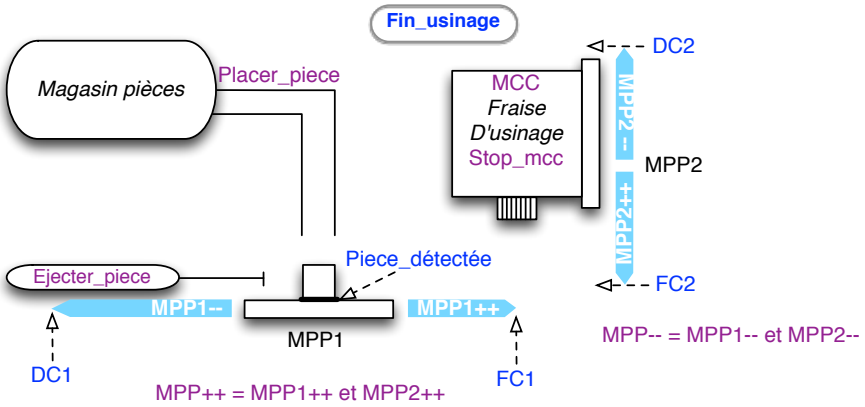


Figure 1: Poste d'usinage

2 Le Grafcet

2.1 Etiquettes

- Faire le bilan des entrées-sorties : Enumérer les capteurs et les actionneurs mentionnés dans l'énnoncé du problème
- Expliquer la façon dont vous aller coder les étiquettes des capteurs et celles des actionneurs. Pourquoi il y a-t-il une différence de codage ? Donner un exemple qui justifie cette différence de codage.
- Ecrire les étiquettes des capteurs et des actionneurs
- Définir un placement des capteurs et des actionneurs sur les ports B, C (D si besoin). On réservera PB_2 pour la MLI de la MCC.

Solution

- Capteurs (6) : DC_1, DC_2, FC_1, FC_2, $Piece\_detectee$, $Fin\_usinage$.
- Actionneurs (6) : $Placer\_piece$, $MPP++$, MCC, $MPP--$, $Stop\_mcc$, $Ejecter\_piece$.

Les entrées sont codées en puissance de 2, les actionneurs sont codées en numéro de bits. En effet cela permet d'éviter toute ambiguité sur les entrées. Avec un codage de numéro de bits sur sur une entrée si on lit une entrée à 5 alors 5 = 4+1 ou 2+3. Ce qui rend la lecture ambigue.

```
#define DC1    1     // PORT C
#define FC1    2
#define DC2    4
#define FC2    8
#define Piece_detectee     16
#define Fin_usinage        32

#define Placer_piece       0    // PORT B
#define MPP++              1
#define MPP--              3
#define MCC                4
#define StopMCC            5
#define Ejecter_Piece      6
```

2.2 Fonction transition

Ecrire la fonction $transition(condition)$ qui teste la combinaison $condition$ d'éventuellement plusieurs capteurs. Cette condition est passée en paramètre. Cette fonction teste et attend la bonne combinaison de capteurs. Dans le cas particulier ou cette condition est $Piece\_detectee$ ou $\overline{Piece\_detectee}$, la fonction teste la condition et doit retourner les résultats 0 ou 1, suivant que la pièce soit présente ou non. Dans tous les autres cas on attend la condition et retourne une valeur quelconque qui sera ignorée.

Solution :

```
int transition(int condition) {
  int lu=0;
  if (condition==0) || (condition == Piece_detectee) {
    lu=PINC & Piece_detectee;
    if (lu==0) return 0
    else return 1;
  }
  else {
    do {
      lu = PINC;
      capteur = capteur & condition;
    } while(capteur != condition);
    return (3);
}
```

2.3 Grafcet

1. Dessiner un G_7 qui décrit le fonctionnement de ce système.
2. Ecrire le code du G_7 qui décrit le fonctionnement de ce système : Main et fonctions nécessaires (ne pas re-écrire le code déjà produit).

Solution :

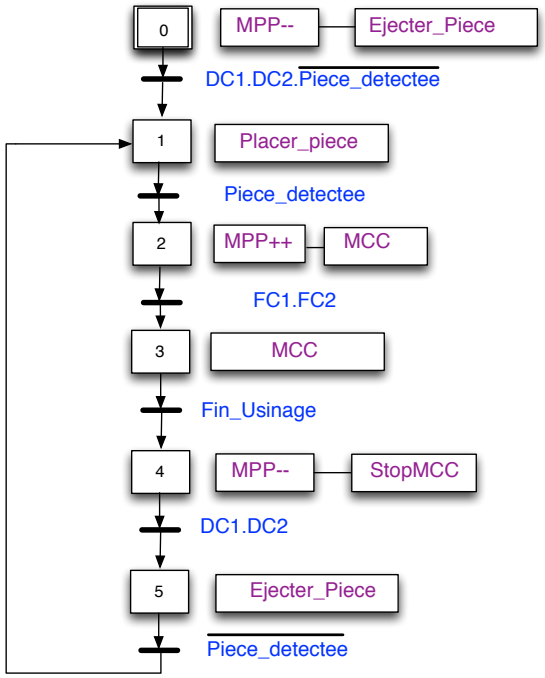


Figure 2: Poste d'usinage

```
void etape0(void){}
void etape1(void){}
void etape2(void){PORTB=(1<<Placer_piece);}
void etape3(void){PORTB=(1<<MPP++)+(1<<MCC);}
void etape4(void){PORTB=(1<<MCC);}
void etape5(void){PORTB=(1<<MPP--)+(1<<Stop_mcc);}
void etape6(void){PORTB=(1<<Ejecter_piece);}

int main(void){
   DDRC = 0x00;   // port C en entree
   DDRB = 0x04;   // port B : en entree sauf PB2 en sortie

    etape0();
    transition(DC1+DC2);
    etape1();
    transition(0);
    do {
      etape2();
      transition(Piece_detectee);
      etape3();
      transition(FC1+FC2);
      etape4();
      transition(Fin_usinage);
      etape5();
      transition(DC1+DC2);
      etape6();
      transition(0);
    } while(1);
}
```

3 Commande de moteur

1. Pourquoi utilise-t-on une machine à courant continu et pas un moteur pas à pas pour la fraise d'usinage ?

2. Commande de MCC : On va activer la MLI en phase et fréquence correcte qui commande par la patte OC_1B (PB_2) une machine à courant continue, via un pont H. Ce moteur est alimenté en $35v$. Cette MCC entraine la fraise. La fraise va accélérer sa vitesse de rotation jusqu'à un plateau définie par une tension de commande de 20 volts. On utilisera une fréquence de hachage de 2khz. Cette fonction définira les valeurs des registres $TCCR_1A$ et $TCCR_1B$. Rappel : Fréquence de hachage $F_h = \frac{12*10^6}{2*N*(1+OCR1A)}$. OCR1A définit le TOP et OCR1B définit le seuil.

   (a) Calculer les valeurs de OCR_1A et N. A quelle valeur de OCR_1B correspondent $0v$ et $20v$?

   (b) Ecrire la fonction MCC(), qui aura le profil de vitesse suivant (figure 3) :

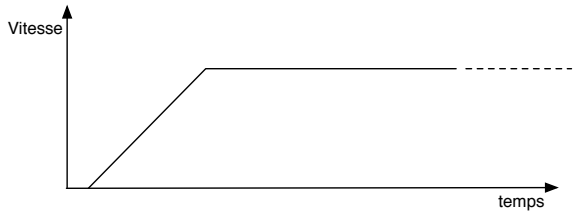


Figure 3: Mise en rotation de la fraise

Solution :

1) On utilise une MCC pour la fraise d'usinage car elle offre à la fois plus de couple et une vitesse maximale plus élevèen que le moteur pas à pas.

2) TCCR1A = (1«COM1B1) + (1«COM1B0) + (1«WGM10);
TCCR1B = (1«WGM13) + (1«CS10);

3a) Posons N=1, $f_h = 2khz \ \rightarrow OCR_1A = (12 * 10^6)/2 * 1 * 2000 = 3000$

OCR_1A est entier donc la valeur trouvée est pertinente. On a un zéro logique à 1500. La valeur 20 volts correspondond à OCR1B=1500+1000=2500.

3b)

```
void MCC(void){
  DDRB |=0x04;
  TCCR1A = (1<<COM1B1) + (1<<COM1B0) + (1<WGM10);
  TCCR1B = (1<<WGM13) + (1<CS10); // N=1 OCR1A+3000
  OCR1A=3000;
  _delay_ms(1000);
  for (i=1500;i<2357;i++) {
    OCR1B=i;
    _delay_ms(1);
  }
  OCR1B=2500;
}
```