Rapport ISE-OC : Serre Connectée

Qualité logicielle – bus de communication

Corentin Bouynot – Tristan sannier

2021

# Table des matières

[Table des matières 1](#_Toc68100461)

[1. Cahier des charges 2](#_Toc68100462)

[1.1 Présentation du projet 2](#_Toc68100463)

[1.2 Les Besoins : 2](#_Toc68100464)

[1.3 Les contraintes : 2](#_Toc68100465)

[1.4 Liste des fonctions à réaliser : 3](#_Toc68100466)

[1.5 Liste des entrées et sorties que devra gérer la serre connectée 3](#_Toc68100467)

[2. Etude préalable 4](#_Toc68100468)

[2.1. Découpage fonctionnel 6](#_Toc68100469)

[3. Tests unitaires 7](#_Toc68100470)

[4. Rapport de tests 8](#_Toc68100471)

# 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.

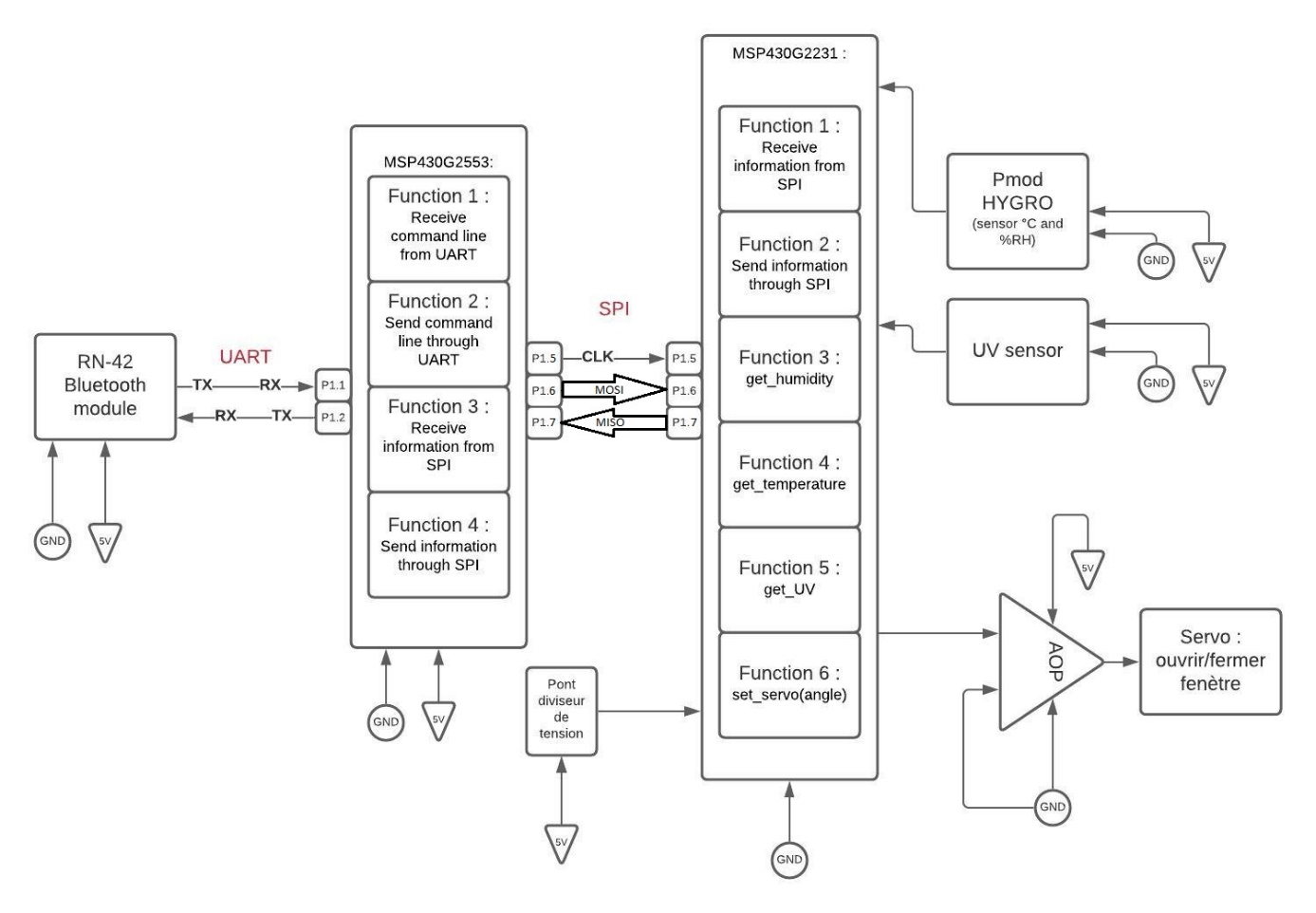
Du coté MSPG2231 nous avons choisi de définir les points suivants concernant la communication

* 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.

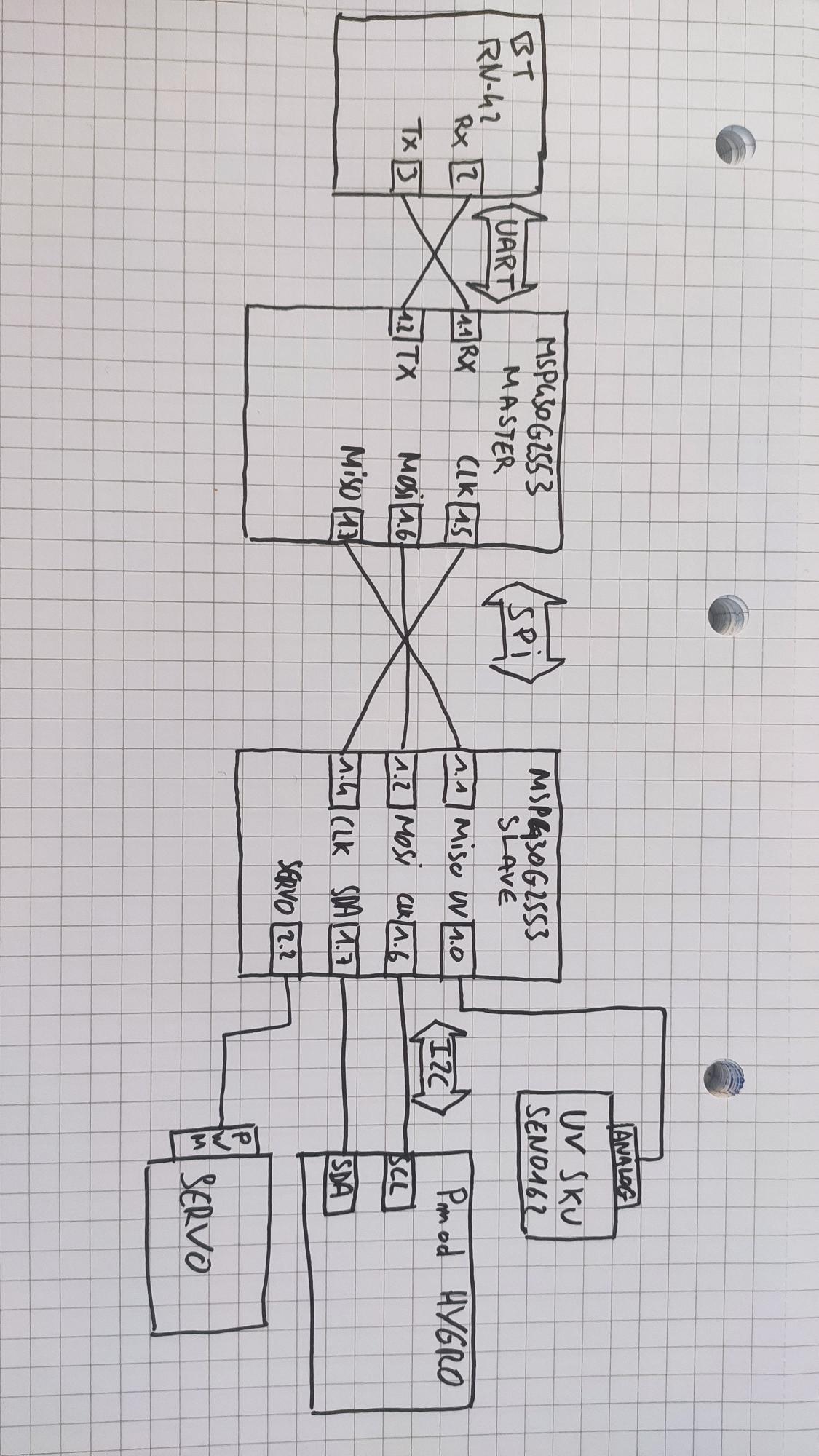
Ainsi le MSP430G2553 le master de deux slaves, cependant cela modifierait nôtre diagramme fonctionnel de la manière suivante : le MSP430G2553 sera master du MSP430G2231 et master de HYGRO.

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.

## 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
  + **void Send\_SPI(unsigned char carac)**: envoie d’un seul caractère au slave via SPI
* 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
  + **unsigned int Read\_UV(void)**: Lecture de la valeur de l’ADC et conversion en indice UV pour la valeur retournée
* Init\_LP :
  + **void init\_LP\_Slave( void ) :** Initialisation de la launchpad
* Servomoteur**:** Initialisation et gestion de la PWM pour le servomoteur
  + **void init\_pwm( void )** : Initialisation de la PWM pour le servomoteur
  + **void set\_servo (unsigned char pos)**: Modification de la PWM en fonction de la position demandée
* SPI\_Slave : Fichier gérant la configuration du SPI
  + **void init\_SPI\_Slave ( void )**: Initialisation du SPI en slave
* 2553\_SLAVE : fichier contenant la fonction main du MASTER
  + **void main( void )**: Comportement du projet
* I2C : Fichier gérant la configuration et la communication I2C
  + **Void init\_I2C (void)**: initialisation et configuration de l’I2C

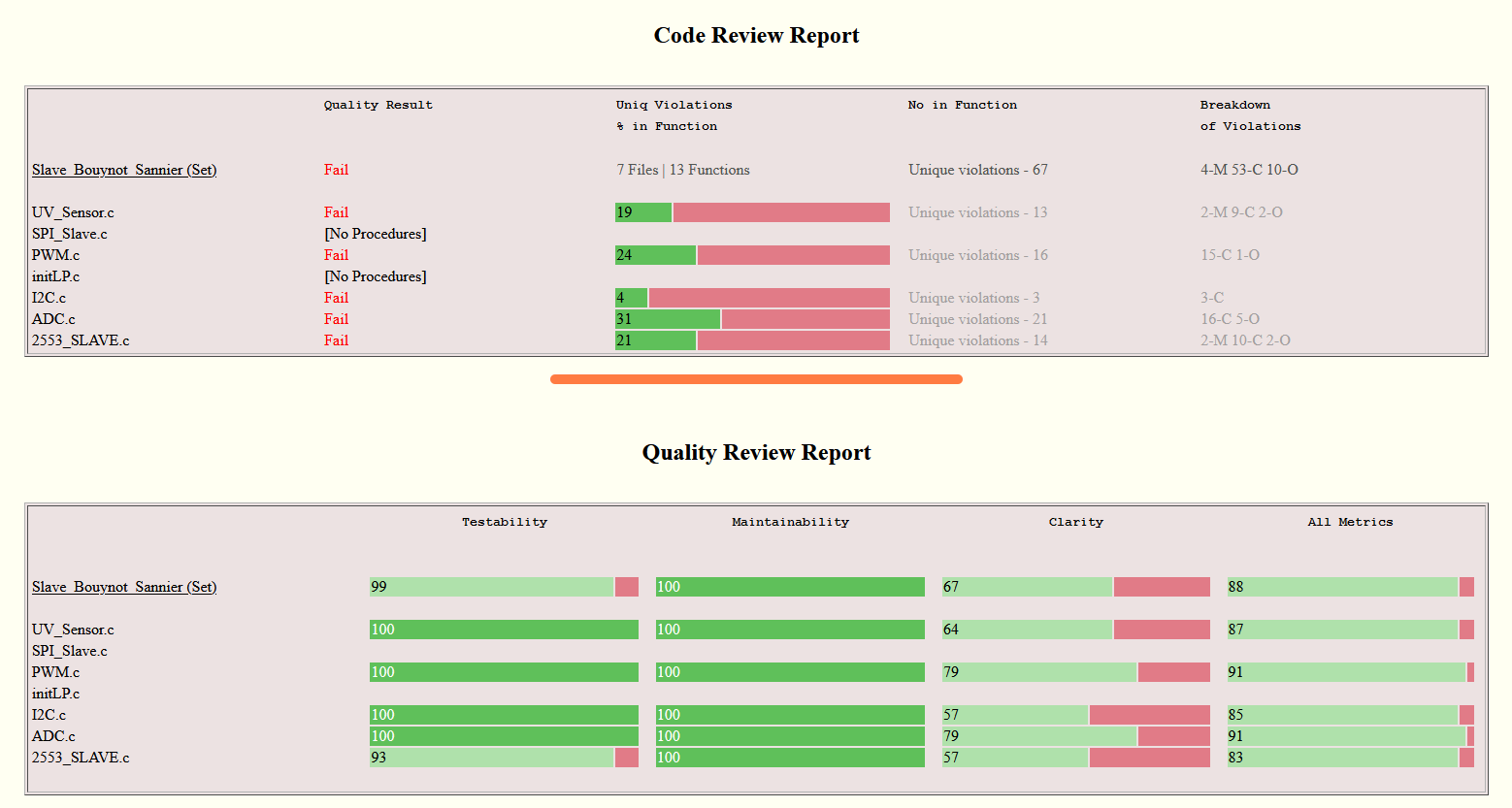
# Tests unitaires

On test le projet avec un terminal Bluetooth sur téléphone.

|  |  |  |  |
| --- | --- | --- | --- |
| Objectif | Entrée | Sortie attendu | Sortie observée |
| Mesure UV | u | Entier (ex : 0) | Renvoie de la valeur UV relevée. |
| Arrêt rotation servo | 0 | Arrêt de la rotation du servo en cours | Arrêt de la rotation du servo en cours |
| Position du servo à 0° | 1 | Position du servo à 0° (TA1CCR1 = 312) | Position du servo à 0° (TA1CCR1 = 312) |
| Position du servo à 0° | 2 | Position du servo à 90°  (TA1CCR1 = 187) | Position du servo à 90°  (TA1CCR1 = 187) |
| Position du servo à 0° | 3 | Position du servo à 180°  (TA1CCR1 = 63) | Position du servo à 180°  (TA1CCR1 = 63) |
| Mesure température et humidité | t ou h | Température (°C)  Humidité (%) | Non testée |
|  |  |  |  |

# Rapport de tests

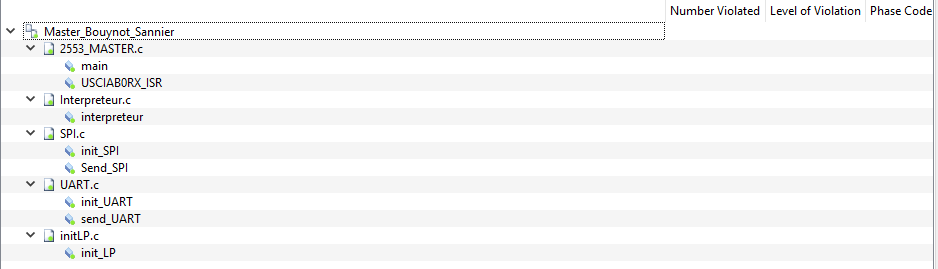
Pour commencer cette partie, nous avons passé nôtre code sous l’Analyse statique de LDRA voici les extraits des rapport obtenus :

* Coté Master :
* Coté Slave :

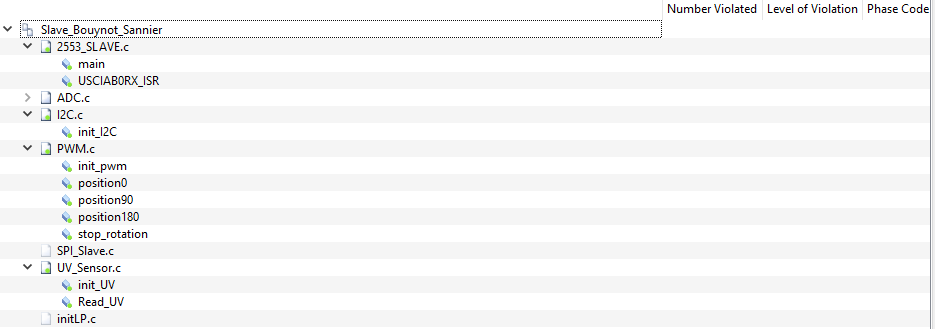
Grâce à ces deux premiers rapports nous avons pu nous faire une idée de l’état général de nôtre code en termes de qualité, de lisibilité et de maintenabilité. De ce fait nous en concluons qu’avant modification nôtre code avait un nombre assez conséquent de normes à appliquer afin de correspondre au standard que nous avons créer (basé sous MISRA 2012). Pour créer ce standard nous nous somme donc basé sur du MISRA 2012 auquel nous nous sommes permis de retirer quelque règle tels que les warnings surs : les opérations bit à bit, les égalités sur des booléens ainsi que la présence de ligne de commentaires avec des « // ». Malgré tout nous pouvons aussi constater que la maintenabilité, la testabilité et la clarté de nos codes obtenaient des valeurs correctes pour un premier passage sous LDRA.

Nous nous sommes par la suite mis à corriger nôtre code en suivant les recommandations de LDRA pour finir par nous retrouver avec une analyse de code tels que visible sur les captures ci-dessous :

* Coté master :



* Coté slave :



Tel qu’il est possible de la voir LDRA ne nous donne plus d’erreurs à corriger. Cependant, malgré cela, le rapport de tests nous indiqué un nombre d’erreurs n’ayant que très sensiblement évolué coté slave et plus grand encore coté master qu’avant les corrections apportées. Nous avons donc pensé à un possible conflit avec d’autres projet LDRA, cependant nous n’avons à ce jour pas trouvés de solution à ce problème.