Rapport ISE-OC : Serre Connectée

Qualité logicielle – bus de communication

Corentin Bouynot – Tristan sannier

2021

# Table des matières

[Table des matières 1](#_Toc68193040)

[1. Cahier des charges 2](#_Toc68193041)

[1.1 Présentation du projet 2](#_Toc68193042)

[1.2 Les Besoins : 2](#_Toc68193043)

[1.3 Les contraintes : 2](#_Toc68193044)

[1.4 Liste des fonctions à réaliser : 3](#_Toc68193045)

[1.5 Liste des entrées et sorties que devra gérer la serre connectée 3](#_Toc68193046)

[2. Etude préalable 4](#_Toc68193047)

[2.1. Schéma 6](#_Toc68193048)

[2.2. Découpage fonctionnel 6](#_Toc68193049)

[3. Tests Réalisés 14](#_Toc68193050)

[3.1. Environnement de test 14](#_Toc68193051)

[3.2. Lecture capteur UV 15](#_Toc68193052)

[3.3. Commande servomoteur 16](#_Toc68193053)

[3.4. Test des fonctions présentes dans la classe UART 17](#_Toc68193054)

[Description 17](#_Toc68193055)

[Câblage réalisé 17](#_Toc68193056)

[Test unitaire de la fonction uart\_send\_byte 17](#_Toc68193057)

[Résultat attendu 17](#_Toc68193058)

[Procédure de test 18](#_Toc68193059)

[Résultat obtenu 18](#_Toc68193060)

[3.5. Test de le communication SPI 18](#_Toc68193061)

[Description 18](#_Toc68193062)

[Câblage réalisé 18](#_Toc68193063)

[Test 19](#_Toc68193064)

[Résultat attendu 19](#_Toc68193065)

[Procédure de test 19](#_Toc68193066)

[Résultat obtenu 19](#_Toc68193067)

[4. Conclusion 20](#_Toc68193068)

# Cahier des charges

## 1.1 Présentation du projet

La serre connectée doit recevoir des informations sur un microcontrôleur MSP430G2553 par l’intermédiaire d’un module Bluetooth RN-42 interfacé en UART. Ces informations doivent être interprétées et transmises à un deuxième microcontrôleur MSP430G2553 via un bus SPI. Ce deuxième microcontrôleur doit, selon la commande reçue

* Lire la valeur d’un capteur UV sur un de ses ports analogiques
* Lire la valeur d’humidité et de température d’un capteur via un interfaçage I2C
* Commander un servomoteur, variant entre 3 positions

Le programme doit être réalisé en C, et devra respecter certaines normes de codage, dérivées du MISRA 2012.

## 1.2 Les Besoins :

Le client nous a communiqué la liste de besoins suivante :

* La serre connectée devra communiquer avec l’utilisateur via un module Bluetooth.
* La serre connectée devra permettre de visualiser sur son smartphone la valeur d’UV relevée.
* La serre connectée devra permettre de visualiser sur son smartphone la valeur d’humidité relevée.
* La serre connectée devra permettre de visualiser sur son smartphone la valeur de température relevée.
* La serre connectée devra permettre à l’utilisateur de fermer ou d’ouvrir la serre par la commande du servomoteur.
* Les commandes d’ouvertures et de fermeture de la serre devront être communiquer par Bluetooth.

## 1.3 Les contraintes :

Ce projet devra contenir au minimum les points suivants :

* Un bus SPI
* Un Bus I2C
* Une communication UART
* Tests unitaires des fonctions réalisée
* Test de non-régression
* Un test d’intégration
* Un test Fonctionnel

## 1.4 Liste des fonctions à réaliser :

Voici la liste des fonctions nécessaire que nous avons identifiées pour mener à bien ce projet :

* Fonctions d’initialisation des launchpad.
* Fonction d’initialisation du SPI.
* Fonction d’initialisation de l’I2C.
* Fonction d’initialisation de l’UART.
* Fonction d’acquisition de la valeur UV.
* Fonction d’acquisition de la valeur d’humidité.
* Fonction d’acquisition de la valeur de température.
* Fonction d’envoi de la communication SPI Master vers Slave.
* Fonction d’envoi de la communication SPI Slave vers Master.
* Fonction d’envoi de la communication I2C.
* Fonction d’envoi de la communication UART.
* Fonction d’interprétation des commande reçus en UART.
* Fonction de PWM pour la rotation du servo-moteur.

## 1.5 Liste des entrées et sorties que devra gérer la serre connectée

Les entrées du système :

* Communication UART.
* Communication SPI.
  + MOSI/MISO
  + (Clock)
* Communication I2C.
* Mesure de la valeur lue sur le capteur UV.

Sorties :

* Communication UART.
  + Vers terminale PUTTY/module bluetooth.
* Communication SPI.
  + MOSI/MISO.
  + (Clock).
* Communication I2C.
* Rotation du servo-moteur via une PWM.

MISRA 2012.

# Etude préalable

Afin d’échanger des données entre les launchpad nous nous sommes basés sur le code que nous avions réalisé au cours de notre premier TP de Bus de communication et nous nous somme aidés des exemples mis à notre disposition a afin de pouvoir reprendre notre code et d’y réaliser les modifications permettant de mener à bien ce projet.

Durant cette partie nous avons rencontrés quelques difficultés pour faire communique les MSP entre eux, dû au fait que nous ne disposions que d’aucun jeu de launchpad fonctionnel (nos deux MSP430G2231 étant défectueuses).

Concernant notre code nous avons commencé par nous questionner sur la configuration de la communication SPI coté MSP430G2553. Nous avons alors défini les points suivants afin de paramétrer la communication :

* Le signal d’horloge sera transmis par la broche 1.5.
* Le MISO sera transmis par la broche 1.6.
* Le MOSI sera transmis par la broche 1.7.

Nous avons ensuite réfléchi aux actions à effectuer pour en arriver aux choix suivants :

* Le caractère ‘1’ allume la led 1.0 du MSP430G2231.
* Le caractère ‘2’ allume la led 1.6 duMSP430 G2231.
* Le caractère ‘0’ éteint les deux leds.
* Afficher un message dans la fenêtre putty de l’opération effectuée afin d’informer l’utilisateur.

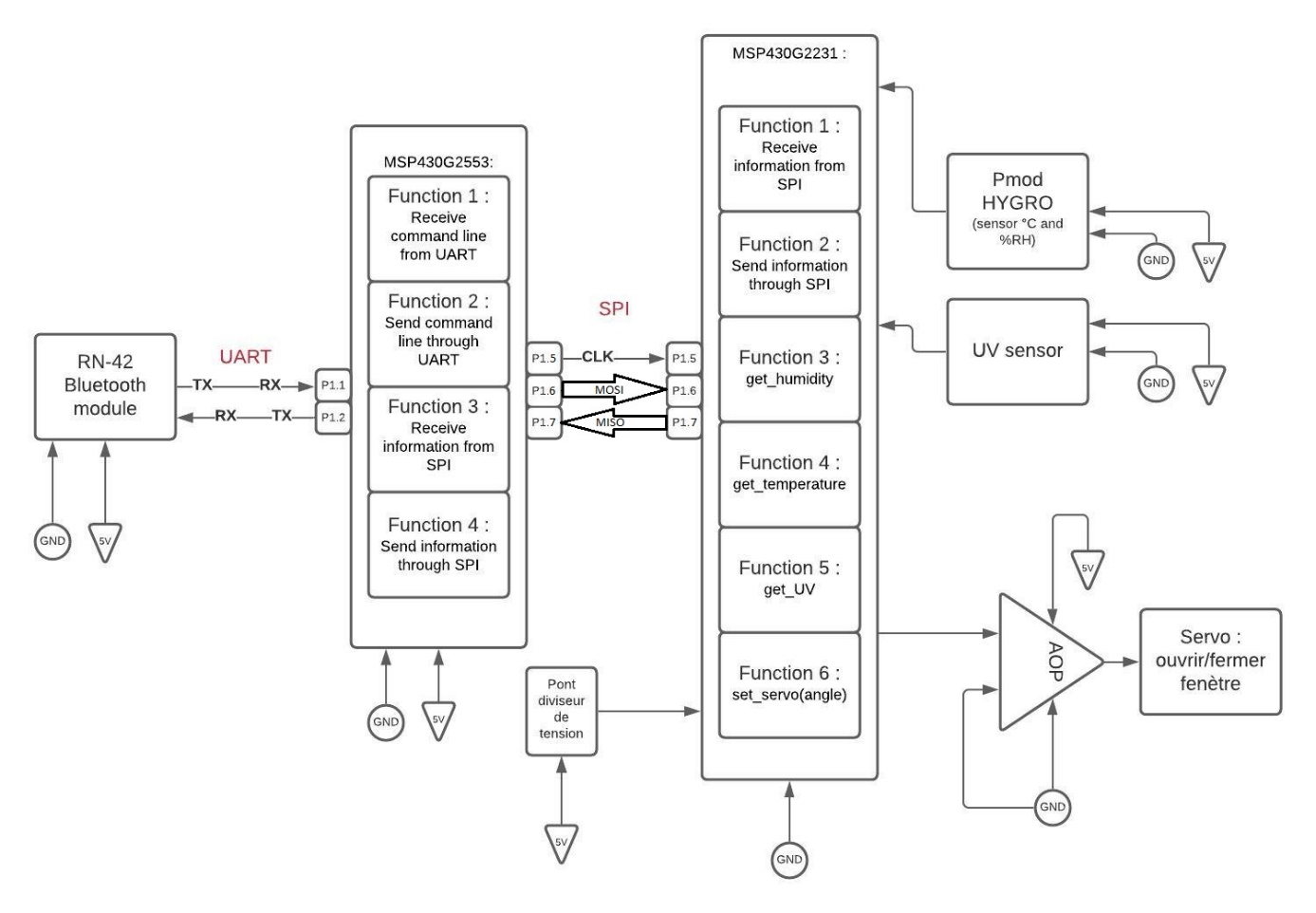
Du coté MSPG2231 nous avons choisi de définir les points suivants concernant la communication

* Le signal d’horloge sera transmis sur la broche 1.5.
* Le MOSI sera transmis par la broche 1.6.
* Le MISO sera transmis par la broche 1.7.

Nous avons ensuite défini les différents cas auxquels nous avions réfléchis précédemment dans le code coté master (dans notre interpréteur de commande) ainsi que du côté MSP430G2231 (slave) :

* Le caractère ‘1’ allume la led 1.0.
* Le caractère ‘2’ allume la led 1.6.
* Le caractère ‘0’ éteint les deux leds.

En partant cela nous avons donc réalisés un diagramme fonctionnel en vue de la réalisation (initialement futur d’un PCB) mais finalement de la maquette réalisée avec nos launchpads. Le diagramme que nous avons réalisé est visible sur la figure ci-dessous :



*Diagramme fonctionnel avec MSP430G2231*

L’étude des différentes documentations techniques ainsi que la réalisation du diagramme fonctionnel du PCB nous a amenés à divers problèmes. Cela impliquant donc que nous nous avons été amenés à réfléchir sur ces derniers. Nous avons ainsi pu constater que les pins présents permettant de communiquer en SPI sont les même que ceux permettant de communiquer en I2C. De ce fait, si nous utilisons une connexion par BUS SPI nous ne disposerions plus de suffisamment de pins pour connecter notre capteur HYGRO.

Nous Avions donc envisagé de passer la connexion entre les MSP en I2C et de communiquer avec le slave et le capteur HYGRO par l’intermédiaire d’un bus I2C.

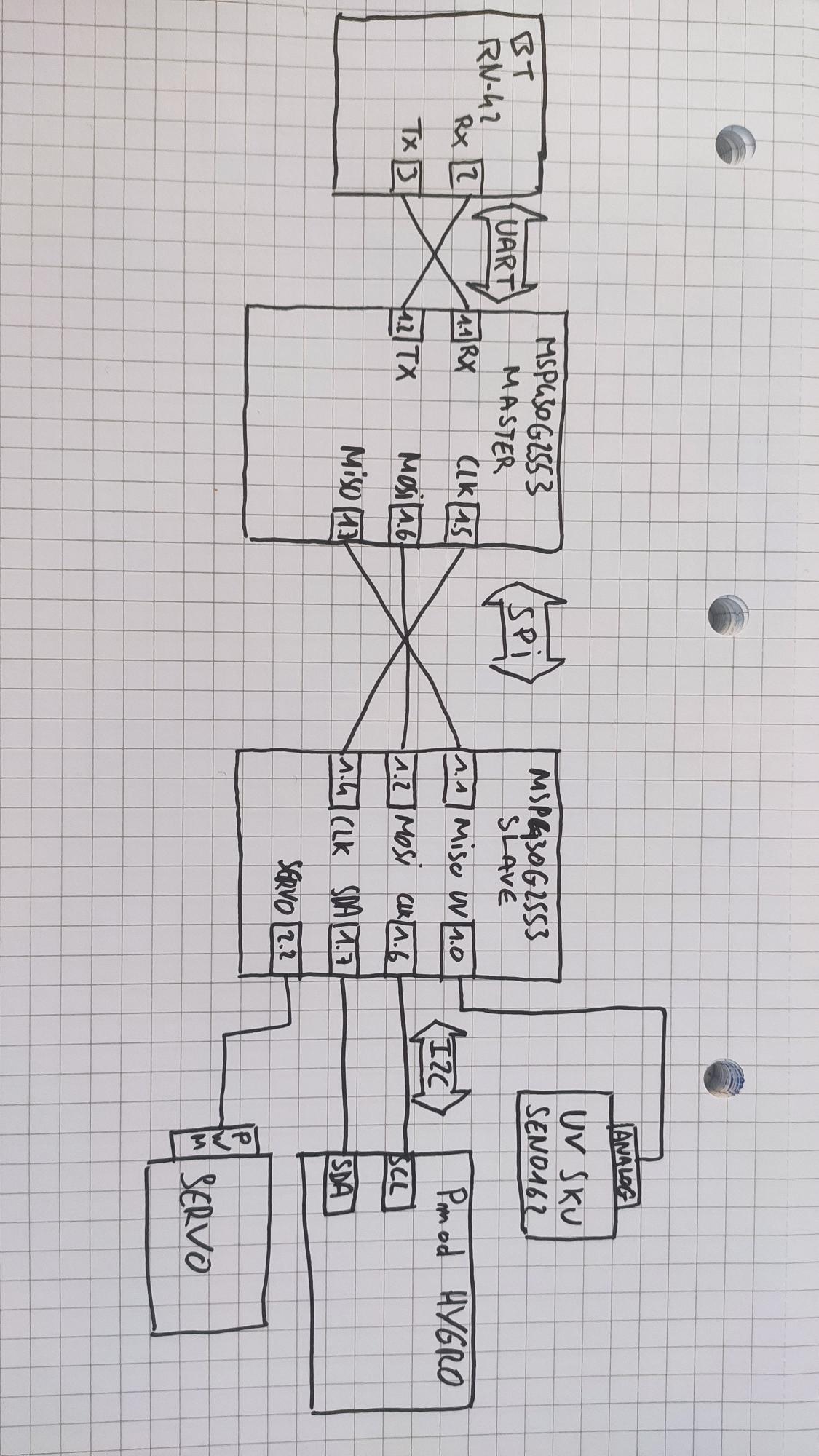
Ainsi le MSP430G2553 le master de deux slaves, cependant cela modifierait nôtre diagramme fonctionnel de la manière suivante : le MSP430G2553 sera master du MSP430G2231 et master de HYGRO.

Nous pensons qu’un fonctionnement comme celui-ci permettrais d’éviter d'avoir à utiliser simultanément du SPI de l’I2C dans le même projet, ce qui aurait posé problèmes si nous voulions les utiliser en même temps

Cependant cela a l’inconvénient de ne pas avoir tous les capteurs/actionneurs déportés sur la même carte (en projet réel, cela aurait été plus pratique).

Cependant le cahier des charges réalisé sur les indications du client nous imposant de réaliser une communication SPI (et n’ayant pas de MSPG2231 fonctionnels), nous avons donc choisis de remplacer les deux MSPG2231 par deux MSPG2553 et de réaliser la communication inter-MSP via une communication SPI trois fils.

De ce fait nous avons donc pu recréer un second digramme fonctionnel, visible sur la figure ci-dessous :



*Diagramme fonctionnel avec deus MSP430G2553*

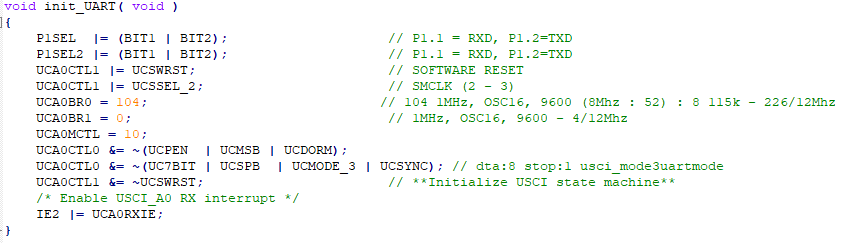
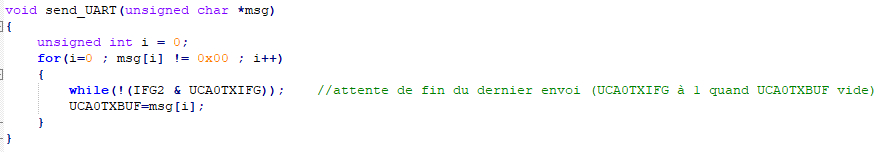
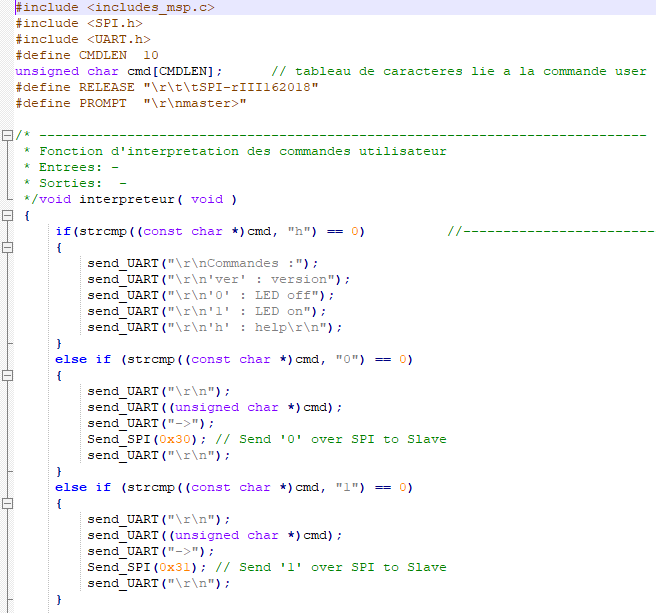
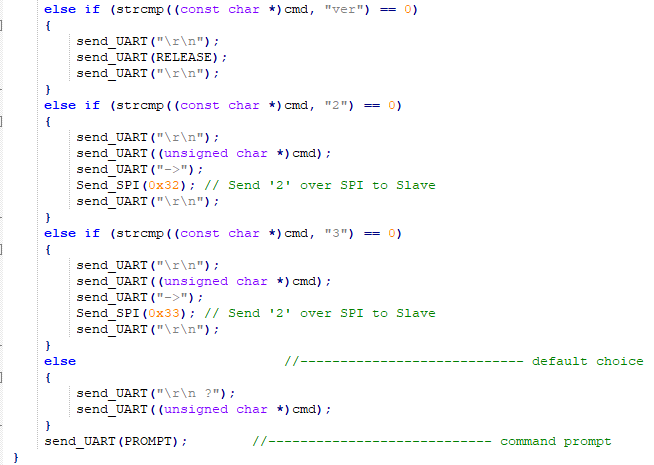
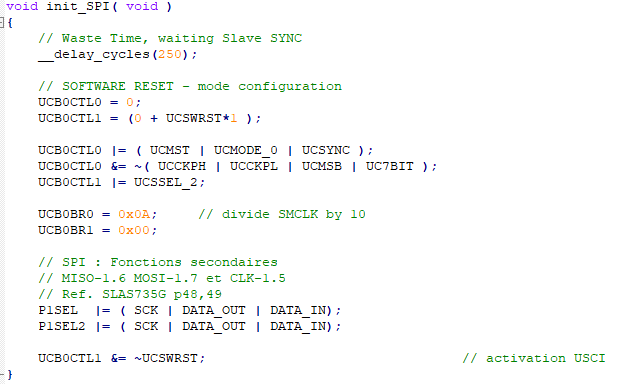
Sur ce schéma il est important de noter que les deux cartes doivent être relier sur leur 5v pour les alimenter et de relier les masses pour obtenir une masse commune.

## Schéma

Ici, tout est alimenté en 3.3V via l’USB du PC. Toutes les masses sont communes.

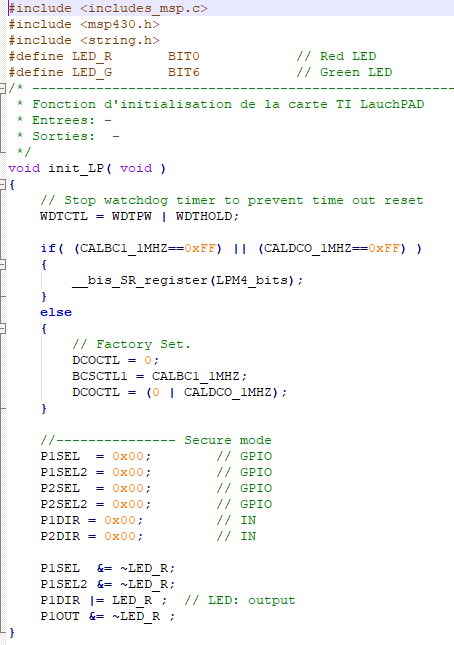
## Découpage fonctionnel

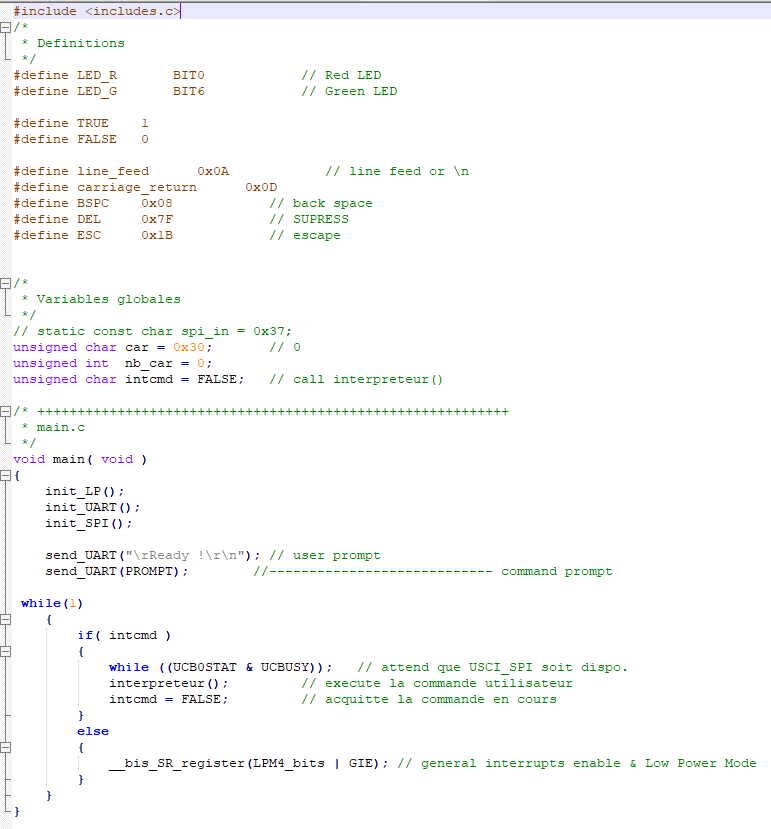
Master :

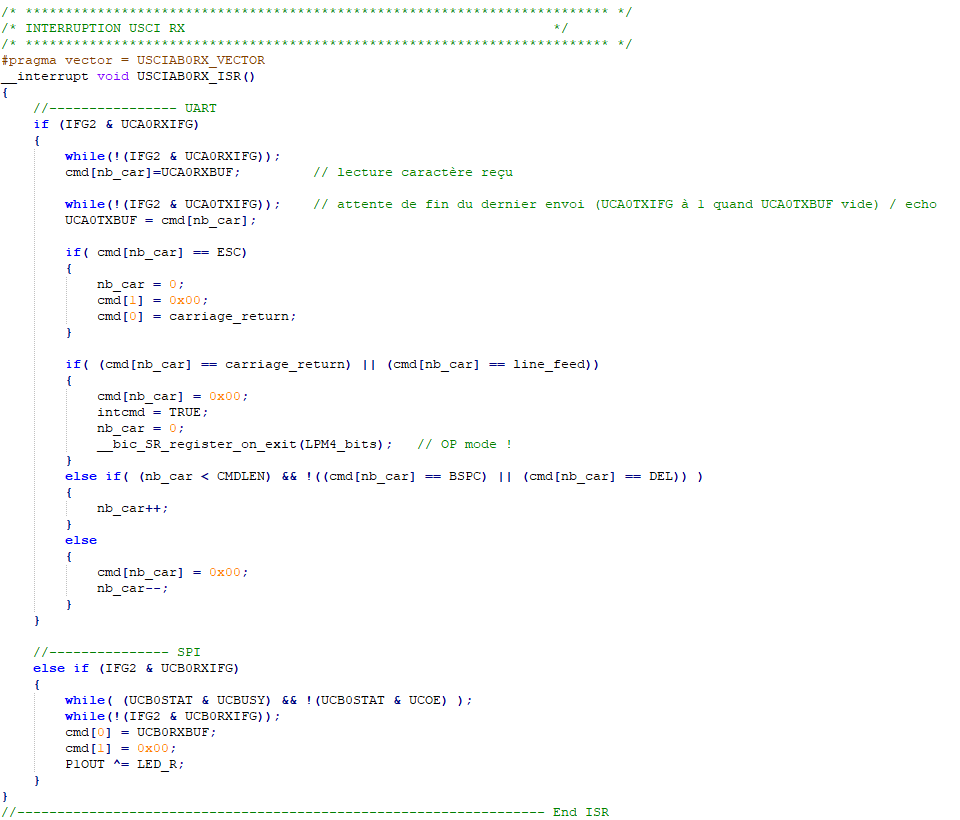
* UART : Fichier gérant la configuration de l’UART et son fonctionnement
  + **void init\_UART( void )** : initialisation de l’UART
  + **void send\_UART(unsigned char \*msg)** : envoie une chaîne de caractères via l’UART
* Interpreteur : Interpreteur de commande : transfert via SPI ou répond via UART selon la commande reçue via UART
  + **void interpreteur( void )**: renvoie un message d’acknowledge via UART et transmet la commande au slave via SPI si besoin de transmettre.
* SPI\_Master : Fichier gérant la configuration du SPI et son fonctionnement
  + **void init\_SPI( void )**: Initialisation du SPI en master sur les pins 1.5 ; 1.6 et 1.7
  + **void Send\_SPI(unsigned char carac)**: envoie d’un seul caractère au slave via SPI

Une image contenant texte, intérieur, capture d’écran

Description générée automatiquement

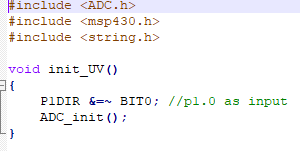
* Init\_LP : Fichier d’initialisation de la launchpad
  + **void init\_LP( void )** : Initlialisation de la launchpad
* 2553\_MASTER : fichier contenant la fonction main du MASTER
  + **void main( void )**: Comportement du projet





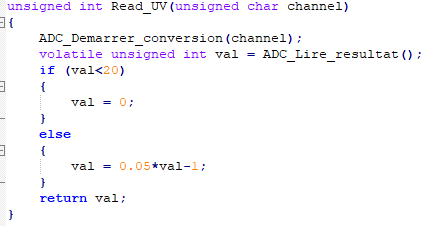
Slave :

* ADC : Bibliothèque gérant l’ADC (fournie par l’ESIGELEC donc non-détaillée ci-dessous)
* UV\_sensor : Fichier gérant l’initialisation et la récupération des données du capteurs UV
  + **void init\_UV( void )** : Paramétrage de l’ADC sur le pin du capteur UV



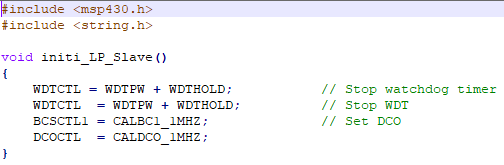
On appelle la fonction d’initialisation de l’ADC après avoir mis le pin 1.0 en entrée.

* + **unsigned int Read\_UV(void)**: Lecture de la valeur de l’ADC et conversion en indice UV pour la valeur retournée

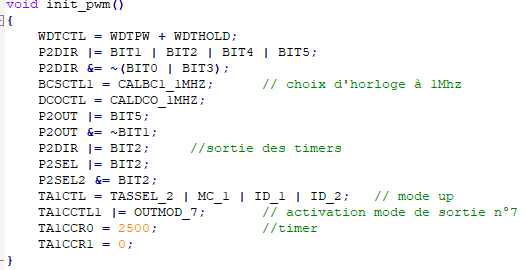


On utilise ici un volatile unsigned int afin de pouvoir débugger le programme. Autrement, CCS fait des optimisations et la valeur n’est jamais réellement stockée.

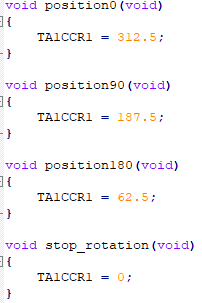
* Init\_LP :
  + **void init\_LP\_Slave( void ) :** Initialisation de la launchpad



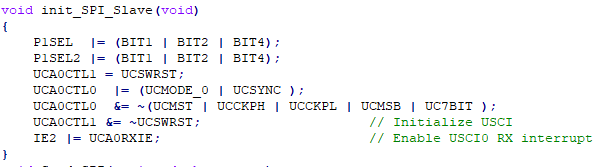
* PWM**:** Initialisation et gestion de la PWM pour le servomoteur
  + **void init\_pwm( void )** : Initialisation de la PWM pour le servomoteur



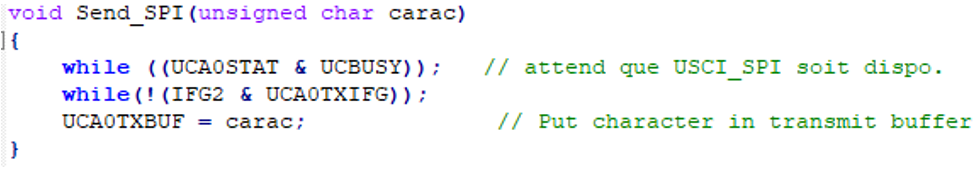
* + **void position***’X’* **(void)**: Modification de la PWM en fonction de la position demandée
  + Ici la valeur entrée dans TA1CCR1 permette grâce au paramétrage de TA1CTL(mode up, prédiviseur de 8, horloge à 1MHz et TA1CCR0=2500) de faire varier le rapport cyclique pour avoir un signal à 50hz avec des rapports cyclique de 0.025 ; 0.075 et 0.125 permettant de piloter le servo-moteur.



* SPI\_Slave : Fichier gérant la configuration du SPI
  + **void init\_SPI\_Slave ( void )**: Initialisation du SPI en slave sur les pins 1.4 ; 1.1 et 1.2 pour pouvoir laisser disponibles les pins 1.5 ; 1.6 et 1.7 pour l’I2C.



* + **void Send\_SPI(unsigned char carac)**: envoie d’un seul caractère au MASTER via SPI



* 2553\_SLAVE : fichier contenant la fonction main du SLAVE
  + **void main( void )**: Comportement du projet



* I2C : Fichier gérant la configuration et la communication I2C

Malgré des heures de recherches et de tentatives, nous ne sommes pas parvenus à faire marcher le bus I2C entre le MSP430G2553 et le capteur Pmod HYGRO. Nous avons tenté plusieurs approches. La première a été de lire les datasheets des composants ainsi que des cours en ligne sur l’I2C, et d’essayer de construire une solution à partir de 0. Nous n’avons malheureusement pas réussi à avoir quoique ce soit de fonctionnel. Nous avons ensuite tenté de trouver un exemple de code I2C pour un MSP430G2553 sur internet, mais aucunes des solutions que nous avons trouvées n’a fonctionné. Cette partie étant celle sur laquelle nous avons le plus passé de temps, nous sommes frustrés de ne pas avoir réussi, mais également satisfait d’avoir appris plus en détail le fonctionnement de ce bus de communication.

# Tests Réalisés

Dans le but d’obtenir un code ayant une qualité permettant de rendre sa compréhension simple et de permettre une bonne maintenabilité de ce code, nous allons tester les fonctions du code réalisé séparément.

L’objectif est donc de tester les fonctions de contrôle des capteurs et des actionneurs.

Nous testerons donc les fonctions suivantes :

* Contrôle du capteur UV
* Contrôle du servomoteur
* Communication UART
* Communication SPI

## Environnement de test

Nous testerons chacune des fonction grâce à code composer studio (éventuellement le mode débogage de code composer studio) ainsi qu’un terminal PUTTY

* Chargement du programme adapté à chaque test dans chacun des deux MSP.
* Un terminal PUTTY ouvert et connecter au master en UART pour envoie et réception des commandes.
* Observation des modifications de position du servo-moteur.
* Mode débogage avec un break point pour vérifier la valeur contenue dans la variable de stockage de l’indice d’UV mesuré

## Lecture capteur UV

La fonction à tester ici est Read\_UV. On lui passe en paramètre la voie analogique sur laquelle se trouve le capteur, et elle retourne un indice UV donné par le calcul présent dans le code si la valeur lue dépasse un certain seuil.

On attend un entier comme valeur de « val », défini comme volatile afin de pouvoir l’observer avec un breakpoint.   
On envoie ‘0’ en paramètre de la fonction pour réaliser une conversion analogique-numérique sur la voie P1.0. On lit bien la valeur du port analogique (fonction testée avec un branchement de P1.0 sur le 3.3V de la Launchpad car le capteur UV en intérieur ne renvoie qu’un très faible signal).

On récupère bien un entier en sortie.

Une image contenant texte

Description générée automatiquement

## Commande servomoteur

Ici, les fonctions à tester sont les fonctions de pilotage du servomoteur.

Une image contenant texte

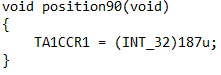
Description générée automatiquement

On observe que le servomoteur se place à 0°.

Une image contenant objets métalliques

Description générée automatiquement

Ici la fonction position0



On observe que le servomoteur se place à 90°.



Ici la fonction position90

Une image contenant texte

Description générée automatiquement

On observe que le servomoteur se place à 180°.

Une image contenant intérieur, accessoire

Description générée automatiquement

Ici la fonction position180

On note que toutes les transitions de positions marchent.

* De 0° à 90°
* De 90° à 0°
* De 0° à 180°
* De 180° à 0°
* De 90° à 180°
* De 180° à 90°

On test également la fonction d’arrêt du servomoteur :

Une image contenant texte

Description générée automatiquement

Cette fonction arrête la rotation du moteur s’il est en rotation. Fonctionne avec toutes les combinaisons listées ci-dessus.

## Test des fonctions présentes dans la classe UART

### Description

Ces fonctions permettent une communication avec un bus UART 8 bit sans parité avec un débit de 9600 bauds.

La classe UART comporte les fonctions suivantes :

* init\_UART
* send\_UART

La fonction init\_UART ne sera pas testée de manière individuelle et sera considéré comme fonctionnelle si la fonction send\_UART est fonctionnelle.

Lors de ces tests on testera la fonction send\_UART avec un terminal PUTTY, considérant que l’utilisation d’une terminale bluetooth n’influe pas sur le comportement de cette fonction.

### Câblage réalisé

Il n’y a pas de câblage à réaliser car TX/RX étant rediriger vers l’USB de la launchpad.

### Test unitaire de la fonction uart\_send\_byte

### Résultat attendu

Description de la fonction send\_UART :

* Cette fonction envoie un tableau de caractère codé sur 8 bits via le bus UART
* Le protorype de cette fonction est :
  + Void send\_UART(unsigned char \*msg)
* Aucun retour n’est attendu par cette fonction
* Apparition du text envoyé sur le terminal PUTTY.

### Procédure de test

Pour tester cette fonction on va appeler la fonction Send\_UART dans la boucle principale de notre master en lui passant en paramètre un message et observé le résultat obtenu via PUTTY :

Une image contenant texte

Description générée automatiquement

### Résultat obtenu

Après exécution de notre programme, nous obtenons le résultat suivant sur le terminal PC



On peut donc voir que cette fonction envoie correctement les données via le bus UART.

## Test de le communication SPI

### Description

Ces fonctions permettent une communication par un bus SPI 3 fils avec une horloge à 1 MHz.

Les classe SPI des MSP comporte les fonctions suivantes :

* init\_SPI
* Send\_SPI

Les fonction init\_SPI ne seront pas testée de manière individuelle et seront considéré comme fonctionnelle si la fonction send\_SPI est fonctionnelle et que les informations transitent correctement.

### Câblage réalisé

Coté master :

* Horloge out broche 1.5
* MOSI broche 1.6
* MISO broche 1.7

Coté slave :

* MISO 1.1
* MOSI 1.2
* Horloge in 1.4

### Test

### Résultat attendu

Description de la fonction send\_SPI coté master :

* Cette fonction envoie un caractère codé sur 8 bits via le bus SPI
* Le protorype de cette fonction est :
  + Void send\_SPI(unsigned char carac)
* Aucun retour n’est attendu par cette fonction
* Rotation du servo-moteur.

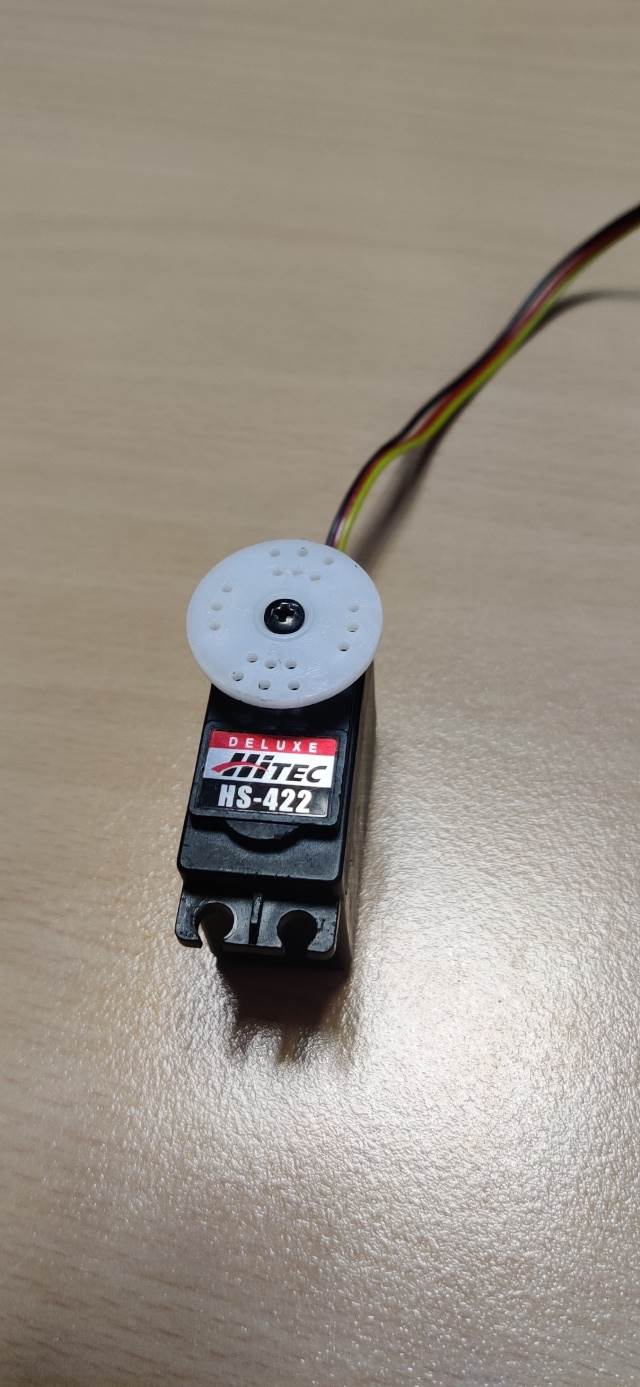
Description de la fonction send\_SPI coté slave :

* Cette fonction envoie un caractère codé sur 8 bits via le bus SPI
* Le protorype de cette fonction est :
  + Void send\_SPI(unsigned char carac)
* Aucun retour n’est attendu par cette fonction
* Affichage du text dans la fenêtre PUTTY.

### Procédure de test

Lors de ces tests on testera la fonction send\_SPI entre le master et le slave en faisant bouger le servo-moteur sur réception des caractère ‘1’, ‘2’ et ‘3’ (position 0°, 90° et 180°) avec le terminal PUTTY. Puis nous testerons d’envoyer une réponse au master via le SPI et de l’afficher dans PUTTY.

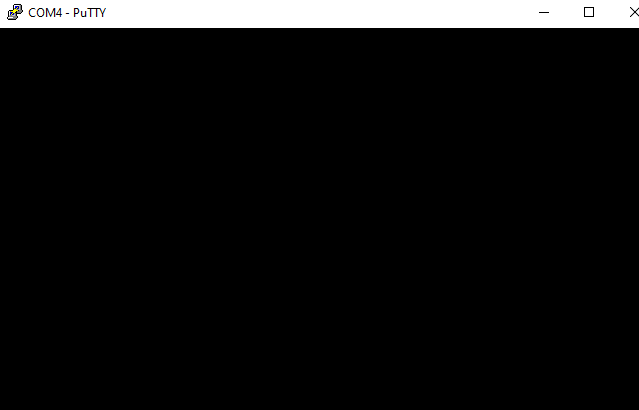
### Résultat obtenu



Envoie de ‘1’ Envoie de ‘2’ Envoie de ‘3’

Après avoir réalisé les tests nous constatons que le servo moteur change bien de position, la communication est bien fonctionnelle dans le sens master vers le slave.

Malheureusement nous constatons aussi que cette communication ne s’effectue correctement que dans le sens master vers slave car nous n’obtenons aucune réponse sur le terminal putty.



Après avoir cherché le problème rencontré, nous constatons que la fonction send\_SPI du notre slave ne fonctionne pas correctement et cela car nous n’arrivons pas à affecter une valeur à notre buffer de communication SPI.

Nous ne sommes malheureusement pas parvenus à faire fonctionner la communication SPI dans ce sens.

# Conclusion

Ce projet nous a permis de découvrir l’importance de l’échange de l’information par des bus de communication au sein d’un projet tel que celui qui nous a été donné. L’expérience que nous avons retiré de ce projet nous aidera dans le cadre de nos activités professionnelles aussi bien à court terme (étant tous deux alternant) que sur le long terme en nous servant de base afin de nous améliorer.

Grâce à l’ensemble des cours et des TP que nous avons pu avoir concernant le Bus de Communication, nous avons pu acquérir de nouvelles connaissances sur des logiciels ainsi que sur du matériel que nous n’avions encore jamais utilisé. Ce projet a été pour nous une opportunité de travailler dans une équipe nouvellement formée sur un projet interdisciplinaire. Cela nous a donc permis de profiter du meilleur des connaissances de chacun et d’apprendre les uns des autres. Au fil des séances, nous avons pu consolider nos acquis aussi bien en programmation en langage C qu’en méthodes et outils pour la qualité logicielle.

De plus nous tenons à remercier l’ensemble des enseignants de Bus de communication et de MOQL qui ont su nous expliquer de manière très pédagogique ce que nous ignorions jusqu’alors.

Au début de ce projet il nous a fallu un peu de temps pour comprendre le fonctionnement de la programmation en utilisant Code composer studio et surtout les subtilités de la programmation sur microcontrôleur car ces notions nous étaient complètement étrangères. Cependant grâce à la méthodologie qui nous a été apprise au travers des différentes séances, nous sommes parvenus à nous en sortir, cela nous a donc donné la chance d’en apprendre davantage en matière de logiciels de programmation et d’approfondir nos connaissances.