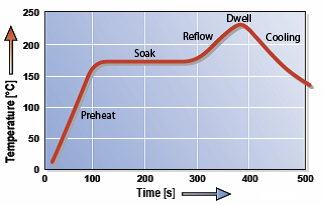


Projet d’électronique numérique

Régulation de température



Antoine Bostem Professeur : F.Triquet

Thomas Herpoel Année : 2015-2016

Table des matières

[Introduction 3](#_Toc439347777)

[OS 4](#_Toc439347778)

[Les callbacks 4](#_Toc439347779)

[La machine d’état 5](#_Toc439347780)

[Serial Peripheral Interface 7](#_Toc439347781)

[Two Wire Interface (I2C) 9](#_Toc439347782)

[Librairie I2C 9](#_Toc439347783)

[Librairie EEPROM 11](#_Toc439347784)

[Librairie RTC (Real Time Clock) 13](#_Toc439347785)

[Régulation via le PID 14](#_Toc439347786)

[Calcul du régulateur 14](#_Toc439347787)

[Code du régulateur 15](#_Toc439347788)

[Modélisation PWM-relai statique-résistance 17](#_Toc439347789)

[Conclusion 18](#_Toc439347790)

[ANNEXES 19](#_Toc439347791)

[Annexe 1 : Ordinogramme de writePage 19](#_Toc439347792)

[Annexe 2 : Ordinogramme de readPage 20](#_Toc439347793)

[Annexe 3 : Schéma du système 21](#_Toc439347794)

[Annexe 4 : script MatLab de modélisation de la puissance 22](#_Toc439347795)

[Annexe 5 : Analyse réflexive : Thomas Herpoel 23](#_Toc439347796)

[Annexe 6 : Analyse réflexive : Antoine Bostem 25](#_Toc439347797)

# Introduction

Notre projet de créer un « reflow oven » se base sur l’utilisation d’un microcontrôleur AVR atmega168 et la théorie de régulation que nous avons vue en cours. Le contrôle de la régulation se fait via le microcontrôleur au travers d’une interface LCD et une machine d’état gère les différents menus de cette interface.

Notre système permet à l’utilisateur de lancer une régulation de la température d’un four pour un palier fixe. Cette température est choisie en fonction de la température de fusion de la pâte à souder.

Le code du microcontrôleur est fait en langage C, avec l’environnement de développement AVR studio 5. Dans ce code, les parties principales sont :

* l’OS (qui contient la gestion de la machine d’état et des callback)
* la gestion du périphérique de mesure de la température via un bus SPI
* le contrôle d’une mémoire EEPROM externe et d’une RTC via un bus I2C (Two Wire Interface).
* la régulation via un PID

Ce sont ces parties qui seront expliquées dans ce rapport.

# OS

L’OS contient la gestion des callbacks et de la machine d’état. Il se compose de 6 fonctions.

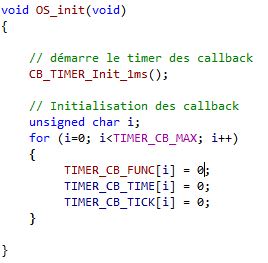
* OS\_init( ) : sert à initialiser les variables liées aux callback
* OS\_start( ) : démarre le système
* OS\_getNextState(state,stimuli) : retourne le prochain état de la machine en fonction de l’entrée sur le clavier
* OS\_addCallback(function,period) : ajoute une callback qui appelle la fonction *function* toutes les *period* millisecondes et retourne l’identificateur de cette callback
* OS\_removeCallback(CB\_ID) : supprime la callback ayant l’identificateur *CB\_ID*
* OS\_setCallbackPeriod(CB\_ID,period) : met à jour la période de la callback *CB\_ID* à la valeur *period*

La fonction la plus importante est la fonction OS\_start. En effet elle contient principalement une boucle infinie qui est le processus principal qui tourne sur le microcontrôleur.

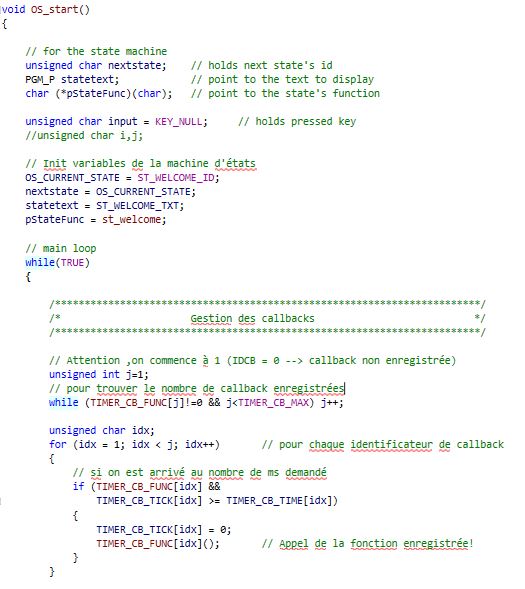
## Les callbacks

Voyons plus précisément le fonctionnement des callbacks. Une callback contient plusieurs attributs : une fonction à exécuter, un compteur qui s’incrémente toutes les millisecondes et un temps qui indique la période de répétition de la fonction à exécuter. Etant donné que le programme utilise plusieurs callbacks, 3 tableaux vont servir à regrouper ces attributs : TIMER\_CB\_FUNC, TIMER\_CB\_TIME et TIMER\_CB\_TICK. Un identifiant permet également de manipuler chaque callback dans le reste du code.

La fonction OS\_init( ) sert à initialiser le timer qui met à jour les compteurs de chaque callback et à mettre à 0 les différents tableaux liés aux callback.

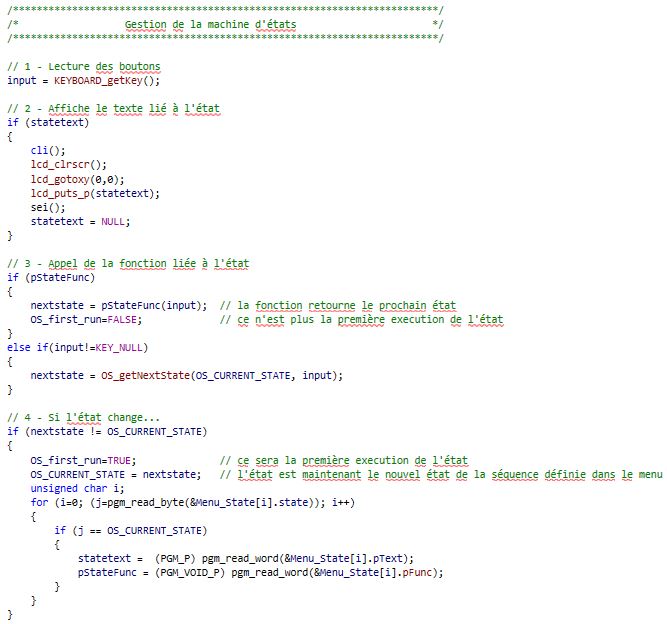


C’est dans la boucle infinie de l’OS que se trouve le code qui gère ces callback à chaque exécution de cette boucle. Une boucle for parcours les callback enregistrées. Elle compare pour chacune d’entre elles leur période d’exécution avec la valeur de leur compteur. Si le compteur est plus grand ou égal à cette période, la fonction liéeà la callback est exécutée et le compteur remis à 0.



## La machine d’état

La gestion de la machine d’état se fait en plusieurs étapes. Dans un premier temps, la fonction KEYBOARD\_getKey récupère la dernière touche appuyée par l’utilisateur. Ensuite, le code affiche le texte lié à l’état s’il y en a un. Troisièmement il appelle la fonction liée à l’état, s’il y en a une. Cette fonction retourne le prochain état selon l’entrée utilisateur. Si aucune fonction n’est liée, la fonction OS\_getNextState est appelée pour déterminer le prochain état. Si le prochain état change par rapport à l’état courant, le code met à jour le nouveau texte à afficher, la nouvelle fonction à exécuter et met à TRUE le booléen OS\_first\_run qui permet aux fonctions liées aux états de savoir si c’est la première exécution de ceux-ci.



Pour le fonctionnement de cette machine d’état, 2 structures sont créées :

* MENU\_STATE : contient un champ pour l’identificateur de l’état, un champ pour le texte lié à l’état et un champ pour la fonction liée à l’état (pointeur de fonction).
* MENU\_NEXTSTATE : contient un champ pour l’identificateur de l’état, un champ pour une entrée utilisateur, et un champ pour le nouvel état dans lequel aller pour cette entrée utilisateur.

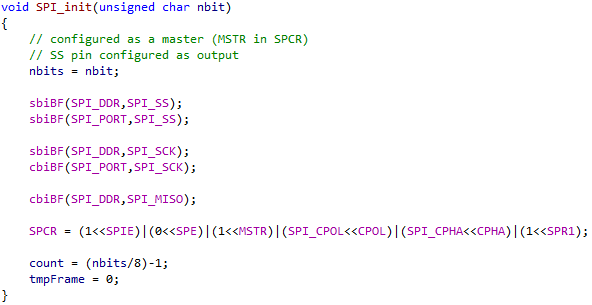
Des tableaux de ces structures sont créés, nommés Menu\_State et Menu\_NextState. Menu\_State aura une ligne pour chaque état et Menu\_NextState aura une ligne pour chaque transition possible entre 2 états.

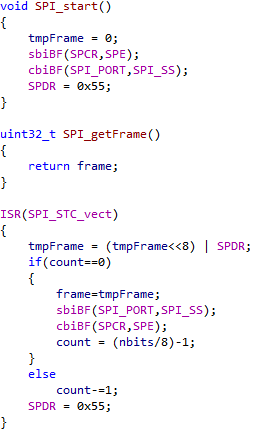
# Serial Peripheral Interface

Dans notre projet, le SPI (Serial Peripheral Interface) est utilisé pour gérer la communication du capteur de température MAX31855 jusqu’au microcontrôleur. Il faut noter que le périphérique SPI de l’ATmega lit les données par paquets de 8 bits, alors que le MAX31855 les envoie par frame de 32 bits. Pour maintenir une structure claire, une librairie de gestion du SPI a été créée ainsi qu’une autre de gestion du capteur.

Commençons par la libraire de gestion du SPI, elle contient les fonctions suivantes :

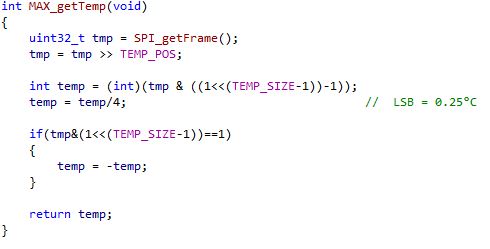
* SPI\_init(nbit) : initialise le port SPI, *nbit* indique le nombre de bits à lire pour chaque frame
* SPI\_start( ) : démarre la communication SPI et met un buffer temporaire tmpFrame à 0
* SPI\_getFrame( ) : retourne la dernière frame entièrement reçue par le SPI



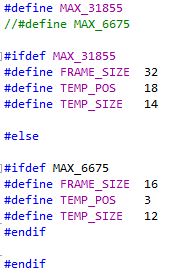


Voyons la libraire MAX. Elle contient 3 fonctions :

* MAX\_init( ) : lance simplement la fonction SPI\_init
* MAX\_start( ) : lance la fonction SPI\_start
* MAX\_getTemp( ) : cette fonction récupère le dernier frame complet reçu par le SPI et en extrait les données de température



TEMP\_POS et TEMP\_SIZE sont définis dans le MAX.h, ainsi le code peut être utilisé pour d’autres capteurs du même type. En effet le MAX31855 n’existe pas dans proteus mais le MAX6675 existe. Ainsi le code n’a besoin que de très peu de modifications pour gérer l’un ou l’autre des capteurs.



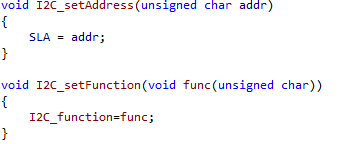
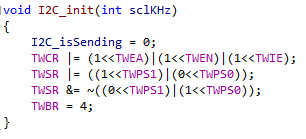
# Two Wire Interface (I2C)

Le bus I2C servira à communiquer avec 2 périphériques : la RTC et l’EEPROM. Ici encore, une libraire séparée est créée pour le bus I2C et pour chaque périphérique. Une interruption unique est déclenchée pour chaque évènement sur le bus I2C. Il faut donc dans cette interruption gérer les différents cas en fonction de l’utilisation que l’on veut faire de ce bus. Le problème est que l’utilisation peut varier si l’on veut utiliser l’EEPROM ou la RTC. Pour cela, la solution mise en place est de passer à la librairie I2C un pointeur vers une fonction. Cette fonction est appelée à chaque fois qu’une interruption intervient, et c’est dans celle-ci que les différents cas sont gérés.

## Librairie I2C

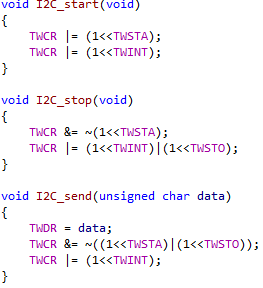
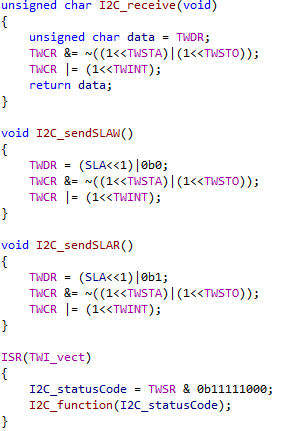
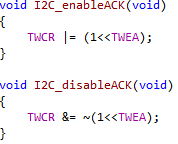
Les fonctions créées pour l’I2C sont séparées en 2 parties. D’abord voyons celles qui servent à configurer la communication :

* I2C\_init( ) :
* I2C\_setAddress(addr) : indique l’adresse du périphérique avec lequel on veut communiquer
* I2C\_setFunction(func) : indique la fonction à exécuter à chaque interruption



Les autres fonctions sont utilisées pendant une communication. Pendant cette communication en mode maitre, le microcontrôleur dit être capable de faire plusieurs choses :

* I2C\_start( ) : envoie un start bit
* I2C\_sendSLAW( ) : envoie l’adresse et initie un mode Write
* I2C\_sendSLAR( ) : envoie l’adresse et initie un mode Read
* I2C\_send(data) : envoie le byte *data*, utilisé en mode Write
* I2C\_receive( ) : retourne un byte lu
* I2C\_enableACK( ) et I2C\_disableACK : en mode Read, il est nécessaire de pouvoir activer ou désactiver l’envoie d’un acknowledge par le microcontrôleur.
* I2C\_stop( ) : envoie un stop bit



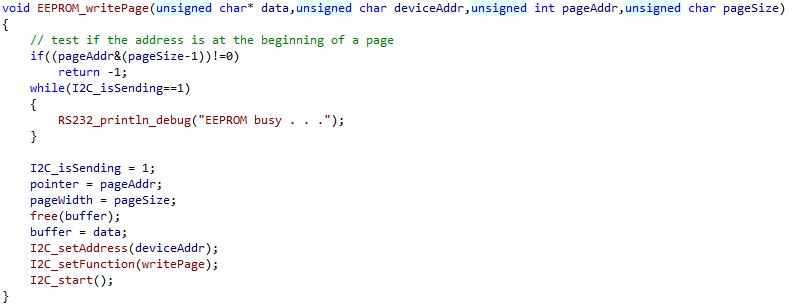
On voit bien que la routine de l’interruption liée à l’I2C lit le status code dans le registre TWSR et le passe à la fonction pointée par I2C\_function.

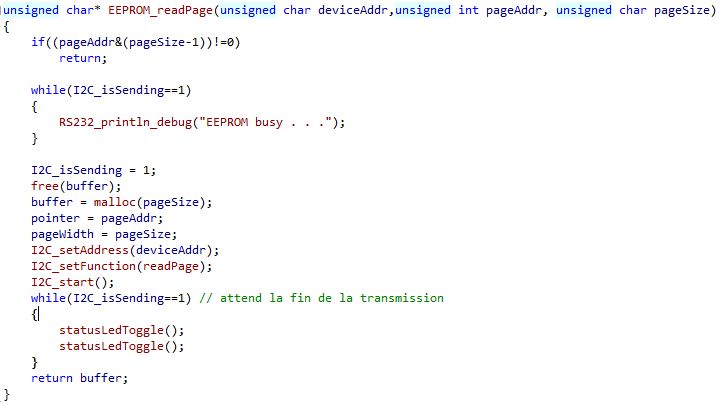
## Librairie EEPROM

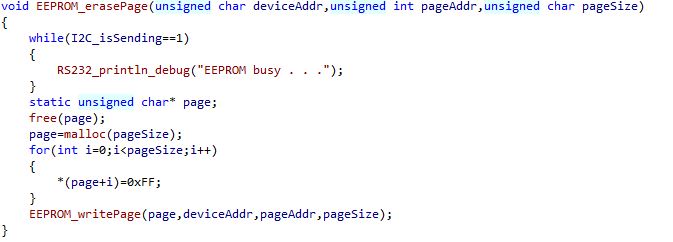
L’EEPROM choisie est une EEPROM 32k de chez microchip. La libraire possède plusieurs fonctions utilisables de l’extérieur :

* EEPROM\_readPage(deviceAddr,pageAddr,pageSize) : retourne un pointeur vers un buffer qui contient la page lue. Le paramètre *deviceAddr* indique l’adresse du périphérique, *pageAddr* désigne la position de la page à lire et *pageSize* la taille de la page (ici 16 bytes).
* EEPROM\_writePage(data, deviceAddr,pageAddr,pageSize) : écrit les données *data* sur le périphérique à l’adresse *deviceAddr*.
* EEPROM\_erasePage(deviceAddr,pageAddr,pageSize) : efface la page située à *pageAddr*.

Ces fonctions vérifient tout d’abord que l’on a bien indiqué le début d’une page comme adresse à laquelle lire, et attendent que le bus I2C soit libre. Ensuite elles utilisent les fonctions I2C\_setAddress, I2C\_setFunction et I2C\_start pour lancer la communication.



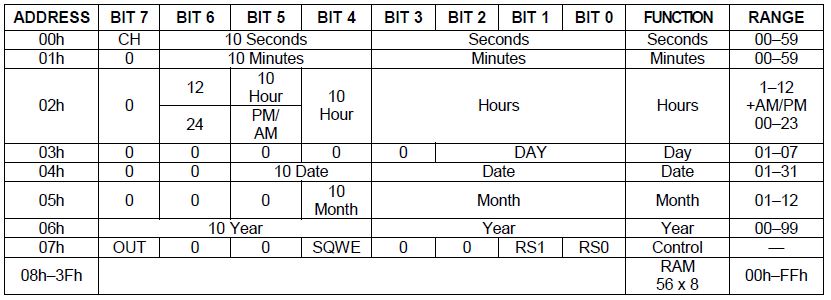




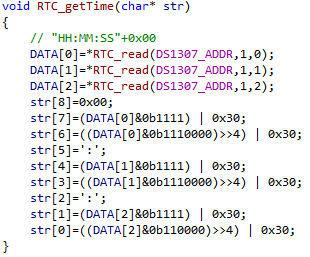
Le code contient également 2 autres fonctions internes : readPage et writePage. Ce sont ces fonctions qui sont envoyées à I2C\_setFunction. Les organigrammes de ces fonctions sont présentés en annexes 1 et 2.

## Librairie RTC (Real Time Clock)

Le fonctionnement de la RTC DS1307 est semblable à celui de la mémoire EEPROM. Voici le contenu de la mémoire interne de cette RTC :



Il suffit donc d’implémenter les mêmes méthodes que celles de la librairie EPPROM pour récupérer le contenu de cette mémoire. La fonction RTC\_getTime( \*str ) s’occupe de lire cette mémoire et de renvoyer dans *str* une chaine de caractères contenant l’heure, les minutes et les secondes actuelles sous la forme HH:MM:SS.

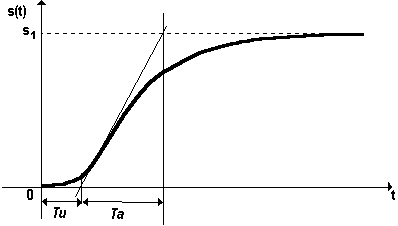


# Régulation via le PID

## Calcul du régulateur

Pour la régulation de notre procédé, nous avons besoin de plusieurs parties du code de notre microcontrôleur, puisque la simple identification s’est fait grâce au datalogger et la connexion SPI.

En effet, pour réguler un système, on a besoin de connaitre la fonction de transfert de celui-ci. On peut donc, soit étudier les équations différentielles et par la suite en faire la transformée de Laplace, ou soit utiliser l’identification. Etudier les équations différentielles est une partie très complexe car pour avoir de bon résultat, il faut prendre en compte beaucoup de paramètre comme l’inertie thermique, l’échange calorifique avec les parois du four etc…

On a donc préféré utiliser l’identification. Celle-ci s’est fait avec la méthode de Strejc. Pour identifier le procédé, on a besoin d’une réponse à un échelon du process. Lorsque l’on a cette réponse, il faut tracer une tangente au point d’inflexion (point ou la dérivée est la plus grande). Cette tangente nous donnera deux valeur, Tu et Ta. Une fois que l’on a ces valeurs, on obtient grâce à des tables, l’ordre du process et la valeur de sa constante de temps. De plus on connait le gain K du process en prenant le rapport de l’entrée sur la sortie.

Pour obtenir : avec n l’ordre du process, τ la constante de temps, et K le gain

Dans notre cas, on obtenait un gain de 2,4, un τ de 550 et un ordre 2.

Nous avons donc calculé un régulateur par une méthode empirique appelé « méthode de constante de temps ». Cette méthode ne demande pas de fonction de transfert du process, mais juste une réponse à un échelon. Pour trouver le gain du régulateur, il suffit de faire l’inverse du gain du process. Dans un second temps, pour trouver la constante de temps Ti, il suffit de prendre le temps écoulé lorsque l’on arrive à 63% de la valeur finale. Enfin, pour Td, la méthode nous dit que l’on doit prendre Ti/4. Cette méthode nous a donné le régulateur suivant : avec K=0.42 ;Ti=400 ;Td=100.

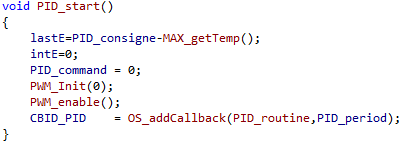
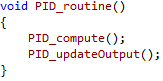
Un paramètre a ensuite été modifié pour avoir un plus faible dépassement. On a donc augmenté la constante de temps Ti=600.

## Code du régulateur

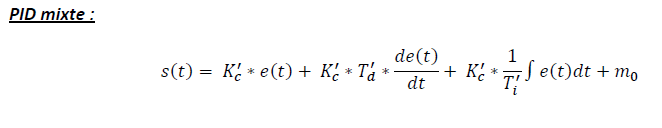
Plusieurs fonctions sont dans la libraire PID pour réaliser la régulation. Les fonctions utilisables dans le reste du programme sont les suivantes :

* PID\_setParams(K,Ti,Td) : sert à fixer les paramètre de notre correcteur
* PID\_setConsigne( consigne ) : sert à fixer la consigne à atteindre
* PID\_stop( ) : arrête la régulation
* PID\_start( ) : démarre la régulation

La fonction PID\_start( ), pour démarrer la régulation, initialise les différentes variables et lance une callback. Cette callback va lancer périodiquement la fonction PID\_routine( ), qui s’occupe d’une part de calculer la prochaine valeur de commande, et d’autre part d’envoyer cette commande sur la sortie PWM.



C’est donc la fonction PID\_compute qui s’occupe de calculer la commande en implémentant un PID de type mixte. L’équation d’un tel PID est donnée :

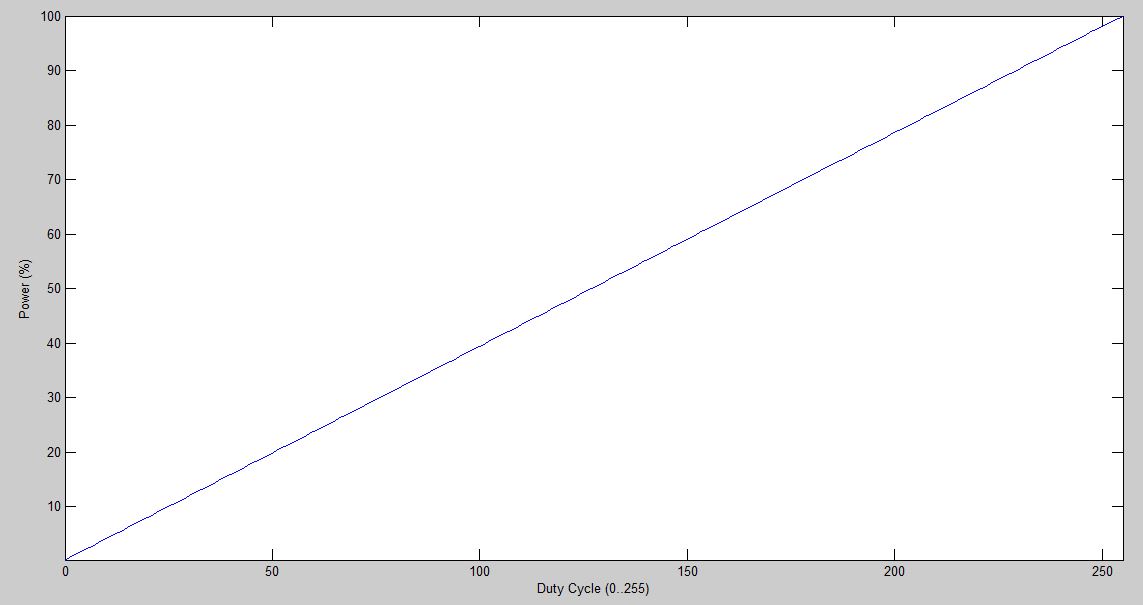


Le code de cette fonction est donné en page suivante. La sortie PWM présentant des valeurs maximum et minimum, une saturation est possible. Pour cette raison, un système anti-saturation est implémenté, si la valeur à assigner à la commande dépasse la plage de fonctionnement du PWM, le terme lié au calcul de l’intégrale de l’erreur est gelé. De plus, cette fonction s’occupe également d’envoyer différente données sur le port série pour permettre un suivi de l’évolution du process.



## Modélisation PWM-relai statique-résistance

Le fait d’utiliser un PWM qui va commander un relai statique pour ouvrir ou fermer l’alimentation de la résistance chauffant par le réseau 220 volts alternatifs pose la question de l’évolution de la puissance effective développée dans la résistance en fonction de la valeur du duty cycle du PWM. En effet, la puissance doit évoluer linéairement par rapport au duty cycle pour que la théorie de la régulation soit valide. Un script matlab a permis de simuler ce système PWM-puissance, dont voici le résultat :



La puissance est exprimée en pourcentage de la puissance maximale (pour un PWM toujours ouvert). Au vu de ces résultats, la linéarité ne fait aucun doute. Le script matlab ayant permis cette modélisation se trouve en annexe 4.

# Conclusion

Dans ce projet, nous avons dû énormément nous pencher sur la programmation, ce qui était le but principal du travail demandé. Cependant la partie hardware a également été très chronophage. Le bruit, les faux contacts et les problèmes d’oscilloscope n’ont jamais lieux sur Proteus.

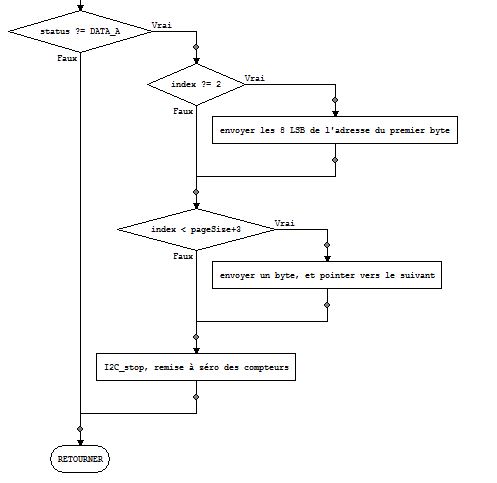
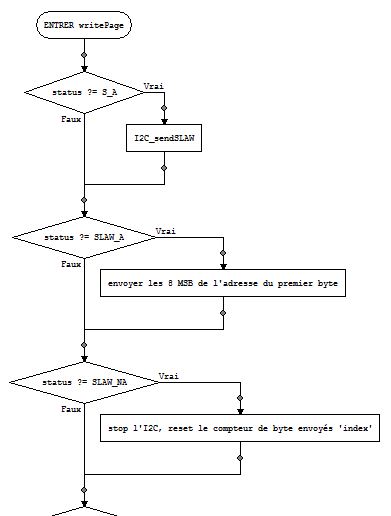
La programmation d’un microcontrôleur nous apparaissant, avant ce travail, surtout comme des travaux sur l’implémentation de l’une ou l’autre fonctionnalité. Grâce à ce projet, nous avons pu nous mesurer à un travail de plus grande envergure, touchant aux limites d’utilisation de notre matériel. En effet, le code final occupe plus de 90% de la mémoire d’un ATmega168, alors que le plan original était de fonctionner avec un ATmega88, ayant moitié moins de mémoire programme.

La nécessité d’avoir une méthode de travail bien structurée a également très vite fait son apparition pour être en mesure de travailler à 2 sur un même projet. Nous avons, dans cette optique, tenté d’approcher l’utilisation d’un logiciel de versions tel que git.

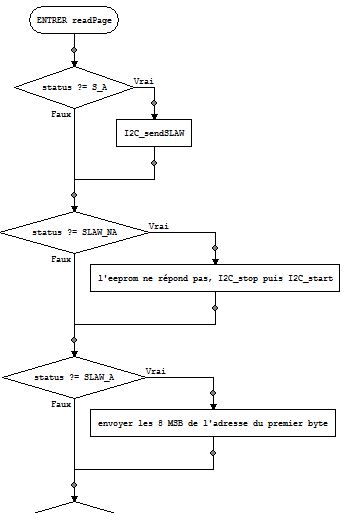
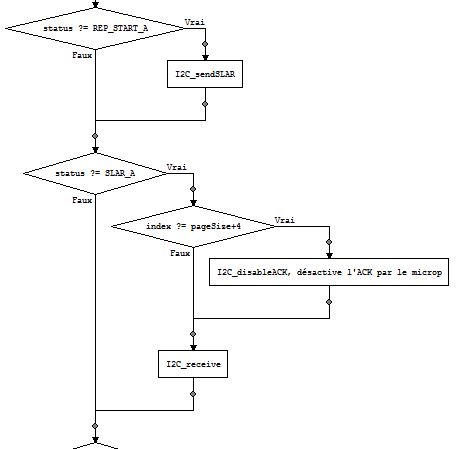
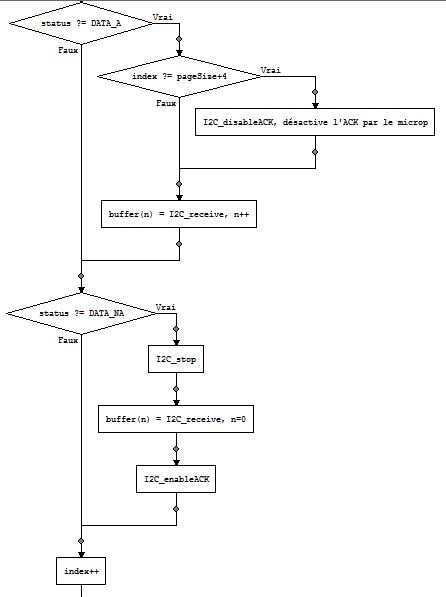
Enfin, nos analyses réflexives personnelles se trouvent en annexes 5 et 6.

# ANNEXES

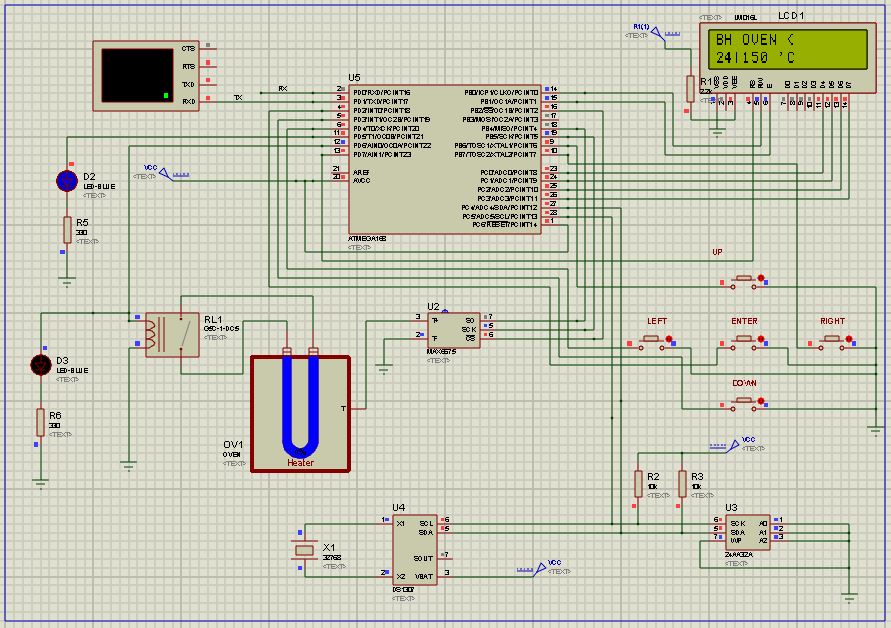
## Annexe 1 : Ordinogramme de writePage



## Annexe 2 : Ordinogramme de readPage



## Annexe 3 : Schéma du système



## Annexe 4 : Script MatLab de modélisation de la puissance

function u = pwm\_gen(f,dc,t)

% counter 8 bit: dc = 0..255

u=zeros(1,length(t));

u(1)=1;

Tpwm=1/f;

Ton=Tpwm\*dc/255;

cnt=0;

dt=t(2)-t(1);

for i=1:length(u)

if(cnt<Ton)

u(i)=1;

else

u(i)=0;

end

if(cnt>Tpwm)

cnt=0;

u(i)=1;

else

cnt=cnt+dt;

end

end

end

clear all;

close all;

f\_res=50;

f\_pwm=30;

T=5;

t=linspace(0,T,5\*30\*2\*255);

u=sqrt(2)\*220\*cos(2\*pi\*f\_res.\*t);

R=120;

P=zeros(1,256);

for dc=0:1:255

pwm=pwm\_gen(f\_pwm,dc,t);

P(dc+1)=mean(((u.\*pwm).^2)/R);

end

plot(0:1:255,P.\*100/P(256));

axis tight;

## Annexe 5 : Analyse réflexive : Thomas Herpoel

#### Quelles notions vous semblent importantes pour réaliser ce genre de projet? Pourquoi?

Bien entendu, le domaine principal à connaitre c’est la programmation en C. Seulement dans un projet de cette envergure, il faut le maitriser relativement en profondeur. En effet, utiliser un code structuré, mais aussi concevoir des fonctions réutilisables dans plusieurs cas (comme notre librairie I2C) permet de simplifier énormément le codage. On voit donc bien l’importance d’un fichier header paramétrable. Cela permet aussi de faire des modifications sur le système sans devoir faire trop de modification dans le code. Par exemple changer la localisation d’une led ou d’une broche juste en changeant un define dans un header.

Au niveau du hardware, l’utilisation de proteus est incontournable. Surtout dans notre cas. Un process de température est extrêmement lent, et parfois l’occasion de tester rapidement une modification sur le process simulé dans proteus a été bien utile.

#### Quelles difficultés avez-vous éventuellement rencontrées? Comment avez-vous procédé pour trouver une solution éventuelle?

Les principales difficultés sont venues des communications I2C ou SPI, et surtout l’I2C. La gestion de cette communication est assez complexe. Encore une fois, le logiciel de simulation proteus a été d’une grande aide grâce à ses outils de debugage de ces communications. Le capteur de température en SPI nous a également posé problème, il envoyait des valeurs de température faussées. Après plusieurs heures perdues, nous avons fini par trouver que le problème était le bruit sur le thermocouple et qu’une simple capacité pouvait régler le problème.

L’écran LCD nous a également posé beaucoup de problème au début, en effet, parfois l’affichage se figeait et il fallait obligatoirement redémarrer le système. Ce problème était due à une « bête » erreur, c’est-à-dire la broche d’alimentation de l’écran mal raccordée au +5V. Cependant, cela illustre un problème plus global : le prototypage sur breadboard. Pour un système qui comprend autant d’éléments, il est assez courant qu’une connexion ne se fasse pas correctement entre un fil et la breadboard, parfois uniquement à cause d’un déplacement d’une table à l’autre du prototype. Le fait de devoir déplacer le breadboard entre notre domicile et l’école n’a probablement pas aidé non plus.

#### Qu’avez-vous modifié dans votre démarche, votre analyse, votre recherche et vos stratégies d’études?

Dans un premier temps, le temps impartis pour réaliser ce projet semblait suffisant. Cependant, il est vite apparu qu’en faisant du développement « au hasard », c’est-à-dire sans établir une liste concrète des choses à faire, et de quoi faire en premier, nous perdions beaucoup de temps. Il est par exemple apparu qu’avoir une communication série et un affichage fonctionnel était une priorité. Cela permet d’envoyer un retour d’information au programmeur pour le debugage.

L’utilisation du logiciel de gestion de version git est également une bonne solution de travail en équipe sur un même projet. Cependant il nous est apparu assez tard et nous avons manqué de temps pour apprendre correctement son fonctionnement.

#### Après avoir analysé votre démarche et pris connaissance de celles des autres étudiants, quels changements souhaitez-vous éventuellement apporter à vos stratégies pour être plus compétent dans votre vie professionnelle future face à un projet de ce type?

Le principal changement serait de commencer par une phase d’analyse du problème. Le but serait de fixer clairement et le plus complètement possible la liste de choses à réaliser, et la priorité de chacune de ces taches. Ensuite, utiliser des outils performants de travail en équipe me semble aussi important.

## Annexe 6 : Analyse réflexive : Antoine Bostem

#### Quelles notions vous semblent importantes pour réaliser ce genre de projet? Pourquoi?

Pour un projet d’une telle ampleur, une connaissance de la programmation est essentielle. Les notions I2C, USART, interruption, callback etc… sont indispensables. De plus, une connaissance assez poussée d’automatique, régulation est nécessaire car elles font partie d’une grosse partie du travail. Outre les connaissances en électronique numérique et programmation, les notions d’électronique analogique sont également utiles pour ce projet, car une partie du projet se situe sur une partie hardware.

#### Quelles difficultés avez-vous éventuellement rencontrées? Comment avez-vous procédé pour trouver une solution éventuelle?

Nous avons eu énormément de difficulté avec le capteur de température, qui fonctionne en SPI. Le codage du SPI s’est fait assez sereinement, c’est la partie hardware qui a été plus complexe. En effet, le capteur ne donnait pas la bonne température. Le problème était finalement simple : « le bruit ». Il nous a suffi de mettre une capacité de filtrage pour neutraliser le bruit qui était ensuite amplifié et donc faussait la température. La régulation a également posé problème car pour faire une bonne régulation, il faut identifier le process. Notre problème était que l’identification prenait beaucoup de temps car on utilisait un process de température. Après avoir finalement identifié le process, nous avons cherché un régulateur correct sur simulateur. Et quand nous l’avons finalement trouvé, nous n’avions pas du tout la même chose sur le process. Nous avons voulu trouvé une réponse trop rapide à cette erreur en mettant le dépassement sur le dos de l’identification. Pour finir, en relisant le code, nous nous sommes rendu compte que l’on s’était trompé dans la formule du PID.

#### Qu’avez-vous modifié dans votre démarche, votre analyse, votre recherche et stratégie d’études?

Au départ, nous avions mal évalué la grandeur du programme, et donc la mémoire prise par celui-ci. C’est donc après avoir fait un over-flow que l’on s’est dit qu’on aurait besoin d’un ATmega plus gros. Des sacrifices au niveau des fonctionnalités ont également été faites car nous manquions de temps pour les réalisées.

#### Après avoir analysé votre démarche et pris connaissance de celles des autres étudiants, quels changements souhaiter-vous éventuellement apporter à vos stratégies pour être plus compétent dans votre vie professionnelle future face à un projet de ce type?

Dans ce projet, nous avons (en séance de projet) toujours travaillé ensemble, on formait un assez bon duo avec des profils différents, ce qui était assez constructif, mais on aurait probablement été plus vite en séparant le travail dès le début. Je pense que nous avons assez vite vu les points principaux à travailler. Nous avons d’abord rendu fonctionnel le hardware, pour pouvoir sortir des valeurs sur nos PC. Après avoir donc dégrossi le hardware et les fonctionnalités utiles, nous sommes passés à la régulation. Nous avons sans doute commencé l’identification trop tard car on ne s’imaginait pas que ça prendrait autant de temps et nous avons passé des séances entières à faire une identification pour finalement se rendre compte que le capteur était mal réglé et qu’il fallait tout recommencer. La méthode de calcul du régulateur aurait dû être plus vite choisie. Les méthodes empiriques donnent de si bon résultat que c’est perdre du temps que de calculer des régulateurs qui ne fonctionnent pas toujours.