

Projet de Fin d'Année II

**Commande à but didactique de convertisseurs par
microcontrôleur :
Hacheurs, Redresseurs ou Gradateurs monophasé et triphasé**

Réalisé par :

Tarek BEN DAHOU et Ahmed LIMEM

Classe : 2AGE2

Encadré par : Férid KOURDA

Soutenu le : Mercredi 09/05/2018

Devant le Jury :

Président	: M. Faouzi BOUANI
Rapporteur	: M. Béchir REBHI
Encadrant	: M. Férid KOURDA

Année universitaire 2017/2018

Remerciements

Nous tenons à exprimer nos profonds remerciements :

A notre encadreur de PFA : Monsieur Férid KOURDA pour tous ses efforts, ses conseils précieux, son savoir-faire et pour la confiance qu'il nous a témoignée, ce qui a permis d'aboutir à ce travail.

Au groupe du laboratoire SYSTEME ELECTRIQUE de l'Enit, en particulier à Monsieur Khaled JLASSI, pour son encouragement et sa disponibilité.

Aux enseignants qui ont contribué à notre formation.

Enfin nous tenons également à remercier gracieusement toute personne qui a contribué de près ou de loin à la réalisation de ce travail.

Résumé

Il s'agit de modifier les commandes existantes en mode analogique et à composants discrets par une commande numérique avec affichage. Ces commandes sont destinées pour :

- Hacheur élévateur de tension
- Redresseur monophasé et triphasé
- Gradateur monophasé et triphasé

Pour ses convertisseurs fonctionnant en alternatif, il est nécessaire de concevoir un signal de synchronisation sur secteur stable, puis d'établir la commande par retard de phase.

Un décalage de 120° puis 240° nous permettra de déterminer précisément les temps de passage par zéro des autres phases de synchronisation.

Mots clés : microcontrôleur, hacheur, gradateur, redresseur monophasé et triphasé, système embarqué.

Abstract

It involves modifying the existing commands in analog mode and discrete components by a digital control with display. These commands are intended for:

- Chopper Voltage Lift.
- Single-phase rectifier.
- Single-phase and three-phase dimmer.

For its AC converters, it is necessary to design a stable sector synchronization signal and then establish the phase delay control.

An offset of 120° then 240° will allow us to accurately determine the zero crossing times of the other synchronization phases.

Key words: microcontroller, chopper, dimmer, rectifier single-phase and three-phase, embedded systems.

Table des matières

Tables des figures	4
Liste des tableaux	5
Introduction générale	6
1 Convertisseur électrique statique	7
1.1 Introduction.....	7
1.2 le hacheur	7
1.2.1 Définition	7
1.2.2 Le rapport cyclique α	7
1.2.3 Etude de hacheur élévateur boost	8
1.2.3.1 Définition	8
1.2.3.2 hypothèses.....	8
1.2.3.3 Principe de fonctionnement	8
1.2.3.4 Chronogramme.....	10
1.2.4 Domaines d'application de hacheur élévateur	10
1.3 Le redresseur	10
1.3.1 Définition	10
1.3.2 différents types de redresseurs monophasé	10
1.3.2.1 redresseur monophasé simple alternance à charge résistive	10
1.3.2.2 redresseur monophasé simple alternance à charge inductive	12
1.3.2.3 Redresseur monophasé a point milieu P2 à charge R.....	13
1.3.2.4 Redresseur monophasé en pont tout thyristor PD2	14
1.3.2.5 Redresseur monophasé en pont tout thyristor	16
1.3.3 différents types de redresseurs triphasé	17
1.3.3.1 Redresseur triphasé a point milieu	17
1.3.3.2 Redresseur triphasé à parallèle double PD3.....	19
1.3.3.3 Redresseur triphasé à pont mixte	20
1.4 Le gradateur	21
1.4.1 gradateur monophasé charge à charge resistive	21
1.4.2 gradateur monophasé charge à charge inductive	22
1.4.3 gradateur à train d'ondes	23
1.5 Domaines d'application de différents types de gradateurs et de redresseurs.....	24
1.6 Le thyristor	25
1.6.1 Principe et composition	25
1.6.2 Amorçage d'un thyristor.....	25
1.6.2 blocage d'un thyristor.....	26
1.6 Conclusion	26
2 La carte à microcontrôleur Arduino	27
2.1 Introduction.....	27
2.2 Connaissance générale sur l'arduino.....	27
2.2.1 Arduino uno	27

2.2.2	Famille des microcontrôleurs AVR.....	28
2.2.3	Les outils logiciels de l'arduino : Arduino IDE	29
2.2.4	langage de programmation de l'arduino	29
2.2.5	Arduino Uno contre Mega	30
2.3	Manipulation des entrées / sorties d'une carte Arduino	30
2.3.1	les commandes prédéfinies des I/O pins	30
2.3.2	Manipulation directe des GPIO ports de l'arduino	31
2.3.2.1	Pourquoi on recourt à la manipulation directe des ports en laissant à coté les commandes prédéfinies d'arduino ?.....	31
2.3.2.2	Les registres des GPIO ports pour l'arduino	32
2.3.2.3	Les instructions de la manipulation directe des bits des GPIO ports d'arduino	34
2.3.3	Les timers et leurs interruptions	35
2.3.3.1	Introduction aux timers de la carte arduino	35
2.3.3.2	Architecture des timers de la carte arduino.....	35
2.3.3.3	Configuration de la fréquence des timers	36
2.3.3.4	Les différentes modes des timers de la carte arduino	37
2.3.3.5	Les interruptions des timers	37
2.3.4	Les interruptions externes.....	38
2.3.4.1	L'intérêt des interruptions externes	38
2.3.4.2	Les routines d'interruptions	38
2.3.4.2	Manipulation des interruptions externes par la carte arduino	38
3	Démarche et développement du projet	40
3.1	Commande du hacheur élévateur.....	40
3.2	Commande de gradateur et de redresseur, en monophasé et en triphasé	42
3.2.1	Détection de la période de signal de synchronisation	42
3.2.2	Génération des signaux de commutation des interrupteurs	45
3.2.3	Modulation des signaux de commutation	45
3.2.4	Passage de la commande an triphasé.....	46
3.2.5	Affichage	47
3.3	Résultats expérimentale.....	48
3.3.1	Commande de gradateur et redresseur monophasé	48
3.3.2	Commande de gradateur et redresseur en triphasé.....	50
3.3.3	Commande de hacheur élévateur	50
	Conclusion générale	52
	Bibliographie	53
	Annexe 1	54
	Annexe 2	55
	Annexe 3	59
	Annexe 4	68

Table des figures

1.1 convertisseur continu-continu (dc-dc)	7
1.2 hacheur boost.....	8
1.3 circuit équivalent de hacheur en phase 1	9
1.4 circuit équivalent de hacheur en phase 2	9
1.5 Chronogramme de hacheur boost	10
1.6 redresseur monophasé simple alternance	11
1.7 chronogramme redresseur monophasé simple alternance.....	12
1.8 redresseur monophasé simple alternance sur charge inductive.....	12
1.9 chronogramme de redresseur monophasé simple alternance sur charge résistive	13
1.10 Redresseur monophasé a point milieu	13
1.11 chronogramme de Redresseur monophasé a point milieu	14
1.12 Redresseur monophasé en pont tout thyristor	15
1.13 Redresseur monophasé en pont tout thyristor	15
1.14 Redresseur monophasé en pont mixte	16
1.15 chronogramme de redresseur en pont mixte	16
1.16 Référence de retard à l'amorçage d'un système triphasé.....	17
1.17 Redresseur triphasé a point milieu.....	17
1.18 chronogramme de Redresseur triphasé à point milieu.....	18
1.19 Redresseur triphasé a point milieu.....	19
1.20 chronogramme de Redresseur triphasé PD3	20
1.21 Redresseur triphasé à pont mixte.....	20
1.22 chronogramme de Redresseur triphasé en pont mixte	21
1.23 gradateur monophasé charge R	21
1.24 chronogramme de gradateur monophasé à charge R.....	22
1.25 gradateur monophasé charge RL	22
1.26 chronogramme d'impulsions de courte durée	23
1.27 chronogramme d'impulsions de longue durée	23
1.28 tension de sortie d'un gradateur à train d'onde	24
1.29 différentes jonctions d'un thyristor.....	25
2.1 l'arduino Uno R3	27
2.2 microcontrôleur Atmega16 dans l'ancienne version de la carte Arduino Uno.....	28
2.3 Les éléments principaux d'Arduino IDE	31
2.4 Registre TCCR1	36
2.5 Tableau de CS pour la configuration de Prescaler	37
3.1 Diagramme de génération de signal PWM	42
3.2 structure de base de la boucle de verrouillage à phase	43
3.3 circuit de commande de gradateur monophasé	48
3.4 tension de commande de gradateur monophasé en 50Hz	48
3.5 tension de commande de gradateur monophasé en 60Hz	49
3.6 modulation du signal de sortie.....	49
3.7 tension de commande de gradateur triphasé en 50Hz	50
3.8 de commande de gradateur triphasé en 50Hz et en 60Hz.....	50

3.9 circuit de commande de hacheur élévateur	50
3.10 commande de hacheur élévateur pour des différentes valeurs de rapport cyclique	51
3.11 commande de hacheur élévateur pour des différentes valeurs de fréquence	51

Liste des tableaux

1.1 les intervalles de conduction de chaque thyristor de P3	18
1.2 les intervalles de conduction de chaque thyristor de PD3	19
1.3 domaines d'application de différents types de redresseurs	24
1.4 domaines d'application de différents types de gradateurs	25
2.1 comparaison des caractéristiques entre Arduino Uno et Arduino Mega	30
2.2 lien entre la configuration des registres GPIO et les fonctions des pins.....	33
2.3 les numéros d'interruptions associés aux pins pour les cartes Uno et Mega	39
3.1 les pins I2C arduino	47

Abréviations

μC : microcontrôleur

- **PLL : Phase Locked Loop**

Introduction générale

L'électronique de puissance est un terme général qui englobe les systèmes et les produits nécessaires à la conversion et au contrôle de la distribution de l'énergie électrique. D'où l'importance de cette discipline dans la formation de l'ingénieur de génie électrique.

Nos étudiants de l'ENIT ont des travaux pratiques en électronique de puissance en deuxième année. Les maquettes de ce TP ont été faites en 1996 par une équipe d'étudiants encadrés par M. Ferid KOURDA.

Certes ces maquettes sont encore fonctionnelles et permettent aux étudiants de voir l'influence des différents paramètres électriques sur le comportement de convertisseurs, mais plusieurs améliorations peuvent être faites.

La commande de convertisseurs est analogique, pour le gradateur et le redresseur est basée sur le circuit TCA 785. Un détecteur de tension nulle évalue les passages à zéro et les transfère au registre de synchronisation. Ce registre de synchronisation commande un générateur de rampe si la tension de rampe dépasse une tension de commande (Angle de déclenchement ϕ), un signal est traité dans la logique.

A la sortie de ce circuit on obtient deux signaux, un pour chaque demi-période retardée par un angle ϕ qui dépend de la commande. Ces signaux sont ensuite multipliés numériquement par un signal PWM généré à partir de circuit NE555, on obtient alors deux signaux en peigne d'impulsion, chacun de ces signaux va être envoyé vers le driver d'un thyristor.

Cette commande analogique est complexe, demande beaucoup d'entretien et nécessite plusieurs circuits intégrés surtout en triphasé.

On propose de modifier les commandes existantes en mode analogique et à composants discrets par une commande numérique avec affichage numérique à l'aide de la carte arduino.

Dans le premier chapitre, nous présentons des généralités sur les composants de puissance utilisée qui sont les thyristors et les différents types de convertisseurs.

Le deuxième chapitre présente le fonctionnement de la carte arduino et les différentes instructions qu'on a utilisées dans notre programme.

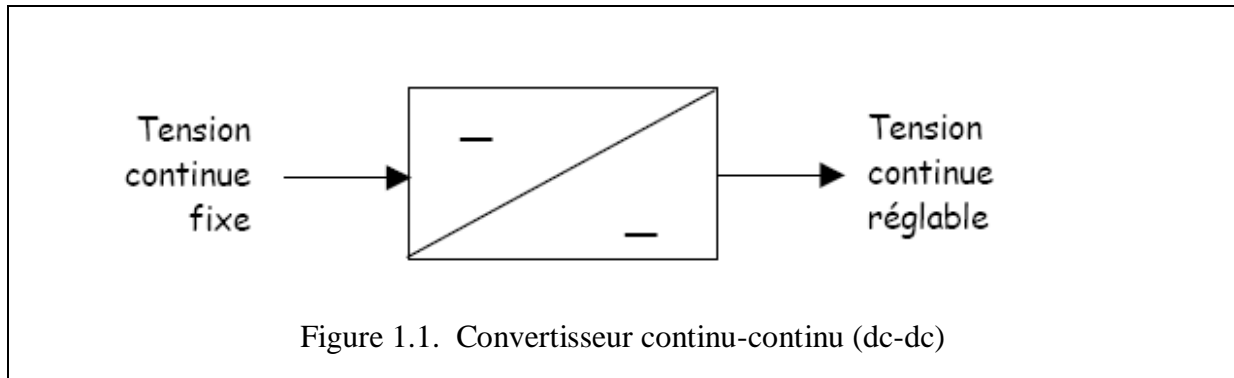
Le troisième chapitre présente les étapes de la réalisation expérimentale de la commande de nos convertisseurs ainsi que les résultats trouvés.

Une conclusion générale vient mettre en exergue les points les plus importants à retenir et les perspectives envisageables pour l'amélioration de ce travail.

Chapitre 1 : Convertisseur électrique statique

1.1 Introduction

Le convertisseur statique est un dispositif qui permet d'adapter une source d'énergie à un récepteur. On distingue plusieurs familles de convertisseurs statiques qui diffèrent par la nature des sources d'énergies, soient monophasé ou triphasé et soient continue ou alternatif.



Un convertisseur comporte essentiellement :

- des interrupteurs qui fonctionnent en mode de commutation et d'une manière périodique choisie selon l'application demandée.
- des composants de stockage intermédiaire d'énergie qui sont essentiellement les inductances et les condensateurs.

On distingue quatre types de convertisseurs :

- convertisseur continu-continu : hacheur.
- convertisseur continu-alternatif : onduleur.
- convertisseur alternatif-alternatif : gradateur.
- convertisseur alternatif-continu : redresseur.

Nous allons étudier que le hacheur, le redresseur et le gradateur puisque notre travail consiste à la synthèse de la commande de ces convertisseurs.

1.2 LE HACHEUR

1.2.1 Définition :

Le hacheur est un convertisseur continu-continu qui permet d'avoir une tension continue à la sortie de valeur moyenne réglable à partir d'une source de tension continue fixe.

Il existe plusieurs types de hacheurs qui diffèrent par la position de commutateur et leurs tensions d'entrées et de sorties. Nous allons étudier le hacheur dit élévateur ou Boost.

1.2.2 Le rapport cyclique α

Le rapport cyclique α définit le temps pendant lequel le thyristor est amorcé 'ton' sur la période totale de fonctionnement de montage 'T'. $\alpha = \text{ton} / T$

Ce rapport α est compris entre 0 et 1

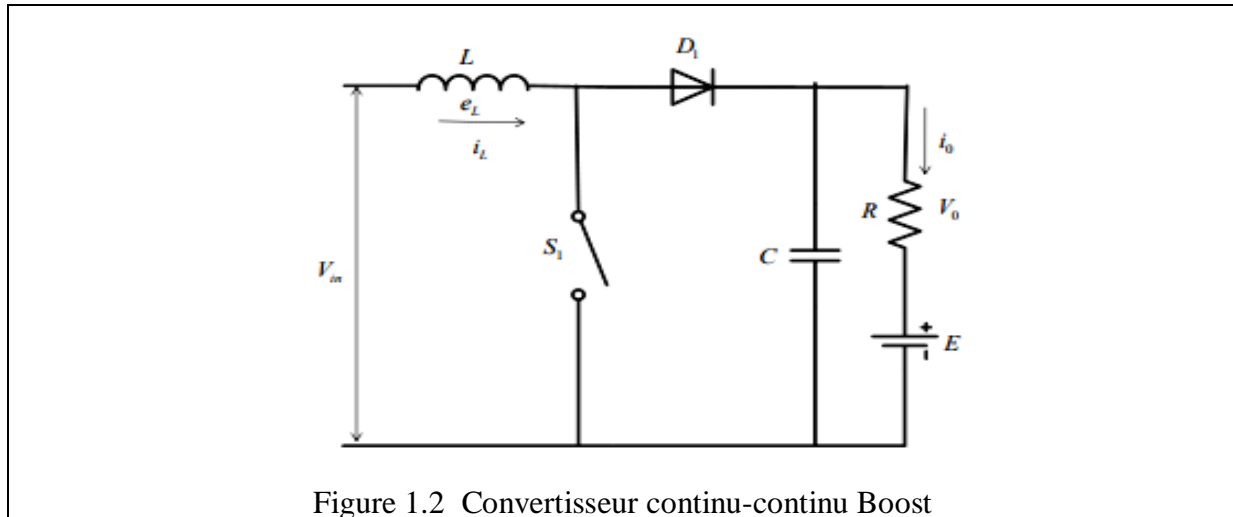


Figure 1.2 Convertisseur continu-continu Boost

1.2.3 Etude de hacheur élévateur Boost :

1.2.3.1 Définition :

Le hacheur Boost est utilisé lorsque la tension demandée par la sortie est supérieure à celle délivrée par la source de tension d'entrée. La figure 1.1 montre la composition d'un hacheur élévateur dit aussi parallèle ou Boost.

Notre circuit est alimenté par une tension d'entrée V_{in} et la charge est de type résistive R parcourue par un courant i_o .

On distingue deux modes de fonctionnement de notre circuit qui sont le mode de conduction continue et le mode de conduction discontinue, vu son importance, nous allons faire l'étude de mode de conduction continu ($E < V_{in}$).

1.2.3.2 Hypothèses :

- On considère que la capacité de condensateur est très grande et la tension entre ses bornes est constante.
- Les composants sont parfaits.
- La fréquence est fixe.

1.2.3.3 Principe de fonctionnement :

On a deux phases :

Phase 1 : lorsque $0 < t < \alpha T$

L'interrupteur S_1 est fermé, donc la diode D_1 est bloquée ; on aura ce circuit équivalent :

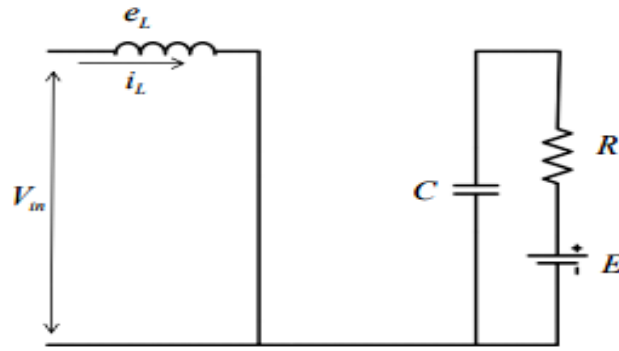


Figure 1.3 : circuit équivalent de hacheur en phase 1

On a :
$$V_{in} = L \frac{di_L}{dt} \quad (1)$$

D'où :
$$i(t) = I_m + \frac{V_{in}}{L} t \quad (2)$$

Avec I_m est la valeur initiale de courant, pour $t = \alpha T$; la valeur finale à la fin de cette phase de courant est :

$$I_M = I_m + \frac{V_{in}}{L} \alpha T \quad (3)$$

Phase 2 : lorsque $\alpha T < t < T$

L'interrupteur S1 est ouvert, donc la diode D1 est passante; le circuit équivalent devient :

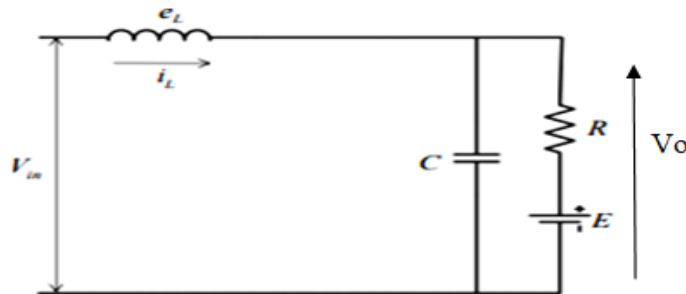


Figure 1.4 : circuit équivalent de hacheur en phase 2

On a :
$$V_{in} - V_o = L \frac{di_L}{dt} \quad (4)$$

D'où :
$$i(t) = I_M + \frac{V_o - V_{in}}{L} (t - \alpha T) \quad (5)$$

Pour $t = \alpha T$ la valeur finale à la fin de cette phase de courant est :

$$I_m = I_M + \frac{V_o - V_{in}}{L} (1 - \alpha) T \quad (6)$$

D'après l'équation (3) :
$$\Delta I = I_M - I_m = \frac{V_{in}}{L} \alpha T \quad (7)$$

D'après l'équation (6) :
$$\Delta I = I_M - I_m = \frac{V_{in}}{L} \alpha T \quad (8)$$

(7) et (8) donnent :
$$V_o = \frac{V_{in}}{(1-\alpha)} \quad (9)$$

Tous les composants sont supposé parfaits donc on n'a pas une perte de puissance :

$$V_{in} \cdot I_{in} = V_o \cdot I_o \quad (10)$$

Donc :
$$I_o = \frac{I_{in}}{(1-\alpha)} \quad (11)$$

L'équation (9) montre le fait que la tension de sortie V_o ne dépend que de la tension d'entrée et la valeur de rapport cyclique α qui est toujours inférieur à 1 , donc la tension de sortie est strictement supérieur à la tension d'entrée donc ce convertisseur est toujours élévateur de courant et de tension .

1.2.3.4 Chronogramme :

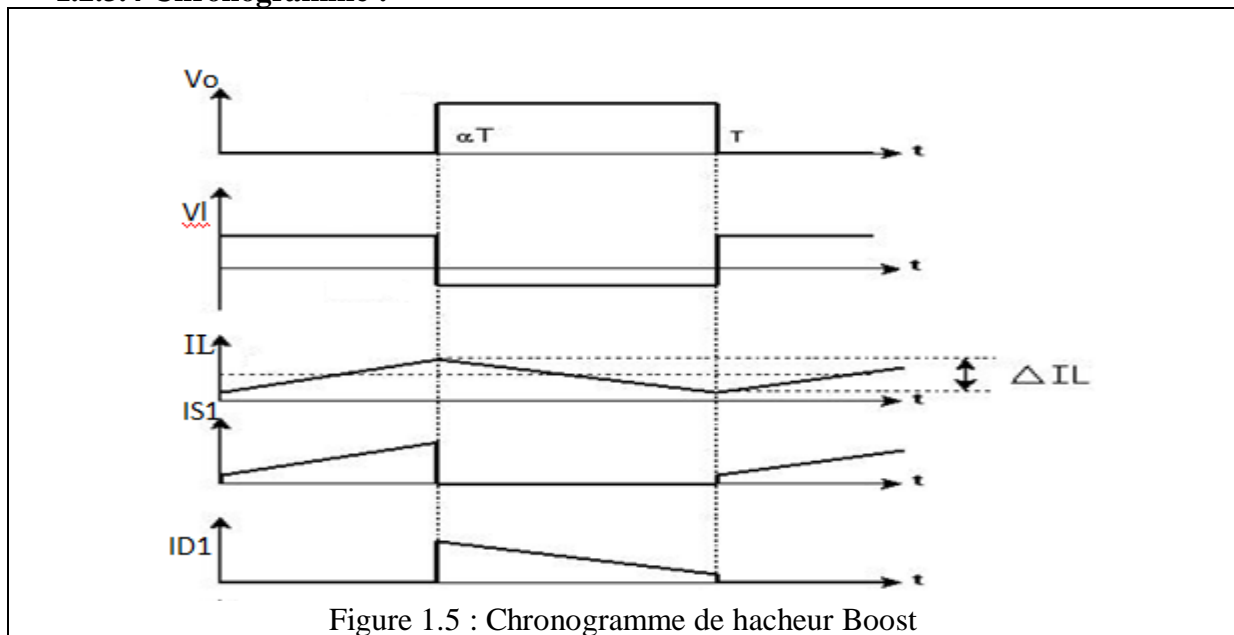


Figure 1.5 : Chronogramme de hacheur Boost

1.2.4 Domaines d'application de hacheur élévateur :

- **L'usage personnel** : il est utilisé dans plusieurs appareils électroniques alimentées par des piles alcalines ou Li-Ion.
- **Automobile** : Les convertisseurs Boost sont largement utilisés dans les applications d'info divertissement et de cluster, de passerelle, d'ADAS et d'électronique de sécurité du corps.
- **Industrie** : le freinage des moteurs à courant continu.
- Les Panneaux solaires alimentant les onduleurs.

1.3 Le redresseur :

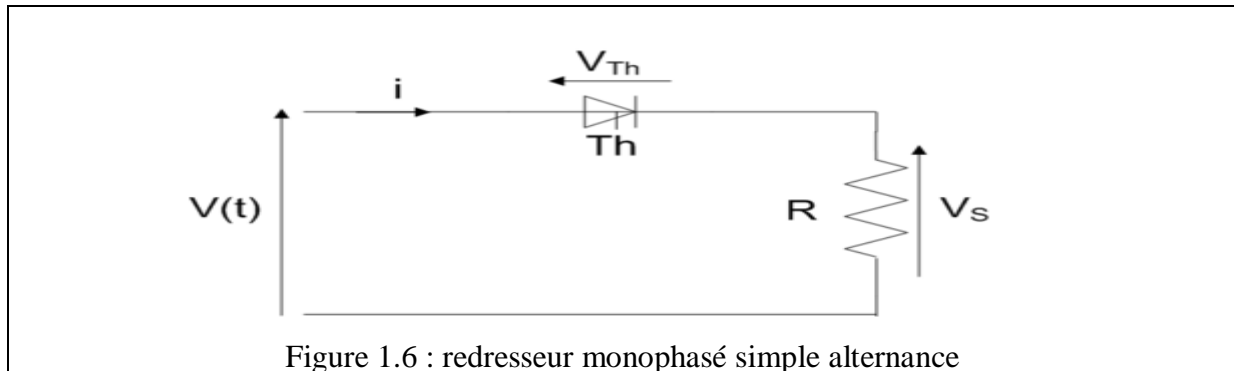
1.3.1 Définition :

Le redresseur est un convertisseur alternatif-continu (AC-DC) qui permet d'avoir à partir d'une tension alternative monophasé ou triphasé une tension continue fixe ou variable.

1.3.2 Différents types de redresseurs monophasés :

1.3.2.1 Redresseur monophasé simple alternance à charge résistive :

Dans cette partie on va étudier le redresseur monophasé commandé par thyristor, le figure suivant montre la composition d'un redresseur commandé avec une tension d'entrée $V(t)=\sqrt{2} V \sin(\omega t)$



D'après la loi des mailles :

$$V(t) = V_s(t) + V_{th}(t) \quad (1)$$

*Pour $0 < t < \alpha T / 2\pi$: Le thyristor th est bloqué donc

$$V_s(t)=0 \quad (2)$$

*Pour $\alpha T / 2\pi < t < T/2$: Le thyristor th est amorcé donc :

$$V_s(t)=V(t)=\sqrt{2} V \sin(\omega t) \quad (3)$$

*Pour $T/2 < t < T$:
Le thyristor th est bloqué donc

$$V_s(t)=0 \quad (4)$$

La valeur moyenne de la tension de sortie V_s d'après les équations (2), (3) et (4) est :

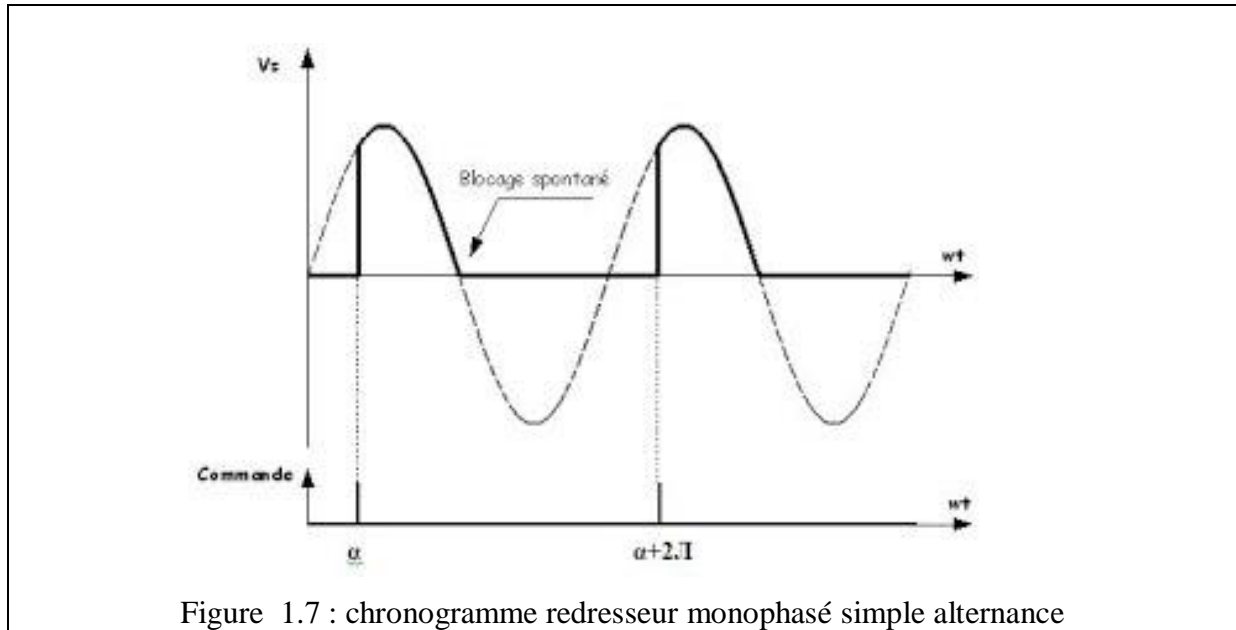
$$\langle V_s(t) \rangle = \frac{1}{T} \int_0^T V_s(t) dt = \frac{1}{T} \int_{\alpha T / 2\pi}^{T/2} \sqrt{2} V \sin(\omega t) dt \quad (5)$$

Donc :

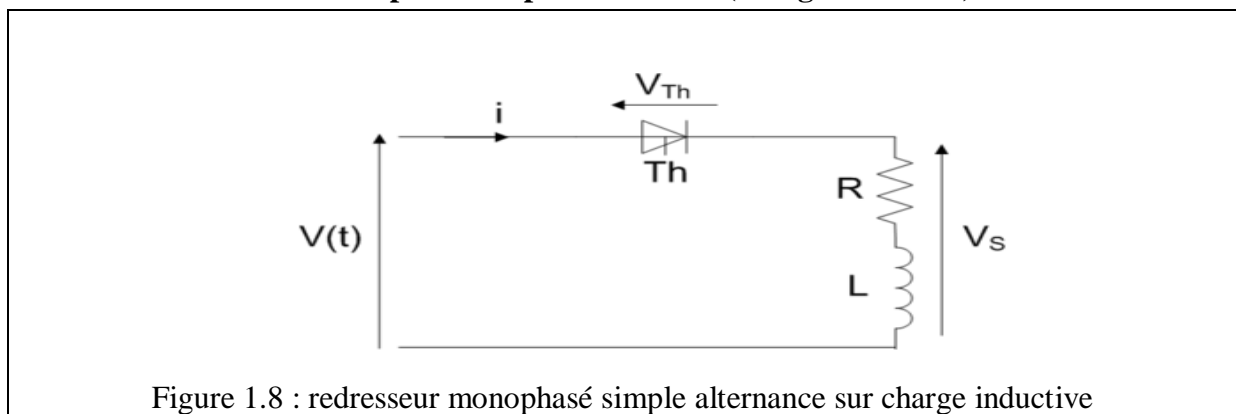
$$\langle V_s(t) \rangle = \frac{\sqrt{2}}{2\pi} V (\cos(\alpha) + 1) \quad (6)$$

Chronogramme :

Les impulsions de commande de thyristor sont retardées d'un angle α par rapport au zéro du secteur, le thyristor est amorcé aux instants $\alpha, \alpha+2\pi, \alpha+4\pi...$ etc.



1.3.2.2 Redresseur monophasé simple alternance (charge inductive):



Dans cette partie on va étudier le redresseur monophasé simple alternance sur une charge inductive commandée par thyristor.

On pose $\theta_0 = t_0 \omega$ et $t_0 > \alpha T / 2\pi$

D'après la loi des mailles :

$$V(t) = V_s(t) + V_{th}(t) \quad (1)$$

*Pour $0 < t < \alpha T / 2\pi$:

Le thyristor th est bloqué donc

$$V_s(t) = 0 \quad (2)$$

*Pour $\alpha T / 2\pi < t < t_0$:

Le thyristor th est amorcé donc

$$V_s(t) = V(t) = \sqrt{2} V \sin(\omega t) \quad (3)$$

*Pour $t_0 < t < T$:

Le thyristor th est bloqué donc

$$V_s(t) = 0 \quad (4)$$

La valeur moyenne de tension de sortie V_s d'après les équations (2), (3) et (4) est :

$$\langle V_s(t) \rangle = \frac{1}{T} \int_0^T V_s(t) dt = \frac{1}{T} \int_{\alpha T/2\pi}^{T/2} \sqrt{2} V \sin(\omega t) + \frac{1}{T} \int_{T/2}^{t_0 - T/2} \sqrt{2} V \sin(\omega t) \quad (5)$$

Donc :

$$\langle V_s(t) \rangle = \frac{\sqrt{2}}{2\pi} V (\cos(\alpha) - \cos(\theta_0)) \quad (6)$$

Chronogramme :

Les impulsions de commande de thyristor sont retardées d'un angle α par rapport au zéro du secteur, le thyristor est amorcé aux instants $\alpha, \alpha+2\pi, \alpha+4\pi...$ etc.

Le thyristor se bloque à l' instant $t_0 > T/2$ puisque le courant s'annule après $T/2$.

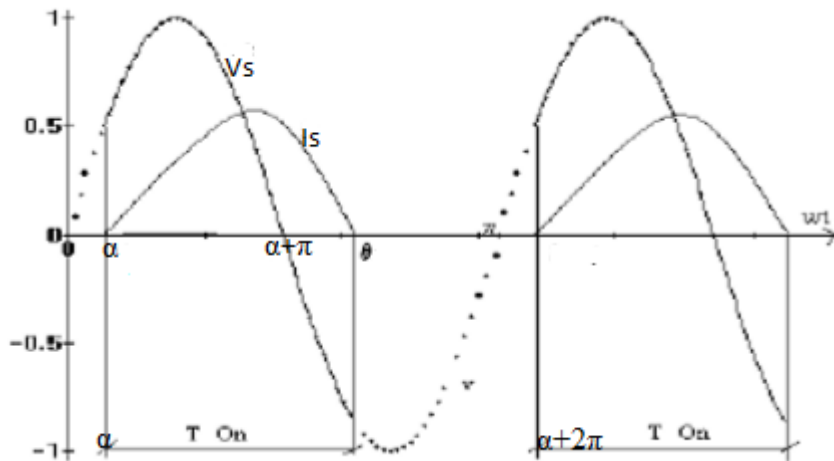


Figure 1.9 : chronogramme du redresseur monophasé simple alternance sur charge résistive inductive

1.3.2.3 Redresseur monophasé à point milieu P2 à charge R :

L'alimentation du convertisseur n'est pas directe mais par un transformateur monophasé à point milieu au secondaire. La figure suivante montre la composition d'un redresseur avec une tension d'entrée $V_1(t) = \sqrt{2} V \sin(\omega t)$ et $V_2(t) = \sqrt{2} V \sin(\omega t + \pi)$

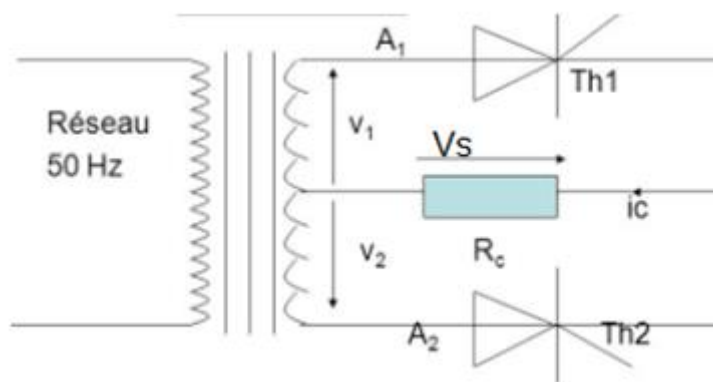


Figure 1.10 : Redresseur monophasé à point milieu

*Pour $0 < t < \alpha T / 2\pi$:

Le thyristor th est bloqué donc $V_s(t)=0$ (1)

*Pour $\alpha T / 2\pi < t < T/2$:

Le thyristor th est amorcé donc $V_s(t)=V_1(t)=\sqrt{2} V \sin(\omega t)$ (2)

*Pour $T/2 < t < \alpha T / 2\pi + T/2$:

Le thyristor th est bloqué donc $V_s(t)=0$ (3)

*Pour $\alpha T / 2\pi + T/2 < t < T$:

Le thyristor th est amorcé donc $V_s(t)=V_2(t)=\sqrt{2} V \sin(\omega t)$ (4)

La valeur moyenne de tension de sortie V_s d'après les équations (2), (3) et (4) est :

$$\langle V_s(t) \rangle = \frac{1}{T} \int_0^T V_s(t) dt = \frac{1}{T} \int_{\alpha T/2\pi}^{T/2} \sqrt{2} V \sin(\omega t) - \frac{1}{T} \int_{\alpha(\frac{T}{2\pi})+T/2}^T \sqrt{2} V \sin(\omega t) dt \quad (5)$$

Donc : $\langle V_s(t) \rangle = \frac{2\sqrt{2}}{\pi} V(\cos(\alpha))$ (6)

Chronogramme :

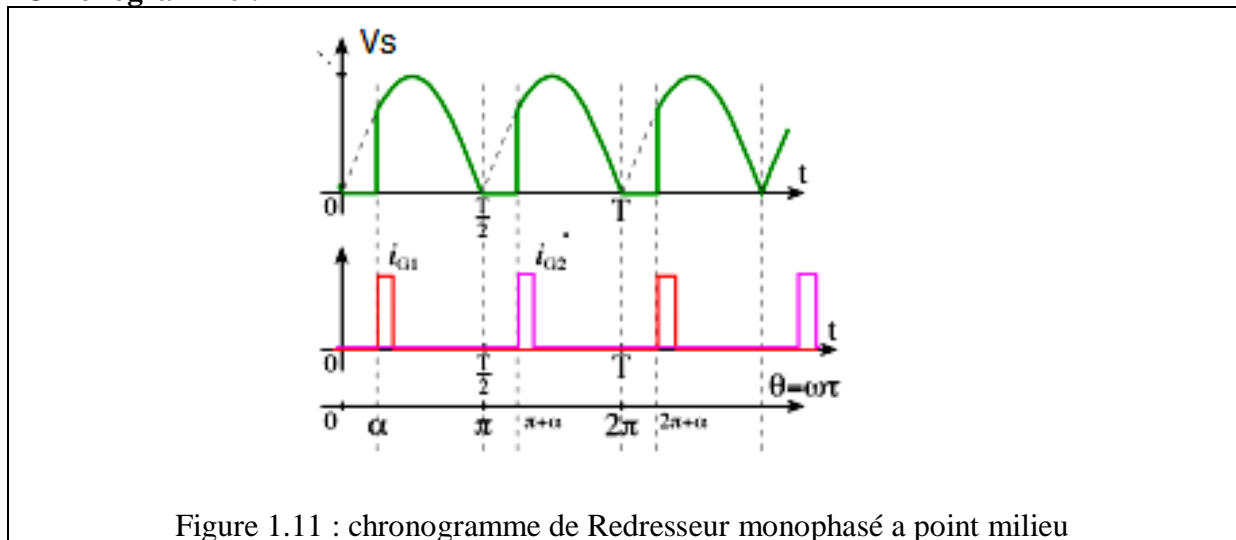


Figure 1.11 : chronogramme de Redresseur monophasé a point milieu

3-d Redresseur monophasé en pont tout thyristor PD2:

Ce type de redresseur ne contient que des thyristors pour sa commande. Quatre thyristors sont utilisés et dans chaque période deux entre eux sont passants. La figure suivante montre la composition de ce redresseur avec une tension d'entrée $V(t)=\sqrt{2} V \sin(\omega t)$ et $\omega = \frac{2\pi}{T}$

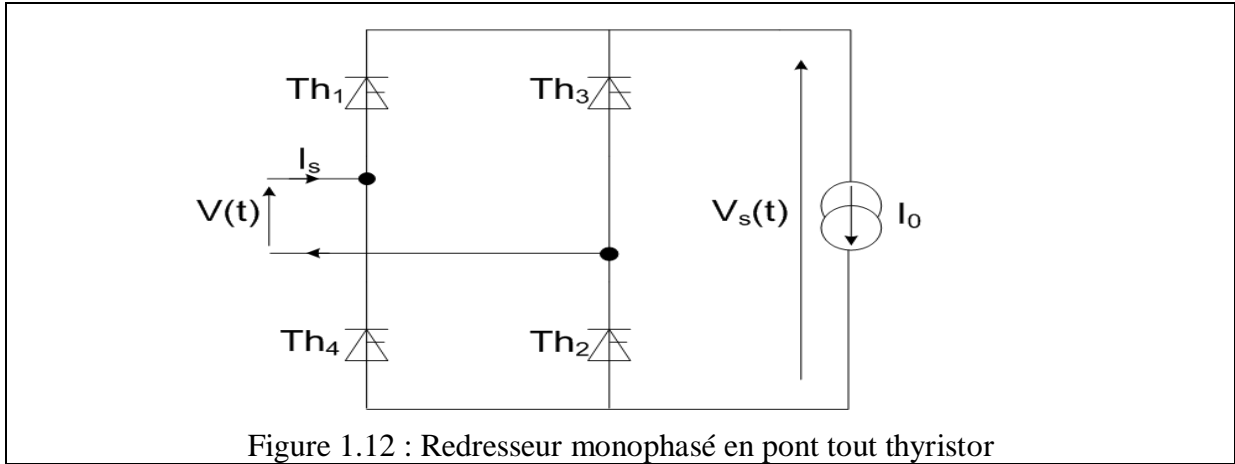


Figure 1.12 : Redresseur monophasé en pont tout thyristor

Les thyristors Th1 et Th2 conduisent le courant lorsque $v(t)$ est positive, et Les thyristors Th3 et Th4 conduisent le courant lorsque $v(t)$ est négative.

*Pour $\frac{\alpha}{\omega} < t < \frac{T}{2} + \frac{\alpha}{\omega}$:

Les thyristors Th1 et Th2 conduisent : $V_s(t) = V(t) = \sqrt{2} V \sin(\omega t)$ (1)

*Pour $\frac{T}{2} + \frac{\alpha}{\omega} < t < T + \frac{\alpha}{\omega}$:

Les thyristors Th3 et Th4 conduisent : $V_s(t) = V(t) = -\sqrt{2} V \sin(\omega t)$ (2)

La tension de sortie est périodique de période $\frac{T}{2}$, La valeur moyenne de tension de sortie V_s d'après les équations (1) et (2) est :

$$\langle V_s(t) \rangle = \frac{1}{T} \int_0^T V_s(t) dt = \frac{1}{T} \int_{\frac{\alpha}{\omega}}^{\frac{T}{2} + \frac{\alpha}{\omega}} \sqrt{2} V \sin(\omega t) dt = \frac{\sqrt{2}}{\pi} V (\cos(\alpha) - \cos(\alpha + \pi)) \quad (3)$$

Donc : $\langle V_s(t) \rangle = \frac{2\sqrt{2}}{\pi} V (\cos(\alpha))$ (4)

Chronogramme :

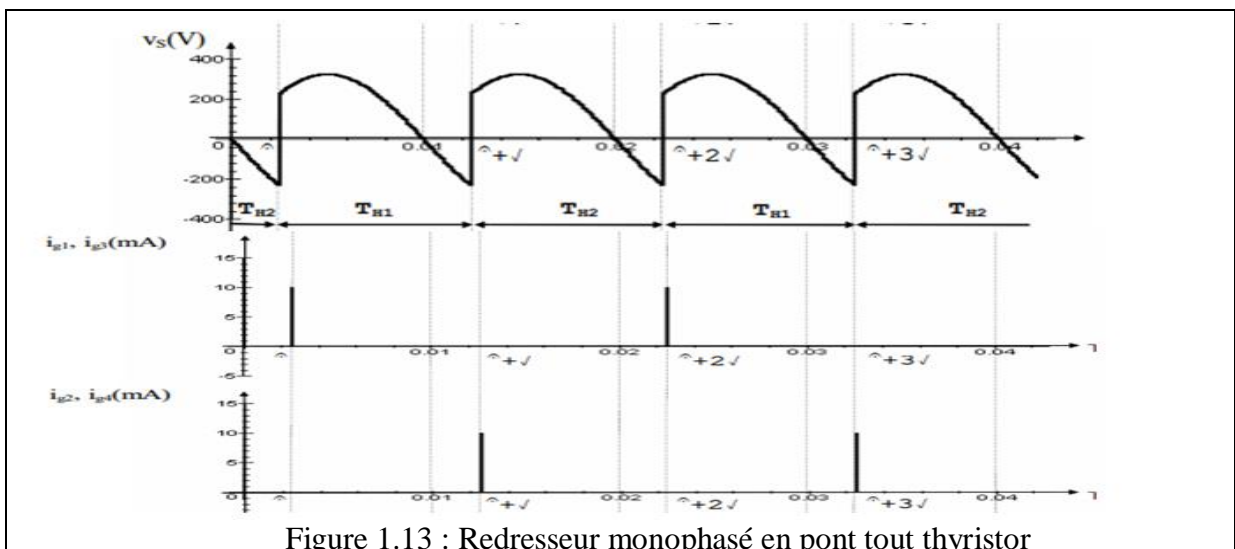
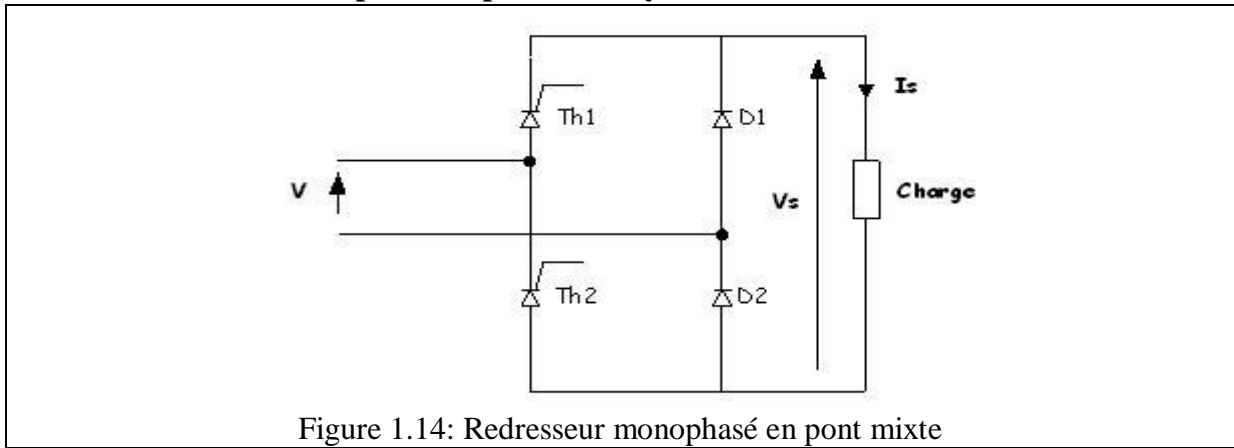


Figure 1.13 : Redresseur monophasé en pont tout thyristor

1.3.2.4 Redresseur monophasé en pont tout thyristor :



Il consiste en deux thyristors seulement. Les deux autres sont remplacés par des diodes. La différence par rapport à un pont complet est que la tension V_s ne pourra plus devenir négative (conduction simultanée de D1 et D2). La figure suivante montre la composition de ce redresseur avec une tension d'entrée $V(t) = \sqrt{2} V \sin(\omega t)$ et $\omega = \frac{2\pi}{T}$

*Pour $\frac{\alpha}{\omega} < t < \frac{T}{2}$:

Le thyristor Th1 est amorcé et la diode D2 conduit : $V_s(t) = V(t) = \sqrt{2} V \sin(\omega t)$ (1)

*Pour $\frac{T}{2} < t < \frac{\alpha}{\omega} + \frac{T}{2}$:

D1 conduit et la charge se trouve en court-circuit sur les diodes D1 et D2.: $V_s(t) = 0$ (2)

*Pour $\frac{T}{2} + \frac{\alpha}{\omega} < t < T$:

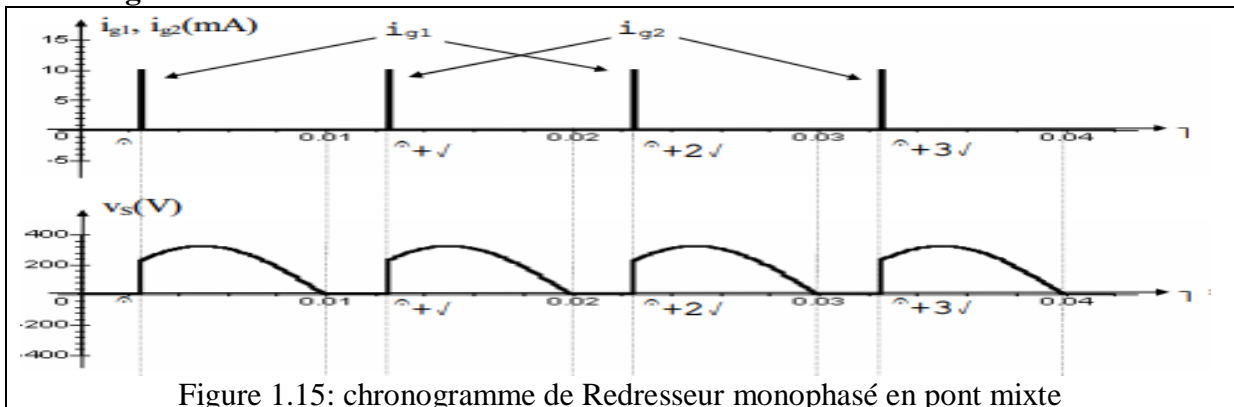
Le thyristor Th1 est amorcé et la diode D2 conduit : $V_s(t) = -V(t) = -\sqrt{2} V \sin(\omega t)$ (3)

La tension de sortie est périodique de période $\frac{T}{2}$, La valeur moyenne de tension de sortie V_s d'après les équations (1), (2) et (3) est :

$$\langle V_s(t) \rangle = \frac{1}{T} \int_0^T V_s(t) dt = \frac{1}{T} \int_{\frac{\alpha}{\omega}}^{\frac{T}{2}} \sqrt{2} V \sin(\omega t) dt \quad (4)$$

Donc : $\langle V_s(t) \rangle = \frac{\sqrt{2}}{\pi} V (\cos(\alpha) + 1)$ (5)

Chronogramme :



1.3.3 Différents types de redresseurs triphasés :

1.3.3.1 Redresseur triphasé a point milieu :

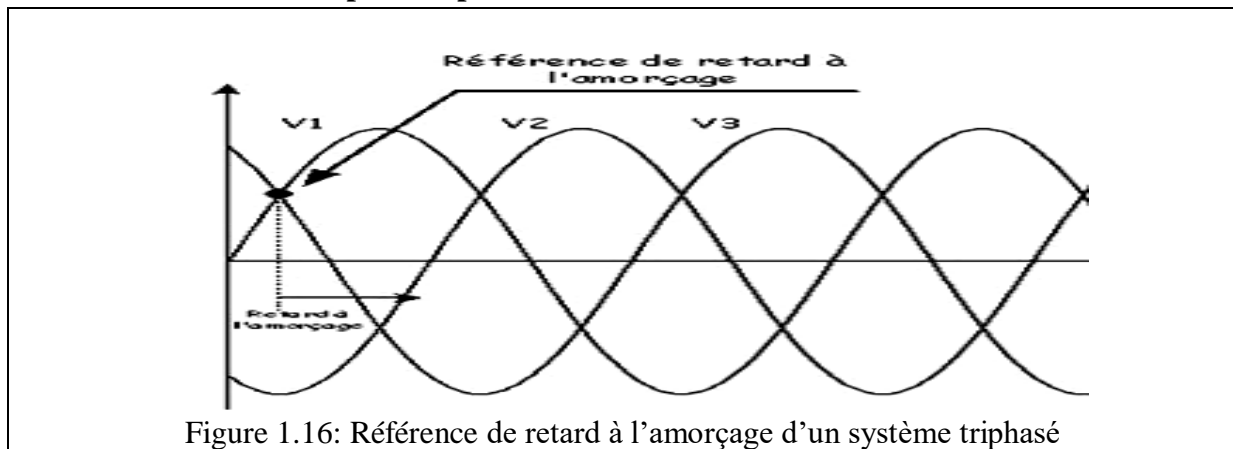


Figure 1.16: Référence de retard à l'amorçage d'un système triphasé

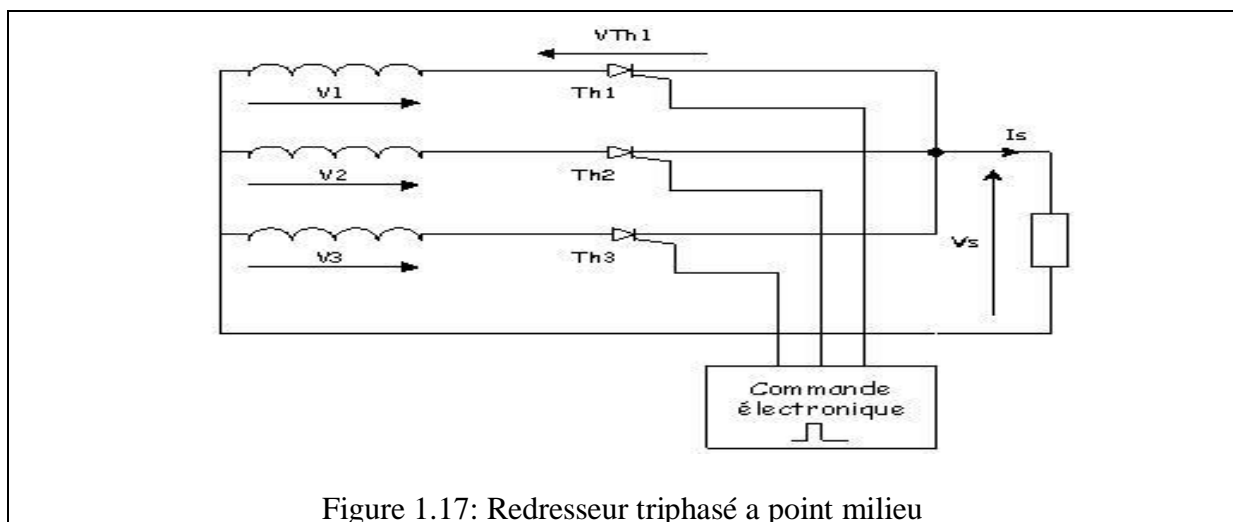


Figure 1.17: Redresseur triphasé a point milieu

Dans le système triphasé, l'angle de retard à l'amorçage α des thyristors n'est pas référencé par rapport aux passages à zéro de signal de secteur, en triphasé le point de référence est l'instant où deux tensions composant le système triphasé équilibré deviennent égales.

Pour le montage P3 le retard à l'amorçage α est pris à partir de l'angle de conduction naturel des diodes qui est égale à $\frac{\pi}{6}$. La figure suivante montre la composition de ce redresseur.

A l'instant α on amorce le thyristor th1 : $V_s(t)=V_1(t)$ (1)

A l'amorçage de thyristor Th2, le thyristor Th1 voit une tension $V_{Th1} = V_1 - V_2 = U_{12}$ négative et se bloque.

Dans ces conditions, on a : $V_s(t)=V_2(t)$ (2)

A l'amorçage de thyristor Th3, le thyristor Th2 voit une tension $V_{Th2}= V_2 - V_3 = U_{23}$ négative et se bloque.

Dans ces conditions, on a : $V_s(t)=V_3(t)$ (3)

Les intervalles de conduction de circuit P3 est le suivant :

Intervalle	Thyristor en conduction	Thyristors bloqués	Tension de sortie Vs
$[\frac{\pi}{6} + \alpha, \frac{5\pi}{6} + \alpha]$	Th1	Th2 et Th3	V1
$[\frac{5\pi}{6} + \alpha, \frac{9\pi}{6} + \alpha]$	Th2	Th1 et Th3	V2
$[\frac{9\pi}{6} + \alpha, \frac{13\pi}{6} + \alpha]$	Th3	Th1 et Th2	V3
$[\frac{13\pi}{6} + \alpha, \frac{17\pi}{6} + \alpha]$	Th1	Th2 et Th3	V1

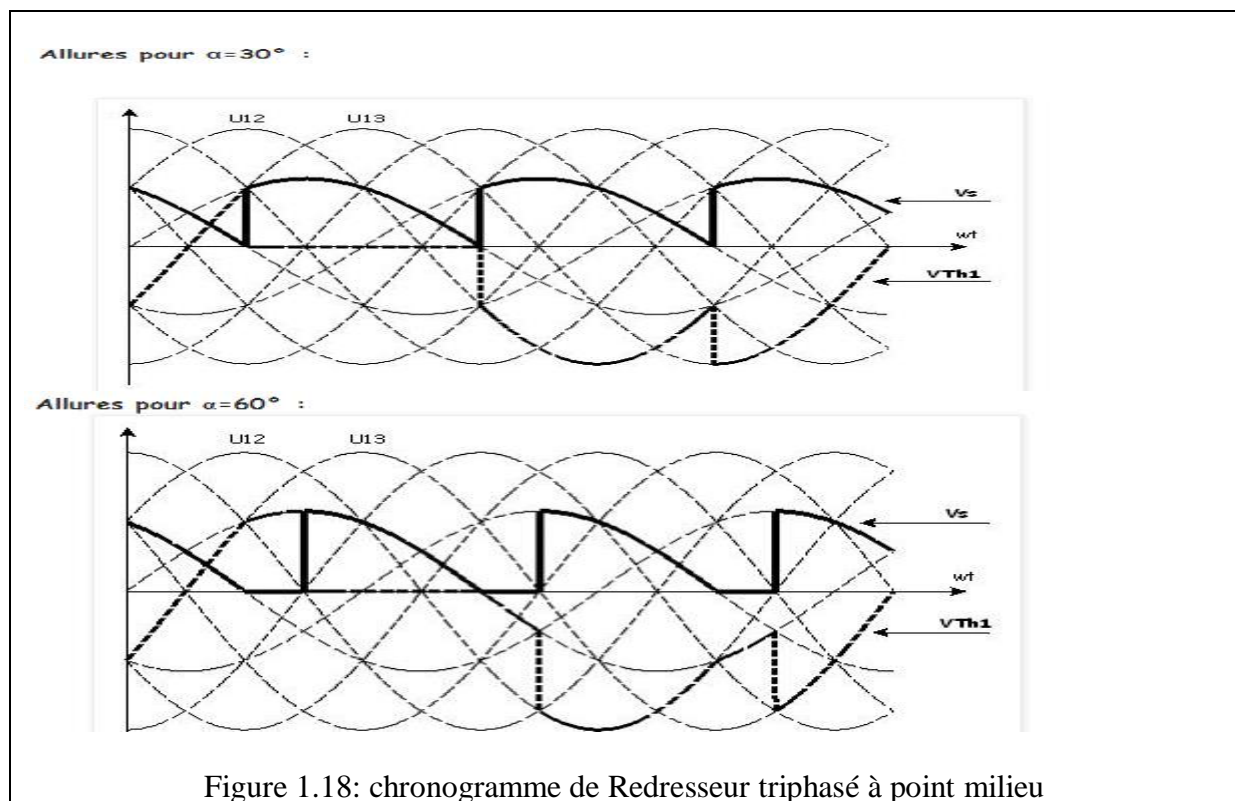
Tableau1.1 : les intervalles de conduction de chaque thyristor de P3

La tension de sortie est périodique de période $\frac{T}{3}$, La valeur moyenne de tension de sortie Vs

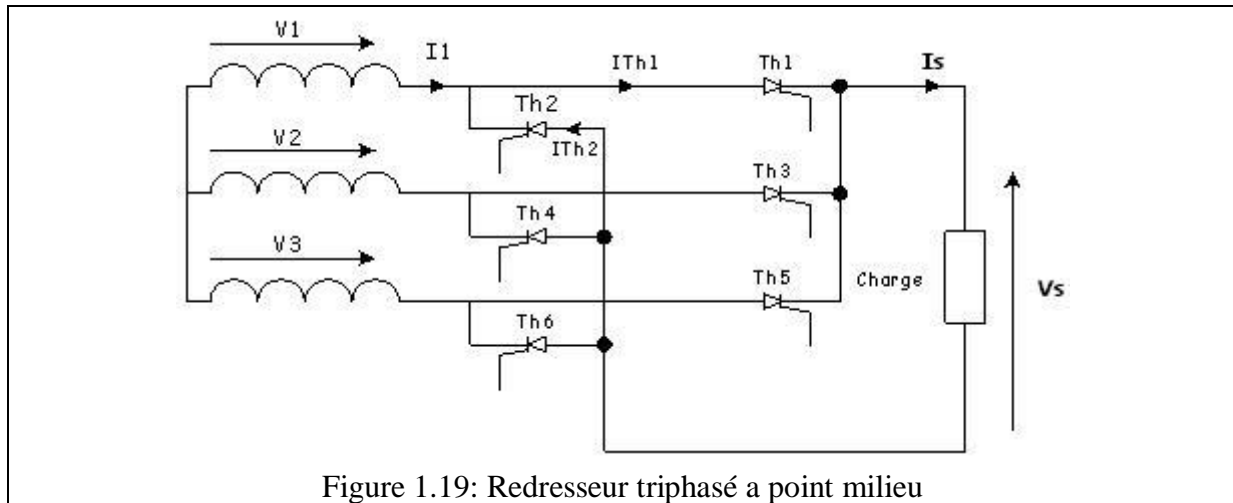
$$\text{est : } \langle V_s(t) \rangle = \frac{1}{\frac{T}{3}} \int_0^{\frac{T}{3}} V_s(t) dt = \frac{3}{2\pi} \int_{\alpha + \frac{\pi}{6}}^{\alpha + \frac{5\pi}{6}} V_m \sin(\omega t) d\omega \quad (4)$$

$$\text{Donc : } \langle V_s(t) \rangle = \frac{3\sqrt{3}}{2\pi} V_m \cos(\alpha) \quad (5)$$

Chronogramme :



1.3.3.2 Redresseur triphasé à parallèle double PD3:



La particularité est de ne comporter que des thyristors permet une réversibilité en puissance et de renvoyer de la puissance au réseau alternatif. Le courant de sortie est toujours positif mais la tension moyenne peut devenir négative (onduleur assisté) si l'angle d'amorçage est compris entre 90° et 180° .

Parmi les thyristors Th1, Th3, Th5, celui qui a la tension la plus positive sur son anode conduit.

Parmi les thyristors Th2, Th4, Th6, celui qui a la tension la plus positive sur son anode conduit.

Les intervalles de conduction de circuit P3 sont les suivants :

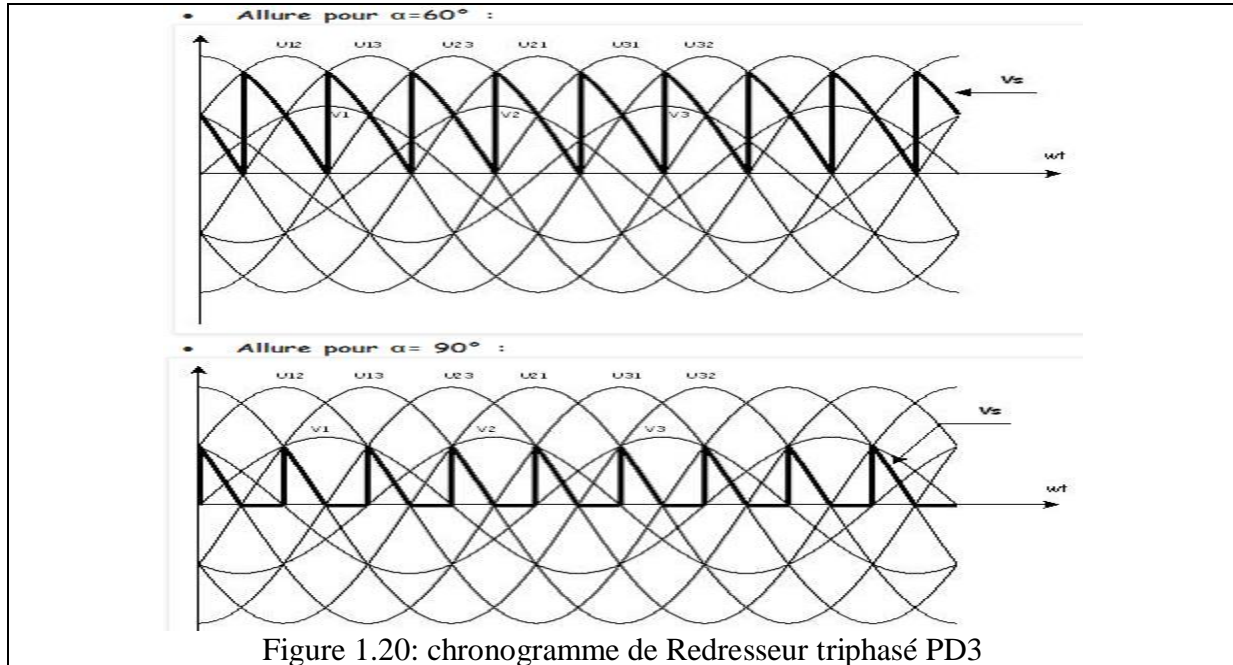
Intervalle	Thyristors en conduction	Tension de sortie Vs
$[\frac{\pi}{6} + \alpha, \frac{\pi}{2} + \alpha]$	Th1 et Th4	$V1 - V2 = U12$
$[\frac{\pi}{2} + \alpha, \frac{5\pi}{6} + \alpha]$	Th1 et Th6	$V1 - V3 = U13$
$[\frac{5\pi}{6} + \alpha, \frac{7\pi}{6} + \alpha]$	Th3 et Th6	$V2 - V3 = U23$
$[\frac{7\pi}{6} + \alpha, \frac{3\pi}{2} + \alpha]$	Th3 et Th2	$V2 - V1 = U21$
$[\frac{3\pi}{2} + \alpha, \frac{11\pi}{6} + \alpha]$	Th5 et Th2	$V3 - V1 = U31$
$[\frac{11\pi}{6} + \alpha, \frac{13\pi}{6} + \alpha]$	Th5 et Th4	$V3 - V2 = U32$

Tableau1.2 : les intervalles de conduction de chaque thyristor de PD3

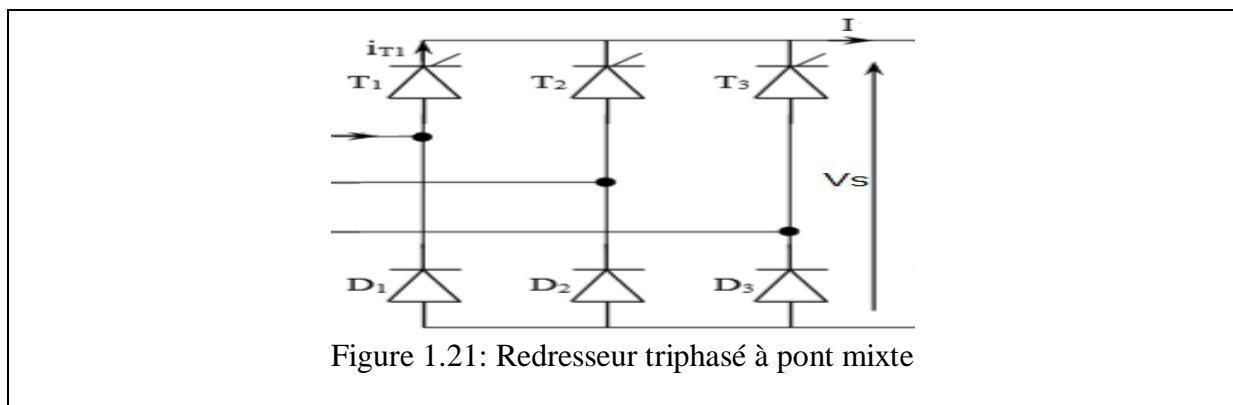
La tension de sortie est périodique de période $\frac{T}{6}$, La valeur moyenne de tension de sortie V_s est :

$$\langle V_s(t) \rangle = \frac{3\sqrt{3}\sqrt{2}}{\pi} V \cos(\alpha)$$

Chronogramme :



1.3.3.3 Redresseur triphasé à pont mixte :



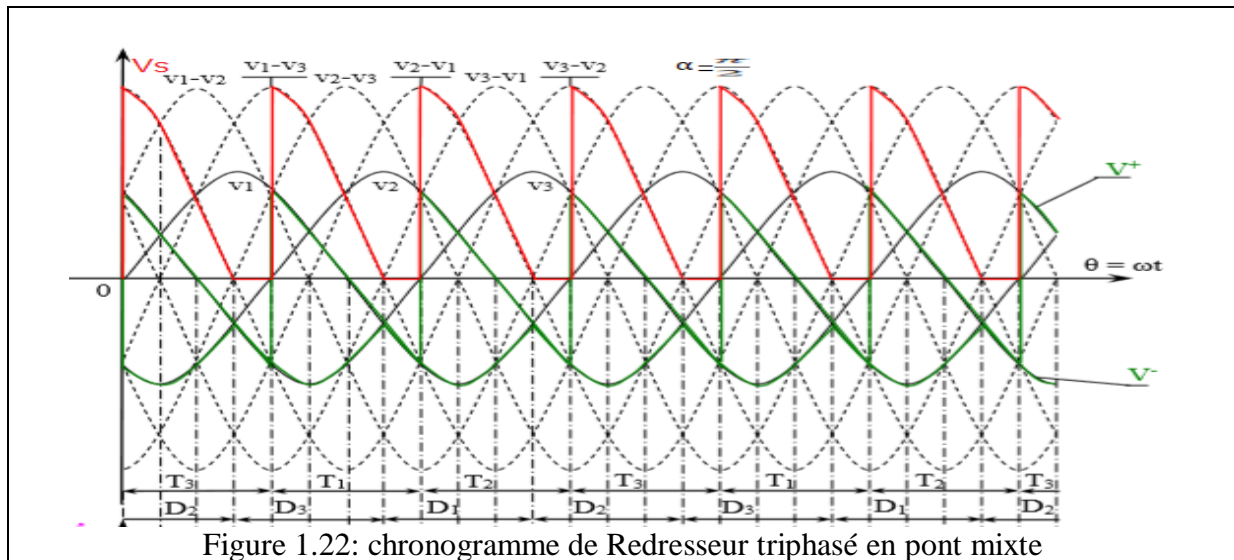
Ce pont est composé de 3 thyristors et de 3 diodes. Les thyristors T1, T2 et T3 sont amorcés pendant l'alternance positive des tensions v_1 , v_2 et v_3 . Les diodes D1, D2 et D3 sont passantes pendant l'alternance négative. La référence des angles d'amorçage est toujours l'angle d'amorçage naturel.

A partir du retard à l'amorçage α , on détermine la tension V_+ en suite, la tension V_- pour la même valeur de l'angle de commande. Il est alors possible de déterminer les grandeurs caractéristiques de ce pont.

Parmi les thyristors T1, T2, T3, celui qui a la tension la plus positive sur son anode conduit. Parmi les diodes D1, D2, D3, une seule diode entre eux conduit à la fois.

La valeur moyenne de tension de sortie V_s est :

$$\langle V_S(t) \rangle = \frac{3\sqrt{3}\sqrt{2}}{2\pi} V (\cos(\alpha) + 1)$$



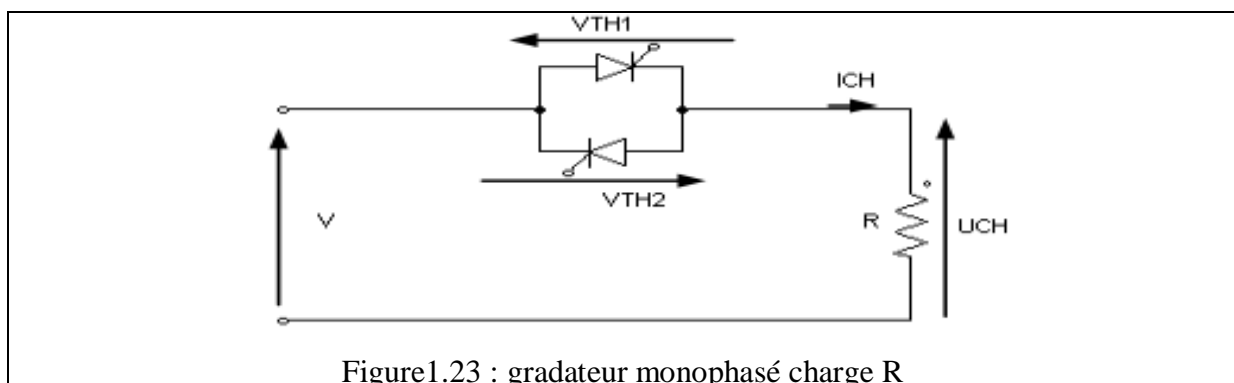
1.4 Le gradateur :

1.4.1 Gradateur monophasé à charge resistive :

Les gradateurs permettent d'obtenir une tension alternative variable à partir d'une tension alternative fixe.

Dans ce cas de montage, on constate que le courant doit pouvoir passer dans les deux sens dans l'interrupteur. La solution consiste à mettre en parallèle 2 thyristors tête-bêche. Chacun des 2 thyristors conduira à tour de rôle en fonction des polarités du réseau.

Les thyristors TH1 ou TH2 sont passants qu'à partir du moment où l'on envoie le signal de gâchette et à la condition que la tension VAK soit positive.



Pour $\alpha < \theta < 0$:

On a $V > 0$, pas d'impulsion sur la gâchette donc TH1 et TH2 sont bloqués : $U_{ch} = 0$ (1)

Pour $\pi < \theta < \alpha$:

On a $V > 0$, envoie une impulsion de commande donc TH1 devient conducteur, alors :

$$U_{ch} = \sqrt{2}V\sin(\theta) \quad (2)$$

Pour $\alpha < \theta < \alpha + \pi$:

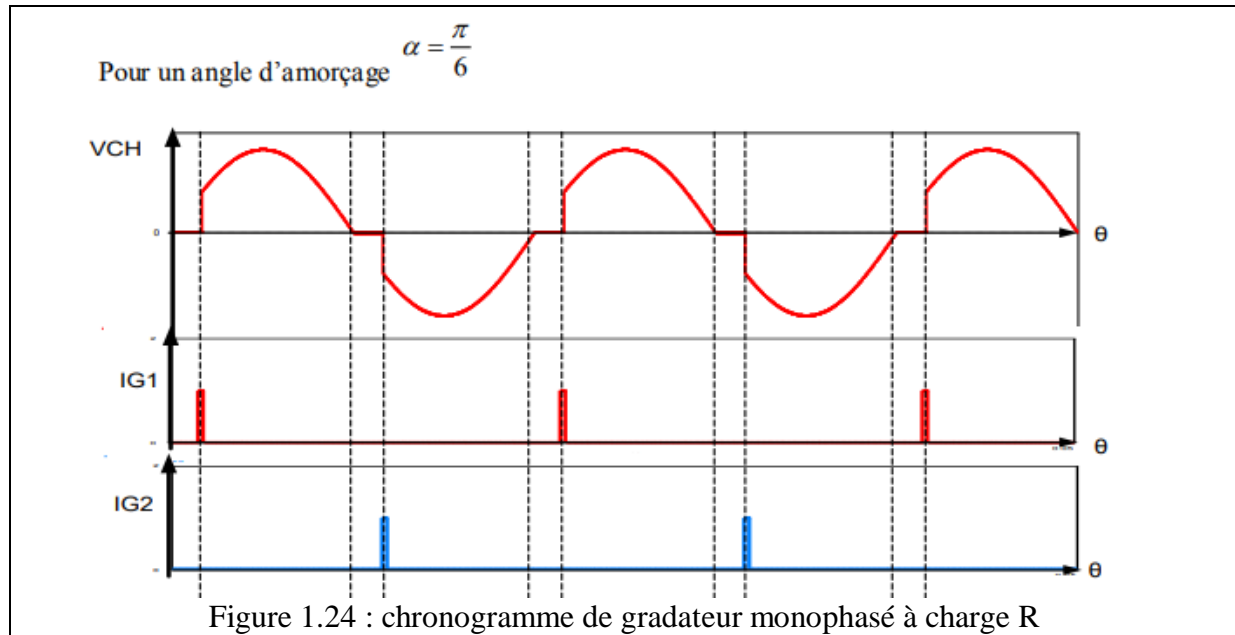
Le thyristor TH1 se bloque : $U_{ch} = 0$ (3)

Pour $\alpha + \pi < \theta < 2\pi$:

TH2 reçoit une impulsion. Il est alors polarisé en direct, Il s'amorce : $V_{ch} = \sqrt{2}V \sin(\theta)$ (4)

La valeur moyenne de U_{ch} est :
$$\langle U_{ch} \rangle = \frac{\sqrt{2} V}{2} \sqrt{1 + \frac{\sin 2\alpha}{2\pi} - \frac{\alpha}{\pi}}$$
 (5)

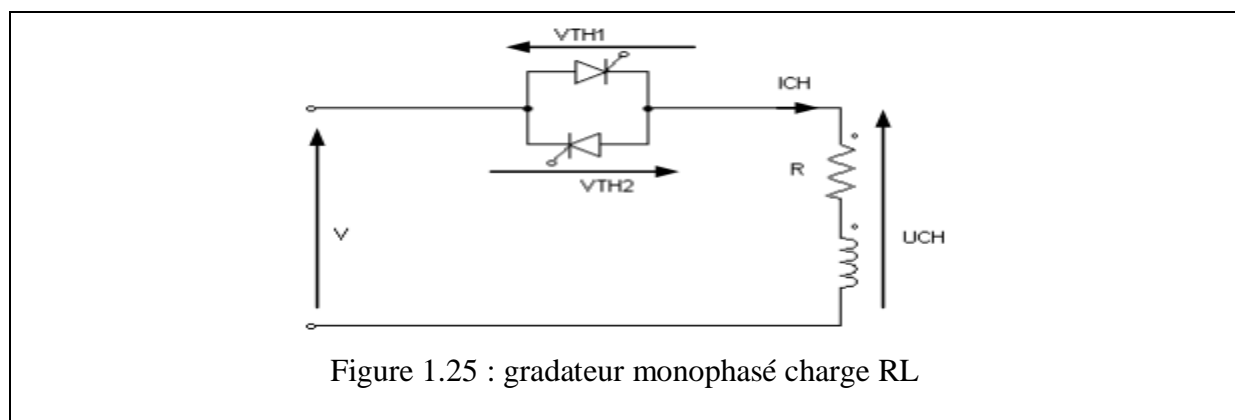
Chronogramme :



1.4.2 gradateur monophasé charge RL :

À cause de l'effet inductif, la conduction se poursuit après la fin de l'alternance, jusqu'à l'instant d'annulation du courant $i(\theta)$.

Lorsque l'angle devient inférieur à ϕ , l'argument du récepteur, le fonctionnement dépend de la nature des signaux appliqués aux gâchettes :



Cas d'une impulsion de courte durée :

Lorsque la commande délivrée est une impulsion de courte durée le thyristor Th2 ne peut pas s'amorcer, le courant va s'annuler. Le thyristor Th1 pourra être réamorçé à l'instant $T+\alpha$: les valeurs moyennes de la tension et de l'intensité de sortie ne sont pas nulles, le gradateur fonctionne en redresseur mono-alternance.

Chronogramme dans le cas d'une impulsion de courte durée :

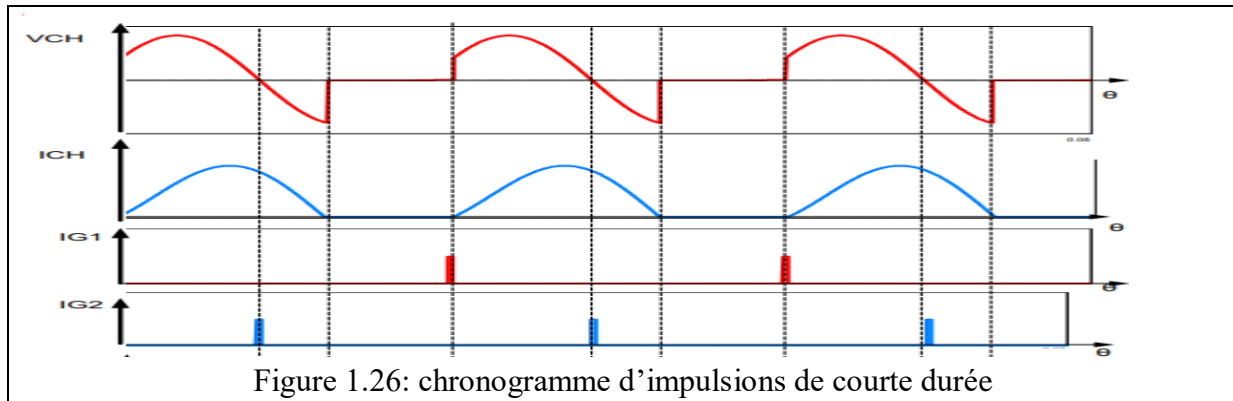


Figure 1.26: chronogramme d'impulsions de courte durée

Cas d'une impulsion de longue durée :

Lorsque la commande délivrée est une impulsion de longue durée, le thyristor Th2 s'amorce lorsque le courant devient négatif. Au bout d'un certain temps, le régime transitoire disparaît, le courant devient sinusoïdal : le gradateur se comporte comme un interrupteur fermé, l'action sur l'angle de retard à l'amorçage est inopérante tant que $\alpha < \varphi$.

Chronogramme dans le cas d'une impulsion de longue durée :

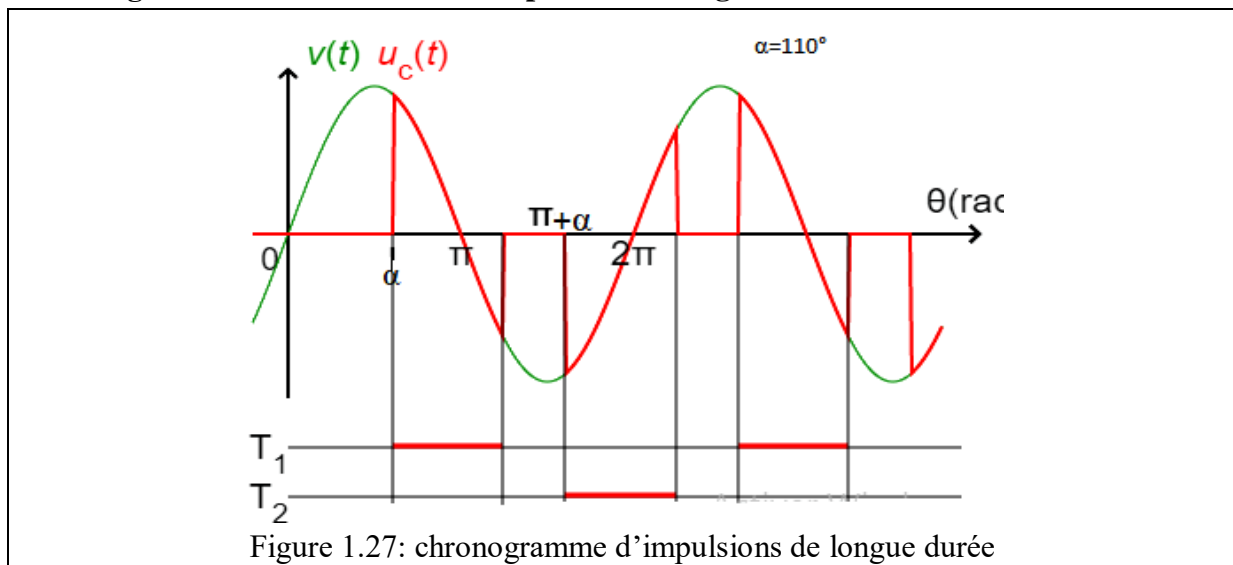


Figure 1.27: chronogramme d'impulsions de longue durée

1.4.3 Gradateur à train d'ondes :

Il fournit à partir du réseau alternatif monophasé ou triphasé, des périodes entières par séries consécutives de tension (amplitude et fréquence du réseau) séparées par des absences totales de tension à un rythme modulable. Il permet donc le réglage d'énergie. Le chronogramme suivant illustre la sortie U_c d'un gradateur à train d'onde.

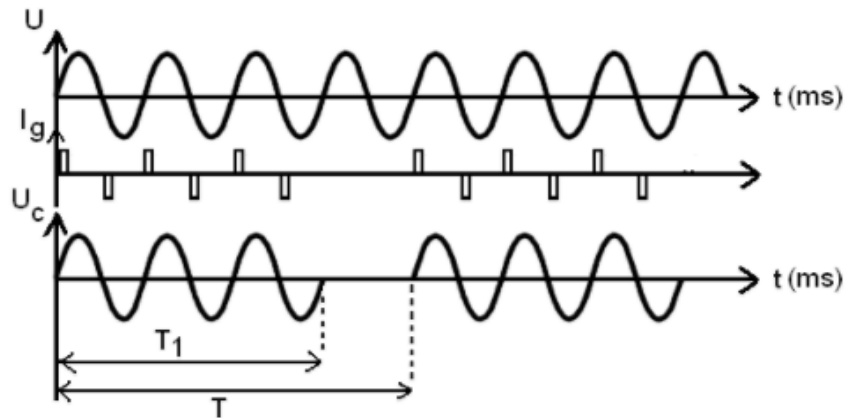


Figure 1.28 : tension de sortie d'un gradateur à train d'onde

Le rapport cyclique α désigne, pour un phénomène périodique, le ratio entre la durée du phénomène et la période du phénomène.

Dans ce cas : $\alpha = \frac{T_1}{T}$ (1)

La puissance est fournie pendant le temps de conduction. Le rapport du temps de conduction sur la période de modulation donne le pourcentage de la puissance max transmise

Donc : $P = \alpha \times P_{max}$ (2)

On se limite dans notre étude par le gradateur monophasé puisque les gradateurs triphasés de différents types ont une commande de thyristor qui ressemble à celle de redresseur triphasé de type PD3.

1.5 Domaines d'application de différents types de gradateurs et de redresseurs :

Certes ces différents types de réalisations de convertisseurs ont un rôle pédagogique important pour la formation de l'élève ingénieur dans l'électronique de puissance mais aussi chaque type d'eux a son propre utilisation dans l'industrie et dans la vie réelle, le tableau suivant montre quelques domaines d'utilisation de chaque convertisseur.

TYPE DE CONVERTISSEUR	DOMAINE D'APPLICATION
Redresseur simple alternance	*Les très faibles charges
Redresseur P2	*Variation de vitesse MCC *Variation de tension continu
Redresseur PD2	*Variation de vitesse MCC *Variation de tension continu
Redresseur PD2 mixte	*Variation de vitesse MCC *Variation de tension continu
Redresseur P3	*utilisé pour les très forts courants et bases tensions
Redresseur PD3 mixte	*Variation de vitesse MCC *Variation de tension continu

Tableau 1.3 : domaines d'application de différents types de redresseurs

TYPE DE CONVERTISSEUR	DOMAINE D'APPLICATION
Gradateur monophasé	*Variation de lumière *Four électrique
Gradateur a train d'onde	*Chauffage *Utilisés sur des systèmes présentant une inertie thermique importante
Gradateur triphasé	*Lumière *Chauffage *Soft Start de moteur asynchrone *Interrupteur statique

Tableau 1.4 : domaines d'application de différents types de gradateurs

1.6 Le thyristor :

1.6.1 Principe et composition :

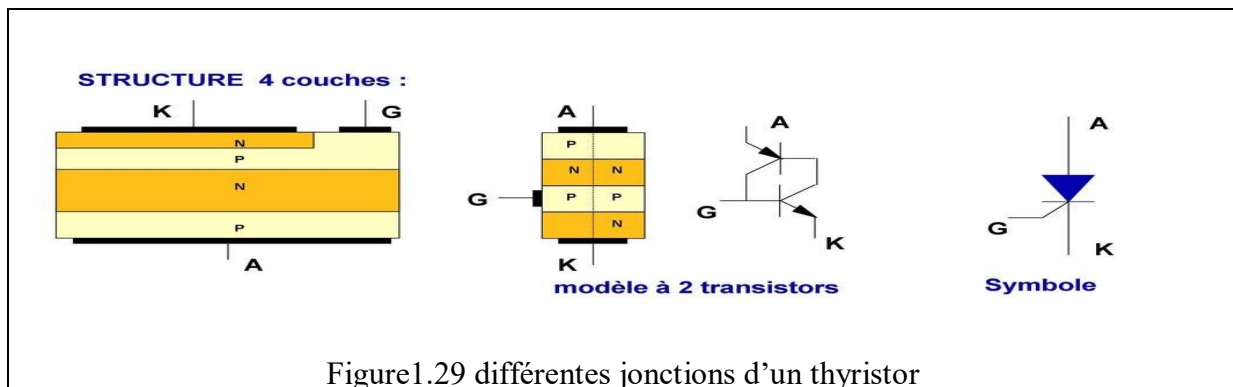


Figure1.29 différentes jonctions d'un thyristor

Le thyristor est un interrupteur statique réversible en tension, il possède trois bornes : l'anode, la cathode et une gâchette qui représente l'électrode de commande. Le thyristor est unidirectionnel en courant, et ne permet le passage que le courant positif de l'anode vers la cathode suite à une impulsion de commande appliquée sur la gâchette.

Il est formé de quatre couches semi-conducteurs dopées alternativement P et N. Il existe donc 3 jonctions P-N dans un thyristor comme deux transistors (fig. 29).

1.6.2 Amorçage d'un thyristor :

L'amorçage d'un thyristor lui rend conducteur de l'anode vers la cathode.

Une impulsion de commande sur la gâchette est nécessaire pour amorcer un thyristor, une fois le thyristor amorcé il reste conducteur sans prendre en considération si le courant de gâchette est maintenu ou coupé.

Lorsqu'il est passant, il se comporte comme une diode avec une tension de l'ordre de 1.2V entre ses bornes.

1.6.3 Blocage d'un thyristor :

Afin qu'un thyristor se bloque il suffit que la tension entre ses bornes devient négative, cette tension peut être naturelle dans un réseau électrique alternatif, forcé par un circuit d'extinction qui lui impose une tension négative mais aussi lorsque le courant s'annule.

Après son blocage le thyristor nécessite un temps pour retrouver son pouvoir bloquant, une tension positive appliquée entre ses bornes avant la fin de ce temps engendre son réamorçage même en absence d'impulsion de courant de commande sur la gâchette.

1.6 CONCLUSION :

Dans ce chapitre nous avons présenté des notions générales sur les convertisseurs électrique et leurs fonctionnement, et on a met l'accent sur l'importance de thyristor dans la commande de ces convertisseurs.

Chapitre 2

La carte à microcontrôleur Arduino

2.1 Introduction

Arduino est une plateforme de développement embarqué Open Source qui a connu une popularité vu son simplicité d'emploi et son écosystème. Il est fourni comme étant tout un package de cartes matérielles et outils logiciels. La plus connue et commercialisée de ses cartes est la fameuse Arduino Uno.

2.2 Connaissances générales sur l'Arduino

2.2.1 Arduino Uno

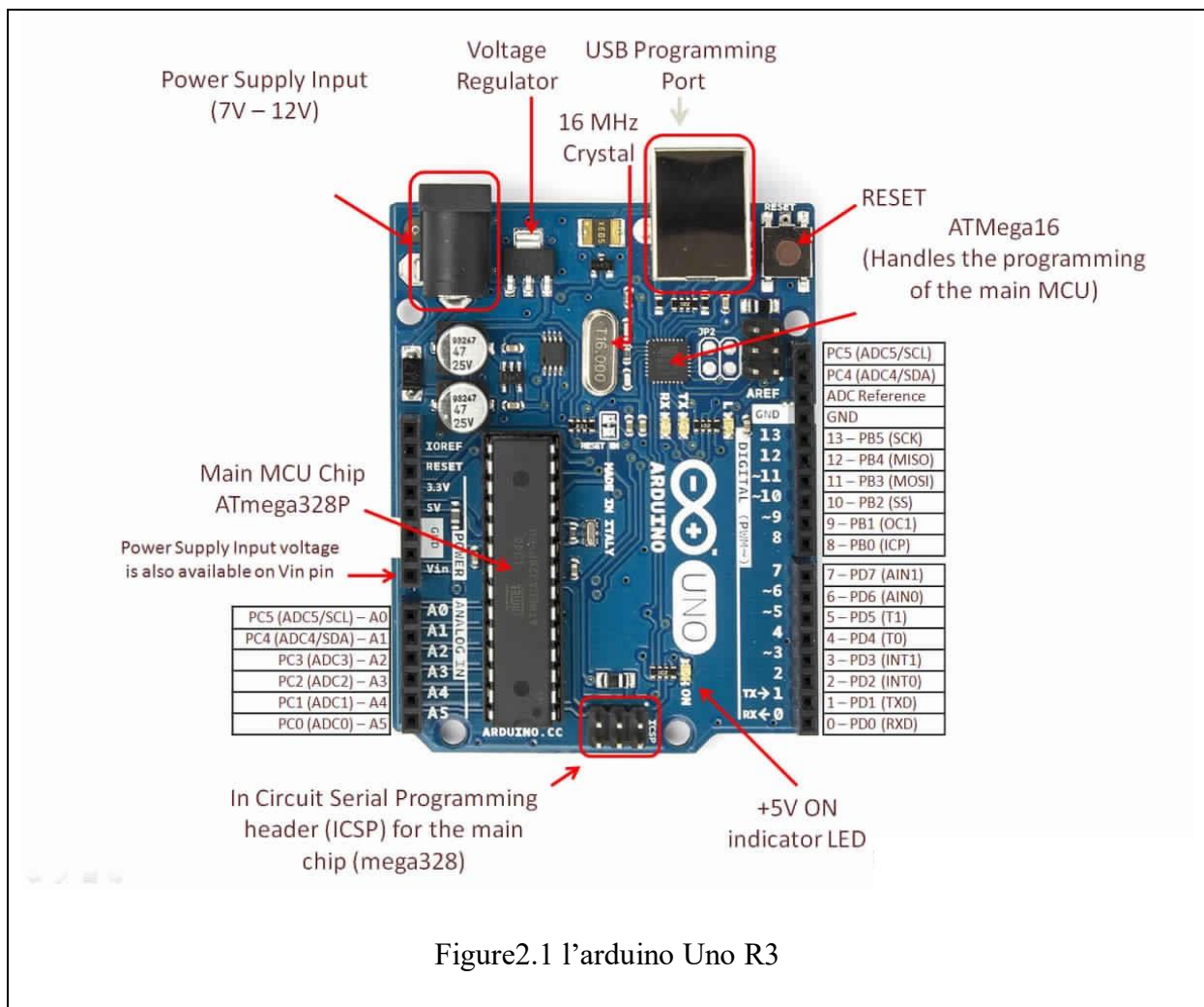


Figure2.1 l'arduino Uno R3

Cette carte est conçue autour du microcontrôleur ATmega 328 de la famille AVR. Ce dernier est un microcontrôleur à 8 bits avec 32 KB de mémoire flash, 2 KB de SRAM, 1 KB de EEPROM et un ensemble des périphériques : timers, ADC, I2C, SPI et UART.

D'autre part, l'arduino uno possède 14 pins numériques d'entrée/sortie et 6 pins d'entrée analogique qui peuvent être employés en mode ADC (conversion analogique-numérique) réparties sur 3 ports, à savoir :

- PORTD : 8 pins (pins numériques de 0 à 7)
- PORTC : 6 pins (pins numériques de 8 à 13)
- PORTB : 5 pins (pins analogiques de 0 à 5)

En plus, elle fournit 6 sorties PWM sur les même pins numériques (on les distingue par un ~). L'oscillateur de synchronisation d'horloge de MCU tourne à 16 MHz.

2.2.2 Famille des microcontrôleurs AVR

Les microcontrôleurs AVR fabriqués par Atmel sont un bon choix afin de concevoir des systèmes embarqués de complexité moyenne. Ce sont des contrôleurs 8 bits de type RISC. Ils sont commercialisés sous plusieurs variétés, les plus connus sont ATmega8 et ATmega16 (les anciennes versions de l'arduino uno (v1 et v2) emploient ATmega16 comme μ C principal alors que les dernières versions (v3) emploient ce μ C comme esclave programmeur en plus du μ C principal ATmega 328p).

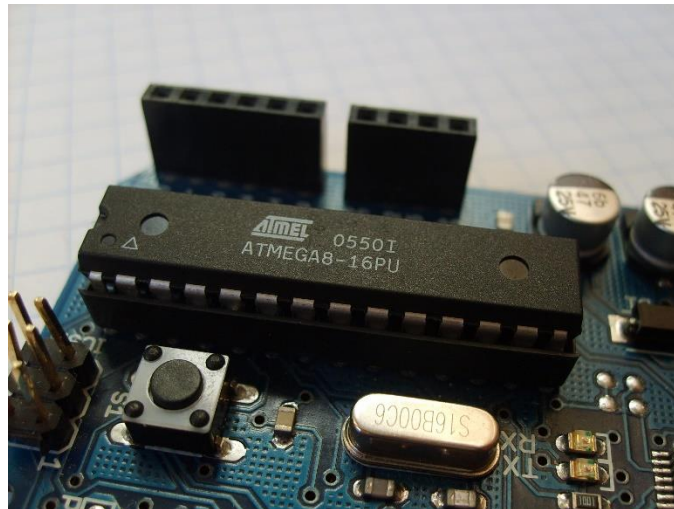


Figure 2.2 microcontrôleur Atmega16 dans l'ancienne version de la carte Arduino Uno

Le point fort de cette gamme, c'est qu'elle fournit un puissant jeu d'instructions de fait que la majorité de ces dernières s'exécutent en un seul coup d'horloge (chose qui n'est pas assurée par les μ C PIC par exemple) et en considérant qu'elles tournent à 16 MHz, on aura une puissance de traitement de 16 MIPS (méga instructions par seconde).

2.2.3 Les outils logiciels de l'arduino : Arduino IDE

C'est un environnement de développement intégré qui offre les fonctions de compilation et téléversement du programme vers le matériel. Il est fourni avec un riche ensemble de librairie, ce qui explique la simplicité de l'emploi de l'arduino.

Pratiquement tout programme développé pour une carte arduino par son IDE doit inclure obligatoirement deux parties :

- Setup() : pour toute déclaration et initialisation, à ce niveau on déclare toute variable et pin qu'on recourra dans le corps de notre programme.
- Loop() : comme la logique d'un microcontrôleur est de n'arrêter jamais l'exécution des instructions, ce bloc de code constitue la partie qui se mettra à s'exécuter infiniment en tant que boucle.

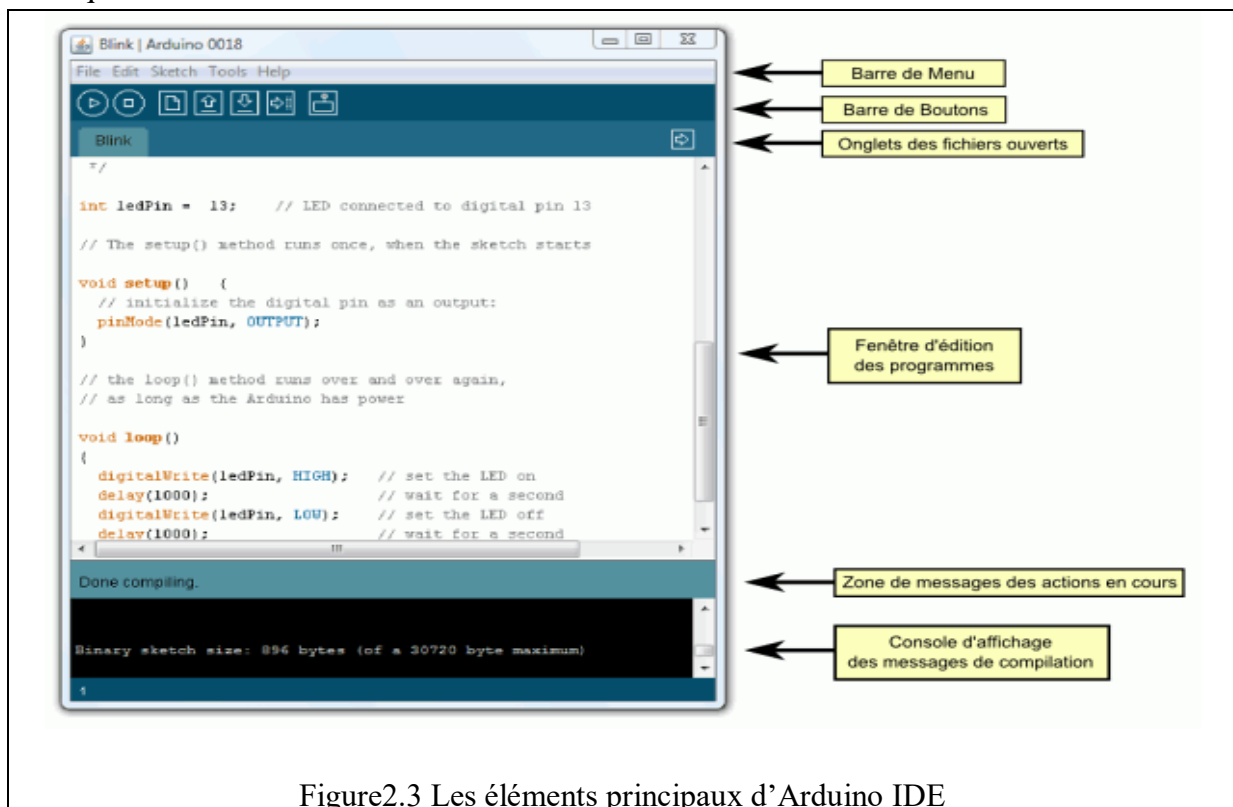


Figure2.3 Les éléments principaux d'Arduino IDE

2.2.4 Langage de programmation de l'arduino

Comme le C/C++ est le langage le plus adapté pour accéder aux fonctionnalités de hardware, dans ce cas le microcontrôleur, arduino a adapté une plateforme de programmation simplifiée dérivant de C/C++ et se basant sur la richesse des librairies.

Les contraintes :

La simplicité de l'environnement arduino n'est atteinte gratuitement, leur "Software Framework" a sacrifié la capacité de l'utilisateur de tout contrôler sur le code de MCU.

En contrepartie, si on cherche le contrôle complet et donc exploiter à fond la puissance totale du microcontrôleur, il sera mieux de migrer vers Atmel Studio, le logiciel fourni par le concepteur même du μ C mais en conservant le support de la carte arduino comme on a toujours besoin du programmeur (dans le cas de l'arduino uno, c'est le hardware ATmega16).

2.2.5 Arduino Uno contre Mega 2560

L'arduino Mega emploie le microcontrôleur Atmel AVR ATmega 2560 possédant beaucoup plus de ressources matérielles. Le tableau suivant montre une comparaison des caractéristiques des deux modèles :



Caractéristiques	Uno	Mega 1560
MCU	ATmega 328p	ATmega 2560
Figure		
Pins numériques IO	14	54
Sorties PWM	6	15
Pins d'entrée analogique	6	16
Mémoire Flash	32 KB	256 KB
SRAM	2 KB	8 KB
EEPROM	1 KB	4 KB
Prix (2018)	40 DT	70 DT

Tableau 2.1 : comparaison des caractéristiques entre Arduino Uno et Arduino Mega

2.3 Manipulation des entrées / sorties d'une carte Arduino

2.3.1 Les commandes prédéfinies des I/O pins

L'environnement de développement d'arduino fournit un ensemble de commandes prédéfinies pour gérer les entrées et les sorties.

D'abord, pour manipuler les entrées/ sorties d'une arduino, il faut initialiser les pins qu'on va utiliser dans la partie Setup() du programme en indiquant si elle sera une entrée (Input) ou bien une sortie (Output) et ça se fait par appel de la commande pinMode() de la façon suivante :

pinMode(numéro de pin, Input/Output);

Il faut indiquer que la carte arduino différencie 3 types de pins :

1-ANALOG pour les entrées analogiques :

La carte arduino uno compte 6 pins analogiques notés de A0 à A5, visant essentiellement à lire les signaux en entrée. Dans ce cadre, l'appel de la commande analogRead() associe à un signal électrique qui varie entre 0 et 5V une variable entière dans l'intervalle 0..1023. Elle s'écrit de la façon suivante :

Pour lire le pin A0 : X=analogRead(A0) ;

2-DIGITAL pour les entrées/ sorties numériques :

On compte pour arduino uno 14 pins numériques allant de 0 à 13 (les pins 0 et 1 servent à élaborer une communication série Tx – Rx donc vaut mieux les utiliser prudemment). Après avoir choisit de mettre les pins en entrée ou sortie comme indiquer avant, soit on cherche à lire un pin Input donc on appelle la commande digitalRead() comme suit :

X=digitalRead(numéro de pin) ;

Soit on vise régler l'état logique d'un pin Output par la commande `digitalWrite()` qui s'écrit :
`digitalWrite(numéro de pin, LOW/HIGH);`
avec LOW pour coder un 0 et HIGH pour coder un 1.

3-PWM pour les sorties spéciaux MLI :

Il y en a 6 qu'on distingue par un tilde ~ sur la carte arduino uno. Ces pins simulent les signaux analogiques de commande MLI à travers la commande :

`analogWrite(numéro de pin, valeur de rapport cyclique);`

avec cette commande on a la possibilité de changer le rapport cyclique qui est codé par une valeur entière de 0 à 255 sans toucher à la fréquence.

2.3.2 Manipulation directe des GPIO ports de l'arduino

2.3.2.1 Pourquoi on a recourt à la manipulation directe des ports en laissant à coté les commandes prédéfinies d'arduino ?

Au premier coup d'œil, on pense que c'est une mauvaise idée pour les raisons suivantes :

- Le code est beaucoup plus difficile à déboguer et maintenir vu qu'on s'approche plus du matériel de microcontrôleur et le langage machine et donc il est plus difficile à le faire expliquer aux gens (l'intérêt d'un code est non seulement d'être fonctionnel mais aussi compréhensible).

- le code est moins portable dans le sens qu'il faut faire attention en basculant entre les différents modèles d'arduino, avec ce même code, on doit vérifier les pins des ports, par contre on saute cette étape par les commandes prédéfinies.

- on peut causer plus facilement un mal-fonctionnement de la carte arduino. En tapant différemment cette instruction (`DDRD=B11111101` ; à expliquer plus tard) on risque de toucher au pin 1 qui doit être impérativement une entrée comme elle représente la ligne de réception de la communication série de la carte arduino.

De l'autre côté, il y a plusieurs aspects positifs pour l'accès direct des ports, on cite :

- On gagne en termes de taille de programme qu'on cherche à exécuter et on profite de quelques bytes de code compilé vu qu'on configure les pins simultanément et on peut même l'adapter à la mémoire flash du microcontrôleur.

- C'est une solution lorsqu'on cherche à configurer des sorties en même temps, surtout lorsque le microcontrôleur est couplé avec des systèmes électroniques sensibles au temps.

Au lieu de faire appel successivement au commande `digitalWrite()` ce qui cause un déphasage de quelques microsecondes suite à l'exécution séparée, l'instruction `"PORTB |= B1100 ;"` touche les pins au même moment.

- Lorsqu'on a intérêt à basculer les pins très rapidement, les commandes prédéfinies de l'arduino ne sont pas le bon choix vu que leurs code source occupent des dizaines d'instructions et donc compilé à plusieurs cycles de machine (avec l'arduino uno, chaque exécution d'une machine instruction se fait à un coup d'horloge de 16 MHz). La manipulation directe des ports peut faire le même travail à un nombre plus petit de coup d'horloge.

2.3.2.2 Les registres des GPIO ports pour l'arduino

Atmel AVR est un microcontrôleur 8 bits, par conséquent tous ses ports sont de largeur 8 bits. Il est associé à chaque port 3 registres, eux-mêmes de 8 bits. Chaque bit d'un tel registre configure un pin donné du port. Par exemple, le Bit0 de ces registres touche le fonctionnement de Pin0 de ce port et etc.

Ces 3 registres mentionnés sont appelés :

- DDRx, -PORTx -PINx

Avec le x réfère le nom du port. Dans le cas de l'arduino Uno, ce sont les ports B, C et D.

• DDRx register :

Le Data Direction Register configure la direction des données pour les pins des ports, dans le sens que son paramétrage détermine est-ce-que les pins seront utilisés en tant qu'une entrée ou sortie. En fait, écrire un 0 dans le bit de registre DDRx rend le pin correspondant un Input (entrée) tant que écrire un 1 rend ce dernier un Output (sortie).

• PINx register :

PortIN est utilisé pour lire les données des pins. Dans le but de lire les informations tirées d'un tel pin, tout d'abord, il faut changer la direction du data comme étant une entrée, ça se fait en mettant un zéro dans le bit correspondant du registre DDRx.

Si ce pin est configuré comme Output, alors lire les données de PINx reflète l'état de sortie.

Remarque: il faut savoir qu'il y a deux modes pour configurer une entrée : en tant qu'une entrée à trois états (tri-stated input) ou bien une entrée avec le pull-up interne activé.

• PORTx register :

Il est utilisé pour deux raisons :

1-controler les données des sorties quand les pins associés sont configurés comme Output : en effet, lorsqu'un bit de DDRx est mis au niveau-haut permettant au pin de se comporter comme une sortie, écrire dans son bit de PORTx change directement son état de sortie selon l'état logique écrit.

En résumé, si on cherche à modifier les informations de sortie d'un pin, il faut nécessairement passer par l'écriture dans son bit de registre PORTx, bien-sûr sans oublier de le configurer en Output avec un 1 dans son bit de DDRx.

2-controler les résistances pull-up quand les pins associés sont configurés comme Input : cas le cas où on utilise les pins comme des entrées en mettant un 0 dans leurs bits de DDRx, leurs bits de PORTx permettent d'activer/ désactiver les pull-up résistances liées à ces pins. Pratiquement, pour les activer, il suffit de mettre un 1 dans le bit associé de PORTx et dans le cas contraire, c'est un 0 et on passe à un pin à 3 états.

Pour mieux comprendre ce point, lorsqu'on est en mode d'entrée et le pull-up est activé, l'état par défaut de pin est le niveau-haut et donc même si on n'a rien connecté et qu'on cherche à lire son état, ça donne toujours 1, seulement une connexion de l'extérieur avec la masse "pull-down" changera son état à 0 et donc on peut lire le niveau bas.

Il est conseillé d'activer les résistances pull-up pour les pins d'entrée car le cas contraire le pin aura une impédance élevée du mode des 3 états et si elle est flottante (non-connectée) la simple charge statique présente à l'entour de pin changera son état logique et donc on sera incapable de prévoir son état à un instant donné lorsqu'on cherche à le lire, ça peut poser des problèmes lorsque le programme dépend de l'état de cet Input. L'exception est lorsqu'on cherche d'activer l'ADC, le pin doit être configuré comme "tri-stated input".

Remarque: chaque port est lié à plusieurs pins et donc changer la configuration d'un port revient à toucher le paramétrage de tous les pins relatifs à ce port. Pour configurer un seul pin, il faut accéder au bit relatif à ce dernier dans les registres de son port.

Résumé : Le tableau suivant présente le lien entre la configuration des registres GPIO et les fonctions des pins :

Fonction de pin	DDRx.n	PORTx.n	PINx.n
Entrée à 3 états	0	0	Lire le bit d'état : $X = \text{PINx.n}$;
Entrée pull-up	0	1	Lire le bit d'état : $X = \text{PINx.n}$;
Sortie	1	écrire le bit d'état : $\text{PORTx.n} = X$;	-

Tableau 2.2 : lien entre la configuration des registres GPIO et les fonctions des pins

Remarque : "x.n" est un pseudo code pour designer l'accès à un bit donné d'un registre, les instructions complètes qui font appel aux opérations des bits seront explicitées dans la partie de la manipulation directe des ports.

A retenir :

La manipulation des registres des ports permet un accès de bas niveau plus rapide aux I/O pins de la carte arduino par rapport à ses fonctions prédéfinies.

Le microcontrôleur employé par la carte Arduino Uno (ATmega 328p AVR) possède 3 ports :

- port B (pins numériques de 8 jusqu'à 13)
- port C (pins d'entrée analogique)
- port D (pins numériques de 0 jusqu'à 7)

Chaque port est contrôlé par 3 registres :

- registre DDR : détermine est-ce-que le pin est une entrée ou sortie.
- registre PORT : contrôle est-ce-que le pin est au niveau-haut "HIGH" ou bas "LOW".
- registre PIN : lire l'état des pins configurés comme des entrées.

Les registres DDR et PORT peuvent être accédés par écriture ou lecture alors que le registre PIN ne peut être accédé que par lecture comme il présente l'état des entrées.

Chaque bit de ces registres correspond à un seul pin. Par exemple, le faible bit des registres DDRB, PORTB et PINB réfère au pin PB0 qui correspond au pin numérique 8 de l'arduino uno.

Le mapping des ports montrent clairement le lien entre les bits des registres et les pins de l'arduino.

Voir l'annexe pour la figure complète de "pin mapping" de la carte arduino uno.

2.3.2.3 Les instructions de la manipulation directe des bits des GPIO ports d'arduino

Lorsqu'on programme n'importe quel système embarqué et en particulier la carte arduino, avoir la connaissance de manipuler des bits particuliers est une compétence essentielle et on peut le sentir qu'on doit configurer des bits précis dans les registres de contrôle du matériel tel que les ports.

Les opérations du système binaire compatibles avec l'environnement de développement arduino sont celles adaptées par le langage c/c++.

En particulier, l'arduino permet de définir les nombres binaires en ajoutant le préfixe 0b, par exemple (0b11==3) et les constantes définissant les bytes, qui peuvent être affectées à des registres par exemple, sont mentionnées par le préfixe B, par exemple B11111111.

Plus pratiquement :

Lorsqu'on programme en niveau-bas un microcontrôleur, on a généralement besoin de mettre en 0 ou 1 un bit donné d'un I/O registre. La documentation du microcontrôleur fournit des noms mnémoniques pour les différents bits de ces registres.

Ces noms des bits qui contrôlent les GPIO sont :

-DDxn pour le registre DDRx (équivalent à Input/Output pour pinMode())

- PORTxn pour le registre DDRx (équivalent à HIGH/LOW pour digitalWrite())

avec x pour le nom du port (B, C, D) et n le numéro du bit qu'on détermine à partir du pin mapping

Le moyen pour associer ce nom (qui représente finalement le numéro de bit dans son registre qui est un octet "byte") et la valeur qu'on veut qu'elle soit affectée directement au registre est à travers la macro-instruction _BV() (BV pour bit value).

Cette macro-instruction (qui représente finalement un simple "bit left shift" c.-à-d. _BV(3) » 1<<3 »0x08) rend le code plus lisible.

Par exemple : DDRD=_BV(PD7) ; permet de mettre un 1 dans le haut bit du registre DDRD, c'est équivalent à écrire PD7=0b1 ; donc finalement configurer son pin associé (pin 7 pour arduino uno) comme sortie.

Et donc on peut configurer un registre en un seul coup comme suit :

DDRD=_BV(PD7) | _BV(PD6) | _BV(PD0) ;

Le résultat est de configurer les pins 7, 6 et 0 comme des sorties.

Les instructions utiles pour configurer les registres des IO ports :

- définir le pin Arduino 13 comme sortie :

DDRB |= 1 << DDB7;

- définir le pin Arduino 13 comme entrée :

```
DDRB &= ~(1 << DDB7);
```

- Régler le pin Arduino 13 au niveau haut :

```
PORTB |= 1 << PORTB7;
```

- Régler le pin Arduino 13 au niveau bas :

```
PORTB &= ~(1 << PORTB7);
```

2.3.3 Les timers et leurs interruptions

2.3.3.1 Introduction aux timers de la carte arduino

Sans aucun doute, les timers sont l'une des fonctionnalités puissantes d'un uC qu'on ne peut pas le manipuler sans les servir vu que dans le monde industriel, le milieu naturel d'un système de commande, tout est lié au temps.

En particulier pour l'Arduino, plusieurs macro-fonctions prédéfinies font intervenir les timers de manière cachée, telles que celles liées au temps (`delay()`, `millis()`..) et celle qui génère un signal PWM (`analogWrite()`).

En générale, il est toujours associé un compteur (counter) à un timer, c'est une pièce de matériel du uC capable de gérer des événements temporels.

Plus précisément, l'arduino Uno possède 3 timers : Timer0, Timer1 et Timer2. Timer0 et Timer2 sont tous les deux à 8 bits alors que Timer1 est à 16 bits. Dans ces deux cas, la différence importante est la résolution du comptage : alors que 8 bits donne 256 valeurs, 16 bits peut atteindre 65536 comme la plus longue valeur de comptage.

L'arduino mega possède en plus des timers cités pour l'uno, 3 autres timers similaires 0 Timer1 (16 bits) : Timer3, Timer4 et Timer5.

2.3.3.2 Architecture des timers de la carte arduino

Les timers sont configurés à travers des registres spéciaux. Les plus importants de ces derniers sont :

- **TCCR_x** : Timer/ Counter Control Register, on peut configurer le "prescaler" dans ce registre à travers les 3 bits CS12, CS11 et CS10. C'est la valeur par laquelle on divise la fréquence d'horloge de système (16MHz) pour avoir la fréquence désirée de comptage de timer.

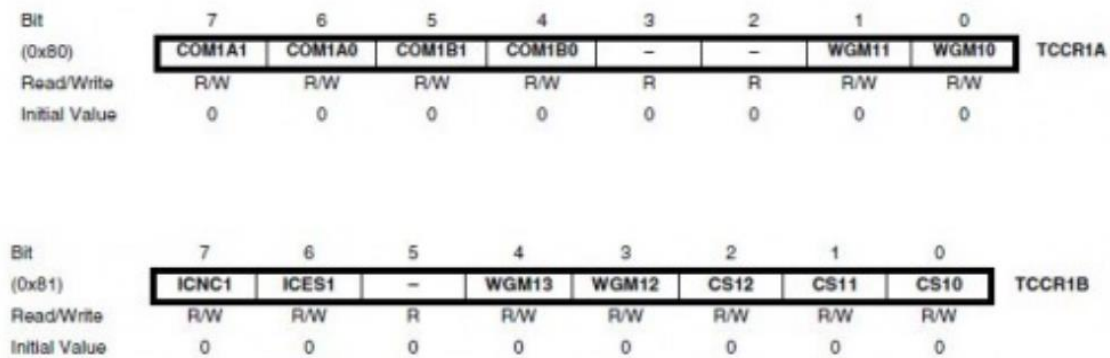


Figure 2.4 Registre TCCR1

- **TCNTx** : Timer/ Counter Register, il enregistre la valeur actuelle du timer.
 - **OCRx** : Output Compare Register
 - **ICRx** : Input Capture Register
 - **TIMSKx** : Timer Interrupt Mask Register, pour activer/ désactiver les interruptions des timers
 - **TIFRx** : Timer Interrupt Flag Register, indique si l'interruption du timer est déclenchée
- Remarque : x réfère au numéro du timer

2.3.3.3 Configuration de la fréquence des timers

On peut associer à chaque timer une source d'horloge indépendante, ce qu'on appelle Clock Select (CS).

Pratiquement, pour configurer la fréquence d'un timer, par exemple 2Hz pour Timer1, on suit les étapes suivantes :

- 1- On sait bien que la fréquence de CPU de l'arduino est 16 MHz
- 2- Suivant le timer choisi, on sait la valeur maximale du compteur qui peut atteindre 256 pour 8bits et 65536 pour 16bits
- 3- On divise la fréquence de CPU par la valeur de "prescaler" choisi (comme indiqué avant), exemple : $16000000/8256=62500$. D'après le tableau du Clock Select, la valeur 256 pour le prescaler est donnée par CS12=1, CS11=0 et CS10=0.
- 4- Diviser le résultat par la fréquence cherchée ($62500/2\text{Hz}=31250$)
- 5- On vérifie si la valeur trouvée est plus petite que la valeur maximale du compteur ($31250 < 65536$: bien), sinon on choisit une valeur de Prescaler plus grande.

Table 16-5. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	clk _{IO} /1 (No prescaling)
0	1	0	clk _{IO} /8 (From prescaler)
0	1	1	clk _{IO} /64 (From prescaler)
1	0	0	clk _{IO} /256 (From prescaler)
1	0	1	clk _{IO} /1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Figure2.5 Tableau de CS pour la configuration de Prescaler

2.3.3.4 Les différentes modes des timers de la carte arduino

Les timers peuvent être configurés selon différents modes, les plus intéressants sont :

- **Le mode PWM** (Pulse Width Modulation mode) configure les sorties OCx (Output Compare) pour générer des signaux MLI.
- **Le mode CTC** : met à zéro le timer lorsque son comptage atteint la valeur du registre du comptage.

La sélection du mode du compteur se fait par les bits WGM13, WGM12, WGM11 et WGM10 du registre TCCR_x (TCCR_{xA} et TCCR_{xB})

2.3.3.5 Les interruptions des timers

L'intérêt des interruptions du timer :

A plusieurs reprises dans le programme, on cherche à exécuter des blocs de code de manière régulière tout un intervalle de temps fini. Avec l'arduino, de premier coup, on pense à la méthode la plus facile : appeler la fonction **delay()** mais il faut savoir que cette dernière met en pause tout le programme pour cette durée de temps, ce qui consiste un gaspillage des capacités du **µC** surtout si vous avez besoin de faire d'autres traitements en attendant. Dans ce cadre entrent en jeu les interruptions des timers.

Comment manipuler des interruptions du timer pour la carte arduino ?

Le programme principal télé-versé au **µC** s'exécute normalement de manière séquentielle instruction par instruction et l'interruption est un événement externe à ce programme et comme indique son nom, elle interrompt l'exécution de ce programme pour exécuter un autre, plus urgent à cet instant, qu'on appelle routine de l'interruption. Après ceci finit, le **µC** reprend l'exécution du programme principal ou il s'est arrêté.

En ce qui concerne les timers, avant qu'une interruption peut déclencher sa routine, il y a une condition qui doit être vérifiée : il faut activer le masque correspond à l'interruption (Interrupt Mask), cela est fait en mettant à 1 le bit correspondant dans le registre TIMSK_x.

Lorsque une interruption est produite, son drapeau (Interrupt Flag) est mis à 1 dans le registre TIFR_x se qui déclenche sa routine comme décrit avant. Finalement, le bit dans le registre des drapeaux d'interruption se met à 0 automatiquement.

Mise en œuvre pratique :

Dans notre travail, on a servi des bibliothèques des timers. Ces bibliothèques constituent des fonctions prédéfinies pour mettre en emploi les différentes fonctionnalités des timers.

Dans notre cas, on en sert pour configurer les interruptions par les commandes suivantes :

- **Initialize**(période) : il faut appeler cette commande tout d'abord pour spécifier la période du timer en microsecondes, elle est par défaut mise à 1 seconde.
- **attachInterrupt**(fonction, période) : elle permet de faire appel à cette fonction tous les périodes spécifiées. Il faut faire attention de ne pas exécuter une interruption très compliquée à une fréquence très élevée car on risque que CPU n'entre pas la boucle principale (main Loop) et il se bloque.

2.3.4 Les interruptions externes

2.3.4.1 L'intérêt des interruptions externes

Ils sont impliqués essentiellement lorsqu'on cherche que notre microcontrôleur, d'un part, synchronise son travail avec des signaux venant des systèmes électriques dont il y est couplé et d'autre part, suit l'évolution de plusieurs de ses périphériques de manière automatique sans la peine de le perturber. Le μC est employé pour suivre les phénomènes de son environnement qui sont en général non-conditionnelles et imprévisibles dans le temps et les suivre périodiquement en s'appuyant sur les capacités de chronométrage des timers n'est pas évident vu qu'il perturbe le fonctionnement principal du microcontrôleur et il est pénalisant en terme de temps d'exécution.

Dans ce cadre intervient les interruptions externes, pour résoudre les problèmes de timing en surveillant les entrées du système de commande.

Pratiquement : si on cherche que notre μC suit un tel capteur qui envoie des impulsions à chaque fois qu'un phénomène est observé, il sera très puissant et rusé d'appeler une interruption qui soulage le programme principal du μC quand il est en train d'exécuter d'autre chose et simultanément il ne perd pas les données provenant de son capteur.

2.3.4.2 Les routines d'interruptions

ISR (Interrupt Service Routines) est la fonction qu'on appelle lorsque la cause de l'interruption se produit.

Elles sont un type particulier des fonctions : elles ne possèdent aucun paramètre et ils ne retournent rien. Généralement, il est recommandé que les routines d'interruptions soient le plus court et vite possible. Si dans notre code, on emploie plusieurs ISRs, seulement une seule peut être exécutée à un instant donné, les autres vont attendre jusqu'à la fin de la routine présente dans un ordre de priorité.

Les variables globales ont la possibilité d'échanger les données entre la routine et le programme principal, en le déclarant comme des variables volatiles.

2.3.4.3 Manipulation des interruptions externes par la carte arduino

La fonction prédéfini `attachInterrupt()` s'occupe des interruptions externes. En effet, l'arduino fournit des pins numériques (Digital Pins) qui peuvent réagir aux interruptions externes. Par exemple, ce sont les pins 2 et 3 pour l'Uno.

Chacun de ces pins est lié à une interruption par un numéro et donc c'est le premier paramètre à vérifier pour servir de la fonction `attachInterrupt()`. Normalement, on utilise la commande `digitalPinToInterrupt()` pour associer à un pin numérique un numéro d'interruption spécifique.

Par exemple, si on veut connecter une interruption externe au pin 3. On écrit `digitalPinToInterrupt(3)` comme premier paramètre de la fonction `attachInterrupt()`.

Les numéros d'interruptions :

Il est conseillé d'utiliser la commande `digitalPinToInterrupt(pin)` au lieu d'utiliser les numéros d'interruptions directement dans le code vu que leurs pins associés changent d'un model arduino à un autre (il est indiqué dans le mapping les pins associés aux numéros d'interruption). C'est une astuce pour garantir la portabilité du code.

Pratiquement, le tableau suivant indique les numéros d'interruptions associés aux pins, pour les cartes Uno et Mega :

modèle	INT.0	INT.1	INT.2	INT.3	INT.4	INT.5
Uno	2	3	-	-	-	-
Mega 2560	2	3	21	20	19	18

Tableau 2.2 : les numéros d'interruptions associés aux pins pour les cartes Uno et Mega

Syntaxe :

On a deux façons de déclarer l'interruption externe :

```
attachInterrupt( digitalPinToInterrupt(pin), ISR, mode) ;
attachInterrupt( numéro d'interruption, ISR, mode) ;
```

avec mode définit la cause pour laquelle l'interruption doit être déclenchée :

- RISING (lorsque l'état de pin change de bas vers haut)
- FALLING (lorsque l'état de pin change de haut vers bas)
- CHANGE (chaque fois que le pin change de valeur)

Précautions :

- Lors de l'utilisation de cette fonction d'interruption, il faut savoir que la commande `delay()` permet de faire tourner le μC à vide pendant un certain délai désiré et donc gagner des temps de pause et `millis()` retourne le temps passé dès l'instant que le microcontrôleur a commencé l'exécution du programme) ne donnent pas les résultats envisagées et détourne mal le fonctionnement du code vu qu'elles même dépendent des interruptions.

- Si on cherche à avoir des variables modifiables dans le corps du code de l'interruption, il faut les déclarer comme des variables volatiles, pratiquement en ajoutant "volatile" devant le type de la variable lors de sa déclaration.

Chapitre 3

Démarche et développement du projet

3.1 Commande du hacheur élévateur

Notre commande de ce convertisseur électronique est à travers la génération d'un signal PWM à rapport cyclique et fréquence variables et à chaque fois qu'on varie une d'eux, l'autre paramètre reste intouchable. En plus, le cahier des charges de projet exige une limitation des valeurs de ces derniers, énoncé de la façon suivante :

Commande à rapport cyclique variable de 0.1 à 0.9 et fréquence fixe de 1 kHz à 30 kHz, choisit par l'utilisateur

Afin de résoudre cette problématique, on a pensé au mode PWM qui offre les timers de notre microcontrôleur.

Dans le but de générer un signal PWM de bonne précision, on a recouru au timer avec la haute résolution, celle de 16 bits. Et comme on emploie l'arduino uno, c'est le Timer1.

On sait bien qu'à chaque timer est associé un compteur, dans ce cas, c'est le compteur numéro 1 noté TCNT1 dont on a servi, qui est un registre 16 bits. En fait, ce compteur s'incrémente par un seul pas à chaque top d'horloge précisée par notre configuration. Il compte 2^{16} fois, à savoir jusqu'à sa valeur maximale 65536.

On avait le choix de configurer la fréquence de l'horloge du compteur : soit elle est celle de l'horloge principale de l'uC (16 MHz), soit un sous-multiple. En se référant au Datasheet du μC , on a compris que les bits de cette configuration sont du Prescaler et qui fonctionnent de la façon suivante :

$$p = (CS12, CS11, CS10)_{base2}$$

Exemples :

$p=0=(000)_{base2}$ » » timer inactif

$p=1=(001)_{base2}$ » » fréquence principale f_p

Pratiquement dans notre travail, on a choisi de prendre $p=2=(010)_{base2}$ » » $f_t = f_p/8 = 2\text{MHz}$ » » Ce qui donne comme résultat que le compteur parcourt tous ses valeurs pendant un cycle de 32.768 ms, dans le sens que la fréquence de comptage des valeurs d'un cycle du compteur est 30.5 KHz, c'était une valeur confortable pour notre travail comme l'intervalle de fréquence de notre PWM mentionné dans le cahier des charges est 1KHz..30KHz, donc on a atteint la valeur maximale de fréquence demandée.

On a configuré notre Timer1 avec deux registres de contrôle 8 bits :TCCR1A et TCCR1B (Timer/ Counter Control Register). Les bits CS12, CS11 et CS10 sont les bits 2, 1 et 0 du registre TCCR1B.

Le Timer1 comporte aussi deux registres 16 bits : OCR1A et OCR1B (Output Compare Register) qui sont comparés au registre TCNT1 pour déclencher différentes actions. Les deux sorties associées sont CO1A et CO1B (Compare Output).

En ce qui concerne notre démarche de travail, on s'est servi de la sortie OC1B (la sortie B de Timer1) qui est liée au pin 10 de l'arduino uno, donc on remarque dans notre code que ce pin est la sortie de génération du signal PWM, explicitée par cette instruction :

```
pinMode(10, OUTPUT);
```

En plus, on a choisi entre les différents modes offerts par un timer celui qui répond le mieux à notre besoin de créer un signal PWM. C'était le mode Phase and Frequency Correct avec contrôle de la valeur maximale du compteur par le registre ICR1. Ce mode est sélectionné par les bits WGM13=1, WGM12=0, WGM11=0 et WGM10=0 de registre TCCR1B.

Cette première partie de configuration de Timer1 au niveau de registre TCCR1B est exprimée par cette ligne de code : `TCCR1B = _BV(WGM13) | _BV(CS11);`

On a utilisé le registre ICR1 (Input Control Register) ici pour fixer la valeur maximale que prend le compteur dans son registre TCNT1. Lorsqu'il atteint cette valeur maximale, il est décrémenté jusqu'à revenir au 0. Il y a donc une phase croissante (up-counting) suivi d'une phase décroissante (down-counting).

L'intérêt de registre OCR1B pour notre travail est de contenir la valeur du signal de consigne. On cherche à faire basculer la sortie OC1B à chaque fois que le registre TCNT1 est égal à OCR1B.

Dans ce cadre, il nous est prioritaire de continuer la configuration de la sortie B de Timer1, cette fois, il faut agir sur les bits COM1B1 et COM1B0 de registre TCCR1A. Nous choisissons la configuration COM1B1=1 et COM1B0=0, qui conduit au comportement suivant : mettre à 0 OC1B lorsque la comparaison est vérifiée lors de up-counting et à 1 lors de down-counting.

Cette instruction de setup() répond à ce besoin : `TCCR1A = TCCR1A | _BV(COM1B1);`

La période T du signal que nous avons obtenue est le double de la période de l'horloge choisie multiplié par la valeur de ICR1. En effet, $f_s = f_T / (2 * ICR1)$ avec f_s fréquence de signal et selon la valeur de Prescaler adéquate, on a la relation entre la fréquence CPU et celle de compteur f_T , donc finalement : $f_s = (f_p / p) / (2 * ICR1)$.

Par exemple : Comme on a choisi $f_T = f_p / 8$ ($p = (010)_{base2}$, d'après le tableau de Prescaler, c'est une division par 8 comme indiqué avant) et $ICR1 = 400$, la fréquence de signal sera $f_s = 2.5\text{KHz}$. Donc une variation du registre ICR1 a un effet direct sur la valeur de la fréquence, sans à priori, toucher au rapport cyclique, donc l'un des solutions de notre problématique est atteint, reste à varier le "duty-cycle".

Pour mettre à jour le registre OCR1B avec la valeur de signal de consigne, on fait appel à une interruption déclenchée lorsque le compteur TCNT1 atteint la valeur maximale ICR1. Pour cela, il faut activer le bit TOIE1 (Timer Overflow Interrupt Enable) de registre TIMSK1 (Timer Interrupt Mask Register).

Comme résultat, on génère un signal MLI ou on a accès à varier son rapport cyclique et sa fréquence : le rapport cyclique est proportionnel à la valeur de registre OCR1B et la fréquence est inversement proportionnelle à la valeur de registre ICR1.

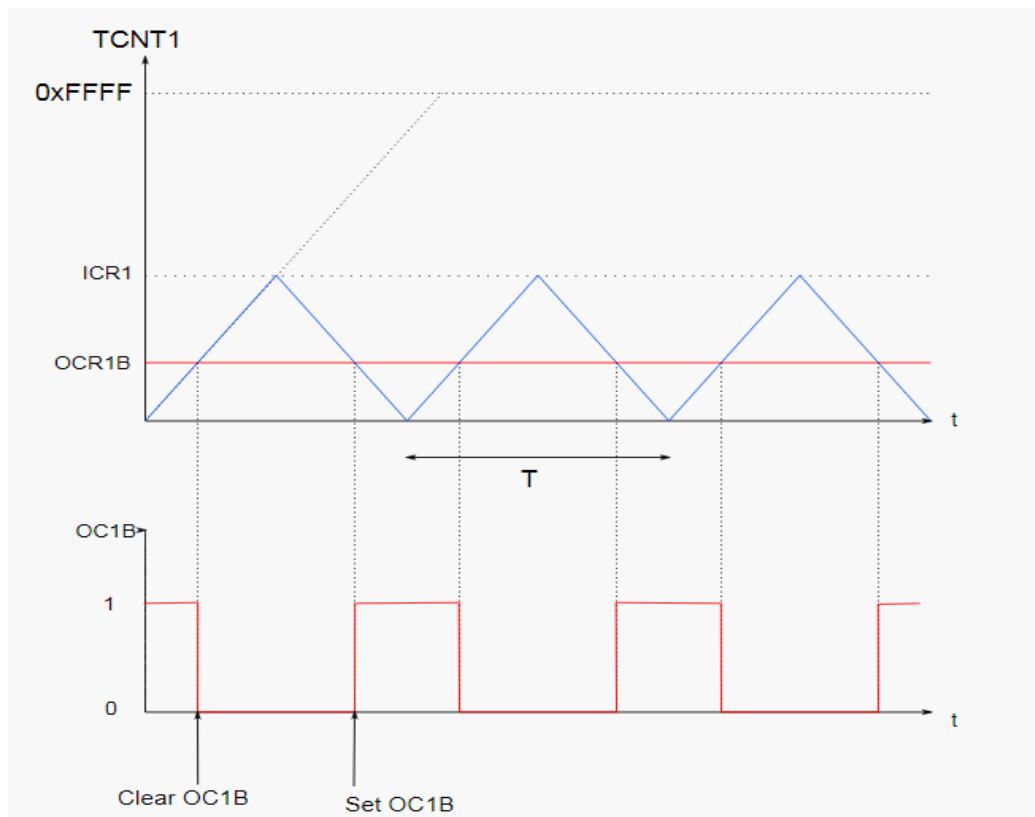


Figure3.1 Diagramme de génération de signal PWM

Finalement, la variation de ces valeurs est assurée par des potentiomètres associés à des entrées analogiques (pour notre travail, ces sont A0 et A3) et la conversion numérique est faite par l'ADC caché de la macro-fonction analogRead. Une correspondance entre les valeurs acquises et les valeurs renvoyées aux registres manipulés est nécessaire afin de garantir la compatibilité. La limitation des intervalles de nos paramètres est assurée par des simples boucles conditionnelles.

3.2 Commande de gradateur et de redresseur, en monophasé et en triphasé

La commande de ces deux convertisseurs électroniques est identiquement sauf que la fonctionnalité souhaitée change en fonction du montage.

3.2.1 Détection de la période de signal de synchronisation

La première problématique de cette partie de travail est de détecter un signal carré alternatif synchrone avec le secteur afin de coordonner les signaux de commutation des interrupteurs.

Comme on sait que la fréquence de secteur varie légèrement en fonction de la consommation de la charge liée au réseau (on peut sentir cette variation entre les différentes périodes de jour et nuit), il n'est pas question de se comporter comme si la période de signal de synchronisation est constante dans notre synthèse.

L'idée initiale de notre travail était de détecter les passages par zéro de ce signal par la conception d'une PLL numérique.

En effet le PLL (boucle de verrouillage à phase) correspond à un système asservi à retour unitaire. Elle est constituée par les éléments suivants :

- un comparateur de phase (CDP)
- un filtre passe-bas (FPB)
- un oscillateur contrôlé en tension (VCO).

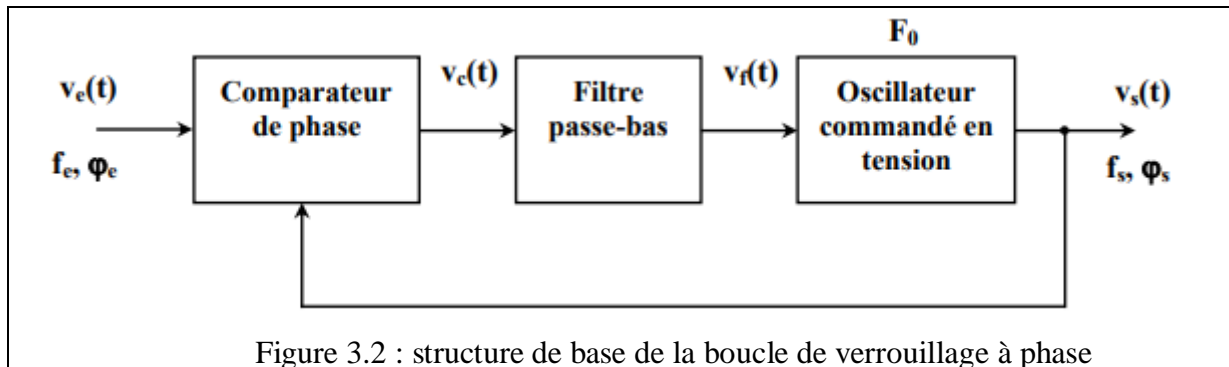


Figure 3.2 : structure de base de la boucle de verrouillage à phase

L'objet d'un tel dispositif est de synchroniser le signal d'un oscillateur modulable en fréquence avec un signal de référence (entrée), le synchronisme étant assuré par un asservissement de la fréquence des signaux. Une fois la boucle est verrouillée ou accrochée, la fréquence f_e d'entrée peut varier dans la plage de verrouillage sans que cette boucle ne décroche et on a toujours $f_s = f_e$

Dans le système électrique la PLL est généralement utilisée pour l'interconnexion d'une source d'énergie au réseau électrique.

Dans notre cas on a besoin de détecter le front montant d'un signal carré synchronisé et élaboré à partir de réseau électrique. Alors que le réseau est généralement bruité et sa fréquence n'est pas toujours la même et dépend de la consommation en électricité.

Durant la phase de bibliographie de notre projet, on a tenu compte qu'une telle solution est complexe techniquement (il demande en plus de développer un code au μC , la conception d'une carte analogique complémentaire).

Après réflexion et discussion avec notre encadrant, on a décidé d'implanter une solution moins compliquée mais toujours efficace, l'idée était de faire la moyenne sur 10 périodes et de synchroniser à la 11^{ème}. Il est inutile de détecter le front descendant pour l'alternance négative, mais connaissant la 11^{ème} période, on déduit chaque de mi-période.

Le résultat était qu'on a un retour d'information sur la fréquence de signal de synchronisation et le système de commande réagit à toute variation de manière progressive.

On a pu même pousser de plus notre travail pour qu'il soit fonctionnel dans une marge de fréquence allant de 30Hz jusqu'à 100Hz (notre projet peut être toujours exploitable même en l'implantant aux pays américains dont la fréquence de secteur est 60Hz).

Techniquement parlant, pour aboutir à ce résultat, on a mélangé deux fonctionnalités de notre μC , à savoir les interruptions externes et les timers.

En effet, tout d'abord on a déclaré une routine d'interruption externe qu'on a appelé `zero_cross_detect`, elle se déclenche lors des fronts montants du signal de synchronisation pour réinitialiser un ensemble des variables qu'on va les expliquer à fur et mesure, entre autres celle liée à la détection de la période qui est la variable compteur k qui s'initialise à 0 à chaque front montant puis elle reprend sa incrémentation pas à pas.

L'instruction de cette déclaration est : `attachInterrupt(0, zero_cross_detect, RISING);`

Et donc à priori, si on trouve une manière de fixer la durée de chaque incrémentation du compteur k , on aboutira directement à la période de signal par une simple multiplication : $\text{période} = k \cdot \text{durée}$.

Et puisqu'on parle de fixer la durée d'un événement temporel, c'est sans doute le domaine d'opération des timers, et plus précisément leurs interruptions.

Et donc, à partir de ce niveau une deuxième interruption prend le relief, qu'on a appelé `dim_check`. Tout d'abord on a initialisé notre timer à la durée de temps qu'on veut qu'à chaque fois il lance l'interruption, on l'appelle `TimeStep` et on l'a fixé à 17 us (on expliquera le choix de cette valeur plus tard).

Les instructions de déclaration de cette interruption de `Timer1` sont:

```
Timer1.initialize(TimeStep);  
Timer1.attachInterrupt(dim_check, TimeStep);
```

A chaque fois que cette interruption se déclenche, ce qui correspond au passage d'une durée de `TimeStep`, sa routine fait elle-même incrémenter le compteur k .

Ce processus continue à se répéter périodiquement jusqu'à que le front montant suivant de signal de synchronisation est détecté par l'interruption `zero_cross_detect` qu'elle lance sa routine et cette fois elle exécute l'opération de multiplication qui calcule la période avant de mettre à zéro de nouveau le compteur k . Cette portion du code illustre cette synthèse :

```
void zero_cross_detect() {  
    T=TimeStep*k;  
    k=0;
```

On a parvenu jusqu'à maintenant à détecter la période de signal de synchronisation de façon instantanée, or on a décidé de faire une moyenne afin de dépasser les défauts de fréquence de réseau.

Le reste de développement de cette solution prend place au niveau du programme principal, qui se sert de la variable volatile T résultante de l'interruption `zero_cross_detect` pour l'implanter dans un tableau de taille 10 à remplir en FIFO (First In First Out).

Chaque fois qu'une nouvelle période prend sa place dans le tableau on calcule la moyenne sur ces 10 valeurs, puis elle sera décalée case par case jusqu'à ce qu'elle soit éliminée du tableau.

La valeur moyenne est notée T_{moy} et c'est elle qui va continuer à s'exécuter dans le reste du programme pour générer les signaux de commutation des interrupteurs.

La partie du code qui figure ce travail est la suivante :

```
void loop() {  
    Per[9]=T;  
    Tt=Per[9];  
    For (n = 0 ; n < 9 ; n++)  
    {  
        Tt=Tt+Per[n];  
        Per[n]=Per[n+1];  
    }  
    Tmoy=floor(Tt/10);
```

Remarque : on a choisi de développer toute notre analyse autour des fronts montants du signal synchrone au secteur en se méfiant que ce dernier peut fournir un signal déformé dans le sens que les deux alternances peuvent ne pas être symétriques et c'est dû à l'ensemble des charges existant entre notre lieu de test et le centrale qui fournit un signal parfait.

3.2.2 Génération des signaux de commutation des interrupteurs

La deuxième problématique posée pour la commande de gradateur et de redresseur est la génération des signaux de commutation.

Les maquettes où on va implanter notre système de commande font intervenir les thyristors comme des interrupteurs de commutation et donc pour commander un seul convertisseur électronique en mode monophasé on emploie deux de ces composants donc on doit générer deux signaux de commutation à partir du signal de synchronisation.

En effet la génération de ces signaux prend en considération le signe des alternances de signal d'entrée (positive ou négative) d'une part et d'autre la valeur du retard à l'amortissage, qui est elle-même une variable modifiable par un potentiomètre.

En fait, l'interruption `zero_cross_detect`, encore une fois, détecte chaque front montant de signal d'entrée, et comme on a deux signaux de commutation à générer :

- le premier est synchrone avec l'alternance positive donc il commence à décaler le retard de l'amortissage dès la détection du front montant

- le deuxième synchrone avec l'alternance négative, donc il attend que la demi-période de signal d'entrée passe (c'est faisable comme on a déjà pu déterminer la valeur de cette période) et décale lui-aussi le retard d'amorçage avant de créer le deuxième signal de commutation.

Comme le retard de l'amorçage peut se traduire par une durée qu'on doit attendre avant de déclencher des actions, on a recourt, de même, au timers et plus précisément à leurs interruptions.

Puisqu'une même interruption peut déclencher plusieurs actions dans sa routine à condition de rester toujours simple pour s'exécuter brièvement, on s'est servi de la même interruption de `Timer1` déjà utilisée, `dim_check`, et on a ajouté à sa routine les instructions suivantes :

On sait très bien que cette interruption s'exécute tout un intervalle de temps défini `TimeStep`, donc on peut traduire le retard à l'amorçage en une valeur en la divisant par cette durée et en se servant toujours d'un compteur (`i` et `j` dans notre code pour chacun des signaux de commutation).

Cette construction de signal se fait par un changement de signe d'une variable booléenne `'the'` (de `False` elle devient `True`). Cette dernière déclenche une boucle conditionnelle (`if`) qui construit le signal à l'instant désirée (après le passage du retard d'amortissage) et de la façon désirée (la modulation dont on va parler tout de suite).

Tous les variables qui ont contribué à la construction de ce signal s'initialisent par la routine de l'interruption `zero_cross_detect` dans le but que ce cycle se répète à chaque front montant et donc on assure la continuité de signal.

D'une autre part, comme on a mentionné le retard de l'amortissage est acquis de la variation d'un potentiomètre mais en même temps il doit prendre en considération la variation de la fréquence de signal d'entrée.

Face à cette problématique, ce qu'on a trouvé comme solution c'est de prendre cette valeur numérique résultante de la conversion analogique-numérique de l'entrée de potentiomètre et de lui associer un intervalle de valeurs dépendant de la valeur de la fréquence.

3.2.3 Modulation des signaux de commutation

Le troisième point mentionné dans le cahier des charges de notre projet est de faire moduler le signal de commutation lors d'amortissage des thyristors.

En effet, comme on doit assurer une isolation galvanique entre la partie de commande et celle de puissance de notre système, le choix technique adapté lors de la réalisation des maquettes était les transformateurs hautes fréquences.

On sait très bien, d'après la relation de Boucherot, qu'en augmentant la fréquence, on peut diminuer les dimensions du transformateur sans risque de se saturer lors de son fonctionnement et la conséquence directe : c'est qu'on gagne sur le coût de fabrication.

L'exigence de notre projet en terme de fréquence était de la fixer au alentour de 30KHz, comme les anciennes maquettes analogiques.

Sur le plan pratique, 31.25 KHz de fréquence donne comme période de modulation $32\mu s$ et donc un basculement de signal de commutation tous les $16\mu s$.

Au début de l'analyse de cette partie de rapport on a bien mentionné que la durée fixée pour le déclenchement de l'interruption `dim_check` est `TimeStep = 16us`, et donc cette valeur n'est pas choisie arbitrairement mais bien dimensionnée pour se servir de cette interruption une troisième fois pour générer la modulation des signaux de commutation.

Pratiquement, à chaque fois que la routine de cette interruption est actionnée, elle vérifie si la variable booléenne `"the"` est vrai et si c'est bon, elle bascule l'état de la sortie, et pour le faire on a appelé les opérations sur les bits des GPIO ports directement pour alléger la routine le maximum possible.

3.2.4 Passage de la commande en triphasé

Pratiquement, à chaque fois que la routine de cette interruption est actionnée, elle vérifie si la variable booléenne `"the"` est vrai et si c'est bon, elle bascule l'état de la sortie, et pour le faire on a appelé les opérations sur les bits des GPIO ports directement pour alléger la routine le maximum possible.

Pour passer au triphasé, la piste est à peu près prête. Jusqu'à maintenant on a pu générer à partir d'un signal de synchronisation deux signaux de commutation, et pour passer au triphasé, il était demandé de décaler ces deux signaux une première fois par 120 degrés et une deuxième fois par 240 degrés.

Mais agir sur les signaux qu'on a déjà créés n'est pas aussi facile. Notre solution alternative était de décaler le signal de synchronisation par ces deux valeurs et réappliquer le résonnement déjà aboutit en monophasé.

A ce point on s'affronte par le premier blocage, on était juste capable de décaler le signal par 120 degrés et non pas par 240 degrés.

L'idée pour surmonter ce problème était de décaler le signal par 120 degrés et générer le signal de décalage d'après par ce même signal déjà décalé, plus simplement le décalage par 240 degrés est fait par deux décalages de 120 degrés.

A ce niveau, c'était une solution abordable de côté programmation mais elle pose un problème matériel. Chaque décalage doit se faire par une interruption externe à part car elle doit suivre un front montant différent à chaque fois. Donc on a besoin en totalité de 3 interruptions externes alors que l'arduino uno ne présente que 2.

D'autre part, développer tous ces calculs de décalages, de retard d'amortissage et de modulation pour 2 signaux de synchronisation et six signaux de commutation par une même routine d'interruption de `Timer1`, contredit le fait qu'elle doit être simple et légère. Donc on a besoin de plus de timers hors l'arduino uno ne présente qu'un seul `Timer1`.

Après une petite recherche dans les autres gammes de cartes d'arduino, on a constaté que l'arduino mega, comme indique son nom, supporte plus de ressources matérielles entre autres 4 interruptions externes et 4 timers 16 bits, ce qui explique notre migration vers ce μc pour résoudre le problème de commande pour les convertisseurs triphasés.

3.2.5 Affichage

L’afficheur LCD peut être directement relié à la carte arduino, or dans ce cas, il nécessite la mise en œuvre de 11 entrées/sorties.

Grace à l’utilisation d’un module qui contient un circuit intégrés spécifique, appelée « extenseur de port pour bus I2C » on peut commander le même afficheur en ne réclamant que deux ports d’entrées/sorties. Le tableau suivant montre le Pin I2C spécifique d’arduino.

Cartes arduino	Broches I2C
ARDUINO UNO	A4(SDA), A5 (SCL)
ARDUINO MEGA	20 (SDA), 20 (SCL)

Tableau 3.2: les pins I2C arduino

Cette communication fonctionne dans un mode particulier. Les échanges sont démarrés par un périphérique « maître » Celui-ci envoie un signal de départ, puis l’adresse du récepteur, puis le registre, enfin les données et termine avec une séquence de stop. Tous ces signaux se suivent en série sur la ligne de données. Le récepteur signale qu’il a bien reçu le message en envoyant un signal d’acquiescement (ACK).

3.3 Résultats expérimentaux :

3.3.1 Commande de gradateur et redresseur monophasé

La figure 3.3 montre le circuit expérimental de commande de gradateur et de redresseur en monophasé

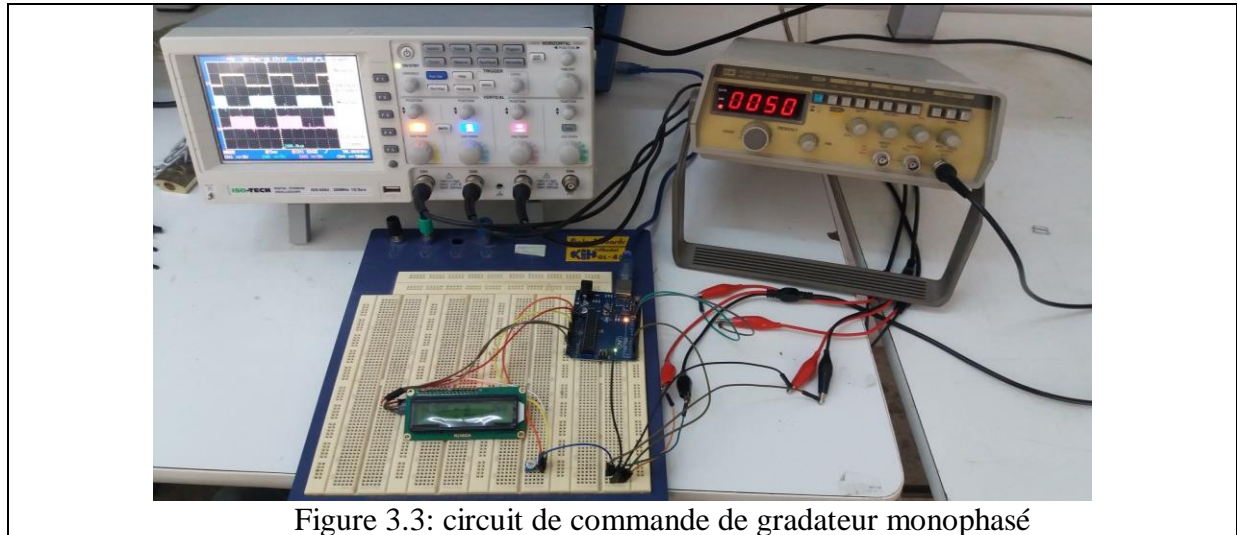


Figure 3.3: circuit de commande de gradateur monophasé

La figure ci-dessous présente la sortie de commande de gradateur pour différents valeurs d'angle d'amorçage α de thyristor ($\alpha=0^\circ$, $\alpha=91^\circ$, $\alpha=180^\circ$) pour une fréquence égale à 50 Hz

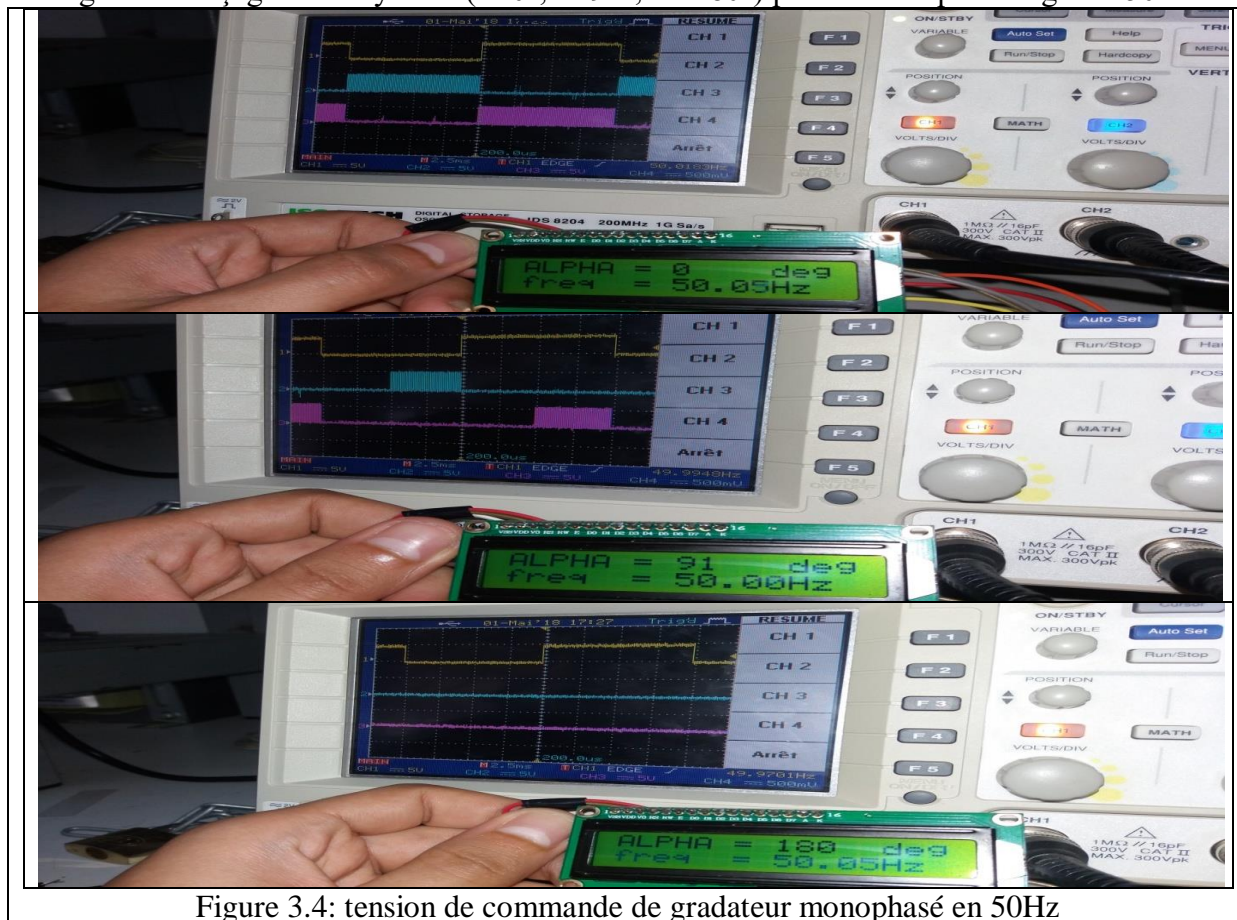


Figure 3.4: tension de commande de gradateur monophasé en 50Hz

La figure suivante présente la sortie de commande de gradateur pour différentes valeurs d'angle d'amorçage α de thyristor ($\alpha=0^\circ$, $\alpha=91^\circ$, $\alpha=180^\circ$) pour une fréquence égale à 60 Hz

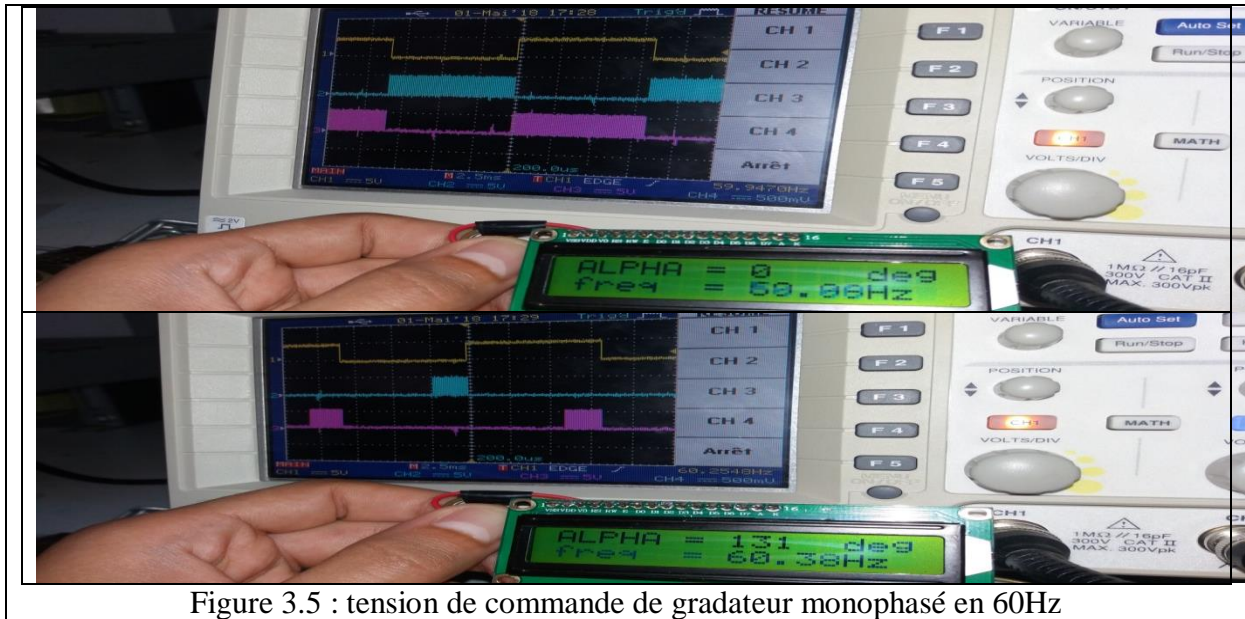


Figure 3.5 : tension de commande de gradateur monophasé en 60Hz

Ce signal de commandé est modulé à 30kHz pour éliminer les pertes et pour réduire de transformateur d'isolation. Montre le résultat de modulation de ce signal.

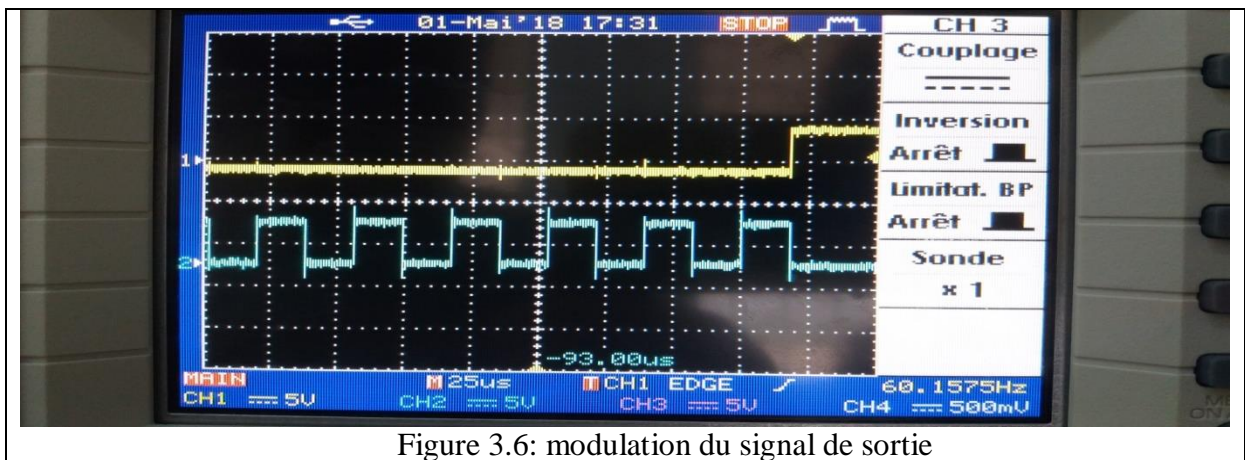


Figure 3.6: modulation du signal de sortie

3.3.2 Commande de gradateur et redresseur en triphasé :

Le figure suivant présente la sortie de commande de la première demi période de gradateur triphasé pour différents valeur d'angle d'amorçage α de thyristor ($\alpha=47^\circ$, $\alpha=90^\circ$, $\alpha=163^\circ$) pour une fréquence égale a 50 Hz



Figure 3.7 : tension de commande de gradateur triphasé en 50Hz

La figure suivante présente la sortie de commande de la deuxième demi période de gradateur triphasé pour différents valeur de fréquence ($f=50\text{Hz}$, $f=60\text{Hz}$)

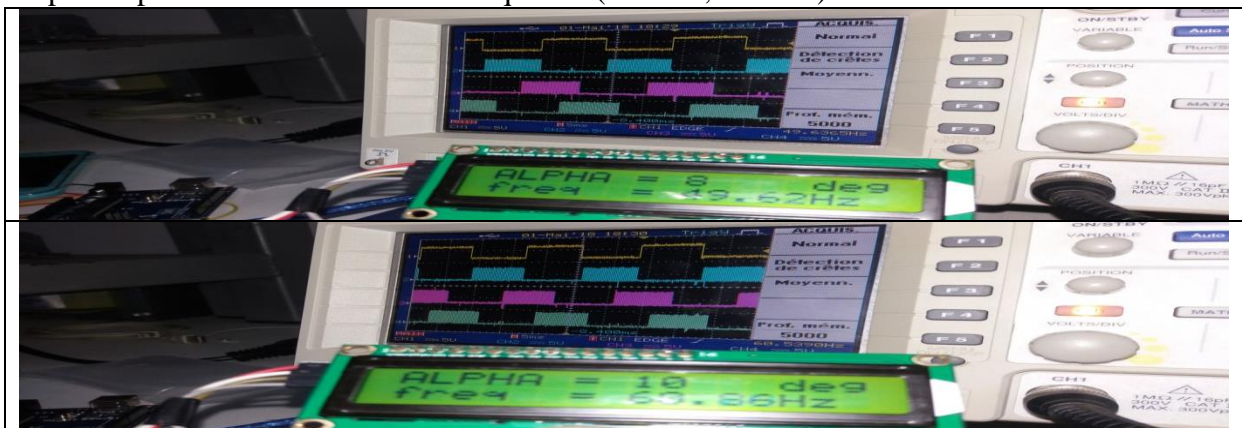


Figure 3.8: tension de commande de gradateur triphasé en 50Hz et en 60Hz

3.3.3 Commande du hacheur élévateur :

La figure suivante montre le circuit expérimentale de commande hacheur élévateur en monophasé

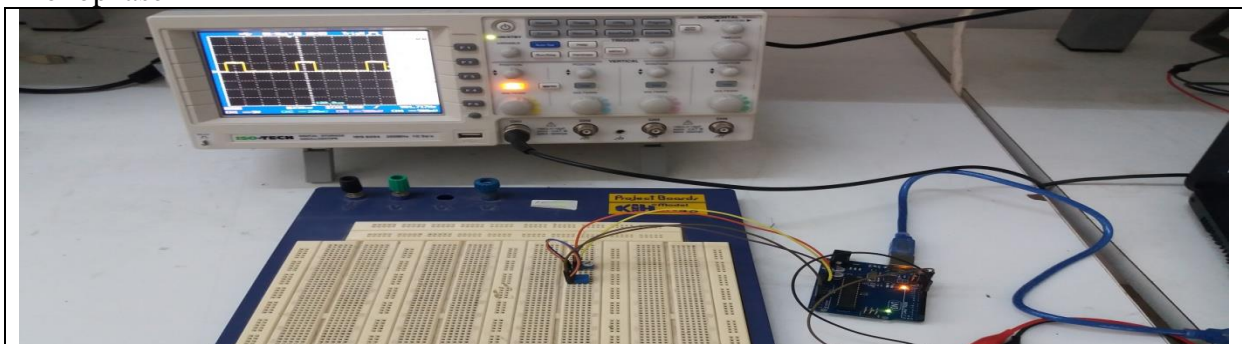


Figure 3.9 : circuit de commande de hacheur élévateur

La figure suivante présente la sortie de commande de hacheur élévateur pour différents valeur de rapport cyclique α ($\alpha=0.25$, $\alpha=0.5$, $\alpha=0.89$) pour une fréquence fixe

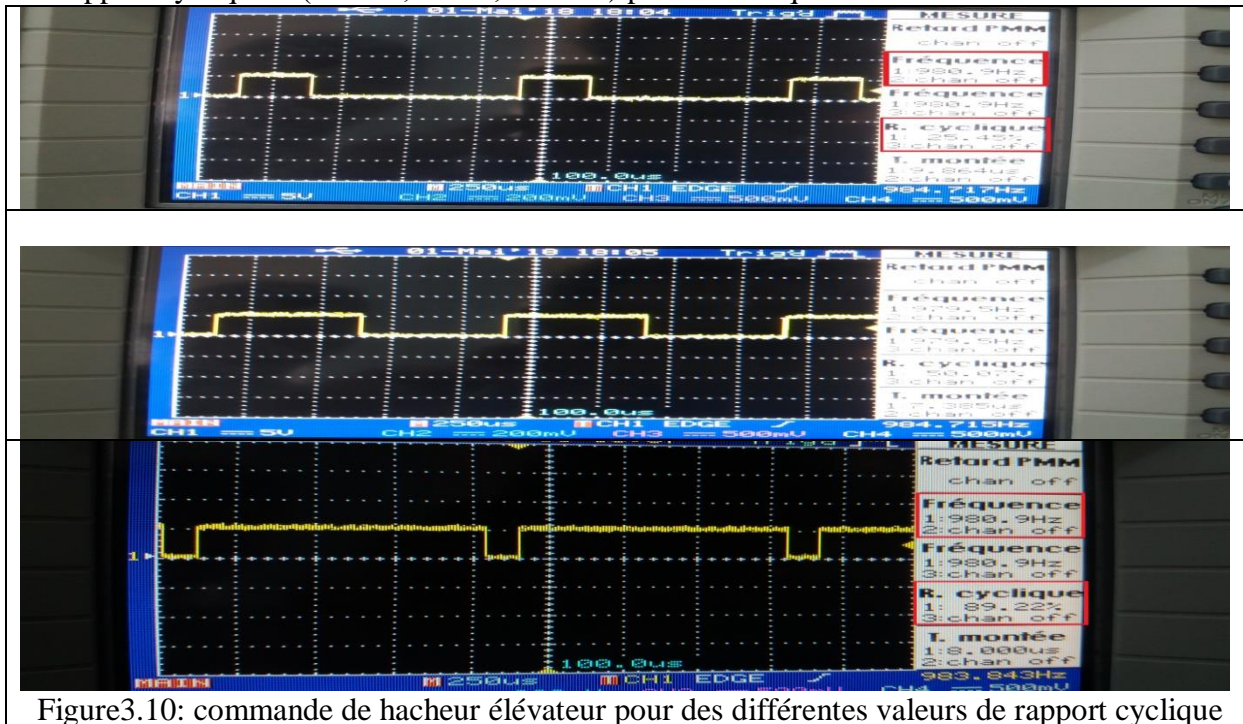


Figure3.10: commande de hacheur élévateur pour des différentes valeurs de rapport cyclique

La figure suivante présente la sortie de commande du hacheur élévateur pour différentes valeur de fréquence ($f=980\text{Hz}$, $f=13\text{kHz}$, $f=33\text{kHz}$) et pour une valeur de rapport cyclique fixe égale a 0.5

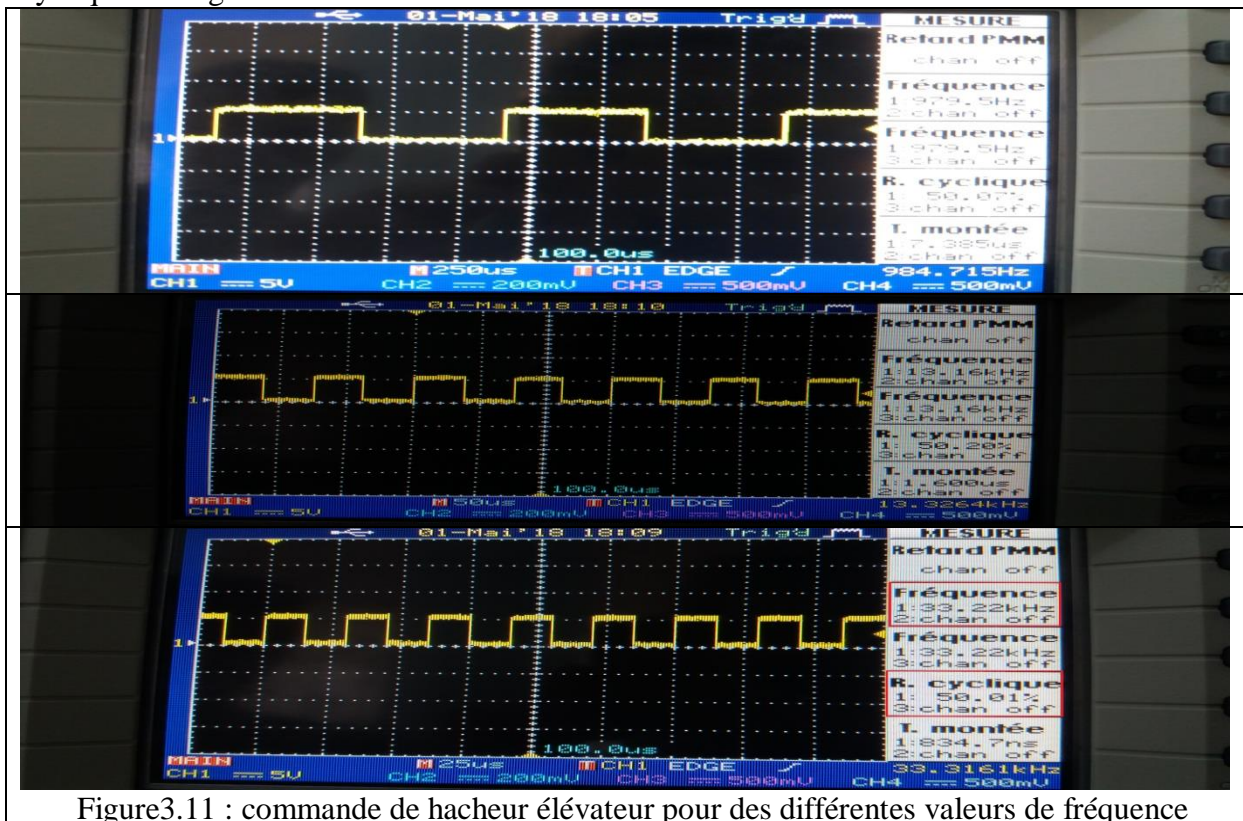


Figure3.11 : commande de hacheur élévateur pour des différentes valeurs de fréquence

CONCLUSION GENERALE :

Dans le cadre de préparation de notre projet de fin d'année, ce travail vise à développer une approche numérique pour la commande de hacheur élévateur, le redresseur monophasé, le gradateur monophasé et triphasé.

Le premier chapitre a été dédié à la présentation du principe de fonctionnement de différents types de convertisseurs statique et leurs caractéristiques.

Dans le deuxième chapitre nous avons présenté carte Arduino et les instructions qu'on a utilisées.

Enfin nous avons expliqué la réalisation de notre commande.

Nous espérons que ce dispositif sera utilisé et amélioré jusqu'à l'obtention d'un produit final.

Perspectives

- ✓ Augmenté la fréquence de modulation de signal de commande des thyristors à quelques centaines de kHz afin de diminuer encore la taille des transformateurs d'isolation galvanique.
- ✓ Réaliser la boucle de verrouillage de phase pour la synchronisation de tension secteur.
- ✓ Implémentation de notre commande dans les maquettes de différents convertisseurs.
- ✓ Aborder l'aspect régulation numérique.

Bibliographie

- [1] Jacques LAROCHE , **Electronique de Puissance Convertisseur** , Dunod ,2005.
- [2] Alain HERBERT, Claude NAUDET et Michel PINARD **Machines Electriques, Electronique de Puissance** , DUNOD,1997
- [3] Gy.Chateiger, Michel Boês, Daniel Bouix, Jaque Vaillant **Manuel de Génie Electrique** , DUNOD,2006
- [4] Jack J Purdum, **Beginning C for Arduino: Learn C Programming for the Arduino**, Apress, 2013.
- [5] Elliot Williams, **Make, AVR Programming**, Shroff Publishers & Distr, 2014.
- [6] John Hughes, **Arduino: A Technical Reference: A Handbook for Technicians, Engineers, and Makers**, O'Reilly Media, Incorporated, 2016.

Annexe 1 : CODE DU HACHEUR BOOST

```

int a =0;
int b =0;
long int time;
float x;
unsigned long pwm;
void setup()
{
// Serial.begin(9600);
pinMode(10, OUTPUT);
TCCR1A = _BV(COM1A1) | _BV(COM1B1) ;
TCCR1B = _BV(WGM13) | _BV(CS11);
time=millis();
}
void loop(){
if(millis() >time+500) {
a = analogRead(A3);
if ( a > 30 )
{ICR1 = a; }
else if (a < 30) { ICR1 = 30; }
b = analogRead(A0);
if ((b > 105) && (b < 950))
{pwm = b;}
else if (b < 105) {pwm = 105;}
else if ( b> 950) {pwm=950;}
x = float(pwm);
x = x * ICR1;
x = x / 1023;
OCR1B = int(x);
time=millis();
}
// Serial.println(a);
}

```

Annexe 2 : CODE DE GRADATEUR MONOPHASE

```

/*single phase dimmer Control
Updated by Ahmed LIMEM_Tarek Ben Dahou_2AGE2
Pins: analog-In()<=A0, Interrupt<=pin2, digital-Out<=pin10-11
pins display LCD-I2C: SDA:A4 SCL:A5
*/
#include <TimerOne.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x3F, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE); //i2c pins
volatile int i=0; // Variable to use as a counter volatile as it is
in an interrupt
volatile int j=0;
volatile int l=0;
volatile int p=0;
volatile int k=0;
volatile boolean zero_cross=0; // Boolean to store a "switch" to tell us if we
have crossed zero
volatile boolean zero_cross1=0;
volatile boolean zero_cross3=0;
volatile boolean zero_cross4=0;
volatile boolean the1=0; //Boolean variable to detect the passage of
amortization delay
volatile boolean the2=0;
unsigned long dim = 0; //the limit of the counter to trigger the amortization of
the thyristor
unsigned long dim_per =0;
int a =0;
int alpha; //thyristor amortization delay in degrees
float freq;
float freqx;
int TimeStep = 17; // This is the delay-per-step in microseconds.
long int tm;
long int time;
unsigned long T;
unsigned long Tt;
unsigned long Tmoy;
unsigned long Per[10];
//unsigned long Per[100]; for systems with high inertia so the frequency varies
slowly within 2 seconds for 50 Hz frequency
int n=0;
// int Xt=21; //the maximum number of toggling signal per cycle de control of
thyristor
// Xt=(timeStep/te)*(2*dim_per)
//te is execution time of 1 toggle in microseconds
// execution frequency is 30KHz when we use arduino
instructions
// execution frequency is 170KHz when we use direct port
manipulation with PORTx registers

```



```

void setup(){
  lcd.begin(16,2);
  lcd.backlight();
  // Set the thyristor pin as output
  DDRB |= 1 << DDB2; //I/O port PB2
  DDRB |= 1 << DDB3; //I/O port PB3
  attachInterrupt(0, zero_cross_detect, RISING ); // Attach an Interrupt to Pin 2
  (interrupt 0) for Zero Cross Detection
  Timer1.initialize(TimeStep); // Initialize TimerOne library for the freq we
  need
  Timer1.attachInterrupt(dim_check,
  TimeStep);
}
void zero_cross_detect() {
  T=TimeStep*k;
  k=0;
  zero_cross = true; // set the boolean to true to tell our dimming function
  that a zero cross has occurred
  zero_cross1 = true;
  the1 = false;
  i=0;
  l=0;
  PORTB &= ~(1<<PORTB2); // turn off thyristor
  //set the Arduino pin Low
  zero_cross3 = true; // set the boolean to true to tell our dimming function
  that a zero cross has occurred
  zero_cross4 = true;
  the2 = false;
  j=0;
  p=0;
  PORTB &= ~(1<<PORTB3); // turn off thyristor
  //set the Arduino pin Low
}
void dim_check() {
  k++;
  if(zero_cross == true) {
    if(i>=dim) {
      the1 = true;
      i=0; // reset time step counter
      zero_cross = false; //reset zero cross detection
    }
    else {
      i++; // increment time step counter
    }
  }
  if(zero_cross1 == true) {
    if(l>=dim_per){
      the1 = false;
      l=0; // reset time step counter
      zero_cross = false;
    }
  }
}

```

```

}
else
{ l++; }
}
// Toggle on the thyristor at the appropriate time
if(the1 == true) {
PORTB =PORTB^_BV(2);
}else{PORTB &= ~(1<<PORTB2);}
if(zero_cross3 == true) {
if(j>=dim+dim_per) {
the2 = true;
j=0; // reset time step counter
zero_cross3 = false; //reset zero cross detection
}
else {
j++; // increment time step counter
}
}
if(zero_cross4 == true) {
if(p>=2*dim_per){
the2 = false;
p=0; // reset time step counter
zero_cross4 = false;
}
else
{p++;}
}
// Toggle on the thyristor at the appropriate time
if(the2 == true) {
PORTB =PORTB^_BV(3);
}else{PORTB &= ~(1<<PORTB3); }
}
void loop() {
Per[9]=T;
Tt=Per[9];
for (n = 0 ; n < 9 ; n++)
{
Tt=Tt+Per[n];
Per[n]=Per[n+1];
}
Tmoy=floor(Tt/10);
dim_per=floor(Tmoy/(2*TimeStep));
if(millis() >time+50){
a = analogRead(A0);
time=millis();
}
dim=map(a, 0, 1023, 1, dim_per);
alpha=floor(dim*180/dim_per);
freqx=100000000/T;
freq=freqx/100 ;

```

```
if(millis() >tm+150){ //Delay used to give a dinamic effect
lcd.clear();
lcd.setCursor(8,0);
lcd.print(alpha);
lcd.setCursor(0,0); //we start writing from the first row first column
lcd.print("ALPHA ="); //16 characters per line
lcd.setCursor(13,0);
lcd.print("deg");
lcd.setCursor(13,1);
lcd.print("Hz");
lcd.setCursor(8,1);
lcd.print(freq);
lcd.setCursor(0,1);
lcd.print("freq =");
tm=millis();}
}
```

Annexe 3 : CODE DE GRADATEUR TRIPHASÉ

```

/*Dimmer Control
Updated by Ahmed LIMEM_Tarek Ben Dahou_2AGE2
to refer: dim<=limitRise, dim_per<=limitWave, Per<=period, dim_check()<=checking
periodically
Pins: analog-In()<=A0, Interrupts-pins<=18(pin9-Ty); 3(pin12-Tx); 2(Square Wave),
digital-Out-pins<=X(10,11); Y(7,8); Z(4,5)
Requirements: potentiometer-1K
to improve: toggle output signals at least 30 khz - developpe PLL
method instead of the average methode - build file mono.h
and call it in main to simplify the code
Changed zero-crossing detection to look for RISING edge only rather
than rising and falling. (originally it was chopping the positive and negative
half
of the AC square wave form).
Also changed the check() to turn on the Thyristor every half
of the AC square wave form, leaving it on until the zero_cross_detect() turn's
it off.
*/

```

```

#include <TimerOne.h> // Available from http://www.arduino.
cc/playground/Code/Timer1
#include <TimerThree.h>
#include <TimerFive.h>
#include <TimerFour.h>
#include <TimerOne.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x3F, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE); //i2c pins
volatile int k=0;
volatile int i0x=0; // Variable to use as a counter volatile as it
is in an interrupt
volatile int l0x=0;
volatile int i1x=0;
volatile int l1x=0;
volatile int ex=0;
volatile int fx=0;
volatile int i0y=0;
volatile int l0y=0;
volatile int i1y=0;
volatile int l1y=0;
volatile int ey=0;
volatile int fy=0;
volatile int i0z=0;
volatile int l0z=0;
volatile int i1z=0;
volatile int l1z=0;
volatile boolean zero_cross1x=0; // Boolean to store a "switch" to tell us if

```

```

we have crossed zero
volatile boolean zero_cross12x=0;
volatile boolean zero_cross21x=0;
volatile boolean zero_cross22x=0;
volatile boolean zero_crossT0x=0;
volatile boolean zero_crossT1x=0;
volatile boolean zero_cross11y=0;
volatile boolean zero_cross12y=0;
volatile boolean zero_cross21y=0;
volatile boolean zero_cross22y=0;
volatile boolean zero_crossT0y=0;
volatile boolean zero_crossT1y=0;
volatile boolean zero_cross11z=0;
volatile boolean zero_cross12z=0;
volatile boolean zero_cross21z=0;
volatile boolean zero_cross22z=0;
volatile boolean th11=0;
volatile boolean th12=0;
volatile boolean th21=0;
volatile boolean th22=0;
volatile boolean th31=0;
volatile boolean th32=0;
unsigned long dim = 0; // Dimming level (0-dim_per) 0 = on, dim_per = 0ff
unsigned long dimT = 0;
unsigned long dim_per =0;
unsigned long dim_perT =0;
int a =0;
int timeStep_toggle = 30;
int timeStep = 155; //145 This is the delay-per-step in microseconds.
// It is calculated based on the number of steps you want.
// Realize that there are 2 zerocrossing per cycle.
//This means to calculate timeStep: divide the length of one full half-wave of
the power
// cycle (in microseconds) by the number of steps.
int alpha; //thyristor amortization delay in degrees
float freqx , freq ;
long int tm;
long int time;
unsigned long T;
unsigned long Tt;
unsigned long Tmoy;
unsigned long Per[10];
int n=0;
void setup() { // Begin setup
lcd.begin(16,2);
lcd.backlight();
// Set the pins as output
DDRB |= 1 << DDB4; //I/O port PB4 // Output to Thyristor
DDRB |= 1 << DDB5; //I/O port PB5
DDRB |= 1 << DDB6; //I/O port PB6 // Output to external interrupt

```

```

DDRH |= 1 << DDH4; //I/O port PH4
DDRH |= 1 << DDH5; //I/O port PH5
DDRH |= 1 << DDH6; //I/O port PH6
DDRG |= 1 << DDG5; //I/O port PG5
DDRE |= 1 << DDE3; //I/O port PE3
attachInterrupt(0, zero_cross_detect, RISING ); // Attach an Interrupt to Pin 2
(interupt 0) for Zero Cross Detection
attachInterrupt(1, zero_cross_detect3, RISING ); // Attach an Interrupt (Int1)
to Pin 3
attachInterrupt(5, zero_cross_detect5, RISING ); // Attach an Interrupt (Int5)
to Pin 18
/**
Timer1.initialize(timeStep); // Initialize TimerOne library for the freq we
need
Timer3.initialize(timeStep);
Timer4.initialize(timeStep_toggle);
Timer5.initialize(timeStep_toggle);
/**
Timer1.attachInterrupt(dim_check, timeStep);
Timer3.attachInterrupt(dim_check3, timeStep);
Timer4.attachInterrupt(dim_check5, timeStep_toggle);
Timer5.attachInterrupt(dim_check4, timeStep_toggle);
/**
// Use the TimerOne Library to attach an interrupt
// to the function we use to check to see if it is
// the right time to fire the Thyristor. This function
// will now run every freqStep in
microseconds.
}
void zero_cross_detect() {
T=timeStep*k;
k=0;
zero_cross11x = true; // set the boolean to true to tell our dimming function
that a zero cross has occurred
zero_cross12x = true;
zero_cross21x = true;
zero_cross22x = true;
zero_crossT0x = true;
zero_crossT1x = true;
th11=false;
th12=false;
i0x=0;
l0x=0;
i1x=0;
l1x=0;
ex=0;
fx=0;
// turn off Thyristor
PORTB &= ~(1<<PORTB4);
PORTB &= ~(1<<PORTB5);

```

```

PORTB &= ~(1<<PORTB6);
//set the Arduino pin Low
}
void zero_cross_detect3() {
zero_cross11y = true; // set the boolean to true to tell our dimming
function that a zero cross has occurred
zero_cross12y = true;
zero_cross21y = true;
zero_cross22y = true;
zero_crossT0y = true;
zero_crossT1y = true;
th21=false;
th22=false;
i0y=0;
l0y=0;
i1y=0;
l1y=0;
ey=0;
fy=0;
// turn off Thyristor
PORTH &= ~(1<<PORTH4);
PORTH &= ~(1<<PORTH5);
PORTH &= ~(1<<PORTH6);
//set the Arduino pin Low
}
void zero_cross_detect5() {
zero_cross11z = true; // set the boolean to true to tell our dimming
function that a zero cross has occurred
zero_cross12z = true;
zero_cross21z = true;
zero_cross22z = true;
th31=false;
th32=false;
i0z=0;
l0z=0;
i1z=0;
l1z=0;
// turn off Thyristor
PORTG &= ~(1<<PORTG5);
PORTE &= ~(1<<PORTE3);
//set the Arduino pin Low
}
void dim_check() {
k++;
if(zero_cross11x == true) {
if(i0x>=dim) { // Thyristor On propogation delay
th11=true;
i0x=0; // reset time step counter
zero_cross11x = false; //reset zero cross detection
}
}

```

```

else {
i0x++; // increment time step counter
}
}
if(zero_cross12x == true) {
if(i0x>=dim_per){
th11=false;
i0x=0; // reset time step counter
zero_cross12x = false;
}
else
{ i0x++;}
}
if(zero_cross21x == true) {
if(i1x>=dim+dim_per) { // Thyristor On propogation delay
th12=true;
i1x=0; // reset time step counter
zero_cross21x = false; //reset zero cross detection
}
else {
i1x++; // increment time step counter
}
}
if(zero_cross22x == true) {
if(i1x>=2*dim_per){
th12=false;
i1x=0; // reset time step counter
zero_cross22x = false;
}
else
{ i1x++;}
}
if(zero_crossT0x == true) {
if(ex>=dimT) {
PORTB |= 1 << PORTB6; //set the Arduino pin High
ex=0;
zero_crossT0x = false;
}
else {
ex++;
}
}
if(zero_crossT1x == true) {
if(fx>=dim_perT){
PORTB &= ~(1<<PORTB6); //set the Arduino pin Low
fx=0;
zero_crossT1x = false;
}
else
{ fx++;}
}

```



```

}
if(zero_cross11y == true) {
if(i0y>=dim) { // Thyristor On propogation delay
th21=true;
i0y=0; // reset time step counter
zero_cross11y = false; //reset zero cross detection
}
else {
i0y++; // increment time step counter
}
}
if(zero_cross12y == true) {
if(i0y>=dim_per){
th21=false;
i0y=0; // reset time step counter
zero_cross12y = false;
}
else
{ i0y++;}
}
}
void dim_check3() {
if(zero_cross21y == true) {
if(i1y>=dim+dim_per) { // Thyristor On propogation delay
th22=true;
i1y=0; // reset time step counter
zero_cross21y = false; //reset zero cross detection
}
else {
i1y++; // increment time step counter
}
}
if(zero_cross22y == true) {
if(i1y>=2*dim_per){
th22=false;
i1y=0; // reset time step counter
zero_cross22y = false;
}
else
{ i1y++;}
}
}
if(zero_crossT0y == true) {
if(ey>=dimT) {
PORTH |= 1 << PORTH6; //set the Arduino pin High
ey=0;
zero_crossT0y = false;
}
else {
ey++;
}
}
}

```

```

}
if(zero_crossT1y == true) {
if(fy>=dim_perT){
PORTH &= ~(1 << PORTH6); //set the Arduino pin Low
fy=0;
zero_crossT1y = false;
}
else
{fy++;}
}
if(zero_cross11z == true) {
if(i0z>=dim) { // Thyristor On propogation delay
th31=true;
i0z=0; // reset time step counter
zero_cross11z = false; //reset zero cross detection
}
else {
i0z++; // increment time step counter
}
}
if(zero_cross12z == true) {
if(l0z>=dim_per){
th31=false;
l0z=0; // reset time step counter
zero_cross12z = false;
}
else
{l0z++;}
}
if(zero_cross21z == true) {
if(i1z>=dim+dim_per) { // Thyristor On propogation delay
th32=true;
i1z=0; // reset time step counter
zero_cross21z = false; //reset zero cross detection
}
else {
i1z++; // increment time step counter
}
}
if(zero_cross22z == true) {
if(l1z>=2*dim_per){
th32=false;
l1z=0; // reset time step counter
zero_cross22z = false;
}
else
{l1z++;}
}
}
// Turn on the Thyristor at the appropriate time

```

```

void dim_check4() {
if(th11 == true) {
PORTB =PORTB^_BV(4);
}else{PORTB &= ~(1<<PORTB4);}
if(th21 == true) {
PORTH =PORTH^_BV(4);
}else{PORTH &= ~(1<<PORTH4);}
if(th31 == true) {
PORTG =PORTG^_BV(5);
}else{PORTG &= ~(1<<PORTG5); }
}
void dim_check5() {
if(th12 == true) {
PORTB =PORTB^_BV(5);
}else{PORTB &= ~(1<<PORTB5);}
if(th22 == true) {
PORTH =PORTH^_BV(5);
}else{PORTH &= ~(1<<PORTH5); }
if(th32 == true) {
PORTE =PORTE^_BV(3);
}else{PORTE &= ~(1<<PORTE3); }
}
void loop() {
Per[9]=T;
Tt=Per[9];
for (n = 0 ; n < 9 ; n++){
Tt=Tt+Per[n];
Per[n]=Per[n+1];
}
Tmoy=floor(Tt/10);
dim_per=floor(Tmoy/(2*timeStep));
if(millis() >time+50) {
a = analogRead(A0);
time=millis();
}
dimT=floor((120*dim_per)/180);
dim_perT=dimT+dim_per;
dim=map(a, 0, 1023, 0, dim_per);
alpha=floor(dim*180/dim_per);
freqx=100000000/T;
freq=freqx/100 ;
if(millis() >tm+150){ //Delay used to give a dinamic effect
lcd.clear();
lcd.setCursor(8,0);
lcd.print(alpha);
lcd.setCursor(0,0); //we start writing from the first row first column
lcd.print("ALPHA ="); //16 characters poer line
lcd.setCursor(13,0);
lcd.print("deg");
lcd.setCursor(13,1);

```

```
lcd.print("Hz");  
lcd.setCursor(8,1);  
lcd.print(freq);  
lcd.setCursor(0,1); //we start writing from the first row first column  
lcd.print("freq =");  
tm=millis();}  
}
```

Annexe 4 : Arduino uno pin mapping

