Serre automatisée

Par Vincent Gherold & Alain Schöbi, Groupe 126, Mai 2021

# Introduction

Dans l’optique d’utiliser un microcontrôleur équipé d’un capteur de température, nous nous sommes penchés vers l’étude de construire une serre automatisée. N’ayant pas vraiment d’outils concrets à notre disposition, notre projet restera principalement théorique.

Dans une serre, de nombreux paramètres entrent en jeu afin d’optimiser la culture de fruits et de légumes. Nous avons évidemment simplifié le modèle de la serre en essayant d’uniquement gérer sa température. Grâce aux rayons du soleil pendant la journée, la serre se réchauffe à travers ses fenêtres. En été, lorsque les rayons du soleil deviennent très intenses, la température de la serre peut dépasser la température optimale pour la culture, il est donc nécessaire de refroidir la serre. Une méthode pour parvenir à baisser la température de la serre est d’ouvrir les fenêtres de cette dernière. On supposera que la température extérieure est toujours inférieure à la température intérieure. En effet, la température nécessaire pour cultiver des fruits et des légumes est rarement inférieure aux hautes températures que l’on atteint pendant l’été.

# Description générale de l’application

Nous pouvons séparer l’application en plusieurs sous modules. Premièrement, nous avons un moteur servo (module *M4*, Moteur SG3003) qui s’occupe d’ouvrir ou de fermer les fenêtres de la serre. Ensuite, nous avons une interface utilisateur affichée sur le *LCD*. Sur cette dernière, nous affichons la température actuelle de la serre et la température de consigne choisie par l’utilisateur. La température actuelle de la serre est mesurée par le capteur de température (module *M5*, *Wire-1*), tandis que la température de consigne est entrée en paramètre via l’encodeur angulaire (module *M2*). Finalement, nous avons un algorithme qui, en fonction des différentes températures, gère l’ouverture ou la fermeture des fenêtres via le moteur servo ou et qui dans certains cas génère un son d’alerte. Ce dernier est généré par le buzzer (module *M2*).

# Mode d’emploi

### Entrer la température de consigne

Une fois le microcontrôleur démarré, l’utilisateur doit entrer la température optimale pour sa culture via l’encodeur angulaire. Pour cela, il suffit de faire tourner ce dernier jusqu’à atteindre la valeur de consigne. Si l’utilisateur n’entre pas de paramètre, la température de consigne sera mise à sa valeur par défaut (20°C).

### Gestion des fenêtres

Le microcontrôleur gère par lui-même l’ouverture et la fermeture des fenêtres. Il va effectuer une vérification tous les 4 minutes :

* si la température de la serre est plus élevée que la température de consigne (avec une marge de 0.5°C), le microcontrôleur va ouvrir les fenêtres
* si la température de la serre est inférieure à la température de consigne (avec une marge de 0.5°C), le microcontrôleur va fermer les fenêtres

### Sons d’alertes

Une fois que la température de la serre est proche de la température de consigne, le microcontrôleur va essayer de garder la température constante (tout en ouvrant/fermant les fenêtres si besoin).

#### Cas critiques

Malheureusement, dans les cas suivants, la température de la serre va s’écarter de la température consigne sans que le microcontrôleur puisse l'en empêcher.

Le microcontrôleur doit donc générer un son d’alerte pour avertir le propriétaire que la température de la serre s’est écartée de la température de consigne. Voici les cas théoriques :

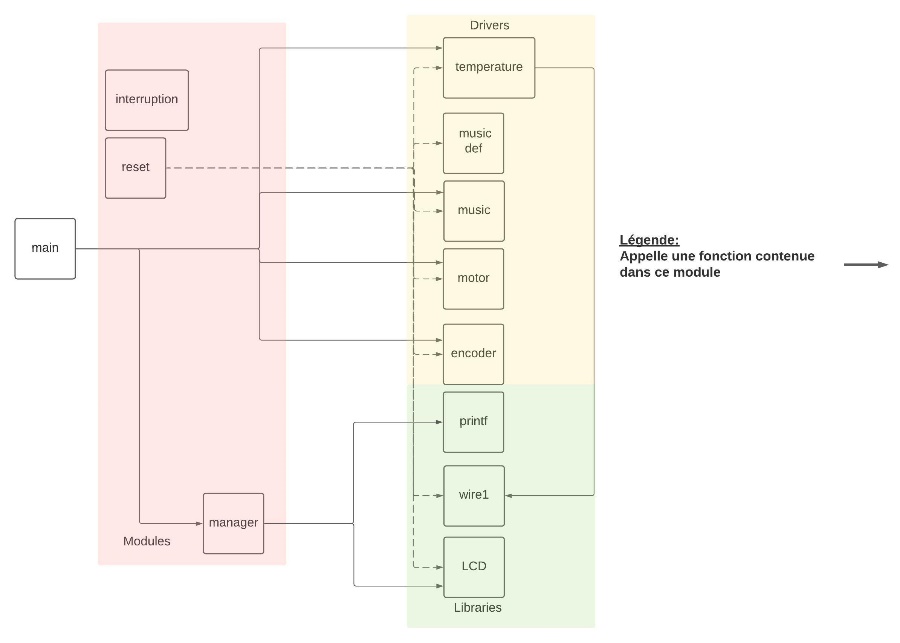
* Le microcontrôleur a gardé la température constante à la température de consigne durant la journée. Une fois la nuit arrivée, un flux de chaleur sortant refroidit lentement la serre. Si la température s’écarte considérablement de la température de consigne, le microcontrôleur va générer un son d’alerte.
* En été, le microcontrôleur a gardé la température constante à la température de consigne durant la matinée. Une fois à midi, comme les rayons de soleil deviennent très intenses, le microcontrôleur ouvre les fenêtres. Malheureusement, même en ayant ouvert les fenêtres, la température à l’intérieur de la serre peut continuer d’augmenter, il faut alors générer un son d’alerte si celle-ci s’éloigne trop de la température de consigne.

# Descriptions technique

Voici les différents éléments de notre projet:

**Libraires**

* wire.1asm
* lcd.asm
* printf.asm
* macros.asm
* definitions.asm



**Drivers**

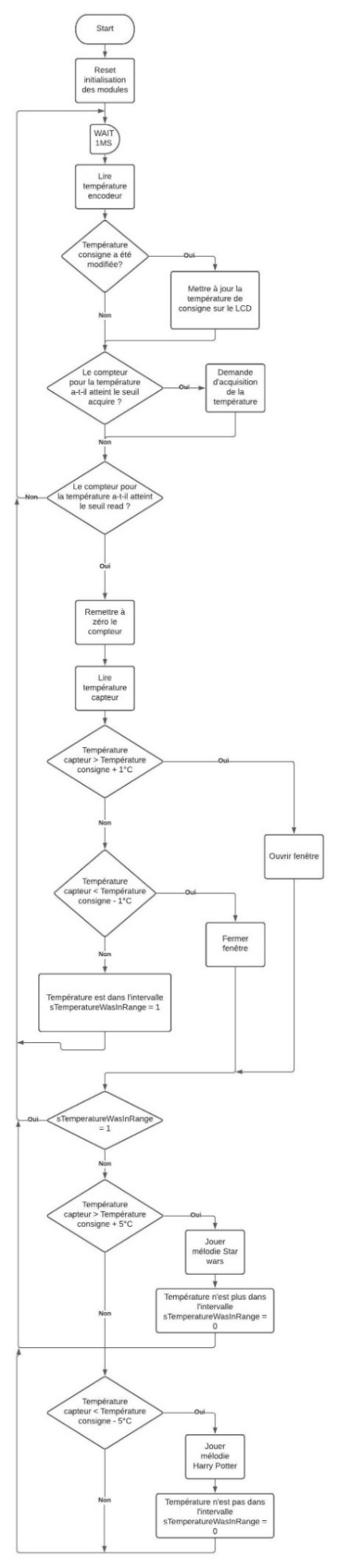
* encoder.asm
* motor.asm
* music.asm et musicdef.asm
* temperature.asm

**Modules**

* manager.asm
* reset.asm
* interruption.asm

Figure 1 Dépendances des différents modules

Ci-dessus vous pouvez voir la boucle principale de notre programme, c’est elle qui appelle les différents modules de notre programme. (Les libraires macro et définitions, m128def n’ont pas été inclus dans la figure 1 car toutes les parties de notre programme dépendent d’elle).

Nous avons choisi les intervalles d’actions suivants  :

(C’est-à-dire à quelles distances de la température de consigne, une action va être effectuée.)

* De plus ou moins 0.5°C autour de la température de consigne pour le contrôle du moteur
* De plus ou moins 1°C pour le contrôle de musique (sons d’alerte).

Ceci est largement suffisant car les plantes ne sont pas sensibles aux faibles variations de température.

La boucle *main* actualise la température toutes les 4 minutes (en mode *production*) et toutes les 2 secondes (pour la *présentation*)

Pour choisir précisément le temps entre deux actualisations de température, nous avons un *timer* qui incrémente une zone mémoire de la SRAM. Nous n’utilisons qu’une zone mémoire de 8 bits c’est-à-dire que la value maximale de la variable incrémentée peut être de 255. Comme nous avons paramétré le *timer* pour générer une interruption toutes 1.04 secondes nous pouvons attendre au maximum 4 min 15 sec entre deux actualisations de la température.

# Modules

## Manager (manager.asm)

Ce module gère l’initialisation du *timer1* qui est dédié à l’actualisation de la température auprès du driver de température. Le *timer1* (16 bits) dont nous avons mis le *prescaler* à 1024 est en mode *Clear On Compare*.

Le module abrite aussi les sous-routines pour rafraichir les deux lignes du LCD (une pour la température mesurée et une pour la température de consigne).

## Reset et interruption (reset.asm et interruption.asm)

Le module *reset* effectue les différentes initialisations des modules, drivers et librairies.

Le module *interruption* met en place la table des vecteurs d’interruption.

Figure 2 Boucle générale

# Périphériques et drivers

## Moteur servo (motor.asm)

Le moteur servo SG3003 se pilote à l’aide d’un signal *PWM* (Pusle Width Modulation) de période 20 ms et d’un niveau haut allant de 1 ms à 2 ms correspondant respectivement à la position angulaire 0° jusqu’à la position 90° du moteur.

Le moteur est connecté sur la broche *P8* du module *M4* et ce dernier est branché sur le *PORTB*.

A partir de la consigne de position envoyée sur la broche P8, le moteur s’occupe lui-même de la respecter. Il faut donc envoyer en tout temps une consigne au moteur. Pour cela, nous avons décidé d’implémenter le driver du moteur à l’aide des interruptions. Le *timer2* du microcontrôleur est dédié au moteur. Nous sauvegardons également 2 zones mémoire dans la *SRAM* afin de se rappeler du temps au niveau haut (*sMOTORHIGH*) et du temps au niveau bas du signal (*sMOTORLOW*). Le *timer2* est utilisé en mode *Clear on Compare Match*  ce qui permet de choisir précisément la durée entre deux interruptions du *timer,* en le préchargeant automatiquement grâce à la variable *OCR2*.

Le graphique à gauche représente le compteur du *timer2* en fonctionnement. Chaque “changement” est dû à une interruption.

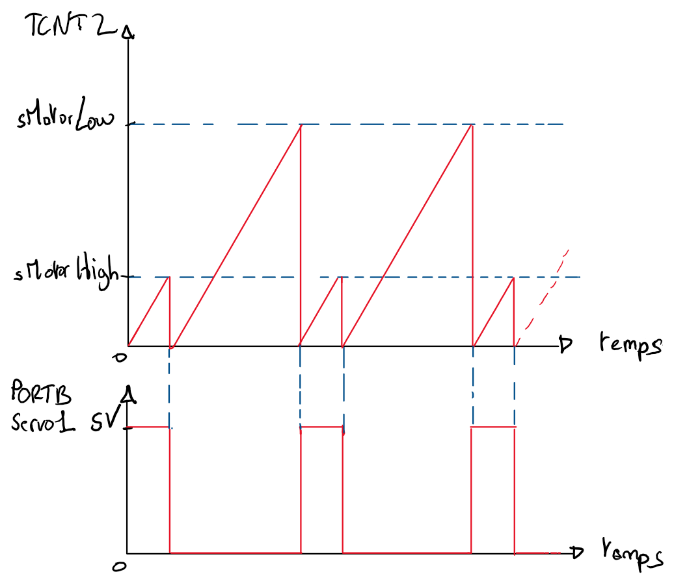
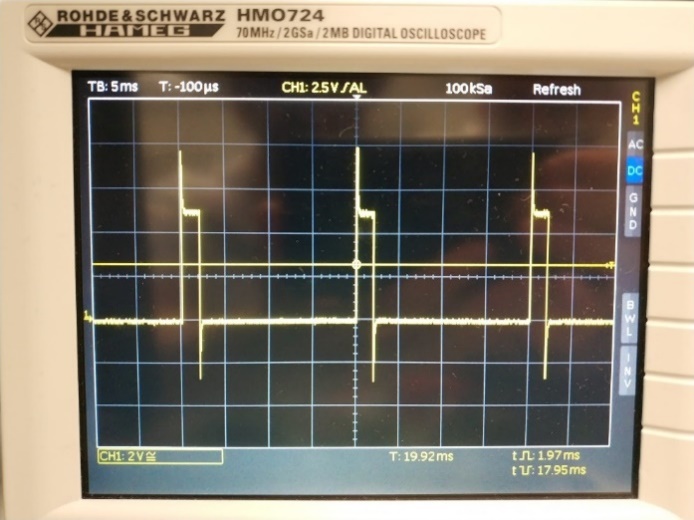


Figure 3 Timer et signal gérant le moteur

Figure 4 Moteur avec 2ms à 5V sur 20ms (fenêtre fermée)

Si nous changeons la consigne de position du moteur trop brutalement, le moteur demandera un courant à la carte trop élevé, ce qui enclenchera le *burn out detector* (nous en disposons deux : un sur la carte STK300 et un directement à l’intérieur de microcontrôleur). Le *burn out detector* compare la tension d’alimentation à une valeur de tension seuil choisie. Si la tension d’alimentation descend en dessous de celle-ci, un signal de *reset* est envoyé et effectue un *reset* du programme. Comme le moteur demande un courant considérable uniquement durant une période courte cela ne pose pas de problème. De plus, nous avons diminué la tension seuil à 3.3V grâce au jumper JP2 présent sur la carte.

## Buzzer (music.asm et musicdef.asm)

Nous utilisons le buzzer piézoélectrique du module *M2* sur le *PORTE*. Nous avons commencé par configurer le *timer3* afin qu’il génère un signal carré à une fréquence donnée. Le *timer3* compte sur 16 bits, ce qui donne une bien meilleure précision que sur un *timer* comptant seulement sur 8 bits. En effet, pour un *la* 440Hz, nous obtenons une fréquence théorique de 440.038Hz, tandis qu’avec le *timer0* (8 bits) nous aurions obtenu 442.811Hz. Ces imprécisions deviennent un problème remarquable, si nous souhaitons jouer une mélodie. Par exemple, avec le *timer0* nous aurions eu un décalage de plus de 24Hz pour la un *si* 5.

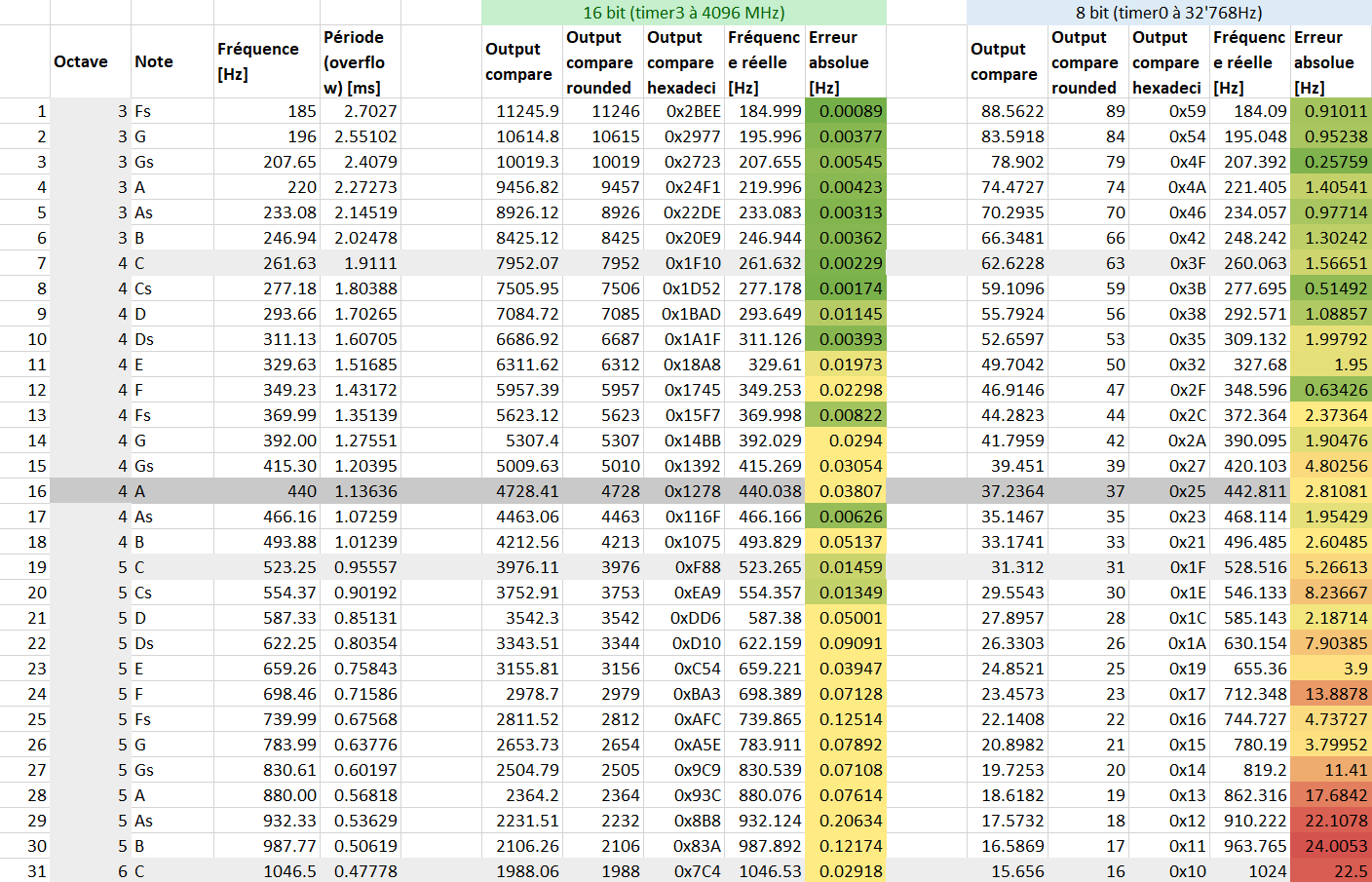


Figure 5 Calculs des fréquences des notes

Nous avons essayé de calibrer les fréquences pour obtenir une fréquence la plus proche d’un *la à* 440Hz. Ci-dessous, un programme inversant la tension tous les 4 cycles. D’après nos mesures, à l’aide d’un oscilloscope, la fréquence de l’horloge interne du microcontrôleur serait de:

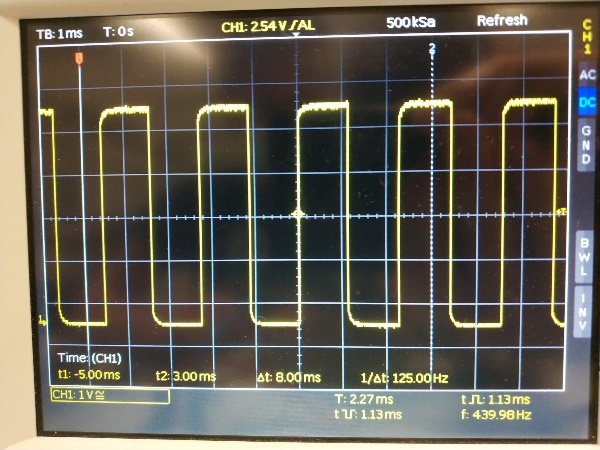
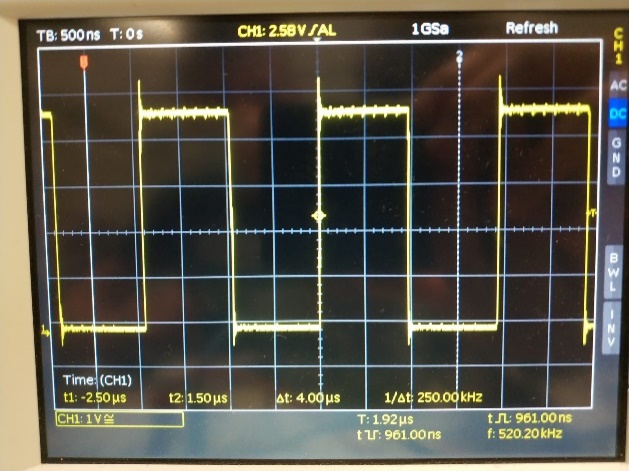
 

Figure 6 Mesure de la fréquence de l'horloge interne avec un programme inversant la tension tous les 4 cycles

Figure 7 Buzzer produisant un la à 439.98 Hz

Grâce à des calculs sur Excel, nous avons pu calculer les différentes valeurs de l’*Output Compare Register* qu’il faut utiliser pour une note donnée. Nous utilisons un *prescaler* de 1 et avons activé le mode *Clear on Compare Match*.

Nous avons ensuite stocké toutes les valeurs des fréquences sur 16 bits (de 31 notes) dans la mémoire programme. Nous pourrons y accéder grâce à l’instruction *lpm*.

Afin de jouer une mélodie, nous utilisons également le *timer0* qui va jouer le rôle de métronome (lecture de la partition). Ce *timer* est également utilisé en mode *Clear on Compare Match* pour pouvoir utiliser des notes de durée différente. Nous avons ensuite défini un standard pour une note: les 3 bits de poids fort (D3-D1) représentent la durée des notes et les 5 bits de poids faible (N5-N1) codent la note elle-même. Nous avons donc 25 = 32 possibilités pour le choix des notes. Ayant défini le code binaire *N = 0b0 0000* comme un silence, il nous reste donc 31 notes à choix (de F#3 à C6). Le code 0x00 représente la fin de la partition.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D3 | D2 | D1 | N5 | N4 | N3 | N2 | N1 |

La partition est, elle aussi, enregistrée dans la mémoire programme. Par exemple :

.cseg ; programm memory

partition: .db nC4+Ronde, nD4+Noire, nE4+Croche, 0

A chaque fois que le *timer0* atteint la valeur de son *Output Compare Register*, le *timer* est automatiquement remis à 0 et l’interruption *iBpm* est appelée, ce qui exécute les étapes suivantes :

* Cherche la position actuelle dans la partition (sauvegardée dans la *SRAM*, dans *sMUSIC*) et l’incrémente
* Lit la note actuelle dans la mémoire programme
* Modifie l’*Output Compare Register* du *timer0* pour définir la durée de la note
* Lit la fréquence actuelle (16 bits) dans la mémoire programme
* Modifie l’*Output Compare Register* du *timer3* pour avoir la fréquence correspondante

L’algorithme de musique est situé dans le fichier *music.asm* tandis que les constantes et les partitions sont enregistrées dans le fichier *musicdef.asm*.

## Encodeur angulaire (encoder.asm)

Nous utilisons l’encodeur angulaire du module *M2* sur le *PORTE* (le module *M2* comporte également le buzzer). Pour notre application, nous nous intéressons seulement à la position angulaire de l’encodeur. Nous n’agissons donc pas sur le bouton de ce dernier.

Nous avons créé un fichier *encoder.asm*, inspiré du code du cours, qui s’occupe de la position angulaire de l’encodeur. Lors d’un appel de la sous-routine *encoder*, la nouvelle valeur de la position est lue via les *pins* *ENCOD\_A* et *ENCOD\_B* du *PORTENCOD*. S’il y a eu un changement de la position de l’encodeur depuis le dernier appel de la sous-routine *encoder*, le fanion T est activé et la nouvelle valeur est placée dans le registre *rENCOD*.

Nous avons également défini deux constantes pour pouvoir gérer la valeur de la position angulaire entre deux bornes (valeur minimale *ENCODMIN* et maximale *ENCODMAX*) ainsi que sa valeur par défaut au démarrage (*ENCODDEFAULT*).

Dans la partie *main* de notre application, nous appelons donc la sous-routine *encoder* toutes les millisecondes et vérifions ensuite si le fanion T est actif ou non. Si oui, alors nous pouvons lire la valeur du registre *rENCOD*.

Comme nous voulons pouvoir régler la température à 0.25°C près, et comme la consigne est comprise entre 15°C et 30°C, nous pouvons n’utiliser qu’une seule variable de 8 bits avec 2 bits pour la partie fractionnaire (2^-2 = 0.25) et 6 bits pour la partie entière. La température de consigne peut donc varier de 0 à 64.75°C ce qui est amplement suffisant.

## LCD

Le LCD de Hitachi utilise par défaut les *PORTA* et *PORTC* de la carte STK-300. Nous l’utilisons pour y afficher les différentes températures. Comme la température de consigne ne s’actualise pas en même temps que la température actuelle de la serre, nous avons créé deux sous-routines séparées (*updateLCDTemperature* et *updateLCDEncoder*). Chacune de ces sous-routines modifie seulement une ligne de l’affichage LCD, la deuxième ligne n’étant pas modifiée. Ainsi, nous pouvons actualiser uniquement une température sans pour autant changer l’affichage complet.

Nous avons ici utilisé les librairies fournies *printf.asm* et *lcd.asm* pour gérer le LCD. Pour démarrer l’écriture du LCD à la deuxième ligne, nous appelons la sous-routine *LCD\_pos* en ayant préalablement posé le registre *a0* à *0x40*. Nous utilisons ensuite la macro *PRINTF* pour l’écriture. La température du capteur thermique étant enregistrée sur 16 bits et en nombre à virgule fixe, nous utilisons la constante *FFRAC2* pour l’indiquer à *PRINTF* (dans le *.db*).

## Capteur de température (temperature.asm)

Le capteur de température se situe sur le module *M5* et ce dernier est branché sur le *PORTD* de la carte. Le capteur utilise le procédé de communication *Wire-1* en mode alimentation externe, c’est-à-dire que le capteur nécessite trois fils (*VCC*, *Ground*, et *D\_WIRE*) pour pouvoir communiquer.

Le module *temperature.asm* utilise la librairie *wire1.asm* fournie dans le cours. Le capteur met 750 ms pour acquérir la température dans sa mémoire *Scratchpad*. Afin que le microcontrôleur ne soit pas bloqué pendant ce temps, nous demandons au capteur d’aller acquérir la température, et 750 ms plus tard, nous prenons cette valeur (que le capteur a pu acquérir entre temps). Pour cela, nous utilisons un *timer* incrémentant une variable *sReadTemperatureRequest* qui gère les demandes avec le capteur. Nous avons ainsi réduit la durée de la communication à 5.30 ms à la place de 755 ms. Le module stocke la température dans la SRAM à l’aide de 2 variables (*sTEMPh* et *sTEMPl*).

# Conclusion

Ce projet nous a permis d’élargir nos connaissances tout en ayant la chance de la liberté du choix du projet, d’autant plus qu’il y avait un nombre considérable de périphériques disponibles. Le travail en groupe a été très bénéfique pour coder en assembleur. Ce langage étant assez complexe et les erreurs apparaissent rapidement si l’on travaille seul. Grâce à ce projet, nous avons désormais une meilleure compréhension du code assembleur et de ses liens avec le code en C, ainsi que le rôle du compilateur.

# Annexes

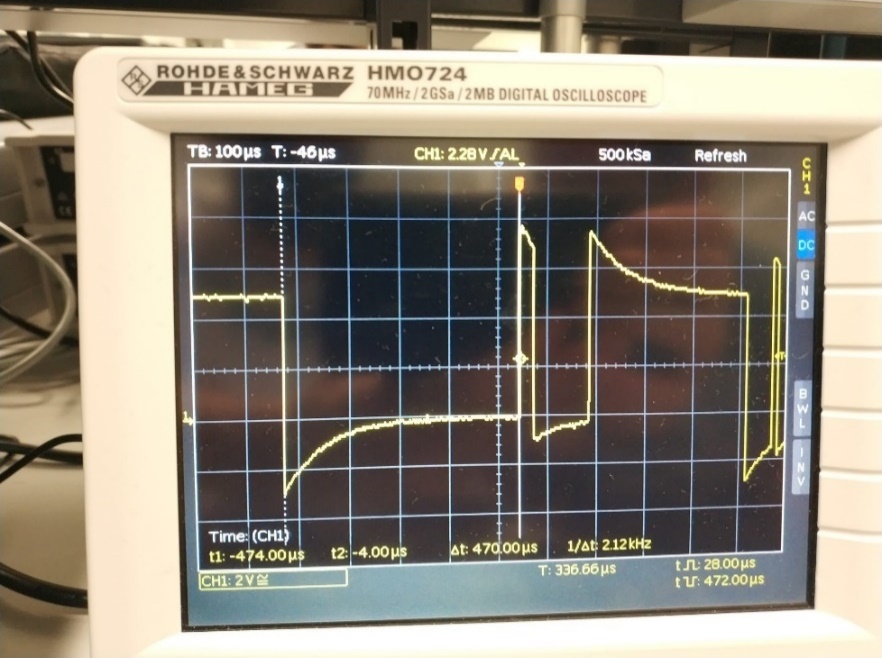
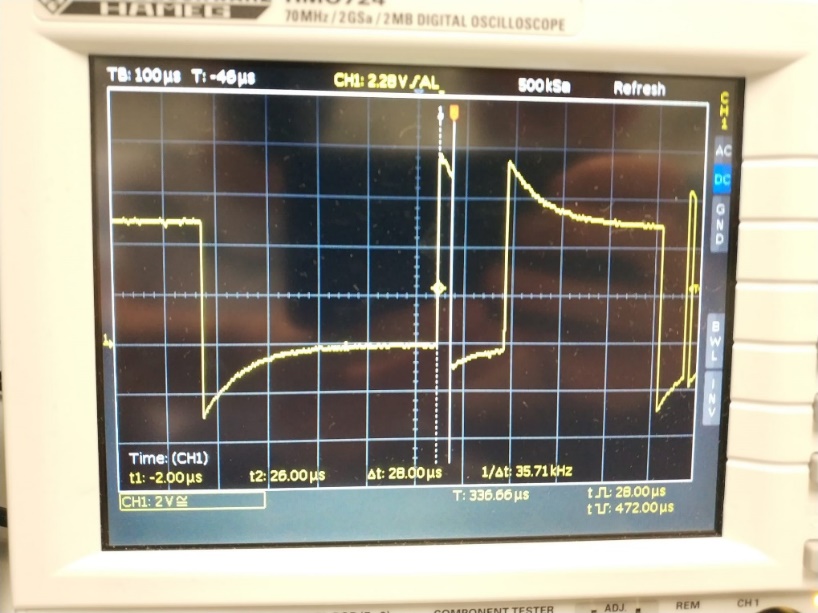


Figure 8 Communication Wire-1 avant le presence pulse (28 μs)

Figure 9 Wire-1 reset pulse (470 μs)

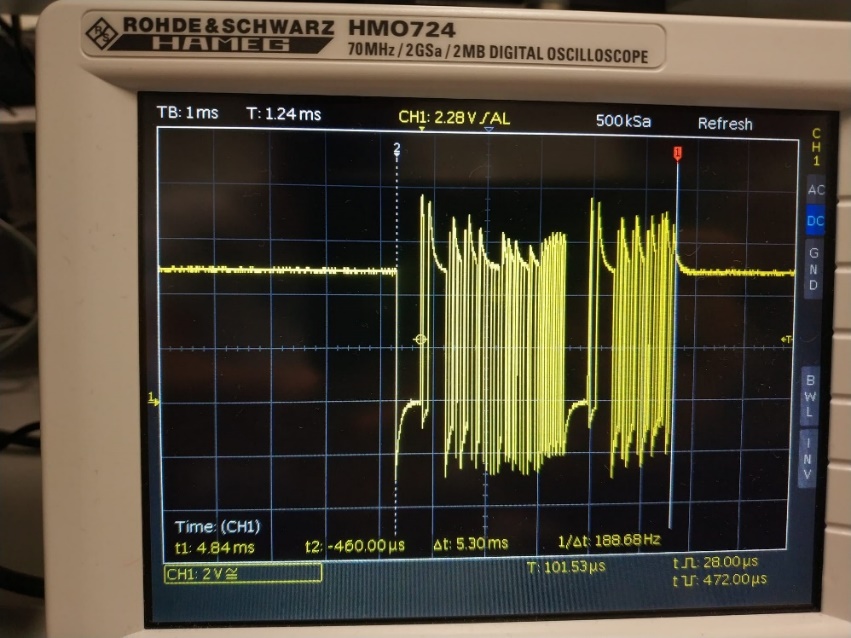


Figure 10 Communication Wire-1 requête complète (durée 5.30ms)

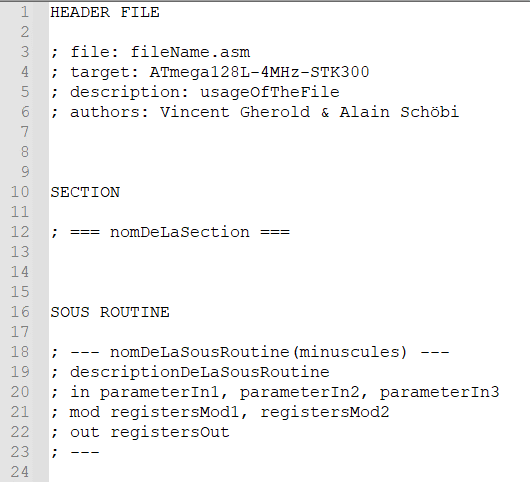
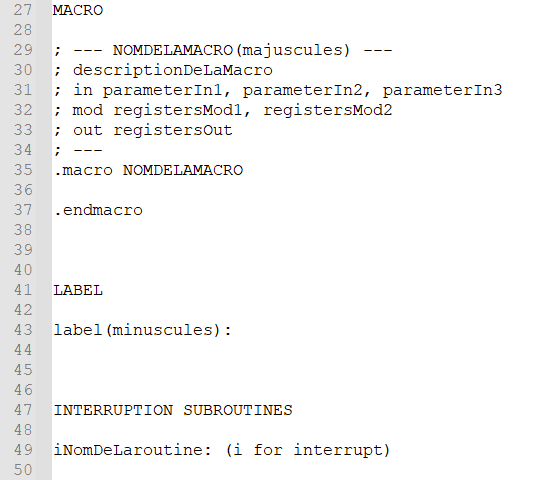
 

Figure 11 Conventions de programme du programme