

Rapport de stage chez HORIBA



Solution de remplacement du MC68HC11 sur le Pentra 80

Maître de stage : Christophe Domergue

Tuteur de stage : Thomas HAETTEL

BUT Informatique

Déploiement d'Applications Communicantes et Sécurisées (DACS)

Remerciements

Je tiens à exprimer ma gratitude envers l'entreprise HORIBA pour m'avoir accueilli durant mon stage de deuxième année de BUT informatique. Un merci particulier à Sofiane Megdoud, responsable métier, et à Christophe Domergue, développeur embarqué, pour leur accompagnement, leurs conseils et leur encadrement tout au long de ce projet.

Je souhaiterais également remercier l'équipe d'électroniciens qui ont contribué de près ou de loin à la réussite de mon projet : Joël Alazetta pour m'avoir appris à souder ainsi que Bruno Canillos pour son support.

Un grand merci à l'équipe de développeurs embarqués, notamment Théo Rovelli et Grégoire Barbon, pour m'avoir partagé leur expertise dans le métier.

De manière plus générale, je tiens à exprimer ma reconnaissance envers l'ensemble des employés de l'entreprise pour leur bienveillance et leurs conseils, qui m'ont permis de m'intégrer pleinement au sein de l'entreprise.

Enfin, je remercie chaleureusement l'équipe pédagogique de l'IUT de Montpellier, sans qui mon parcours en tant que développeur n'aurait pas été possible.

Résumé

Ce document est un rapport de stage effectué au sein de l'entreprise HORIBA et rédigé par un étudiant de l'IUT de Montpellier dans le cadre de sa deuxième année d'études. Il présente le travail effectué durant la période de stage, à la fois dans le développement logiciel et le maquettage électronique.

Ce rapport parle de la thématique abordée durant le stage :

- ❖ La solution de remplacement d'un microcontrôleur devenu obsolète, le MC68HC11, par une nouvelle technologie de microcontrôleur, le STM32G0B1RE, sur une machine produite par HORIBA Medical depuis maintenant 30 ans, le Pentra80.

Ce projet contient deux parties d'un point de vue métier :

- Une partie développement : Portabilité d'un code existant en assembleur vers un code en C reproduisant les fonctionnalités à l'identique.
- Une partie électronique : Maquettage de la solution de remplacement et interfaçage de celle-ci avec la carte mère du Pentra80.

Mots clés : HORIBA, développement embarqué/embedded, langage C/assembleur, maquettage électronique, microcontrôleur, automate, Git, hématologie, portabilité

Sommaire

Remerciements.....	2
Résumé.....	3
Sommaire.....	4
Table des figures.....	6
Glossaire.....	7
1 - Introduction.....	1
1.1 - Présentation de l'entreprise.....	1
1.2 - Automate d'hématologie Pentra 80.....	2
1.2.1 - Présentation.....	2
1.2.2 - Principe de mesure des cellules.....	3
1.3 - Contexte.....	4
1.3.1 - Contexte hématologique.....	4
1.3.2 - Contexte technique.....	5
2 - Définition du stage.....	6
2.1 - Objectif du stage.....	6
2.2 - Analyse de l'existant.....	7
2.3 - Analyse de l'environnement technique.....	8
2.4 - Cahier des charges.....	10
3 - Rapport technique.....	11
3.1 - Architecture électronique.....	11
3.1.1 - Maquettage.....	11
3.1.2 - Configuration du STM32.....	13
3.2 - Arborescence du projet.....	14
3.3 - Fonctionnement du bus SPI.....	15
3.4 - Etudes d'anomalies potentielles.....	17
3.4.1 - Etude de l'anomalie du Chip Select.....	17
3.4.2 - Etude de l'anomalie de gestion de la mémoire.....	18
3.5 - Développement de la partie INIT / BOOT.....	21
3.5.1 - Développement du protocole de synchronisation.....	21
3.5.2 - Développement du protocole de téléchargement.....	22
3.6 - Statut d'acquisition.....	24
3.6.1 - Précisions sur le fonctionnement du SPI.....	24
3.6.2 - Fonctionnement du statut d'acquisition.....	25
3.7 - Conception logicielle de la partie acquisition.....	28
3.7.1 - Types de commandes.....	28
3.7.2 - Communicating commands.....	30
3.7.3 - Processing commands.....	32
3.7.4 - Gestion de la commande RESET.....	34
3.8 - Principes de comptage.....	36
3.8.1 - Types de comptage.....	36
3.8.2 - Protocole du comptage 2D.....	37

3.9 - Comptage 2D.....	40
3.9.1 - Environnement de travail.....	40
3.9.2 - Développement de la fonctionnalité.....	41
3.10 - Fonctionnement du Timer.....	44
3.11 - Timer de comptage.....	45
3.11.1 - Configuration du Timer.....	45
3.11.2 - Interruptions Timer sur front Montant / Descendant.....	46
3.11.3 - Interruptions Timer sur ‘update event’.....	49
3.12 - Compteur immédiat.....	51
3.12.1 - Configuration du Timer.....	51
3.12.2 - Fonctionnement du compteur immédiat.....	52
3.13 - Fonctionnement de l'ADC.....	54
3.13.1 - Configuration de l'ADC.....	54
3.13.2 - Pont diviseur de tension.....	55
3.13.3 - Fonctionnement de l'ADC.....	56
3.14 - Envoi des résultats du comptage 2D.....	57
3.15 - Tests du comptage 2D.....	59
3.15.1 - Tests sur table.....	59
3.15.2 - Environnement de test sur machine.....	61
3.15.3 - Tests sur machine.....	62
3.15.4 - Résultats sur machine.....	63
3.16 - ADJUST 2D.....	64
3.16.1 - Fonctionnement ADJUST 2D.....	64
3.16.2 - Code ADJUST 2D.....	65
4 - Apprentissages critiques mobilisés.....	67
5 - Méthodologie et organisation du projet.....	68
6 - Conclusion.....	70
6.1 - Bilan du travail réalisé.....	70
6.2 - Perspectives.....	70
6.3 - Bilan des acquis techniques.....	71
6.4 - Bilan de l'expérience en entreprise.....	72
6.5 - Bilan de l'expérience personnelle.....	73
7 - Bibliographie.....	74
8 - Annexes.....	75

Table des figures

- Figure n°1 :** Automate d'hématologie Pentra 80
- Figure n°2 :** Principe de mesure de COULTER
- Figure n°3 :** Chaîne d'analyse hématologique
- Figure n°4 :** Rôle du MC68HC11
- Figure n°5 :** Agencement des HC11 en fonction du Maître
- Figure n°6 :** États du MCU
- Figure n°7 :** Maquette d'un des 4 esclaves STM32
- Figure n°8 :** Architecture logicielle
- Figure n°9 :** Transfert des données full duplex
- Figure n°10 :** Exemple de communication sur le bus SPI
- Figure n°11 :** Intervalle de relâchement du Chip Select
- Figure n°12 :** STM32F072RB connecté à un STM32G0B1RE
- Figure n°13 :** Big / Little Endian
- Figure n°14 :** Octets reçus analysés au débogueur
- Figure n°15 :** Résultat des tests Little / Big Endian
- Figure n°16 :** Envoi des octets READY et START
- Figure n°17 :** Commande BLOCK_SIZE
- Figure n°18 :** Echange de 0x15ED octets
- Figure n°19 :** DSI du fonctionnement des interruptions SPI
- Figure n°20 :** Visualisation du test de performance
- Figure n°21 :** Réception d'une commande
- Figure n°22 :** DSI du gestionnaire de commande
- Figure n°23 :** DSI des "Processing Command"
- Figure n°24 :** Cuve optique
- Figure n°25 :** Visualisation de la réception d'une impulsion
- Figure n°26 :** Environnement de test
- Figure n°27 :** Fonctionnement d'une tranche de comptage
- Figure n°28 :** Réception d'une impulsion sur input capture
- Figure n°29 :** Configuration du Timer en horloge externe
- Figure n°30 :** Compteur immédiat versus input capture
- Figure n°31 :** Configuration de l'ADC sur 8 bits
- Figure n°32 :** Schéma électronique du pont diviseur de tension
- Figure n°33 :** Envoi de la taille du paquet
- Figure n°34 :** Analyse de la commande 1 suivie de 33
- Figure n°35 :** Environnement de laboratoire
- Figure n°36 :** Mallette de transport du sang
- Figure n°37 :** Analyse du comptage 2D sur machine
- Figure n°38:** Versions de test
- Figure n°39 :** Résultats d'un Yumizen
- Figure n°40 :** Résultats du Pentra 80 avec la solution de remplacement
- Figure n°41 :** Interruption hardware levée sous ADJUST 2D
- Figure n°42 :** Visualisation de plusieurs tranches de comptage
- Figure n°43 :** Activités en entreprise

Glossaire

Microcontrôleur (i.e MCU) : Le microcontrôleur est un circuit électronique programmable combinant un processeur, de la mémoire et des périphériques d'entrée/sortie.

MC68HC11 (i.e HC11) : Le MC68HC11 est une famille de microcontrôleurs 8 bits développée par Motorola en 1984. Ce sont ici les microcontrôleurs que l'on va remplacer par les STM32. Il sera aussi identifié en tant qu'"Esclave" lors de communications SPI.

STM32 (i.e STM32G0B1RE) : La famille STM32 regroupe plusieurs séries de microcontrôleurs 32 bits réalisés par STMicroelectronics. Les STM32 sont réputés pour leurs performances, leur faible consommation d'énergie et leur polyvalence. Il sera aussi identifié en tant qu'"Esclave" lors de communications SPI.

Pentra 80 : Le Pentra 80 est un analyseur hématologique développé par HORIBA Medical. La solution de remplacement des HC11 est testée sur cette machine.

SPI (Serial Peripheral Interface) : Un bus de communication série utilisé pour la communication de données entre un microcontrôleur et divers périphériques. Le SPI fonctionne en mode maître-esclave, où le maître contrôle la communication et les esclaves répondent aux commandes. Les principaux signaux de l'interface SPI comprennent **MISO** (Master In Slave Out), **MOSI** (Master Out Slave In), **SCK** (Serial Clock) et **SS** (Slave Select) ou **CS** (Chip Select).

MC68331 i.e (68331) : Un microcontrôleur 32 bits conçu par Motorola. Dans le contexte du projet, il est le microcontrôleur principal de la carte mère du Pentra 80, il sera identifié en tant que "Maître" lors de communications SPI.

CPLD : Pour "Complex Programmable Logic Device". Il est constitué de plusieurs blocs logiques configurables appelés macrocells, qui peuvent être programmés pour réaliser des fonctions logiques complexes. Les CPLD utilisent une mémoire non volatile, ce qui leur permet de conserver leur configuration même après une mise hors tension.

ADC (Analog-to-Digital Converter) : Un convertisseur analogique-numérique qui transforme un signal analogique (tel qu'une tension) en une valeur numérique compréhensible par le microcontrôleur.

Timers / Temporiseurs : Des périphériques matériels intégrés dans les microcontrôleurs STM32, utilisés pour générer des événements temporisés et mesurer le temps écoulé. Les timers peuvent être utilisés pour diverses applications,

telles que la génération de signaux PWM, la capture de signaux externes, la mesure de durée et la gestion des tâches périodiques.

DMA (Direct Memory Access) : Une fonctionnalité permettant le transfert direct de données entre la mémoire et les périphériques sans intervention du processeur central.

Interrupts / Interruptions : Des mécanismes permettant de suspendre l'exécution normale du programme pour traiter des événements spécifiques. Les interruptions peuvent être générées par des périphériques internes (comme les timers, le bus SPI..) ou externes (comme les boutons ou les capteurs).

PWM (Pulse Width Modulation) : est une technique utilisée pour contrôler la puissance délivrée à des appareils électriques. Elle fonctionne en modulant la largeur des impulsions dans un signal, permettant ainsi de réguler la vitesse des moteurs, la luminosité des LED, et d'autres applications similaires.

AOP (Amplificateur Opérationnel) : C'est un composant électronique polyvalent utilisé pour amplifier des signaux électriques. On l'utilise dans beaucoup d'appareils, comme les radios, les télévisions et les ordinateurs, pour améliorer le son ou l'image.

Maître / Esclave : Nous parlerons de nombreuses fois d'une relation Maître / Esclave représentant la hiérarchie du bus SPI. Nous associons ici le Maître à 68331 et l'esclave au HC11. Il faut donc bien penser tout du long du rapport à associer ces appellations aux bonnes composantes pour comprendre le plein sens des phrases. Il est plus facile de parler au sens hiérarchique qu'avec un numéro de série.

Acquisition : Lorsque l'on parle d'acquisition, on parle ici de la partie qui analyse le sang. Cela regroupe la qualification des cellules, le traitement ainsi que le tri de la donnée mesurée.

GBF : Pour **Générateur de Basses Fréquences**, est un appareil utilisé en électronique pour tester ou dépanner des équipements. Il permet de produire des signaux électriques de différentes formes (sinusoïdales, carrées, triangulaires) à des fréquences spécifiques. Ces signaux peuvent être observés et analysés à l'aide d'un oscilloscope.

1 - Introduction

1.1 - Présentation de l'entreprise

HORIBA Medical est la branche biomédicale de la société japonaise HORIBA. Située à Montpellier, elle conçoit, développe et distribue dans le monde entier des automates de diagnostic in vitro dans les domaines de l'Hématologie et de la Chimie Clinique. Ces instruments contribuent dès aujourd'hui à préparer la santé de demain.

Créée en 1983, l'entreprise française ABX est devenue une filiale du groupe HORIBA depuis 1996. De 1983 à 1990 l'entreprise connaît une croissance rapide grâce au succès commercial d'un de ses appareils première génération, l'ABX Minos.

En 2008 une nouvelle unité de production de réactifs chimiques avec un investissement de 8 millions d'euros est créée.

Le bâtiment de 4 400 mètres carrés, qui s'ajoute aux 13 000 mètres carrés du site du parc Euromédecine, permet au fabricant de doubler sa capacité de production annuelle de réactifs, la faisant passer à 13 000 tonnes. La fabrication et la commercialisation de réactifs représentent déjà 50 % du chiffre d'affaires de l'entreprise et, plus important encore, une source majeure de croissance de son activité et de son résultat net. HORIBA ABX est aujourd'hui le premier producteur mondial d'automates d'analyse hématologique.

Le groupe est présent dans environ 25 000 laboratoires dans le monde et compte des usines et des centres de recherche en France (Montpellier), aux USA (Los Angeles), au Japon (Kyoto) et au Brésil (São Paulo). De plus, HORIBA ABX possède un vaste réseau de distribution qui comprend 5 agences en Europe (France, Grande-Bretagne, Italie, Espagne et Portugal), 3 filiales (Etats-Unis, Brésil et Pologne) et enfin 75 distributeurs.

Pour fidéliser ses clients, la société développe aussi des automates de biochimie depuis 1998, un secteur de développement en expansion. HORIBA ABX emploie 700 personnes, dont 450 en France, 200 emplois créés en 2 ans et des certifications ISO-9001 pour Montpellier et ISO-9002 pour le Brésil.

La force de HORIBA ABX est basée sur le savoir faire et à la polyvalence du groupe ou l'on peut trouver des équipes mixtes composés d'ingénieurs en optique, en informatique, en développement, en mécanique, en électronique, en biologie, en marketing et des médecins consultant dans le monde entier.

1.2 - Automate d'hématologie Pentra 80

Dans le but de caractériser et compter les cellules, les machines sont équipées d'une technologie utilisant comme principe la cytométrie. Celle-ci convertit les caractéristiques d'une cellule en plusieurs signaux électriques qui peuvent être interprétés, plus particulièrement sa taille. Ces signaux électriques permettent aussi de réaliser un comptage de cellules pour déterminer leur concentration dans le sang analysé.

1.2.1 - Présentation

Figure n°1 : Automate d'hématologie Pentra 80



L'automate Pentra 80 est un analyseur d'hématologie capable de traiter jusqu'à 80 échantillons de sang par heure. Il effectue diverses analyses, telles que le comptage des globules blancs, des globules rouges, des plaquettes, et la mesure de l'hémoglobine. Cet appareil peut contenir jusqu'à 100 tubes de sang, disposés en 10 rangées de 10 tubes.

Avant d'être analysé, le sang doit être préparé à l'aide de réactifs spécifiques, fabriqués par HORIBA, en fonction de l'analyse souhaitée.

Cette préparation inclut la dilution du sang, car à l'état brut, il est trop concentré pour être analysé. Parfois, il est nécessaire de modifier physiquement les cellules à l'aide d'autres réactifs pour isoler une population cellulaire particulière et les analyser séparément, comme dans le cas de la destruction des globules rouges pour analyser les globules blancs.

1.2.2 - Principe de mesure des cellules

Le principe est basé sur une variation d'impédance (détection volumétrique de COULTER) qui permet la mesure du volume des cellules, paramètre indispensable pour la détermination du volume globulaire moyen en ce qui concerne les globules rouges et le volume plaquettaire moyen pour les plaquettes.

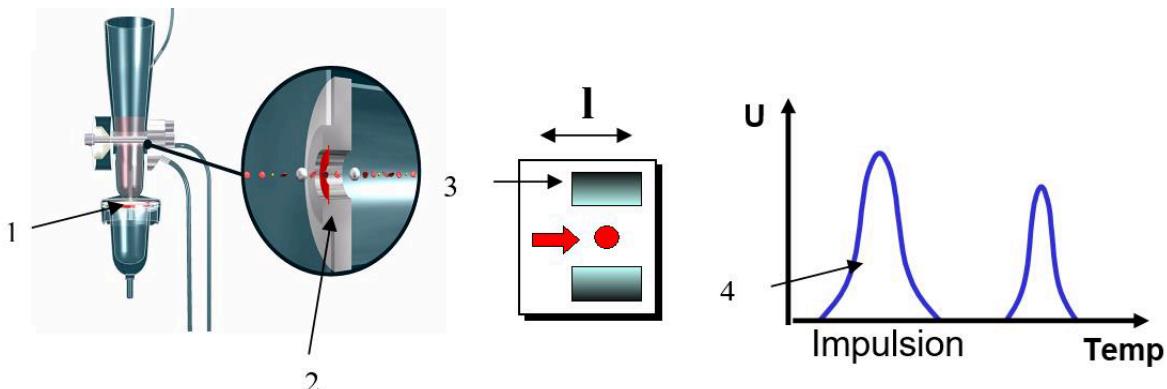
Les cellules sont suspendues dans un liquide conduisant le courant (1).

Une fois la dilution effectuée, celle-ci est aspirée à travers un orifice calibré (2).

Le passage d'une cellule dans l'orifice créé de façon momentanée une augmentation d'impédance entre les deux électrodes (3).

Cette augmentation d'impédance génère une impulsion proportionnelle au volume cellulaire (4).

Figure n°2 : Principe de mesure de Coulter



Ces pics de tension, correspondant aux volumes des cellules, subissent une conversion analogique numérique qui permet d'obtenir donc un nombre de cellules comptées par unité de volume.

Les données sont alors intégrées et tracées par une courbe de distribution sur l'IHM de l'automate Pentra 80.

1.3 - Contexte

1.3.1 - Contexte hématologique

Les analyses hématologiques jouent un rôle crucial dans le diagnostic et le suivi de diverses maladies. Le sang, composé de plasma liquide et de cellules sanguines telles que les globules rouges, les globules blancs et les plaquettes, ainsi que de nombreuses substances (protéines, hormones, vitamines, etc.), constitue un élément central de ces analyses.

Dans le cadre du traitement de certaines pathologies, une analyse hématologique approfondie s'avère indispensable. Cette étape est essentielle pour déterminer avec précision l'état de santé d'un patient et élaborer un plan de traitement adapté.

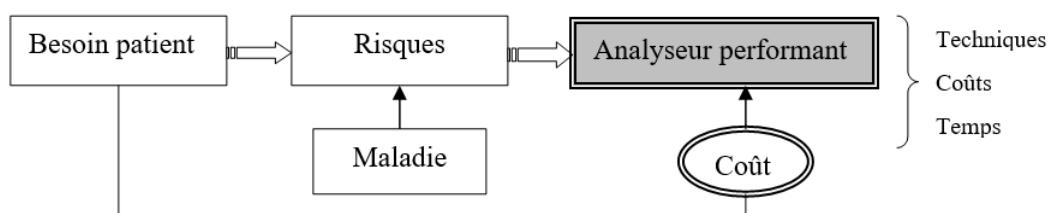
Les avancées technologiques récentes ont permis d'améliorer de manière significative la précision des mesures des cellules sanguines, facilitant ainsi leur différenciation. Les automates d'hématologie et de chimie clinique, tels que ceux développés par HORIBA, jouent un rôle clé dans ces progrès. Ces dispositifs permettent de réaliser des diagnostics plus fiables et plus rapides.

La mise en place de dispositifs d'analyse plus précis vise principalement à répondre aux besoins des patients. Pour certaines maladies, l'analyse hématologique constitue un élément central du diagnostic et du suivi thérapeutique. Les objectifs spécifiques incluent :

- **Amélioration de la précision diagnostique** : Permettre une identification plus précise des maladies.
- **Suivi optimal des traitements** : Faciliter l'évaluation de l'efficacité des traitements en cours.
- **Réduction des temps de diagnostic** : Accélérer les processus diagnostiques pour un traitement plus rapide.

On met en place un dispositif d'analyse dans le but principal de répondre à un besoin du patient. Pour certaines maladies, cela repose principalement sur l'analyse hématologique :

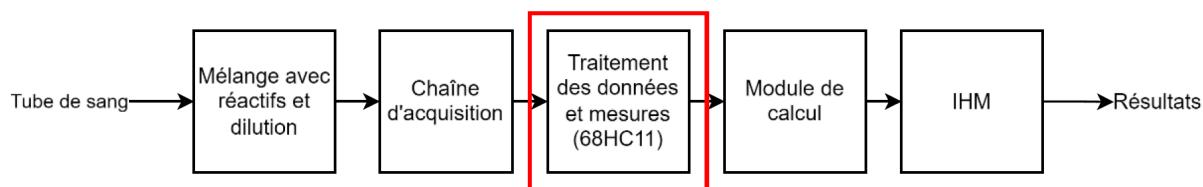
Figure n°3 : Chaîne d'analyse hématologique



1.3.2 - Contexte technique

Le microcontrôleur MC68HC11 est actuellement utilisé dans de nombreux instruments HORIBA (Pentra 80, MHR, Micros...). Aujourd’hui les cartes mère XAA456B du PENTRA 80 sont équipées de 4 microcontrôleurs MC68HC11. Ce microcontrôleur effectue la mémorisation des hauteurs des impulsions électriques générées par le passage des cellules dans le trou (cf [principe de mesure COULTER](#) ou [section 1.2.2](#)).

Figure n°4 : Rôle du MC68HC11



Ce composant date des années 1980/1990, date correspondante au début du développement des appareils. La première société à produire ces composants est la société MOTOROLA. Cette société a ensuite vendu sa division “semi-conducteur” à la société FREESCALE puis FREESCALE à NXP.

Fin 2021, NXP annonçait la fin de production des microcontrôleurs pour juin 2022.

Ce composant est maintenant en fin de vie et présente un rapport performance/prix peu avantageux.

2 - Définition du stage

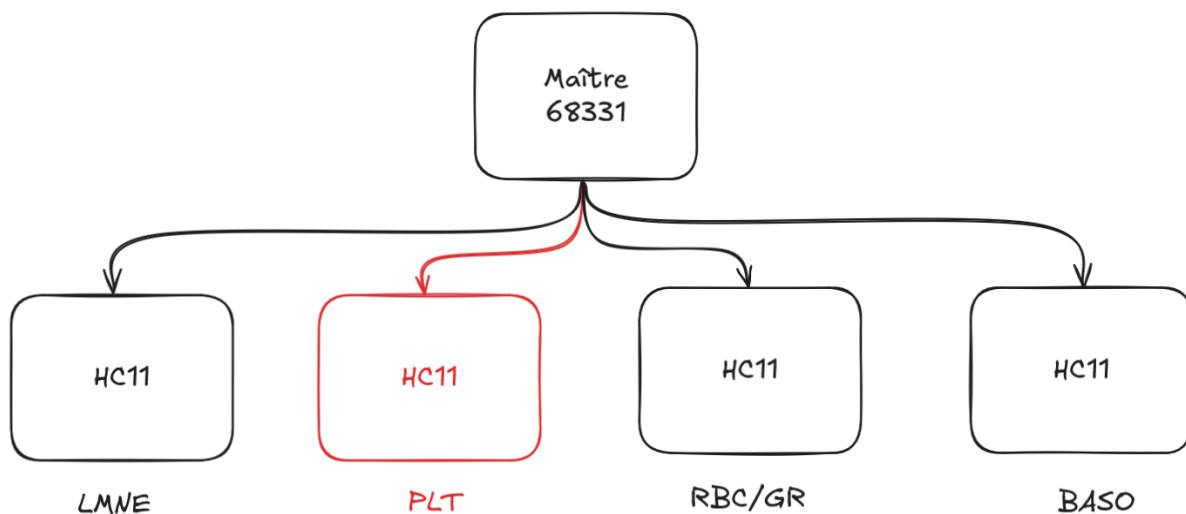
2.1 - Objectif du stage

Le MC68HC11 n'étant plus mis en vente depuis juin 2022, le stock s'épuise au fur et à mesure que les années passent. L'objectif de ce stage est de remplacer ce microcontrôleur par une solution récente, plus particulièrement la technologie STM32. Il faut donc effectuer une portabilité depuis le code assembleur des MC68HC11 vers un code en C imitant le même comportement sur des STM32.

Il faut prendre en compte que l'acquisition comporte **quatre voies** :

- Voie BASO Analyse une population de globules blancs
- Voie LMNE Analyse une sous population des BASO
- Voie RBC/GR Analyse les globules rouges
- Voie PLT Analyse les plaquettes

Figure n°5 : Agencement des HC11 en fonction du Maître



En effet, le code du HC11 contient les fonctionnalités de ces 4 voies. Pour une raison de gestion des versions et étant donné que ces HC11 sont reportés sur plusieurs machines au sein de HORIBA, il est préférable de ne pas dissocier les versions en fonction des machines en plus de garder une trace de celles-ci.

La durée du stage étant légèrement courte pour opérer la portabilité de l'entièreté du code HC11, nous nous concentrerons sur une seule voie d'acquisition : La voie **PLT**. Pour permettre la pérennité du projet, nous proposerons une conception de code propre et documentée.

2.2 - Analyse de l'existant

Un stagiaire de l'ENSI CAEN avait déjà commencé à travailler sur le projet avant mon arrivée pendant une durée de 6 mois.

Cependant, le travail qu'il a pu fournir n'est pas conforme aux normes de développement. J'ai alors jugé, avec l'approbation de mon tuteur, que le code n'est pas maintenable ni améliorable bien que partiellement fonctionnel.

J'ai alors pris la décision de reprendre la conception à zéro en m'inspirant de ce qu'il avait pu entreprendre, notamment des spécifications de son rapport, quelques bouts de codes ainsi que la configuration donnée aux STM32.

Grâce à ce stagiaire, une partie du travail a déjà été réalisée, notamment le câblage du STM32 à la place du HC11 sur la voie plaquette, mais aussi l'analyse d'une partie du code assembleur qui a été documentée dans son rapport.

Nous avons donc à notre disposition :

- La documentation du Pentra 80
- Une carte de Pentra 80 avec une version de test déjà implémentée et flashée (possibilité d'interagir avec la carte via un shell)
- Une carte de Pentra 80 câblée à un STM32 ainsi que la configuration de STM32 correspondante
- Une esquisse de projet avec des bouts de code supposés fonctionnels
- Une trace des anomalies rencontrées concernant le comportement des STM32
- Une analyse partielle des interactions entre le Maître et l'Esclave **PLT**

Il nous manque cependant une documentation détaillée du fonctionnement du HC11 PLT. Ces informations me seront fournies par mon tuteur qui effectue le retro-engineering du code assembleur pour me fournir des organigrammes du fonctionnement des HC11.

2.3 - Analyse de l'environnement technique

Le STM32 a été choisi pour remplacer le HC11 car c'est une gamme de microcontrôleurs récente dont la production ne risque pas de s'arrêter. De plus, son faible coût d'achat ainsi que sa faible consommation d'énergie s'avèrent être des facteurs intéressants sur le long terme, permettant globalement de réduire le coût de production des cartes.

N'étant pas familier avec les microcontrôleurs STM32, j'ai dû suivre une formation spécifique : les cours de Polytech Montpellier [Pomad](#). Cela m'a permis de me familiariser avec cette technologie et de maîtriser l'environnement de travail nécessaire pour mon stage.

Les projets STM32 sont réalisés à l'aide d'un IDE spécifique : STM32Cube IDE. Cet environnement regroupe deux entités :

- **Eclipse** : Intervient dans la partie programmation. Cet IDE intégré nous fournit tous les outils nécessaires pour développer et déboguer l'application.
- **STM32CubeMX** : Apparaît sous la forme d'une interface graphique nous permettant de configurer le STM32(GPIO, vitesse d'horloge..).

Il existe des bibliothèques bien spécifiques au développement STM32 : HAL (**H**ardware **A**bstraction **L**ayer). Ces bibliothèques nous permettent de générer du code en fonction des paramètres définis dans l'interface graphique de STM32CubeMX. Elles simplifient l'interaction avec les microcontrôleurs STM32.

STM32Cube IDE nous permet de développer en C, en C++ ou en assembleur. Ici, le développement a été réalisé en C. Les bibliothèques HAL ont généré le code nécessaire à la configuration du STM32. Certains drivers ont dû être développés séparément, notamment les drivers SPI qui ont été partiellement développés concernant la partie communication et gestion de la mémoire.

Concernant la structure du code, celui-ci est structuré selon une machine à états. Un concept courant en développement embarqué dans la mesure où les microcontrôleurs fonctionnent généralement en autonomie. Leur état change donc en fonction des actions réalisées.

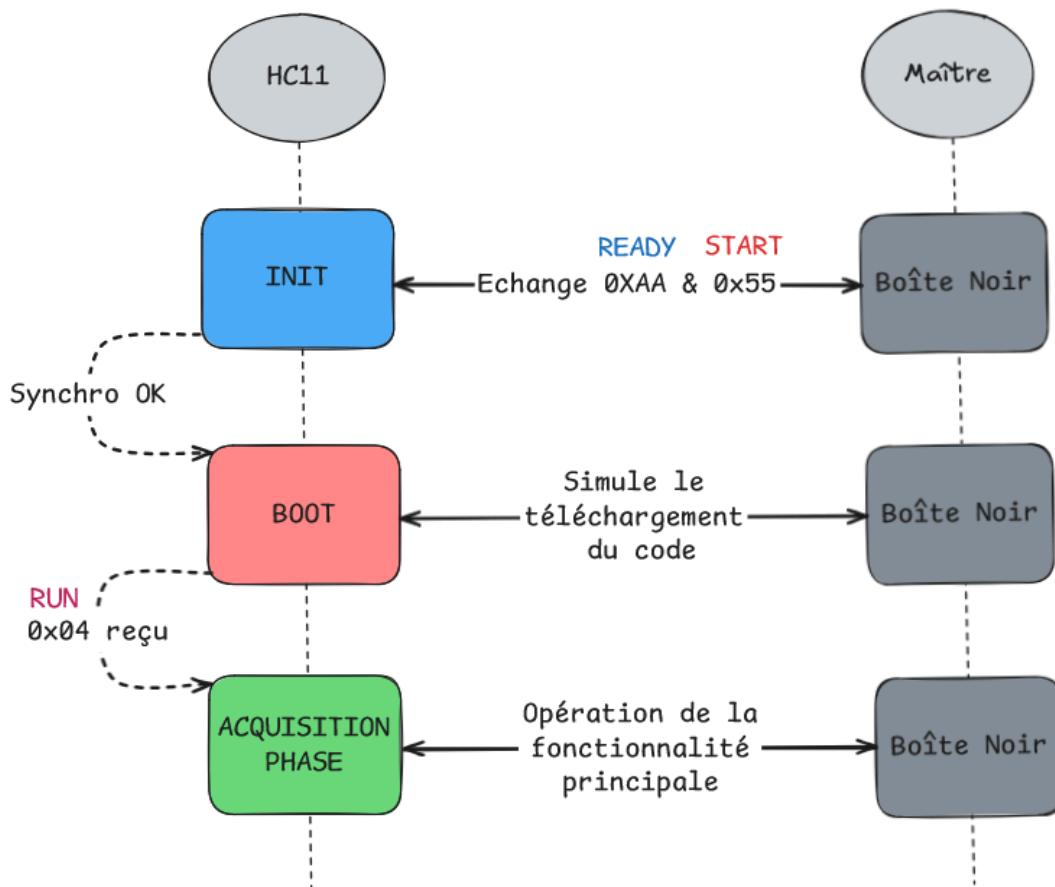
Il faut prendre en compte qu'il existe deux types d'états :

- L'état du MCU qui correspond à un état général (voir ci-après)
- L'état de l'acquisition, qui lui permet de déterminer s'il est en mesure ou non d'exécuter les commandes reçues par le Maître sur le bus SPI. (voir [Rapport Technique](#))

Le MCU est séparé en **trois** états : (cf [annexe DS1](#))

1. **INIT** : Le maître et l'esclave s'échangent 2 octets via le bus SPI : READY & START. Ces octets leur permettent de se synchroniser, si la synchronisation réussit, on passe alors en état de **BOOT**
2. **BOOT** : Pour recontextualiser, à chaque démarrage de la machine, le Maître transfère via le bus SPI le programme des HC11 qu'ils devront exécuter. Etant donné que nous développons une solution de **remplacement**, nous devons reproduire ce comportement. L'état de **BOOT** simule le téléchargement de ce programme. Nous ignorons en réalité les octets reçus sur le bus SPI dans la mesure où ils sont destinés aux HC11. Notre programme est déjà développé et prêt dans la mémoire interne du STM32. Nous passons ensuite dans l'état **ACQUISITION_PHASE**
3. **ACQUISITION_PHASE** : C'est la fonctionnalité principale des HC11, Ils sont dans un état d'attente de réception d'une commande de la part de leur Maître. Dès qu'une commande est reçue, ils vont l'interpréter et l'exécuter en fonction de leur état d'acquisition.

Figure n°6 : États du MCU



2.4 - Cahier des charges

Une première version de cahier des charges a été rédigé par moi-même, une deuxième version à été rédigé avec l'aide de Sophiane Megdoud, responsable métiers. Le cahier des charges ayant été fourni via la planification de mon stage dans un diagramme de gantt, j'ai simplifié la visualisation de celui-ci en passant par un schéma. (cf [annexe cahier des charges](#))

Ce cahier des charges peut se diviser en 3 grandes parties :

- La **prise en main** du projet. Le projet existant est complexe et riche en informations. Il est donc essentiel de bien le comprendre avant de commencer à travailler dessus. La partie d'appropriation est une étape cruciale pour produire une bonne conception derrière.
- La partie **Conception / Développement / Test** du projet. Développer les principales fonctionnalités de chaque périphérique ainsi qu'effectuer les tests associés. Chaque périphérique, que ce soit le bus SPI ou l'ADC, constitue un comportement et une configuration complexe. Il est donc essentiel pour chacun d'entre eux de trouver la bonne configuration ainsi que de développer des fonctions complémentaires pour avoir des drivers appropriés.
- La partie **Intégration de la solution**. Une fois que les fonctionnalités ont été développées et testées dans la limite du possible sur table, il faut maintenant intégrer la solution sur machine et vérifier son bon comportement. On appelle cette étape les tests d'intégration.

3 - Rapport technique

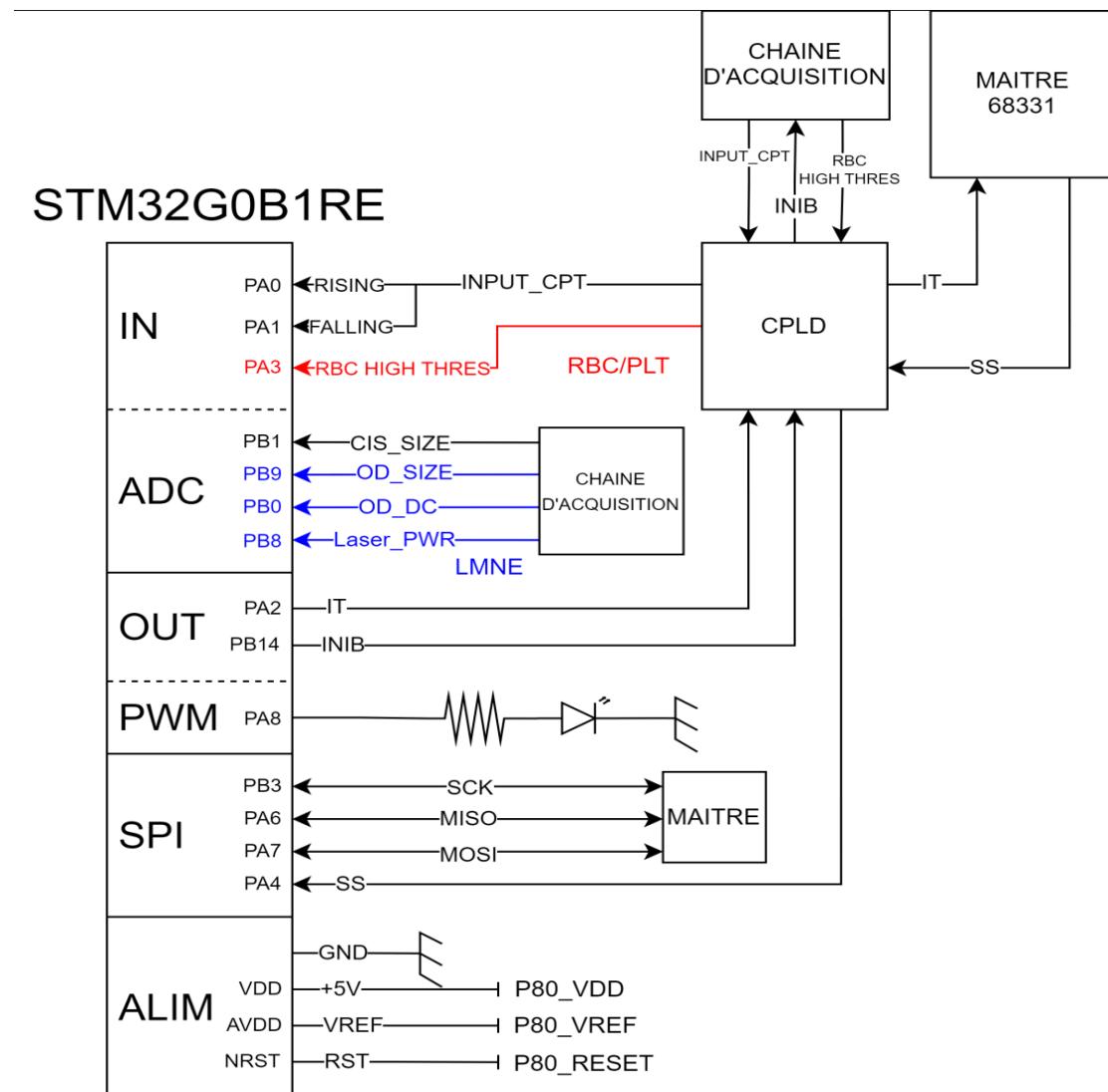
3.1 - Architecture électronique

3.1.1 - Maquettage

Dans le cadre de ce projet, on utilise une carte de développement NUCLEO que l'on peut fixer sur la carte-mère. Cette carte facilite les interactions avec le STM32.

Cette section concerne la configuration électronique du STM32. On y retrouve les entrées et sorties configurées sur la carte de développement NUCLEO.

Figure n°7 : Maquette d'un des 4 esclaves STM32



IN :

INPUT_CPT (i.e Input Capture) : est un signal numérique donnant l'information sur le niveau des impulsions haut ou bas (1 ou 0).

Seuil Haut : Pour les globules rouges et plaquettes, il y a en plus une entrée numérique « Seuil haut » permettant de différencier les globules rouges des plaquettes, car ils partagent le même canal d'acquisition.

Compteur Immédiat : est un signal numérique nous permettant de compter le nombre d'éléments rencontrés durant l'analyse.

CIS_SIZE : est un signal analogique donnant l'information sur la taille des cellules issue de la méthode résistive.

OUT :

IT (i.e Interrupt Hardware) : est mis à HAUT lors de la fin d'une acquisition. Cela permet au microcontrôleur de communiquer avec le maître sans passer par le bus SPI.

INIB : est un signal pour le CPLD. Lors de l'acquisition, le CPLD maintient le signal analogique jusqu'à réception d'une impulsion sur INIB.

PWM : Ce signal imite le comportement des LEDs utilisées par les MC68HC11. Un câble peut être branché directement sur la carte de développement pour relier la LED à l'impulsion générée.

ALIMENTATION : On utilise la tension d'alimentation de la carte-mère pour alimenter la carte de développement.

SPI IN :

SCK : Horloge du bus SPI dictée par le maître 68331.

MOSI : Canal de transfert des données du maître vers l'esclave.

SS : Signal de sélection de l'esclave dicté par le maître

SPI OUT :

MISO : Canal de transfert des données de l'esclave vers le maître

3.1.2 - Configuration du STM32

Nous devons nous inspirer de la configuration des broches du HC11 pour configurer le STM32.

 : Broches **SPI**

 : Broches d'**Acquisition**

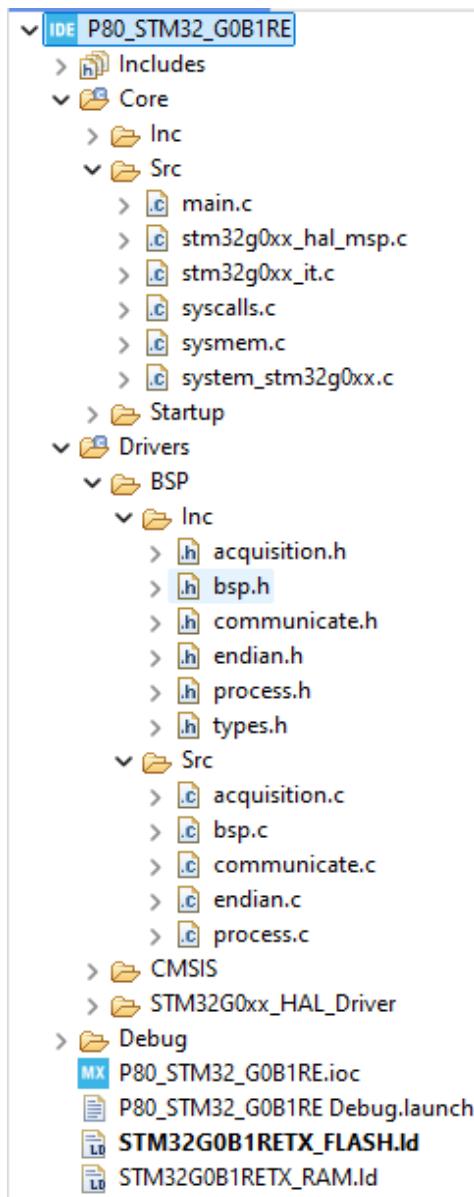
 : Broches de **Test**

Nom de la broche	Fonctionnalité	Commentaire
PB3	SCK	signal analogique dicté par le maître (68331)
PA7	MOSI	signal analogique géré par le maître (68331)
PA6	MISO	signal analogique géré par l'esclave (STM32)
PA4	SS	signal analogique dicté par le maître (68331)
PA0	TIMER 2 Channel 1	input capture sur front montant
PA1	TIMER 2 Channel 2	input capture sur front descendant
PB1	ADC	configuré en DMA
PB14	GPIO Output	INIB (cf 3.1.1)
PA2	GPIO Output	Hardware Interrupt
PA3	GPIO Input	Seuil Haut
PD2	TIMER 3 Channel 1	Compteur immédiat, configuré en clock externe
PA8	TIMER 1 Channel 1	générateur PWM
PB8	GPIO Output	BIT SET / CLEAR Pin
PA5	GPIO Input	LED

3.2 - Arborescence du projet

L'arborescence du projet est générée avec STM32Cube IDE. À la création du projet, il nous est demandé de spécifier le modèle de STM32 sur lequel nous travaillons pour télécharger ses drivers et librairies associés.

Figure n°8 : Architecture logicielle



Le fichier main.c se situe dans ‘Core/Src’, le reste des fichiers dans ce dossier sont des librairies importées. Le fichier main.c contient la boucle principale du programme ainsi que des variables globales.

La majeure partie du code se situe dans ‘Drivers/BSP/Src’. ‘Src’ correspond à Source et ‘Inc’ correspond aux Includes.

L'appellation BSP est une appellation propre au développement embarqué. Cela correspond à “Board Support Package”. Ce dossier contient les drivers et les librairies développées par l'utilisateur.

Le fichier ‘bsp.c’ est un fichier spécial, il contient les fonctions nécessaires pour compléter les drivers générés par HAL. Par exemple des fonctions complémentaires au bon fonctionnement du SPI (voir [section 3.4.2](#)).

On observe, tout en bas de l'arbre, des fichiers en ‘.Id’ Ce sont les linkers nécessaires à la compilation du code. Ils font partie de la toolchain constituée d'un assembleur, d'un compilateur C, d'un **linker** ainsi qu'un débogueur. On retrouve ici la “GNU ARM Embedded ToolChain”. Basée sur le compilateur gcc et le débogueur gdb. Ils constituent une alternative open source aux autres solutions professionnelles et coûteuses.

Le reste des fichiers seront commentés durant le rapport.

3.3 - Fonctionnement du bus SPI

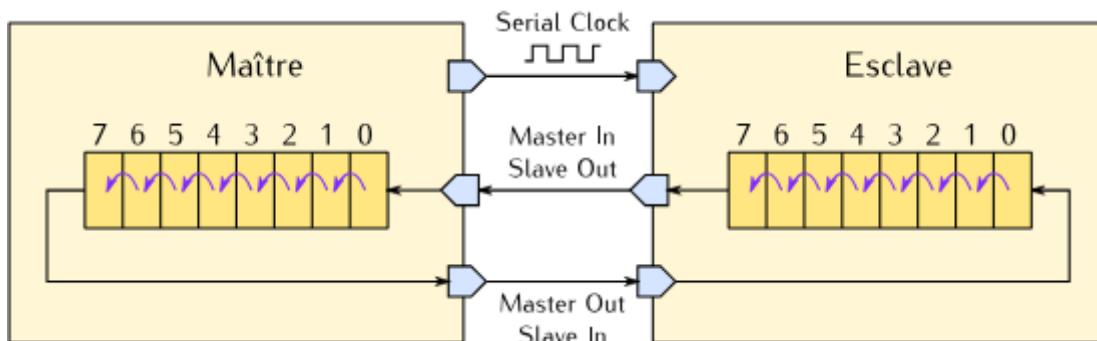
L'entièreté du projet tourne autour du bus SPI.

Le bus SPI permet le transfert de données entre plusieurs microcontrôleurs en mode full duplex. Étant un système de communication, il existe donc un protocole associé. Le bus s'articule autour d'une hiérarchie Maître / Esclave où le Maître va dicter à qui la communication est destinée ainsi que la vitesse à laquelle la communication va s'opérer. Il faut voir cela comme un orchestre où le maître bat le tempo et les esclaves s'activent selon les directives du maître.

Rentrions maintenant dans le technique, pour une communication SPI, quatre câbles sont nécessaires à la communication : (cf [Agencement du bus SPI](#))

- **SCK** pour “Serial Clock”. C'est le tempo dicté par le Maître. À chaque tic généré sur ce câble, un bit est transféré sur le bus SPI. Cette connexion fonctionne donc par série de 8 tics ce qui permet de transférer octet par octet.
- **MOSI** pour “Master Out Slave In”. Cette connexion contient le transfert de données du Maître vers l'Esclave.
- **MISO** pour “Master In Slave Out”. Cette connexion contient le transfert de données de l'Esclave vers le Maître.
- **SS ou CS** pour “Slave Select”. Ce signal est aussi dicté par le Maître, c'est un signal logique qui permet au Maître de choisir l'Esclave auquel il parle. Si le signal est mis à zéro, l'Esclave sait que l'information lui est destinée. Sinon, les octets transmis sur le bus sont ignorés.

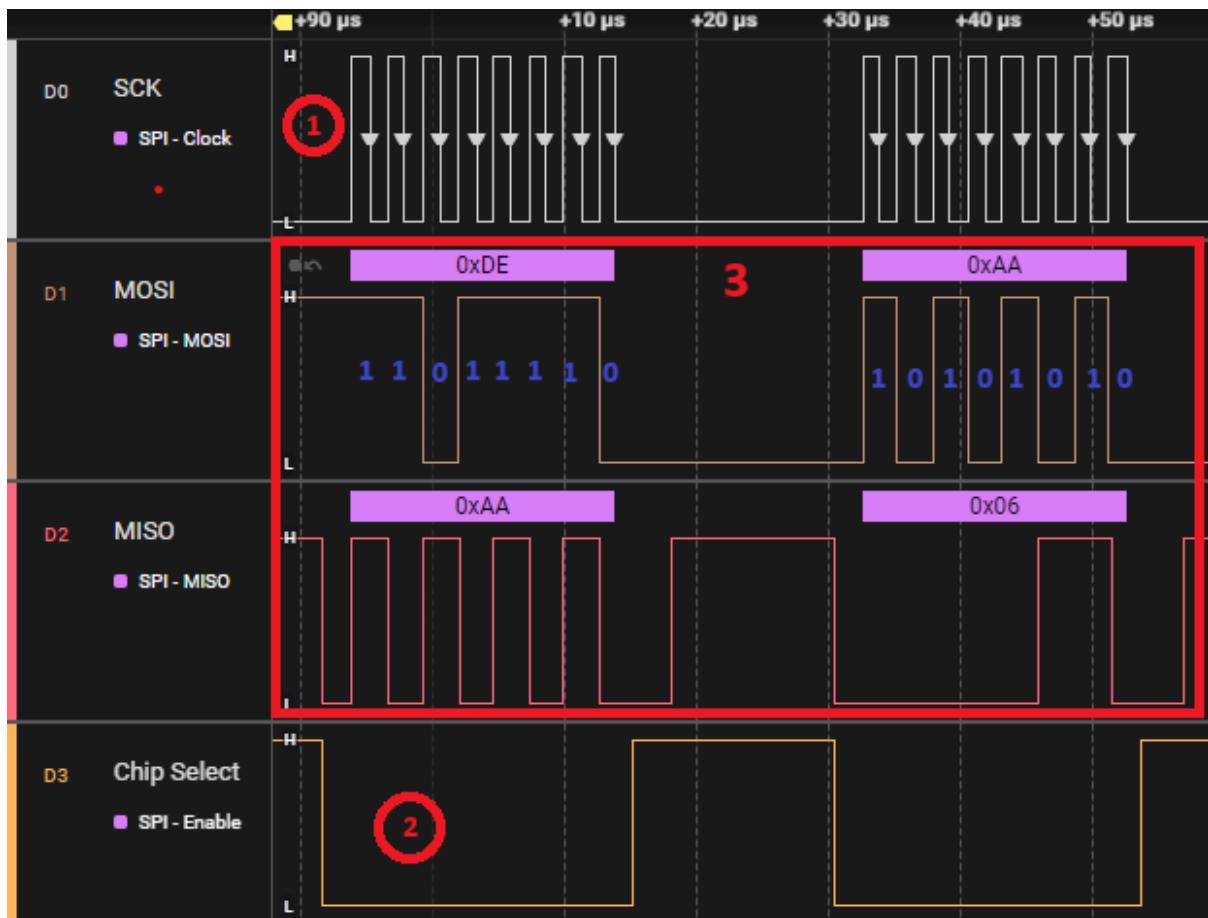
Figure n°9 : Transfert des données full duplex



Le bus full-duplex permet l'échange de données en simultané. Lorsqu'un tic sur le SCK est généré par le maître, un octet est transféré **simultanément** du Maître vers l'Esclave et de l'Esclave vers le Maître.

L'analyse du bus SPI se fait au travers d'une sonde SALAE qui est un analyseur logique. Cet analyseur nous permet d'espionner le bus SPI.

Figure n°10 : Exemple de communication sur le bus SPI



(1) On observe que le SCK fonctionne bien par série de 8 tics. Par exemple, sur le MOSI, on retrouve un premier octet transféré : 1101 1110 correspondant à **DE** ainsi qu'un second octet transféré : 1010 1010 correspondant à **AA**.

L'analyseur logique arrive à déchiffrer les octets transmis sur le bus SPI en se basant sur le signal SCK.

(2) On observe bien le *Chip Select* mis à 0 lors du transfert de données entre le Maître et l'Esclave.

(3) On observe aussi le mode full duplex, un octet transféré simultanément (MISO/MOSI) à chaque fois que 8 tics sont générés sur SCK.

3.4 - Etudes d'anomalies potentielles

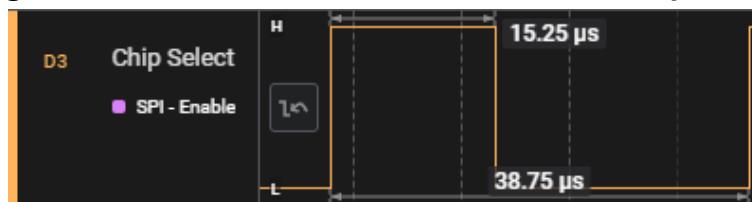
Grâce au précédent stagiaire, on a pu récupérer une trace d'anomalies rencontrées concernant le comportement du STM32 vis à vis du maître. On prend donc le temps de faire des tests sur ce qui a été remonté pour ainsi partir sur une base saine.

3.4.1 - Etude de l'anomalie du Chip Select

Habituellement, le *Chip Select* reste bas tout du long de la communication entre le Maître et l'esclave (et ce sur plusieurs octets). Cependant, il semblerait que le *Chip Select* du Maître descende et remonte pour chaque octet transféré.

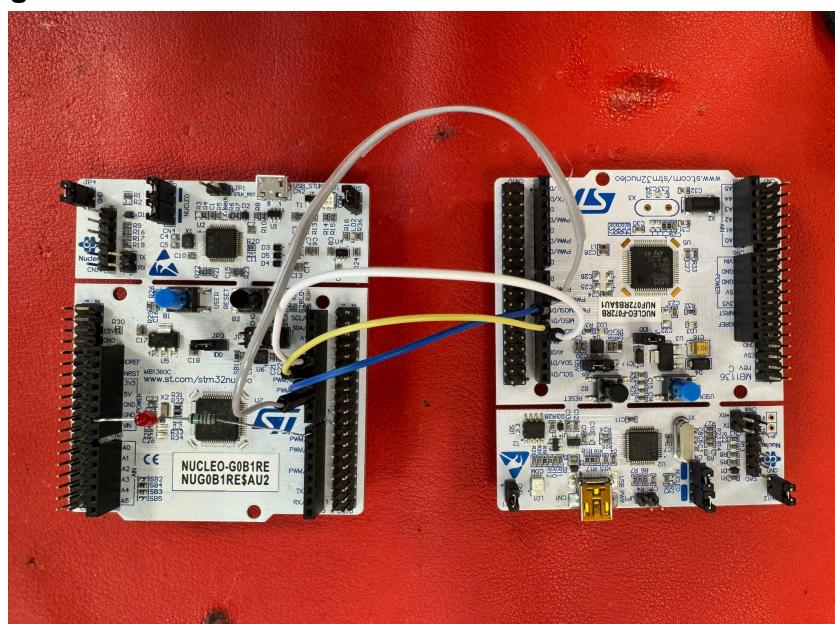
On développe donc un premier programme imitant ce comportement sur un STM32F072RB (qui est le STM des cours pomad) configuré en tant que maître. Il fallait reproduire un *Chip Select* qui relâche le signal pour une durée de 15 microsecondes environ entre chaque octet transmis.

Figure n°11 : Intervalle de relâchement du Chip Select



On développe parallèlement un second programme sur un STM32G0B1RE (qui est le STM visant à remplacer le HC11) configuré en tant qu'esclave pour tenter d'analyser la communication SPI. Reste à câbler leur bus SPI (4 câbles) et tester l'échange de données.

Figure n°12 : STM32F072RB connecté à un STM32G0B1RE



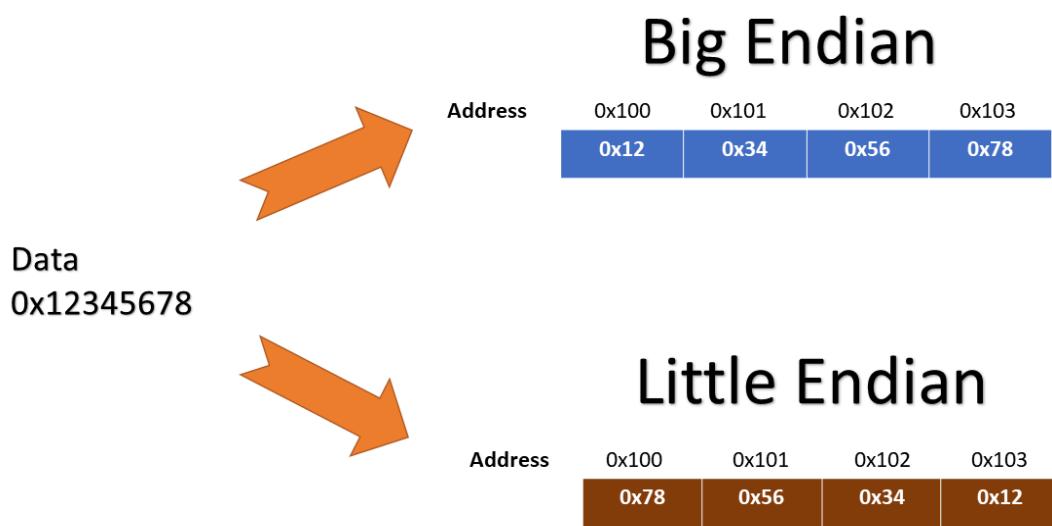
Cette phase de test entre deux STM32 a facilité la prise en main des fonctionnalités proposées par les librairies HAL. En effet, HAL ne se contente pas de générer du code de configuration pour les broches, HAL génère aussi les drivers et les fonctions associées pour faire fonctionner le tout. Rien que pour le SPI, il y a une dizaine de fonctions à appréhender avec chacune leur comportement respectif.

Les tests de transmission et de réception de données sur le bus SPI furent concluants. Il n'y eut aucun déphasage observé, les données étaient correctement reçues d'un côté comme de l'autre. On en conclut que ce comportement ne pose aucun problème pour le développement des fonctionnalités futures.

3.4.2 - Etude de l'anomalie de gestion de la mémoire

Concept : Le *memory mapping* dans les systèmes embarqués implique l'agencement des données dans des emplacements mémoire. Les formats *Big Endian* et *Little Endian* se réfèrent à deux manières différentes de stocker des données multi-octets. En format *Big Endian*, l'octet le plus significatif (MSB) est stocké à l'adresse mémoire la plus petite (à gauche), ce qui facilite la lecture des hexadécimaux par les humains. À l'inverse, le format *Little Endian* stocke l'octet le moins significatif (LSB) à l'adresse mémoire la plus petite (toujours à gauche), s'alignant mieux avec certaines architectures informatiques et opérations arithmétiques.

Figure n°13 : Big / Little Endian



Ce concept ressort lorsque l'on assemble des technologies d'époques différentes. Pour recontextualiser, un changement a eu lieu dans les années 90, pour des raisons de performance, la majorité des nouveaux microcontrôleurs sont passés au format Little Endian.

Le maître (68331) possède un *memory mapping* au format *Big Endian* alors que le STM32 possède un *memory mapping* au format *Little Endian*.

Le fonctionnement du maître est une boîte noire, nous n'avons pas la main dessus. Nous pouvons seulement agir sur le code côté esclave. Il est donc de la responsabilité de la solution de remplacement de gérer l'agencement mémoire.

Dans la mesure où les octets sont transmis un à un sur le bus SPI, toute donnée avec une taille supérieure à un octet doit être inversée. On parle ici de short / uint16_t (2 octets) ou de word / uint32_t (4 octets).

Toujours grâce aux deux STM32 connectés ensemble, on effectue les tests adéquats :

Côté maître, on stocke l'entier 43 981 correspondant à l'hexadécimal 0xABCD dans un buffer, on l'envoie ensuite à l'autre STM32 via le bus SPI.

Côté esclave on stocke le short reçu octet par octet. Grâce au débogueur, on observe que le short envoyé sur le bus SPI est 0xCDAB.

Figure n°14 : Octets reçus analysés au débogueur

Expression	Type	Value
(x)= Received_short	uint16_t	0xcdab (Hex)

L'inversion s'est faite lorsque l'on a stocké 0xABCD dans le buffer, le processeur stocke la valeur en hexadécimal inversée d'une manière à ce qu'il puisse l'interpréter et opérer des opérations arithmétiques. Côté esclave, les octets ont été reçus un à un, la conversion n'est pas opérée.

Une librairie nommée 'endian.c' a donc été développée avec son fichier header associé 'endian.h' contenant les fonctions d'inversion d'octets nécessaires.

Cette fonction inverse les octets d'un short à l'adresse passée en paramètre. Elle est générique et utilisée dans de nombreuses fonctions de la librairie.

Une fois la librairie 'endian.c' développée (cf [contenu librairie](#)), il faut maintenant développer les fonctions SPI associées dans la librairie 'bsp.c'.

On développe alors des fonctions visant à gérer le SPI ainsi que le memory mapping de manière générique pour ne pas surcharger le code de fonctions intermédiaires.

Figure n°15 : Résultat des tests Little / Big Endian



- (1) Envoi d'un word par le maître : 0x55BFDEAA. L'esclave quant à lui se contente de lire les données qui arrivent sur le bus SPI, d'où l'apparition de 0 sur MISO.
- (2) Envoi de l'octet 0x06 par l'esclave signifiant Acknowledge. L'octet 0x06 est largement utilisé dans l'application du HC11, on choisit donc de le réutiliser lors des tests.
- (3) L'esclave a de son côté reçu le word et a opéré la conversion Big / Little Endian. Il le renvoie maintenant sur le bus SPI mais inversé. Le word envoyé sur le bus est donc bien 0xAADEBF55.

Pour la suite du rapport, le fonctionnement du STM32 sera détaillé partie par partie. Un DSI de l'architecture du fonctionnement global du STM32 est disponible en [annexe](#).

3.5 - Développement de la partie INIT / BOOT

A partir de maintenant, les tests ont été effectués en connectant le STM32 à la carte mère du Pentra 80. Les interactions sont donc opérées vis à vis d'une boîte noire.

3.5.1 - Développement du protocole de synchronisation

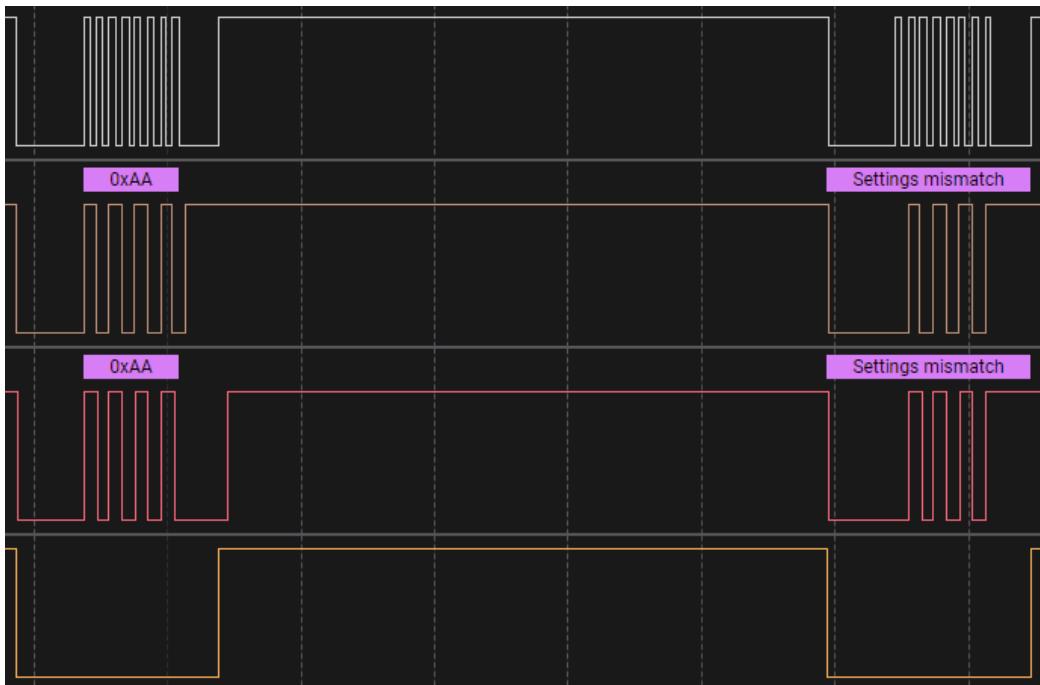
Lors du démarrage de la carte, le maître se synchronise sur le bus SPI avec les esclaves. Cela constitue la partie INIT.

Deux octets sont envoyés simultanément sur le bus SPI : READY et START. Ces deux octets sont représentés par 0xAA et 0x55. Le choix de ces octets n'est pas anodin, ces octets sont envoyés la plupart du temps pour réaliser des tests. En effet, leur représentation binaire est constituée de 0 et de 1 alternés périodiquement :

- 0xAA : 1010 1010
- 0x55 : 0101 0101

Cela nous permet de les repérer facilement à l'analyseur logique.

Figure n°16 : Envoi des octets READY et START



L'analyseur logique se base sur le signal SCK pour interpréter la donnée. Nous avons ici un léger décalage qui rend la lecture impossible pour l'analyseur logique sur le second octet. Nous reconnaissons cependant très bien l'octet 0x55 grâce à sa structure de bits constituée de 0 et de 1 alternés. (commençant par un 0)

3.5.2 - Développement du protocole de téléchargement

Une fois la synchronisation des esclaves avec leur maître réussie, le programme des HC11 est téléchargé via le bus SPI et ce à chaque démarrage de la carte. Cela constitue la partie BOOT du programme.

Nous devons donc reproduire ce comportement. Nous ignorons cependant le programme reçu dans la mesure où il est destiné aux HC11, développé en assembleur et que **notre** programme est déjà développé et prêt en mémoire.

Quatre commandes sont utilisées par le maître pour communiquer avec ses esclaves durant le téléchargement :

Chaque fois qu'un octet de commande est reçu, une action suit en conséquence :

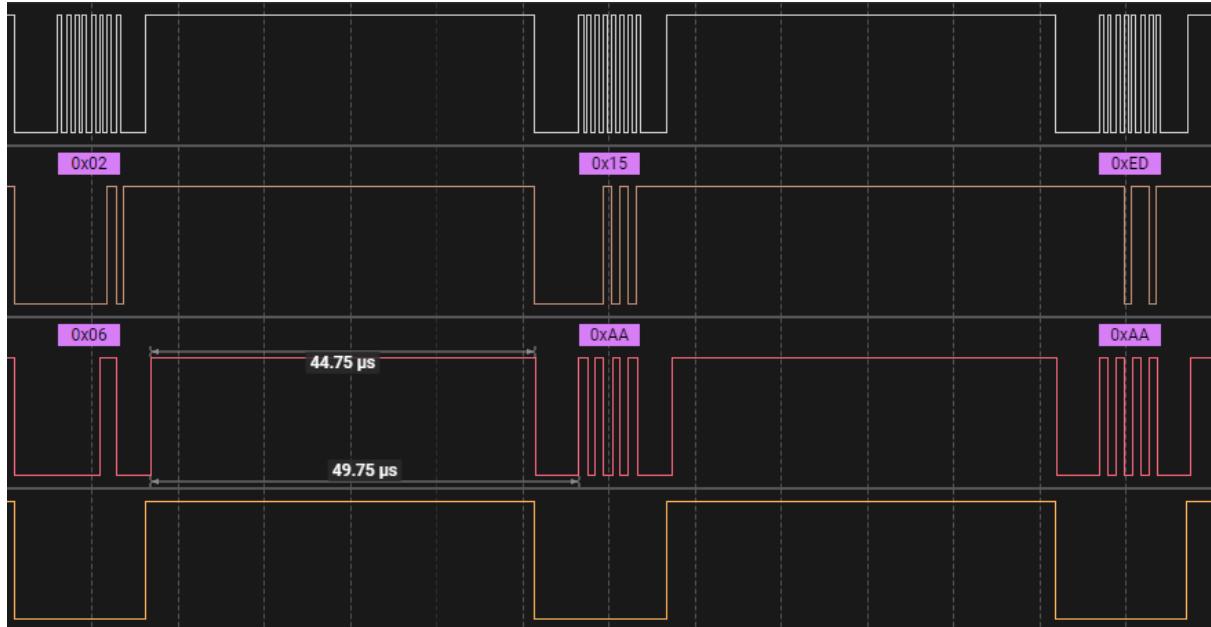
Nom commande	Octet	Action
READ_ADDR	0x01	Le maître envoie l'adresse de téléchargement du programme sur deux octets. Nous ignorons cette adresse.
BLOC_SIZE	0x02	Le maître envoie la taille du bloc à télécharger sur 2 octets. Nous stockons cette taille dans un buffer.
BLOC_READ	0x03	Le maître envoie les octets à "télécharger". Nous ignorons les octets reçus.
RUN	0x04	Cette commande signifie que le protocole de téléchargement est terminé et que nous passons à la phase d'acquisition.

A chaque commande reçue, l'esclave envoie simultanément l'octet ACKNOWLEDGE (0x06). Nous devons donc garder une trace de ce que le maître nous envoie pour ne pas se désynchroniser avec lui. Nous ne pouvons pas non plus envoyer continuellement ACKNOWLEDGE jusqu'à réception de l'octet RUN (0x04) car il sera sûrement présent dans les octets de téléchargement.

Cette partie du code simule le téléchargement, nous devons envoyer les signaux suivant le protocole pour stipuler au maître que le téléchargement s'est bien passé. Si le maître ne reçoit pas les bons octets où que le protocole est corrompu, il envoie un signal, interprété par l'IHM, nous affichant un message pour nous signaler que l'application va redémarrer.

De nombreux tests ont été réalisés à l'analyseur logique pour comprendre le fonctionnement du maître mais aussi pour vérifier que les interactions documentées sont les bonnes et qu'il n'en existe pas d'autres.

Figure n°17 : Commande BLOCK_SIZE



Nous observons la réception de l'octet 0x02 (i.e BLOCK_SIZE) et l'envoi de l'octet 0x06 (i.e ACKNOWLEDGE). S'ensuit le protocole pour la commande BLOCK_SIZE, le maître envoie la taille du bloc d'octets à télécharger tandis que l'esclave se contente de lire les octets reçus.

L'apparition de 0xAA sur MISO est due au fait que le buffer de transmission SPI de l'esclave contient l'octet 0xAA. Étant donné que la communication est en full-duplex, même si l'esclave se contente de lire les données reçues, un octet doit être envoyé sur le bus des deux côtés.

Nous observons donc que la taille du bloc d'octets à télécharger correspond à 0x15ED soit 5 613 octets.

Figure n°18 : Echange de 0x15ED octets

ΔT	341.670411 ms	N_{falling}	5.613 k
N_{rising}	5.613 k	f_{\min}	6.221 kHz
f_{\max}	16.598 kHz	f_{mean}	16.428 kHz
T_{std}	5.47 μ s		

En mesurant les front montant (ou front descendant) du *Chip Select*, on observe bien 5 613 octets transférés sur le bus SPI.

3.6 - Statut d'acquisition

Un comportement assez complexe à reproduire du HC11 est au niveau du statut d'acquisition. Une subtilité concernant le SPI ainsi que la configuration à lui donner.

3.6.1 - Précisions sur le fonctionnement du SPI

Le SPI sur les STM32 peut être configuré de trois manières différentes :

- *Polling mode* : C'est un mode bloquant où l'on utilise la charge du processeur pour la communication sur le bus SPI (en mode esclave). Ce mode est utilisé par exemple dans la partie INIT / BOOT pour communiquer avec le maître. Notre seule tâche est de répondre au maître alors nous consacrons toutes nos ressources pour la mener à bien.
- *Interrupt mode* : Comme son nom l'indique, ce mode fonctionne sur interruptions. C'est-à-dire que l'on peut définir un nombre de données reçues avant de générer une interruption qui interpelle le processeur. Le STM32 possède des registres (mémoire intégrée au MCU) dédiés au SPI. Ces registres permettent de stocker un certain nombre d'informations reçues sur le bus sans avoir à solliciter le processeur. De ce fait, la réception des données peut se faire autrement que par le blocage de la charge du processeur.
- DMA mode : Ce mode fait sensiblement la même chose que le mode Interrupt à la différence qu'il est utilisé pour les transferts de données volumineux. Nous n'utilisons pas ce mode ici.

Pour la partie INIT / BOOT, le SPI est utilisé en *Polling mode*. Cependant, durant la partie ACQUISITION, le mode *Polling* et *Interrupt* sont utilisés simultanément.

À noter que le SPI possède deux buffers se situant dans les registres du MCU. Un buffer pour la transmission (Tx) et un buffer pour la réception (Rx). Lors d'une communication, les octets proviennent des buffers Tx et sont stockés respectivement dans les buffers Rx.

Nous avons déjà observé auparavant des résidus d'octets transmis sur le bus lorsque l'esclave recevait des informations (cf [commande BLOCK_SIZE](#)). Ces dits résidus proviennent en fait des octets présents dans le buffer de transmission.

Lors de réception d'octets en mode interruptions, les octets transmis sur le bus sont les octets présents dans Tx, ce buffer est une file (FIFO).

3.6.2 - Fonctionnement du statut d'acquisition

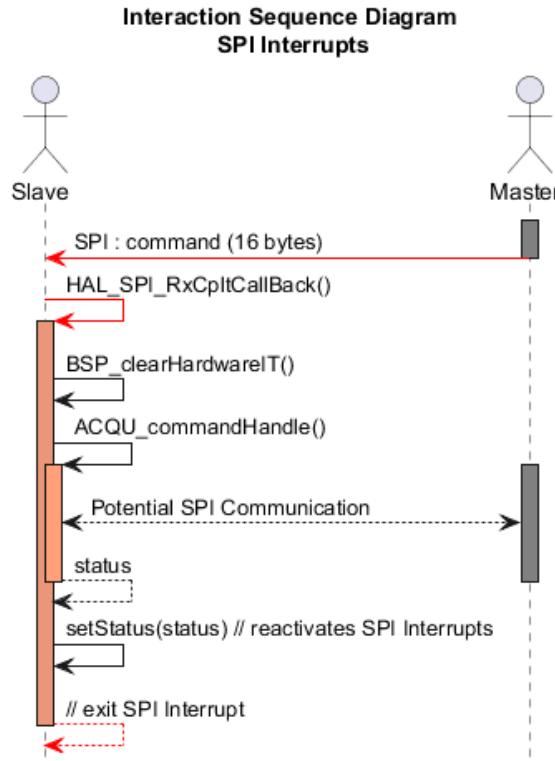
Le code fonctionne avec une architecture de machine à états. Il existe deux types d'états ici : **l'état du MCU** (mcuSTATE) qui correspond aux états INIT / BOOT / ACQUISITION et **l'état d'acquisition** (acquisitionStatus) qui correspond à l'état du MCU **pendant** la partie ACQUISITION.

Cet état peut prendre de nombreuses valeurs, les principales sont FREE (0x51) et BUSY (0x02). Nous appellerons cet état 'statut'. Il peut prendre d'autres valeurs que nous verrons plus en détail par la suite du rapport. Il est défini sur un octet et correspond à l'état actuel du MCU.

En effet, cet état change constamment en fonction des actions réalisées par le STM32. Le fait est qu'il n'est pas simplement utilisé par le STM32 en interne mais il est aussi utilisé par le Maître qui peut le lui demander à n'importe quel moment.

Le choix du fonctionnement sous interruptions du SPI prend alors tout son sens. Nous ne pouvons pas bloquer indéfiniment la charge du processeur au cas où le maître demande notre statut.

Figure n°19 : DSI du fonctionnement des interruptions SPI



Le maître envoie ses commandes sur seize octets. Même si sa commande ne contient qu'un seul octet qui nous sera utile, il remplit les quinze autres par des 0.

Nous savons donc que seize octets seront reçus à chaque commande envoyée par le maître. De ce fait, nous pouvons activer les interruptions SPI sur réception de 16 octets via la fonction donnée par HAL : (IT = Interrupt)

```
HAL_SPI_Receive_IT(hspi, RxCommand, COMMAND_SIZE);
```

Nous avons COMMAND_SIZE qui est une macro égale à 16. 'hspi' est une structure contenant les paramètres du SPI (générée par HAL). 'RxCommand' est le buffer de réception des commandes. Lorsque 16 octets sont reçus, l'ISR (Interrupt Service Routine) copie les 16 octets dans 'RxBuffer' et envoie le processeur dans la fonction de callback associée :

```
void HAL_SPI_RxCpltCallback(SPI_HandleTypeDef * hspi)
{
    BSP_clearHardwareIT();
    ACQU_setStatus(ACQU_commandHandle());
}
```

Une fois dans le callback, suivant les organigrammes fournis par mon maître de stage, le protocole est de lever l'interruption hardware à chaque commande reçue. Nous avons ensuite une fonction qui met le statut d'acquisition selon la valeur de retour du 'commandHandler'.

Avant de rentrer dans les tréfonds de la conception logicielle concernant l'acquisition, revenons à notre statut d'acquisition.

Nous avons parlé précédemment du statut qui est utilisé par les deux entités, nous n'avons cependant pas parlé de comment ce statut était utilisé par ces deux entités et plus particulièrement côté maître.

Le maître possède une commande READ_STATUS (i.e 0x03) qui est envoyée à chaque fois qu'il a besoin de récupérer le statut de l'esclave. Le maître s'attend à recevoir l'octet de statut lors du premier octet transféré en simultané à la commande et non suite à une action dans la continuité de la commande.

Étant donné que nous fonctionnons sous Interruptions, le statut doit être constamment présent et à jour dans le buffer de transmission SPI. Le buffer de transmission est une file (FIFO), le dernier statut en date ne sera donc pas forcément celui lu par le maître lors du transfert de données. Il nous faut alors vider le buffer de transmission avant de pouvoir réécrire le nouveau statut dedans.

Le problème étant que nous pouvons seulement remplir le buffer de transmission et vider le buffer de réception. Nous n'avons pas la main directe sur ces buffers dans la mesure où ce sont des registres (aucune adresse physique).

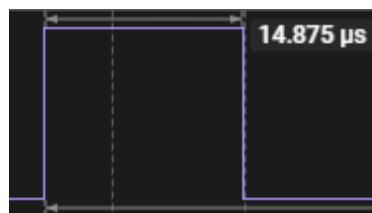
La solution est alors de réinitialiser les périphériques SPI pour vider ces buffers. Une fonction a donc été développée dans la librairie ‘bsp’ :

```
void BSP_SPI_TxFlush(SPI_HandleTypeDef *hspi)
```

Cette fonction est appelée dans la fonction ‘setStatus’. Cela nous permet de contrôler qu'à chaque changement de statut, le statut destiné au maître est aussi mis à jour.

Des tests de performance ont été réalisés à l'analyseur logique pour mesurer le temps de réinitialisation des périphériques SPI.

Figure n°20 : Visualisation du test de performance



A l'aide d'une broche configurée pour les tests en GPIO output, nous pouvons connecter l'analyseur logique dessus. Mettre le niveau logique de la broche à 1 au début puis mettre le niveau logique de la broche à 0 à la fin nous permet de visualiser le temps pris par cette fonctionnalité.

Cette méthode est largement utilisée dans le développement embarqué pour faire des tests de performances. On appelle cela un “bit set bit clear”. Nous observons grâce à cette méthode que la réinitialisation prend environ 15 µs.

On peut remarquer que la fonction ‘setStatus()’ est appelée en sortie d'interruption SPI. En effet, en agissant de la sorte, on dresse un filet de sécurité qui nous permet d'être sûr qu'un statut sera toujours présent dans le buffer. Étant donné que le buffer de transmission est vidé à chaque réception de commande (le maître lit son contenu), il doit être re rempli en sortie.

Il est difficile de suivre la trace du code lorsque l'on développe sous interruptions. Le fait est que le code n'est pas séquentiel, il peut être interrompu à n'importe quel moment. On se fixe alors une règle : Le statut peut être changé **seulement** en sortie d'interruption, **ou bien**, pendant l'opération d'une commande de type “Processing” (cf [prochaine section](#)).

3.7 - Conception logicielle de la partie acquisition

La partie acquisition est la fonctionnalité principale du STM32. Il faut prendre en compte que le code des quatre voies est commun. On doit alors proposer un code générique qui puisse être utilisé par les quatre voies sans entraver les travaux des prochaines voies.

Nous avons besoin d'une architecture avec une base solide si l'on ne veut pas perdre du temps lors du développement des quatres voies. Dès lors, la phase de conception est primordiale, d'autant plus dans un projet comme celui-ci.

3.7.1 - Types de commandes

Le maître envoie des commandes sur le bus SPI à destination de ses esclaves. De ce fait, les esclaves ont besoin d'un mécanisme pour interpréter ces commandes.

Chaque commande correspond à une action. Il en existe une vingtaine, voici une liste des commandes qui nous concernent pour la partie du projet réalisé :

Nom de commande	Octet associé	Action
CDE_STATUS	0x03	Aucune action, le maître a lu le statut lors de l'envoi de la commande
CDE_RESULT_ADJUST	0x06	Le maître demande les résultats d'un ajustement de type 2D
CDE_RESULT_2D	0x0C	Le maître demande les résultats d'un comptage de type 2D
CDE_CNT_2D	0x04	Le maître lance l'esclave sur un comptage de type 2D
CDE_ADJUST_2D	0x0A	Le maître lance l'esclave sur une fonction d'ajustement de type 2D
CDE_RESET	0x07	Le maître demande à l'esclave de réinitialiser et de retrouver un état stable

La compréhension de ces commandes à ce stade du rapport importe peu, ce qu'il faut retenir c'est que l'on différencie deux types de commande. Celles où le maître demande un résultat et celles où le maître demande à l'esclave de travailler.

Pour rester fidèle à la documentation rédigée vis-à-vis du projet, on garde les mêmes termes d'appellation pour les types de commandes. On différencie donc deux type de commandes :

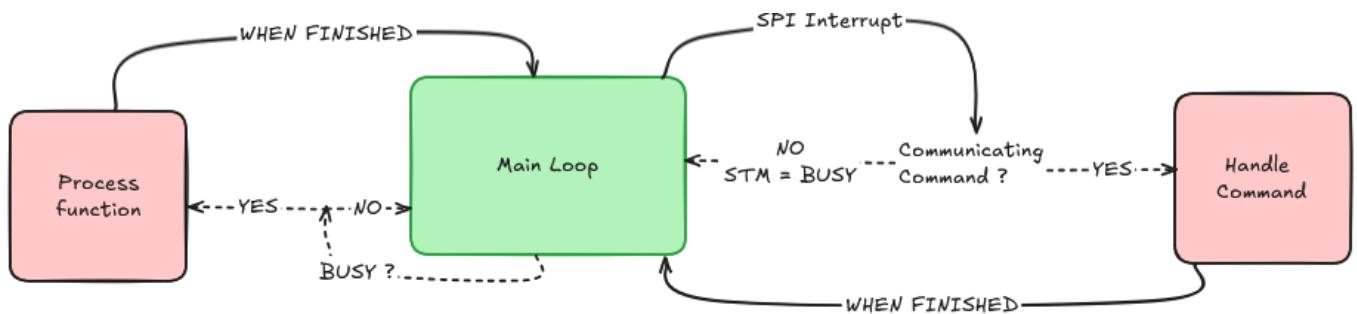
- “Communicating Commands” qui sont les commandes nécessitant une communication directe avec le maître. Ce sont globalement les commandes de demande de résultats.
- “Processing Commands” qui sont les commandes nécessitant au STM de travailler en autonomie.

À noter que toute commande ne nécessitant aucune action de la part du STM est considérée comme une “Communicating Command” et sera traitée en tant que tel.

Le but de différencier deux types de commandes concerne en réalité le fonctionnement du SPI. La réception de commandes fonctionne actuellement par interruptions, il est cependant recommandé de ne pas rester dans le gestionnaire d'interruptions. Bien qu'un gestionnaire d'interruptions intelligent existe (NVIC), il faut exécuter le moins de code possible dans une interruption et essayer d'en sortir au plus vite pour éviter toute superposition d'interruptions.

De plus, la transmission de données volumineuses via le mode d'Interruptions SPI est très complexe à mettre en place. Nous préférons donc passer par le mode *Polling* plutôt que par les interruptions.

Figure n°21 : Réception d'une commande



On observe ici qu'en fonction du type de commande, celle-ci s'exécute soit dans l'interruption SPI, soit l'Interruption SPI renvoie le code dans le main en ayant mis le statut à BUSY (cf [prochaine section](#)).

Les fonctionnalités prochainement détaillées se trouvent dans le fichier ‘acquisition.c’.

3.7.2 - Communicating commands

Par définition, ces commandes nécessitent une communication directe avec le maître. Les commandes ne nécessitant aucune action sont par défaut exécutées au même endroit : dans le gestionnaire de commande. Elles ont la particularité d'être exécutées durant l'interruption SPI et opèrent toute communication en SPI *Polling* mode.

Ce sont des commandes dites "rapides" durant lesquelles on ne risque pas de rater une commande de la part du maître car il est soit en train de nous parler, soit notre commande n'exécute aucune action et sort aussitôt de l'interruption. En faisant cela, aucune 'NVIC error', qui sont les erreurs de gestion d'interruptions, ne sera levée.

Se référer au [DSI](#) sur la prochaine page pour les explications qui vont suivre.

Le cas de la commande RESET est traitée dans une section ci-après ([3.6.4](#)).

Lors d'une réception de commande, le statut d'acquisition doit être mis à jour en fonction de la valeur de retour du gestionnaire de commande qui agit quant à lui en fonction de ce même statut :

- **BUSY** : le STM ignore la commande et renvoie le statut actuel qui est donc le statut BUSY. Lorsque le STM sort de son interruption, il retourne là où il s'en était arrêté dans sa tâche.
- **autre que BUSY** : le STM copie le buffer de réception de commande 'RxBuffer' dans le buffer de traitement 'command'. Ce buffer permet de ne pas perdre les informations si une autre interruption SPI intervient. En effet, lors d'une réception de commande, l'ISR copie directement les octets dans 'RxBuffer'. Il nous faut donc sauvegarder son contenu dans un buffer intermédiaire.

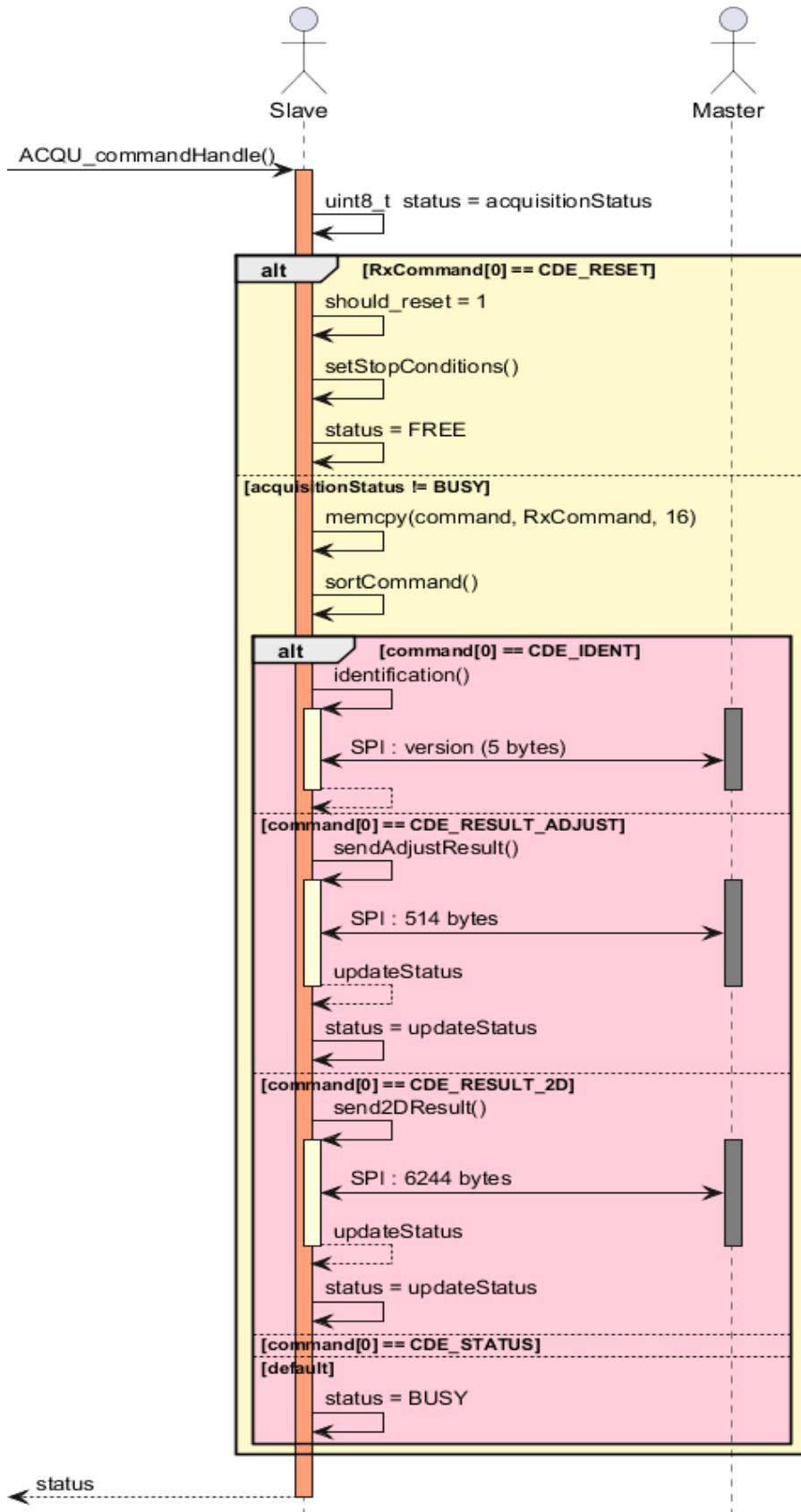
La commande est ensuite réorganisée selon le concept *Big / Little Endian*.

Enfin, le STM exécute la fonction associée au command id situé en index 0 du buffer 'command'.

Le gestionnaire de commande stocke le statut retourné par la fonction exécutée. Il retourne ensuite ce statut qui sera mis à jour en sortie d'interruption.

Si le STM n'est pas BUSY et que la commande n'est pas de type "Processing Command", le STM retourne le statut BUSY. En faisant cela, lorsque le STM sort de l'interruption, il retourne dans la boucle principale pour exécuter la fonction associée (cf [prochaine section](#)).

Figure n°22 : DSI du gestionnaire de commande

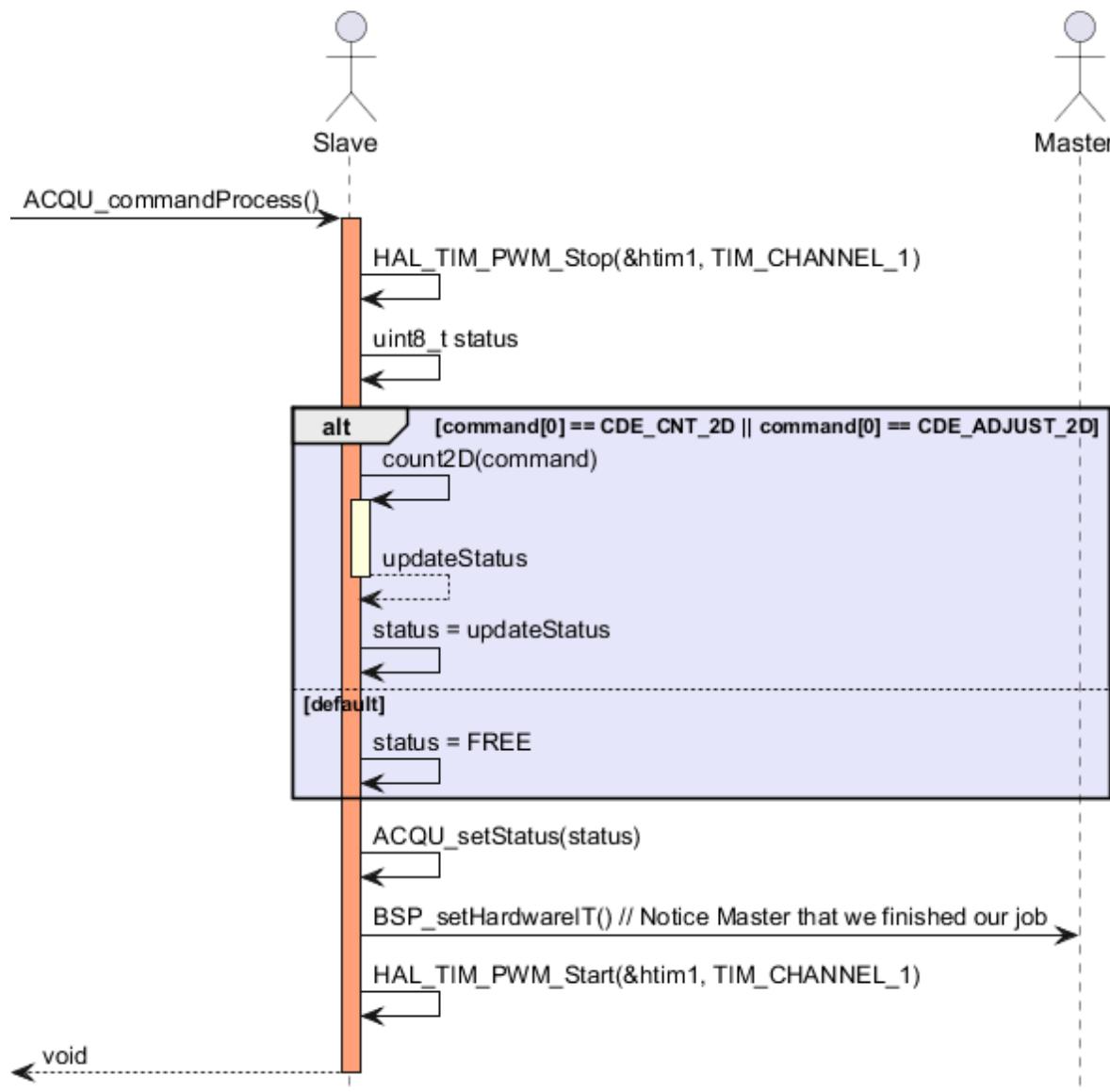


3.7.3 - Processing commands

Par définition, ces commandes lancent le STM en autonomie sur une tâche. Ce sont des tâches qui prennent du temps et qui ne sont donc pas exécutées durant les interruptions SPI.

Sur réception d'une commande de ce type, le gestionnaire de commande renvoie un statut BUSY. Ces fonctions sont exécutées dans la boucle principale d'acquisition qui vérifie en permanence le statut (cf [DSI main](#)). La fonction 'ACQU_commandProcess()' est alors appelée.

Figure n°23 : DSI des “Processing Command”



Pour coller au protocole des HC11, on éteint la LED lorsque le HC11 est en train de travailler. En faisant cela, on obtient un indicateur visuel autre que par le biais de l'analyseur logique ou du débogueur.

Il suffit pour cela de couper la génération de signaux PWM qui est connecté à la LED. Celle-ci n'étant plus alimentée, elle s'arrête de clignoter. On peut ainsi savoir si le STM travaille ou s'il attend une commande.

Pour faire un rappel avec la [section](#) sur le statut d'acquisition. Une règle a été instaurée où celui-ci ne peut être modifié qu'en sortie d'interruption **ou bien** pendant une "Processing Command". Cette précision sur les commandes de ce type est importante. En effet, en sortie d'une "Processing Command", le STM possède un statut particulier rempli d'informations pour le maître . Il nécessite donc de le modifier lorsqu'il termine son travail.

On suppose que si le statut a été mis à BUSY, c'est que l'id de commande présent dans le buffer 'command' correspond bien à une des "Processing Command".

En sortie de fonction, l'interruption hardware est levée pour spécifier au maître que le STM a fini son travail et qu'il est prêt à envoyer ses résultats. On relance alors la génération PWM pour faire clignoter la LED de test.

Ici l'ordre d'exécution est important. Le statut doit être mis à jour avant l'interruption hardware. Il faut rappeler que la mise à jour du statut réinitialise les périphériques SPI et prend un certain temps. Si l'interruption hardware est levée avant, on risque de rater la commande de réponse du maître.

PS : Il faut prendre en compte que pendant l'exécution de ces commandes les interruptions SPI sont activées. Le STM peut recevoir une commande à n'importe quel moment de la part du maître. On peut ainsi interpréter un potentiel RESET reçu sur le bus SPI, ignorer la commande sinon.

3.7.4 - Gestion de la commande RESET

Cette commande est particulière car elle demande un reset propre au STM. En effet, le maître envoie cette commande lors d'un comportement du STM qui ne lui convient pas (mauvais statut, interruption hardware levée au mauvais moment..). Il est donc important de la gérer pour pouvoir se synchroniser avec le maître.

Le problème est que nous ne pouvons pas simplement tout éteindre et tout relancer. Notre programme ne contient pas que la partie ACQUISITION. Il contient aussi la partie INIT et la partie BOOT. Recommencer à l'adresse de départ n'est donc pas la solution.

Une première solution testée a été d'essayer de mapper le point de sortie du programme avec le point d'entrée de la partie acquisition. Grâce à la fonction 'atexit()', sur un appel à 'exit()' des bibliothèques standards de C, on peut lancer une dernière fonction un peu comme un baroud d'honneur. Cependant, la fonction 'exit()' est faite pour terminer un processus. On observe alors des comportements indéterminés que l'on ne peut pas qualifier.

Dans le fichier main.c on mappe le point de sortie avec la phase d'acquisition :

```
atexit(acquisitionPhase); // works with exit() function
```

Dans le fichier acquisition.c on termine le processus sur réception de RESET :

```
// check if Master asked for a RESET
    if (RxCommand[0] == CDE_RESET)
        exit(SUCCESS); //reset main function directly to acquisitionPhase
```

Cette fonctionnalité ne pouvant pas marcher, il fallait trouver un autre moyen de terminer le programme. La problématique ici est que la commande RESET est reçue et interprétée sur interruption SPI. Même si nous avons un flag vérifié en permanence dans la boucle principale, lorsque le programme sort de l'interruption il retourne où il s'est arrêté sans passer par la fonction principale.

Sur réception d'un RESET, le code ne fait pas que mettre le flag de RESET à true, il met aussi des conditions d'arrêt. En effet, le problème étant les "Processing Command" qui sont **probablement** en train d'être exécutées lors du reset. Il faut alors trouver un moyen de les arrêter.

La définition même d'une "Processing Command" est qu'elle lance le STM en autonomie sur une tâche. Qui dit autonomie dit condition d'arrêt.

Nous avons donc par définition n'importe quelle “Processing function” qui est arrêtable à n'importe quel moment en modifiant sa condition d'arrêt à notre avantage. Nous avons maintenant sur réception de RESET :

```
// set reset flag on true
should_reset = 1;
// stop any processing command
setStopConditions();
status = FREE;
```

Avec la fonction ‘setStopConditions()’ qui doit être mise à jour au fur et à mesure du développement des autres voies.

Côté main, le flag de reset est vérifié continuellement (cf [DSI main](#)). S'il est à true, on reset les périphériques et on repart avec un départ propre.

3.8 - Principes de comptage

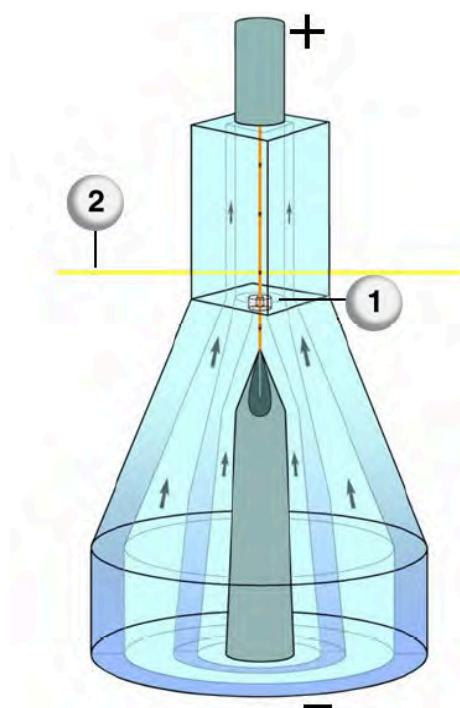
3.8.1 - Types de comptage

Comme vu précédemment, le Pentra 80 contient plusieurs technologies en son sein (COULTER et laser optique). Il regroupe non seulement plusieurs technologies mais aussi plusieurs types de comptages de cellules.

Il existe deux type de comptage :

- Comptage CBC : Ce comptage vise la détection des globules rouges et des plaquettes. Il fonctionne via le principe de mesure de COULTER. On appelle aussi ce comptage le comptage 2D.
- Comptage DIFF : Ce comptage est dit différentiel. Il vise le comptage des globules blancs / basophiles. Ce comptage est réalisé deux fois par deux capteurs différents. Une fois en même temps que les globules rouges. Une autre fois dans la cuve optique. On appelle aussi ce comptage le comptage matriciel.

Figure n°24 : Cuve optique



(1) Le volume cellulaire est mesuré par un courant électrique (variations de l'impédance).

(2) La mesure de la lumière transmise à un angle de 0° qui permet de mesurer la réponse en fonction de la structure interne de chaque cellule et de son absorbance, la lumière non absorbée traversant les espaces du matériel nucléaire de chaque cellule. Il s'agit du principe de diffusion de la lumière.

Nous nous intéresserons seulement au comptage CBC pour la suite du rapport (i.e comptage 2D). La voie développée étant la voie des plaquettes, le comptage différentiel ne nous intéresse pas ici.

3.8.2 - Protocole du comptage 2D

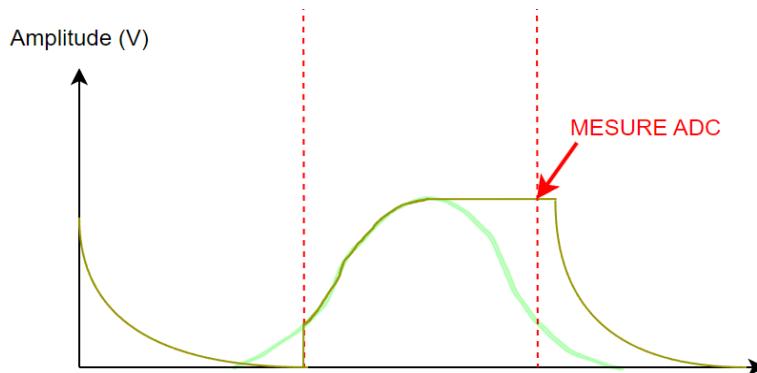
Le comptage 2D fonctionne selon le principe de COULTER (cf 1.2.2). Nous avons donc des impulsions créées par la variation d'impédance lors du passage de la cellule entre deux électrodes. Côté embarqué, on travaille avec ce signal.

Une couche matérielle sépare le STM de la génération des impulsions. Cette couche matérielle est complexe et s'articule autour d'un CPLD qui gère un certain nombre d'actions pour faciliter les interactions avec les HC11.

Ce qu'il faut retenir c'est qu'un pré-traitement est effectué sur les impulsions:

- Un premier pour supprimer le bruit du signal (largeur<10 µS | hauteur<390 mV). En faisant cela, les impulsions parvenant au STM32 sont pré-qualifiées.
- Un second pour maintenir la hauteur du signal.
Ainsi, le STM peut récupérer la valeur du signal une fois prêt sans risquer de la rater. Le STM émet ensuite une impulsion sur une broche spécifique (INIB) qui informe au CPLD que la valeur a été récupérée et qu'il peut relâcher le signal.

Figure n°25 : Visualisation de la réception d'une impulsion



On observe le signal maintenu jusqu'à la lecture de la valeur ADC par le CPLD.

Le STM reçoit donc des impulsions qu'il doit ensuite qualifier et stocker en mémoire selon **deux** paramètres :

- La **hauteur** de l'impulsion récupérée via une conversion ADC.
- La **largeur** de l'impulsion récupérée via un Timer. On l'obtient en effectuant la différence des temps relevés entre le front montant et le front descendant.

Le lancement du comptage 2D côté STM est dû à la réception d'une commande sur le bus SPI. Cette commande est CDE_CNT_2D associée à l'octet 0x04.

Le comptage 2D stocke trois valeurs :

- Le nombre d'éléments en fonction des hauteurs d'impulsion sur 2 octets.
- La somme des hauteurs d'impulsion sur 4 octets
- Le nombre total de cellules sur 4 octets. (compteur immédiat)

Le comptage s'effectue par tranches de comptage. Sur cette commande, 6 tranches de comptage sont effectuées de une seconde chacune.

Cette commande fonctionne avec des numéros de comptage (1 et 2). Chaque numéro de comptage correspond à 6 tranches. Il y a en réalité 12 tranches au total.

La première des trois valeurs est stockée dans un tableau de short :

```
uint16_t data[12][256]; // Buffer to store measured data
```

Chaque tranche possède un index où stocker ses valeurs. À chaque fois que l'on passe à une autre tranche, cet index est incrémenté.

Les hauteurs d'impulsions sont converties par l'ADC sur un octet. Ces valeurs vont donc de 0 à 255. Chaque tranche va donc incrémenter son tableau proportionnellement à la valeur lue par l'ADC.

Par exemple, si l'on est dans la tranche n°4 et que l'on vient de relever un impulsion avec pour hauteur la valeur 153 :

```
data[4][153]++;
```

Ainsi les éléments sont stockés proportionnellement à leur valeur.

Les deux autres valeurs ne nécessitent pas un tableau à deux dimensions dans la mesure où la somme est opérée sur toute la tranche. Nous avons donc pour la somme des hauteurs et le nombre de cellules:

```
pulseHeightSum[n°Tranche] += pulseHeight;
counter[n°Tranche]++;
```

Les paramètres du comptage 2D sont reçus sur le bus SPI. Ce sont les 15 octets qui suivent l'id de commande.

Une structure à été créée pour stocker les paramètres reçus et ainsi pouvoir les réutiliser au travers des fonctions :

```
typedef struct {  
    // params  
} CNT_2D_PARAMS;
```

On y retrouve la largeur de pulse maximale, le nombre de tranches de comptage, le numéro de comptage, la durée d'une tranche de comptage ainsi que le canal concerné. Le canal peut être PLT (plaquettes), RBC (Red Blood Cells)..

Le reste des paramètres ne sont pas reçus par communication SPI mais ont été créés pour opérer les fonctionnalités. On retrouve l'index de stockage des tranches, une variable temporelle pour gérer le changement de tranches via les Timers, le tableau du compteur immédiat ainsi qu'un booléen spécifiant le type de comptage 2D.

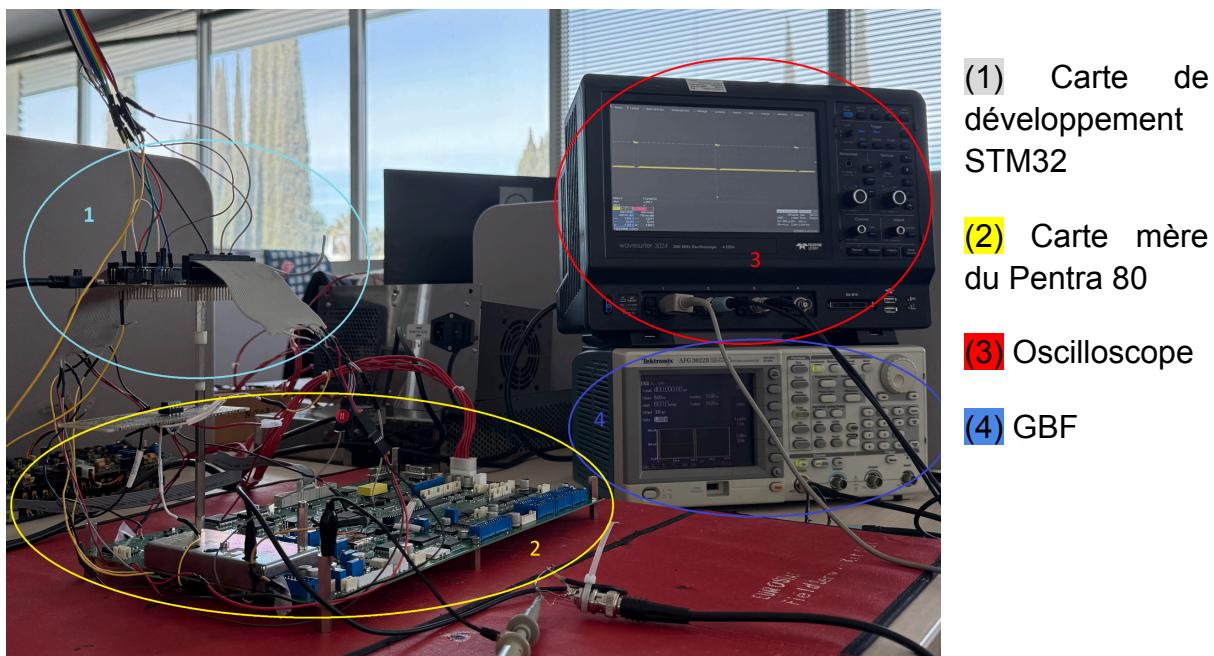
3.9 - Comptage 2D

Le développement du comptage 2D englobe de nombreux concepts. Il y a la partie acquisition / traitement des données mais aussi l'envoi des données sur le bus SPI.

3.9.1 - Environnement de travail

Les tests concernant cette partie ont été effectués sur table dans un premier temps. À l'aide d'un GBF et d'un oscilloscope. Le GBF permet de générer des impulsions. Avec les bons paramètres, ces impulsions s'apparentent aux cellules que l'on veut mesurer. Quant à l'oscilloscope, il nous permet de vérifier qu'il n'y a pas d'écart de signal entre ce que génère le GBF et ce qui est reçu au niveau de la carte.

Figure n°26 : Environnement de test



À noter que deux fonctionnalités sont en réalité présentes dans le comptage 2D. La seconde est une fonctionnalité de "réglage de la machine". On traite ici la fonctionnalité de base destinée aux laborantins.

La partie acquisition fonctionne principalement sous interruptions, la fonction principale que nous allons parcourir prochainement ne fait que activer les interruptions des périphériques, attendre la condition d'arrêt, couper les interruptions des périphériques pour sortir proprement.

3.9.2 - Développement de la fonctionnalité

La fonction principale du comptage 2D est une fonction d'architecture. En effet, cette fonction se contente de gérer les périphériques mais ne possède aucun code quant au traitement des données. La majeure partie du traitement est réalisée sous interruptions et donc, dans les fonctions de callback associés.

À noter qu'une structure de paramètres pour le comptage 2D a été instanciée en tant que variable globale pour pouvoir être utilisée par toutes les librairies.

Il n'y a pas forcément de continuité entre les fonctions, il n'est donc pas possible de déclarer la structure interne à la fonction et de la passer en paramètre.

Cette structure a été instanciée en tant que 'cnt2D'. Chaque appel à ce paramètre fait donc référence à cette structure.

Revenons-en à la fonction principale du comptage 2D. Cette fonction va nous permettre d'avoir un aperçu global de quels périphériques sont utilisés pendant un comptage 2D. La première partie concerne l'initialisation des périphériques ainsi que l'activation des interruptions :

- La fonction 'init2D()' initialise la structure étudiée précédemment en récupérant les paramètres contenus dans le buffer 'command'.
- Une impulsion sur INIB est générée pour notifier le CPLD de relâcher le signal au cas où le signal aurait été maintenu.
- La conversion ADC via le DMA est lancée. En systématique, le paramètre 'pulseHeight' est mis à jour en permanence.
- Les interruptions du Timer numéro 2 sont activées. Dorénavant, une interruption est générée toutes les 10 millisecondes.
- Le compteur immédiat est lancé en comptage 2D.

On active ensuite les interruptions sur front montant et sur front descendant du Timer numéro 2. Les broches PA0 et PA1 sont configurées pour capter ces interruptions.

S'ensuit la partie de comptage des cellules. La fonction principale se contente de bloquer le processeur jusqu'à ce que les tranches de comptage se terminent :

```
/* ----- Processing Counting -----*/
while(cnt2D.slicesRemaining > 0); // while slices are left to count
```

Enfin, une fois les tranches terminées, on termine le programme proprement en terminant les interruptions de tous les périphériques.

Une fois les interruptions coupées, on retourne un statut bien particulier. Il contient l'octet ACKNOWLEDGE , le canal qui s'adresse au maître ainsi que le numéro de comptage. Le décalage d'octet (<<) permet de ne pas *override* les informations :

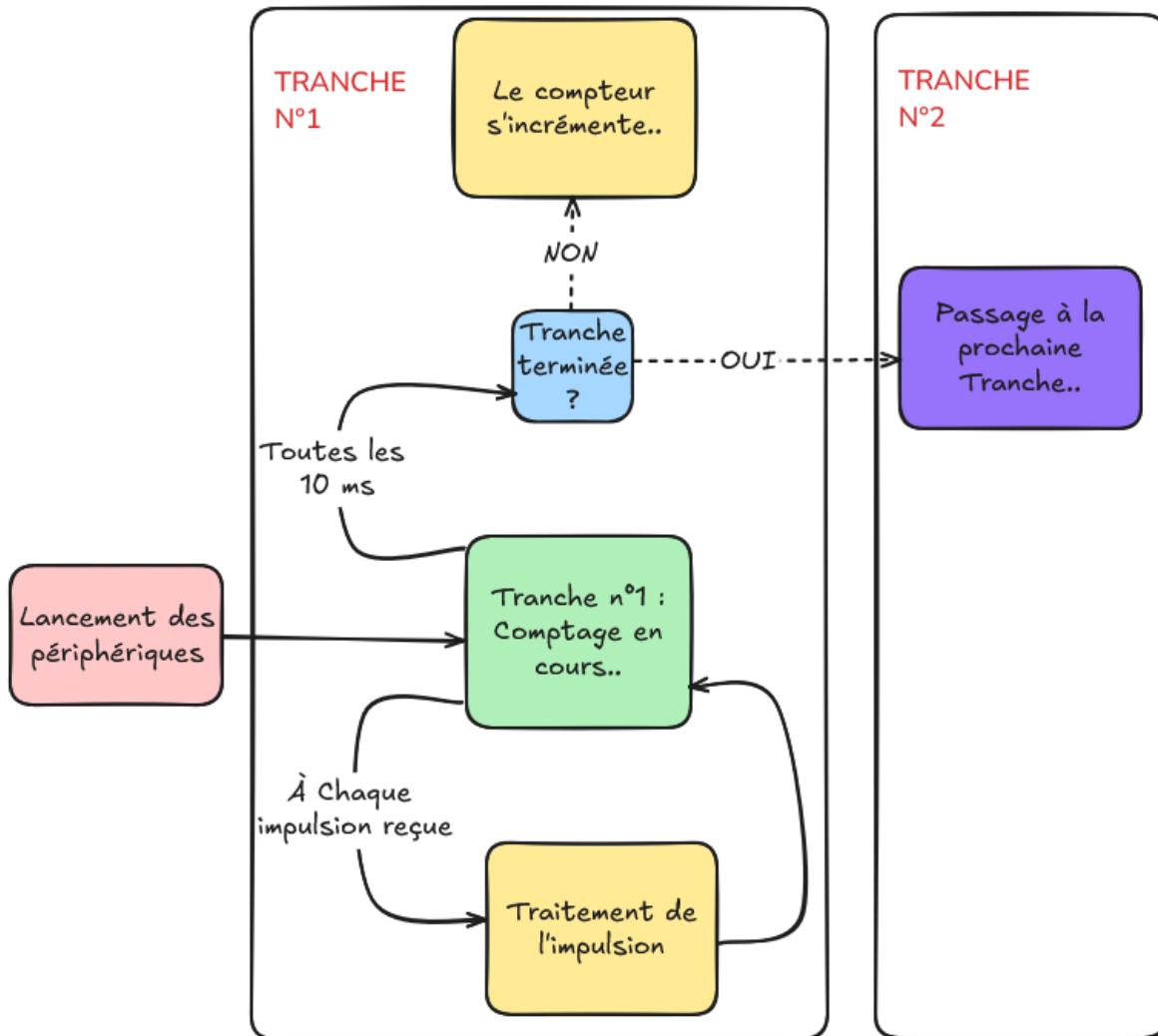
- L'octet ACKNOWLEDGE a pour valeur 0x40.
- Le canal PLT a pour valeur 0x03
- Le numéro de comptage quant à lui peut prendre deux valeurs : 1 ou 2
On prend la valeur 0x01 pour l'exemple.

$$\begin{array}{c} 0x40 \quad | \quad 0x03 \quad | \quad 0x01 \\ \Leftrightarrow 0100\ 0000 \mid 0000\ 0011 \mid 0000\ 0001 = 0100\ 0100 \end{array}$$

$$\begin{array}{c} 0x40 \quad | \quad 0x03 \ll 2 \mid 0x01 \\ \Leftrightarrow 0100\ 0000 \mid 0000\ 1100 \mid 0000\ 0001 = 0100\ 1101 \end{array}$$

Expliquer du code fonctionnant sous interruptions est relativement complexe, voici un schéma du fonctionnement d'une tranche de comptage pour aider à la compréhension du processus complet :

Figure n°27 : Fonctionnement d'une tranche de comptage



On y retrouve les deux types d'interruptions, une pour le traitement des impulsions et une pour l'incrémentation du compteur. On observe aussi la vérification du compteur toutes les dix millisecondes nous permettant de savoir s'il faut passer à la tranche suivante ou non.

3.10 - Fonctionnement du Timer

Un Timer est un périphérique matériel permettant de gérer des tâches de chronométrage ou de comptage et cela indépendamment du processeur. Il est donc autonome et fonctionne en parallèle.

Il se configure selon les valeurs contenues dans ses registres :

- CNT (Counter) : C'est le compteur du Timer. Une valeur sur deux octets incrémentée à chaque tic d'horloge du Timer. À noter que ce registre n'est pas configurable mais important pour la compréhension du fonctionnement.
- ARR (Auto Reload Register) : Il définit la valeur maximale que le compteur peut atteindre avant de se réinitialiser. Lorsque le compteur atteint cette valeur, le Timer génère un événement de mise à jour (update event) et réinitialise le compteur à 0. On peut grâce à cet évènement générer une interruption et agir périodiquement.
- PSC (Prescaler) : Le prescaler divise la fréquence d'horloge du Timer par une valeur définie, ce qui permet de réduire la fréquence à laquelle le Timer compte. Par exemple, si l'horloge du système est de 16 MHz et que le prescaler est réglé à 16, la fréquence de comptage du Timer sera de 1 MHz. Le Timer incrémentera donc son compteur une fois par microseconde

Des relations entre ces registres nous permettent de calculer les valeurs recherchées comme la fréquence de comptage, le délai en secondes.. :

$$\text{fréquence de comptage} = \frac{\text{fréquence d'horloge}}{(PSC + 1)}$$

$$\text{délai (secondes)} = \frac{(ARR + 1)}{\text{fréquence de comptage}}$$

$$\text{fréquence de ticks} = \frac{1}{\text{fréquence de comptage}}$$

Un exemple sera toujours plus parlant :

Supposons que l'on veut configurer un Timer pour générer une interruption toutes les secondes avec une horloge système de 16 MHz. On peut régler le prescaler à 15 999 pour obtenir une fréquence de comptage de 1 kHz. On configure ensuite le registre d'auto-reload (ARR) à 999 pour que le Timer se réinitialise après 1000 ticks, soit 1 seconde.

3.11 - Timer de comptage

Le STM32 est une technologie récente qui peut être très performante. Nous avons cependant la contrainte de devoir coller à l'existant. Le Timer de comptage doit donc fonctionner selon les caractéristiques de son prédecesseur : le HC11.

3.11.1 - Configuration du Timer

Les HC11 possèdent une fréquence d'horloge de Timer à 3 MHz. Ils fonctionnent avec un ARR générant un 'update event' toutes les 10 millisecondes.

Le STM32G0B1RE peut quant à lui augmenter sa fréquence d'horloge jusqu'à 64 MHz. Il faut donc configurer la fréquence d'horloge du STM32 pour qu'il puisse avoir une fréquence d'horloge à 3 MHz vis-à-vis du Timer tout en restant performant.

Cela correspond à avoir une fréquence d'horloge multiple de 3. On choisit donc de configurer sa fréquence d'horloge à 63 MHz.

On a donc la configuration du Timer de comptage suivante (i.e TIMER 2) :

Nom du registre	Valeur
PSC	20
ARR	29999

On obtient ainsi :

$$\text{fréquence de comptage} = \frac{63\,000\,000}{(20+1)} = 3\,000\,000$$

$$\text{délai (s)} = \frac{(29\,999+1)}{3\,000\,000} = 0,01 \text{ secondes} \Leftrightarrow 10 \text{ millisecondes}$$

$$\text{fréquence de tic} = \frac{1}{3\,000\,000} = 333 \text{ nanosecondes}$$

Le Timer possède désormais la bonne configuration matérielle. Il faut maintenant configurer les broches adéquates pour pouvoir s'en servir. Nous avons d'une part, la broche PA0 configurée en 'Input Capture' sur front montant. D'autre part, la broche PA1 est configurée en 'Input Capture' sur front descendant.

Ces broches sont connectées sur le signal d'entrée des impulsions. Cette configuration nous permet donc de générer une interruption sur front montant et front descendant.

3.11.2 - Interruptions Timer sur front Montant / Descendant

Ce Timer nous permet de gérer le temps de comptage de chaque tranche ainsi que les largeurs d'impulsions reçues en entrée sur PA0 et PA1.

Il y a donc deux type d'interruptions à gérer :

- Les interruptions de front montant et de front descendant
- Les interruptions d'update event générées par l'overflow de l'ARR

Des variables globales au fonctionnement du comptage 2D sont déclarées avant les fonctions. Elles permettent de gérer les interruptions et le bon fonctionnement des comptages :

```
// ADC DMA storing variable
uint8_t pulseHeight;
// global timer variables
uint16_t pulseStart;
uint16_t pulseEnd;
uint16_t pulseWidth;
```

Commençons par les interruptions de front. La fonction de callback associée se présente comme suit :

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    switch(htim->Channel)
    {
        // FALLING EDGE
        case HAL_TIM_ACTIVE_CHANNEL_2:
        // RISING EDGE
        case HAL_TIM_ACTIVE_CHANNEL_1:
        // DEFAULT
        default:
            break;
    }
}
```

On rentre dans un switch case qui nous permet de déterminer quel front, ou plutôt quelle broche a levé l'interruption. Chaque front possède une action spécifique que nous allons expliquer en détail. Pour des raisons électroniques, les fronts sont inversés. Le premier front est donc le front descendant et le second le front montant.

Commençons par le front le plus simple : front descendant. On se contente de relever la valeur du compteur et de la stocker dans une variable :

```
pulseStart = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_2);
```

S'ensuit alors le front montant : On commence en faisant la même chose que le front descendant. On vient donc de récupérer le deuxième paramètre pour traiter la largeur de la pulse, on opère alors la différence tout en traitant le cas de l'*ARR overflow* :

```
/*----- Treating pulse width-----*/
if(pulseEnd > pulseStart)
    pulseWidth = pulseEnd - pulseStart;
else
    // Timer went back to 0 between captures
    pulseWidth = htim2.Init.Period - pulseStart + pulseEnd;
```

On passe ensuite à l'étape de qualification de l'impulsion :

```
/*----- Qualification -----*/
if(pulseWidth <= cnt2D.maxPulseWidth &&
    (!BSP_getSeuilHaut() || cnt2D.channel != PLT))
{
    // treating pulse
}
```

On remarque que l'impulsion doit posséder une largeur inférieure à la largeur maximale. La seconde partie de la condition logique possède en revanche un concept qui n'a pas encore été vu.

On introduit donc ici le concept de “**Seuil Haut**”. Ce seuil haut est une broche spécifique configurée en entrée. Cette broche est directement connectée au CPLD qui nous spécifie si le seuil a été atteint ou non. Ce “seuil haut” concerne les plaquettes. Si le seuil haut est dépassé et que le canal concerné est bien les plaquettes, alors, l'impulsion ne doit pas être qualifiée. D'où la condition logique qui vérifie que le canal soit différent des plaquettes, ou bien, que le seuil n'ait pas été dépassé.

On suppose que l'impulsion a été qualifiée, on opère maintenant le traitement de ses données. On incrémente donc le tableau de la somme des éléments selon leur hauteur, puis, on ajoute sa hauteur au tableau qui en fait la somme :

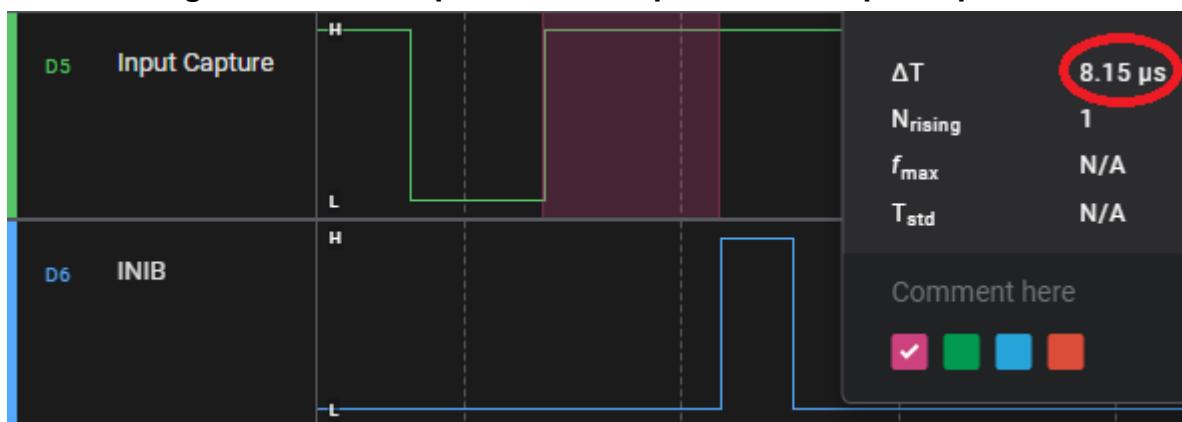
```
// pulseHeight is being updated by DMA
    cnt2D.data[cnt2D.storingIndex][pulseHeight]++;

    if (!cnt2D.isAdjust)
        cnt2D.pulseHeightSum[cnt2D.storingIndex] += pulseHeight;
```

On termine ensuite en émettant une impulsion sur INIB spécifiant au CPLD de relâcher le signal :

```
/* ----- Exiting ----- */
// BIT SET BIT CLEAR EOC pin
BSP_triggerINIB();
```

Figure n°28 : Réception d'une impulsion sur input capture



On observe ici le front descendant suivi du front montant sur input capture. L'impulsion sur input capture est suivie d'une impulsion émise sur INIB à destination du CPLD.

On observe aussi un intervalle de temps entre l'impulsion reçue et l'impulsion émise d'environ huit microsecondes. On peut supposer que cet intervalle de temps correspond au temps mis par le processeur pour qualifier l'impulsion.

3.11.3 - Interruptions Timer sur ‘update event’

L'interruption dite ‘update event’ est générée lorsque le compteur de notre Timer atteint la valeur de l'Auto Reload Register. Ce registre nous permet de générer des interruptions périodiquement. Le Timer de comptage a été réglé de sorte à ce qu'un ‘update event’ surviennent toutes les 10 millisecondes.

En effet, les paramètres de comptage 2D nous sont passés selon une base 10. Si le paramètre de durée d'une tranche de comptage vaut 90, il vaut en réalité 900 millisecondes. En l'occurrence, les tranches de comptage 2D ont une durée d'une seconde, ce paramètre nous est envoyé avec la valeur 100 (0x64).

Nous possédons dans la structure une variable permettant de gérer le changement de tranches vis-à-vis de cet ‘update event’ :

```
uint16_t timerCount; // Incremented parameter, count up to sliceDuration
```

Pour une question de compréhension du code, on choisit de multiplier par 10 le paramètre de durée des tranches lors de l'initialisation des paramètres de comptage.

```
cnt2D.sliceDuration = BSP_CAST_short(&command[5]) * 10;
```

Le paramètre timerCount sera incrémenté dix par dix. On peut ainsi représenter les dix millisecondes ajoutées à chaque ‘update event’ explicitement.

La fonction de callback associée à notre interruption se présente comme suit :

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    // TIM3 Auto Reload Register overflow
    if(htim->Instance == TIM3)
    {
    }
    else // TIM2 Auto Reload Register overflow (every 10 ms)
    {
    }
}
```

Nous gérons deux Timers différents dans cette fonction : Le timer du compteur immédiat ainsi que le Timer de comptage. Nous traiterons ici le Timer de comptage (i.e TIM2).

Si la tranche n'est pas terminée, on se contente d'incrémenter la variable 'timerCount' de dix :

```
if (cnt2D.timerCount < cnt2D.sliceDuration) // SLICE NOT TERMINATED
{
    cnt2D.timerCount += 10; // Interrupts base 10ms
}
```

En revanche, si la tranche est terminée, on doit changer de tranche. On remet alors le compteur à zéro et on décrémente le nombre de tranches restantes. À noter que ce nombre de tranches est aussi notre condition d'arrêt, une fois cette variable mise à 0, la fonction principale terminera les interruptions :

```
else // END OF SLICE
{
    cnt2D.timerCount = 0; // reload sliceDuration
    cnt2D.slicesRemaining--;
    if (!cnt2D.isAdjust)
        endOfSlice_COUNT2D();
    else
        endOfSlice_ADJUST2D();
}
```

On rentre ensuite dans une fonction intermédiaire, en effet, en fonction du mode de comptage, on n'effectue pas les mêmes actions. En réalité, le contenu de cette fonction concerne bien plus le fonctionnement du compteur immédiat. Ce qui nous intéresse dans cette fonction ne concerne qu'une seule ligne :

```
cnt2D.storingIndex++;
```

On vient incrémenter l'index de stockage. En effet, les tranches stockent leurs valeurs en fonction de cet index qui est incrémenté à chaque fin de tranche. Le changement de tranche est ainsi opéré.

Comme vous avez pu le voir durant l'explication du traitement des impulsions, le compteur immédiat fonctionne aussi via un Timer.

3.12 - Compteur immédiat

Le compteur immédiat nous permet de quantifier le nombre de cellules rencontrées dans l'échantillon de sang. Il fonctionne différemment du principe de mesure du Timer de comptage. En effet, il ne se contente pas de compter les cellules pré-qualifiées, il compte aussi les cellules non qualifiées.

Une broche provenant du CPLD à destination du STM émet une impulsion à chaque fois qu'une cellule passe entre les électrodes. Nous pouvons donc quantifier les impulsions émises sur cette broche. C'est là que le Timer intervient (encore).

En effet, le timer est un périphérique polyvalent, jusqu'à présent notre Timer était configuré et cadencé selon l'horloge système. Le comptage s'effectue alors en interne. Ici, le compteur immédiat fonctionne différemment.

3.12.1 - Configuration du Timer

Le Timer du compteur immédiat est le Timer numéro 3. Par définition, un Timer incrémentera son compteur à chaque tic d'horloge. Nous possédons ici une impulsion émise sur une broche que nous devons quantifier. Il suffit alors de configurer le Timer suivant une horloge externe au système. Cela nous permet d'avoir le nombre d'impulsions émises dans le registre CNT.

La configuration du Timer se présente comme suit :

Nom du Registre	Valeur
PSC	0
ARR	0xFFFF (i.e 65 535)

On ne divise pas l'horloge externe, on veut compter toutes les impulsions (PSC = 0). On configure l'ARR à sa valeur maximum pour éviter au plus possible les overflows.

Figure n°29 : Configuration du Timer en horloge externe

The screenshot shows a software interface for configuring the TIM3 timer. At the top, it says "TIM3 Mode and Configuration". Below that is a "Mode" section with a dropdown menu set to "Slave Mode External Clock Mode 1". To the left, there is a grey box containing "(52)-PD2:" and a list item "• [Immediate Counter] TIM3_ETR (Trigger Source ETR1)". To the right of the dropdown, there is explanatory text: "La broche PD2 est quant à elle configurée en ETR (External).".

3.12.2 - Fonctionnement du compteur immédiat

Le compteur immédiat est géré uniquement en fin de tranche. On retrouve la gestion de son overflow dans la fonction de callback associée, celle-ci s'occupe d'incrémenter une variable qui s'appelle ‘nbLoop’, plutôt explicite non ?

On a donc la variable ‘nbLoop’ incrémentée lors d’un ‘update event créé par le Timer 3. Cette variable nous permet de garder une trace d’une potentielle remise à 0 de la valeur du compteur.

À noter que cette remise à 0 ne devrait jamais arriver en systématique. Il faudrait pour cela que plus de 65 535 éléments soient comptés lors d’une tranche. On en compte en moyenne 30 000..

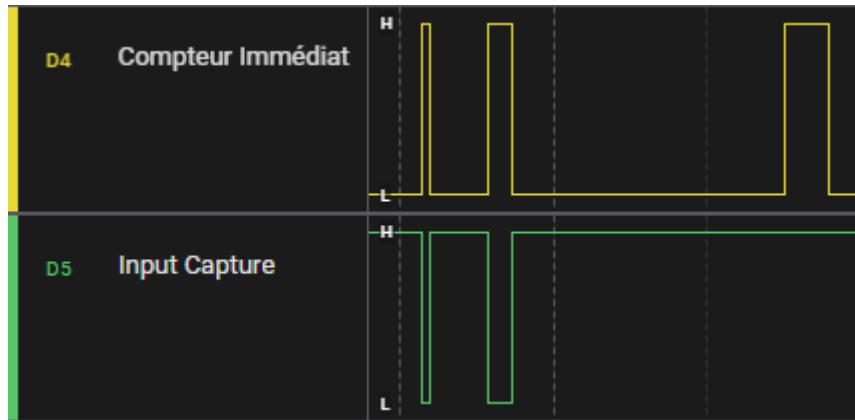
La valeur du compteur immédiat est ensuite récupérée à chaque fin de tranche et stockée dans le tableau associée. Dans la même fonction de callback donc :

```
static void endOfSlice_COUNT2D()
{
    // store slice's elements number
    cnt2D.counter[cnt2D.storingIndex] = __HAL_TIM_GET_COUNTER(&htim3) +
    nbLoop * AUTORELOAD_MAX_VALUE;
    // Reset immediate counter
    __HAL_TIM_SET_COUNTER(&htim3, 0);
    __HAL_TIM_CLEAR_FLAG(&htim3, TIM_FLAG_UPDATE);
    nbLoop = 0;
    cnt2D.storingIndex++;
}
```

On observe ici le stockage de la valeur récupérée du compteur dans le tableau ‘counter’ à l’index de la tranche de comptage en cours. On y retrouve aussi la gestion de l’overflow avec la variable ‘nbLoop’ multipliée par 65 535 puis remise à 0 en même temps que le compteur de notre Timer. On retrouve aussi l’index de stockage de tranche qui est incrémenté pour opérer le changement de tranche.

On peut vérifier que le compteur immédiat compte bien des impulsions qui ne parviennent pas sur les broche d'input capture :

Figure n°30 : Compteur immédiat versus input capture



On observe ici que le compteur immédiat reçoit les impulsions d'input capture. En effet, le signal du compteur immédiat est même symétrique à celui d'input capture. On retrouve en réalité ici l'impulsion telle qu'elle est générée.

Revenons-en à notre comparaison. On observe bien que le compteur immédiat reçoit les mêmes impulsions que sur input capture. Cependant, on observe aussi qu'une impulsion apparaît sur le compteur immédiat et n'apparaît pas sur input capture. On en déduit donc que leurs signaux diffèrent bel et bien et qu'ils ne viennent pas de la même source. Chaque signal est soumis à un pré-traitement différent.

3.13 - Fonctionnement de l'ADC

Maintenant que le fonctionnement du SPI et des Timers a été détaillé, il nous reste le dernier périphérique à documenter : l'ADC.

L'ADC pour **Analog to Digital Converter** permet de convertir des valeurs analogiques en valeurs numériques. On s'en sert ici pour convertir la hauteur des impulsions enregistrées en une valeur interprétable par notre MCU.

3.13.1 - Configuration de l'ADC

Comme pour le SPI, il existe plusieurs modes d'utilisation pour l'ADC. La plus commune étant le *polling mode*. Encore une fois, cette méthode bloque la charge du CPU pour opérer la conversion.

En effet, l'ADC convertit des valeurs analogiques, cependant, demander au processeur de lancer l'ADC, aller récupérer la valeur convertie, la traiter puis arrêter l'ADC peut être demandant en charge de processeur.

C'est là qu'intervient le DMA. Grâce au **Direct Access Memory**, l'ADC fonctionne en continu, la valeur convertie est placée périodiquement à l'adresse spécifiée en paramètre lors du lancement de l'ADC. Selon la configuration, on peut décider d'à quelle fréquence l'ADC doit mettre à jour la variable.

Grâce à ce fonctionnement, on économise de la charge de processeur pour opérer d'autres opérations en parallèle. On configure l'ADC pour rafraîchir la variable tous les 80 cycles d'horloge soit environ toutes les microsecondes.

Les HC11 possèdent un ADC convertissant des valeurs sur 8 bits. On configure alors l'ADC du STM sur 8 bits. Les valeurs converties vont alors de 0 à 255. 255 étant la valeur maximale de conversion de l'ADC. Si l'ADC convertit de 0 à 5V, 5V correspond à 255.

Figure n°31 : Configuration de l'ADC sur 8 bits

▼ ADC_Settings		
Clock Prescaler		Asynchronous clock mode divided by 1
Resolution		ADC 8-bit resolution

On rencontre alors à ce moment un problème matériel concernant l'ADC.

3.13.2 - Pont diviseur de tension

Les HC11 possèdent un ADC numérisant des valeurs entre 0 et 5V alors que les STM32G0B1RE possèdent un ADC numérisant des valeurs entre 0 et 3.3V.

Nous avons donc une valeur de conversion pour une tension de 3.3V d'HC11 qui ne correspond pas à une valeur de conversion pour une tension de 3.3V d'STM32.

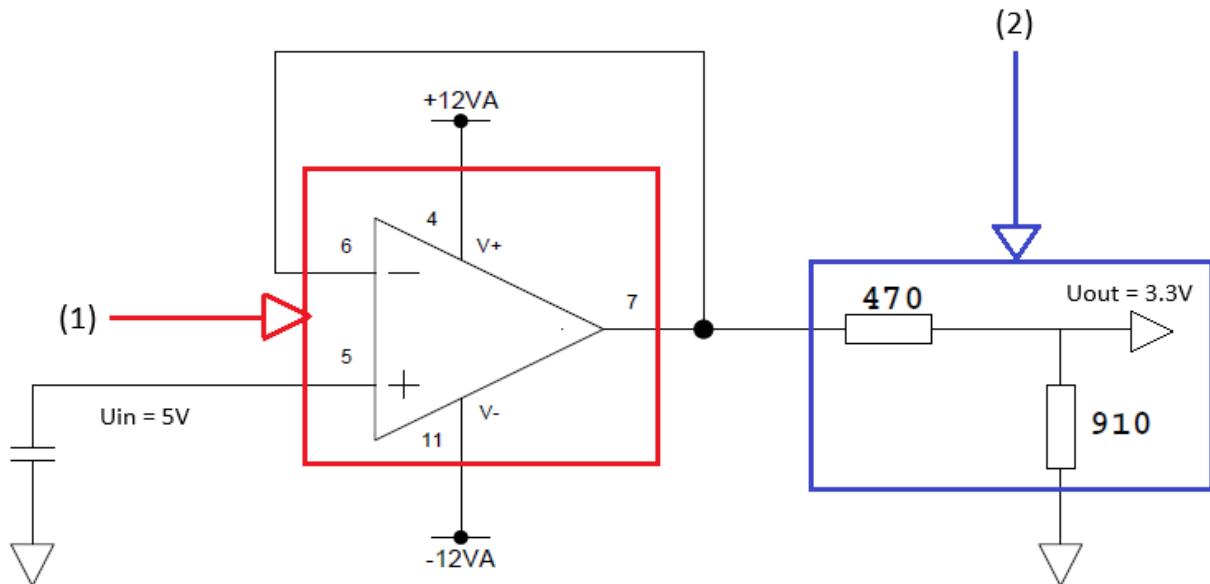
Nous devons donc opérer une conversion de la valeur à numériser pour les STM32 et ainsi obtenir des valeurs cohérentes qui ne font pas saturer l'ADC du STM32.

Ce problème étant de l'ordre du hardware, la solution était alors de rajouter un montage électronique avant l'entrée analogique. Ce montage permet de convertir la valeur à numériser suivant le bon rapport ($5V / 3.3V$ soit rapport = 0,66)

Suivant le schéma électrique de la carte, la tension à numériser est stockée dans un condensateur. On ne peut donc pas simplement insérer un pont diviseur de tension directement après le condensateur, sinon, le condensateur se décharge au travers des résistances ce qui entraîne une modification de la valeur à numériser. Nous insérons donc un AOP avant le pont diviseur pour palier à ce problème.

- (1) Un AOP réalisant la fonction d'isolation galvanique
- (2) Un pont diviseur de tension suivant les valeurs 470K et 910K

Figure n°32 : Schéma électrique du pont diviseur de tension



Ce problème ayant déjà été rencontré sur d'autres cartes au sein de l'entreprise, le montage précédent a déjà été qualifié, nous partons donc du prédicat qu'il fonctionne et qu'il est robuste.

3.13.3 - Fonctionnement de l'ADC

Comme vu précédemment, l'ADC est activé dans la fonction principale de comptage 2D en mode DMA, les valeurs converties sont directement transmises à l'adresse de la variable associée :

```
//start ADC DMA conversion
    HAL_ADC_Start_DMA(&hadc1, (uint32_t *) &pulseHeight, 1);
```

On lance la conversion sur 1 octet.

La valeur est ensuite récupérée et stockée à chaque fois qu'une impulsion est qualifiée dans la fonction de callback sur front montant :

```
/* ----- Qualification ----- */
if(pulseWidth <= cnt2D.maxPulseWidth &&
    (!BSP_getSeuilHaut() || cnt2D.channel != PLT))
{
    cnt2D.data[cnt2D.storingIndex][pulseHeight]++;
    // pulseHeight updated by DMA

    if (!cnt2D.isAdjust)
        cnt2D.pulseHeightSum[cnt2D.storingIndex] += pulseHeight;
}
```

Le tableau de somme des éléments selon leur hauteur est incrémenté suivant la valeur convertie par l'ADC ainsi que le tableau de somme des hauteurs.

3.14 - Envoi des résultats du comptage 2D

L'envoi des résultats sur le bus SPI se fait suite à l'activation de l'interruption hardware par l'esclave. En effet, le STM possède une broche (PA4) à destination du maître qu'il peut activer ou désactiver. Une fois cette broche activée, le maître sait que les résultats sont prêts. Il envoie alors la commande CDE_RESULT_2D soit l'octet 0x0C pour récupérer les résultats.

Il existe tout un protocole d'envoi des données. Pour le comptage 2D basique, on envoie en premier la taille du paquet qui va être transmis sur le bus SPI.

Ce paquet est constitué de plusieurs informations :

- Le nombre de tranches de comptage en deux fois (nombre de tranches par numéro de comptage). Soit l'envoi de l'octet 0x06 deux fois. (**2 octets**)
- Le nombre de cellules rencontrées, soit un tableau de 12 word sur 4 octets. Cela représente l'envoi de **48 octets**.
- La somme des hauteurs d'impulsions, soit un second tableau de 12 word sur 4 octets. Cela représente l'envoi de **48 octets**.
- La somme des éléments suivant leur hauteur d'impulsion, soit un tableau de 12 par 256 short sur 2 octets. Cela représente l'envoi de **6144 octets**.

On se retrouve au total avec un paquet d'une taille de : $2 + 48 + 48 + 6144 = 6242$.

Environ 6 Kilo octets sont transférés sur le bus SPI à chaque fois qu'un comptage 2D est lancé par le maître. La fonction d'envoi se décompose en cinq parties :

- On convertit d'abord les word en mémoire car on s'est rendu compte que cela prenait trop de temps pour les inverser entre les communications SPI.

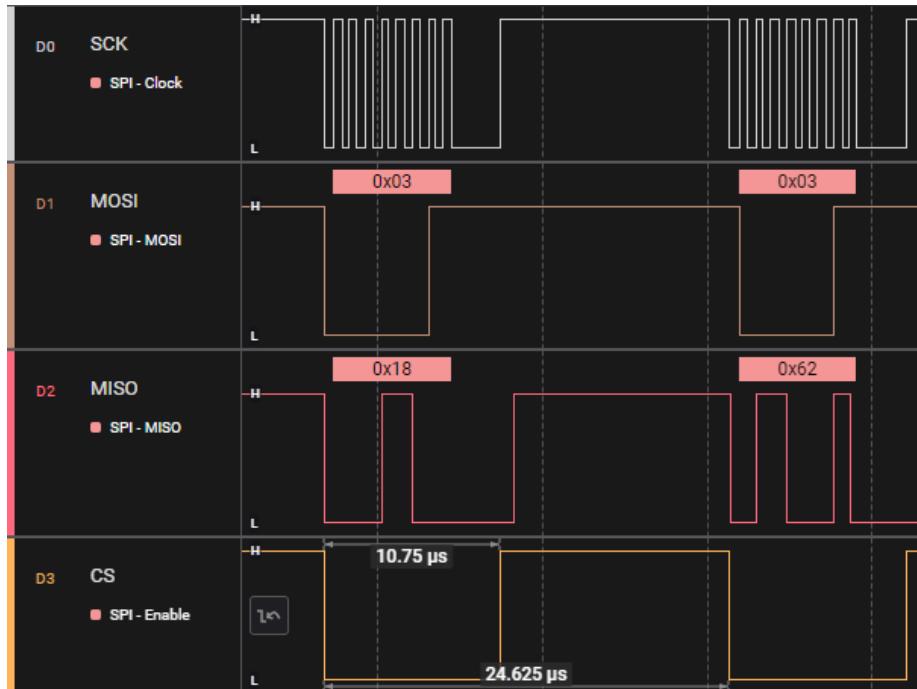
```
uint8_t send2DResult()
{
    // convert uint32_t tabs in memory
    ENDIAN_convertTab32((uint8_t *) cnt2D.counter, 12);
    ENDIAN_convertTab32((uint8_t *) cnt2D.pulseHeightSum, 12);

    // size of bytes to send
    uint16_t size = MAX_SLICE * (SLICE_SIZE * SHORT_SIZE + INT_SIZE * 2) + 2;

    // transmit size to send
    BSP_SPI_Transmit16(&hspi1, (uint8_t *) &size);
```

- On stocke ensuite la taille sur deux octets suivant les explications précédentes. On obtient une taille de 6242 octets. On peut observer que la bonne taille est envoyée sur le bus SPI :

Figure n°33 : Envoi de la taille du paquet



On observe que le short 0x1862 est envoyé sur le bus, cela correspond bien à 6242.

- Nous avons ensuite l'envoi du paquet de données qui n'est qu'une succession de fonctions SPI
- On reconvertit ensuite les tableaux de somme des hauteurs et de somme des éléments pour que le MCU soit capable de les interpréter au besoin :

```
// convert back uint32_t tabs in memory (MCU might need to interpret them)
ENDIAN_convertTab32((uint8_t*) cnt2D.counter, 12);
ENDIAN_convertTab32((uint8_t*) cnt2D.pulseHeightSum, 12);
return FREE;
```

- Enfin, on retourne FREE qui est le statut mis à jour une fois que l'entièreté des résultats ont été envoyés. Le STM retrouve ainsi un état stable et le processus de comptage 2D est terminé.

3.15 - Tests du comptage 2D

Cette partie contient les résultats obtenus pendant les tests sur table ainsi que les résultats obtenus lors des tests d'intégration qui sont les tests sur machine.

3.15.1 - Tests sur table

On possède une version de Test implémentée sur la carte connectée à notre STM. Cette version est une copie de la version en production qui a été ensuite modifiée par mon maître de stage pour pouvoir interagir avec la carte via un shell. Nous pouvons ainsi envoyer des commandes à la carte et lancer des comptages manuellement.

Il existe deux commandes :

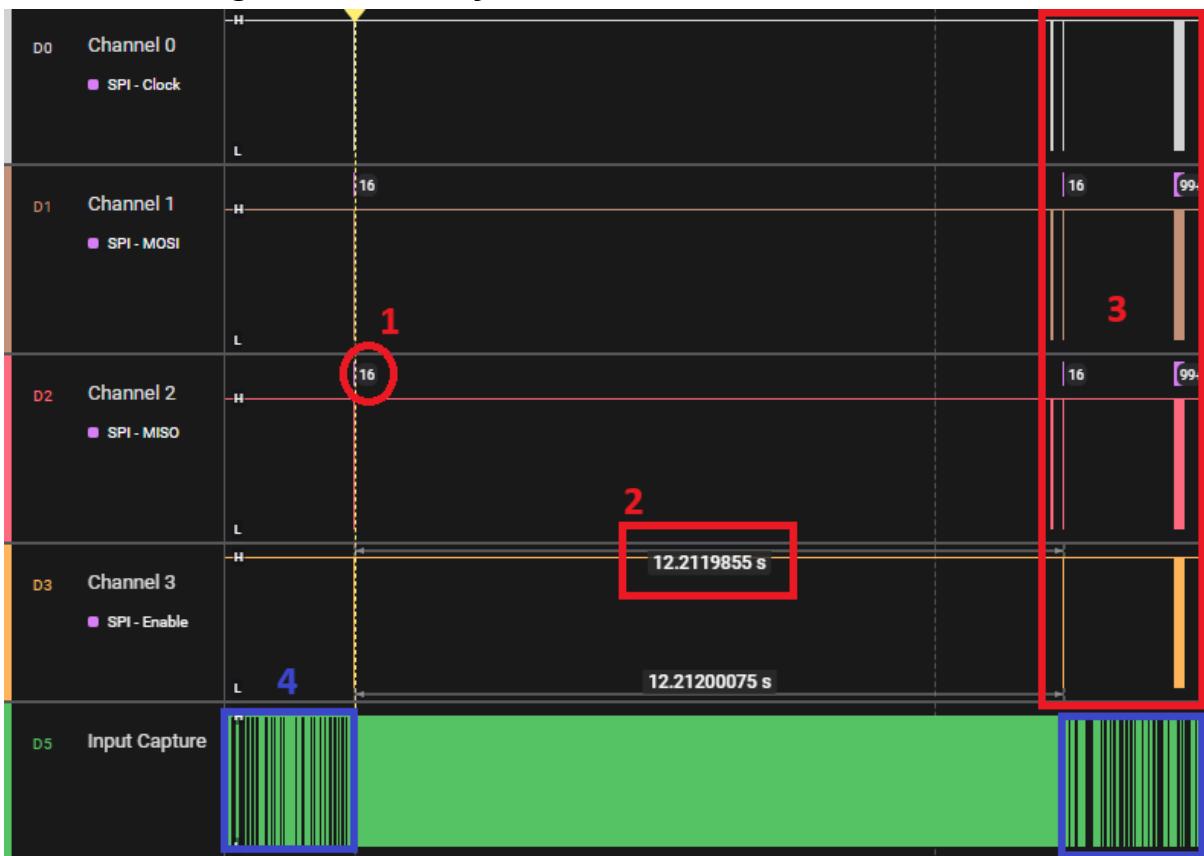
- Une pour lister les erreurs : `\\$ ler`. Cela affiche les logs dans le shell.
- Une pour lancer des actions : `send [nom_du_processus] [id commande]`
En effet, il existe trois processus qui fonctionnent ensemble dans la carte. C'était la période de la guerre des étoiles, on a donc, r2d2, cyspeo et hal.

Le processus qui interagit avec les HC11 et donc qui nous intéresse est r2d2.
La commande ressemble maintenant à : `\\$ send r2d2 [id_commande]`. Il nous manque encore l'identifiant de commande.

L'id de commande nous permet de définir l'action à lancer pour notre STM. Une liste non exhaustive des commandes (celles qui nous intéressent) se présente comme suit :

id commande	Explications
1	Lance un comptage 2D sur 12 tranches avec un seul numéro de comptage. Chaque tranche a une durée de 1 seconde.
13	Lance un comptage 2D sur 1 tranche de 1 seconde.
23	Commande READ_STATUS
33	Demande de résultat pour l'esclave Plaquettes (PLT)

Les commandes utilisées durant les tests en majeure partie ont été les commandes une et trente-trois.

Figure n°34 : Analyse de la commande 1 suivie de 33

On observe donc :

- (1) Une commande sur seize octets est envoyée, cette commande est la commande de comptage 2D sur 12 tranches.
- (2) Une durée d'environ douze secondes entre l'envoi de la commande et la lecture du nouveau statut par le maître.
- (3) Une fois la commande terminée, le maître lit le statut. On envoie ensuite la commande 33 qui demande les résultats. Cette vue est très dézoomée, on ne peut donc pas bien voir le contenu du paquet, mais on le voit bien apparaître à l'écran.
- (4) Des résidus apparaissent sur input capture. Ces résidus ne sont pas interprétés par l'esclave du temps que la commande n'est pas en marche. On peut noter qu'une fois la commande lancée, le CPLD laisse passer la totalité des impulsions, le signal devient alors plus dense.

Le protocole est opérationnel, on peut passer aux tests sur machine.

3.15.2 - Environnement de test sur machine

Les tests sur machine sont faits dans un autre environnement de travail que les tests sur table. Ces derniers étaient réalisés dans l'open space avec un environnement de bureau. Les tests machines sont quant à eux réalisés dans un autre bâtiment avec un environnement de laboratoire.

Figure n°35 : Environnement de laboratoire



Cela inclut le port de la blouse, des gants ainsi que des lunettes de protection (cf [annexe](#)). On manipule du sang pour réaliser ces tests. Dans un premier temps on utilise du sang de contrôle. Une fois que les résultats sont concluants, on récupère du sang frais qui est fourni par les laboratoires d'analyse de sang pour effectuer nos tests.

Il existe tout un protocole de transfert du sang pour éviter les AES (Accident d'Exposition au Sang) au sein de l'entreprise. Le sang est transporté dans une mallette fermée et ne peut être sortie que dans un environnement de laboratoire.

Figure n°36 : Mallette de transport du sang

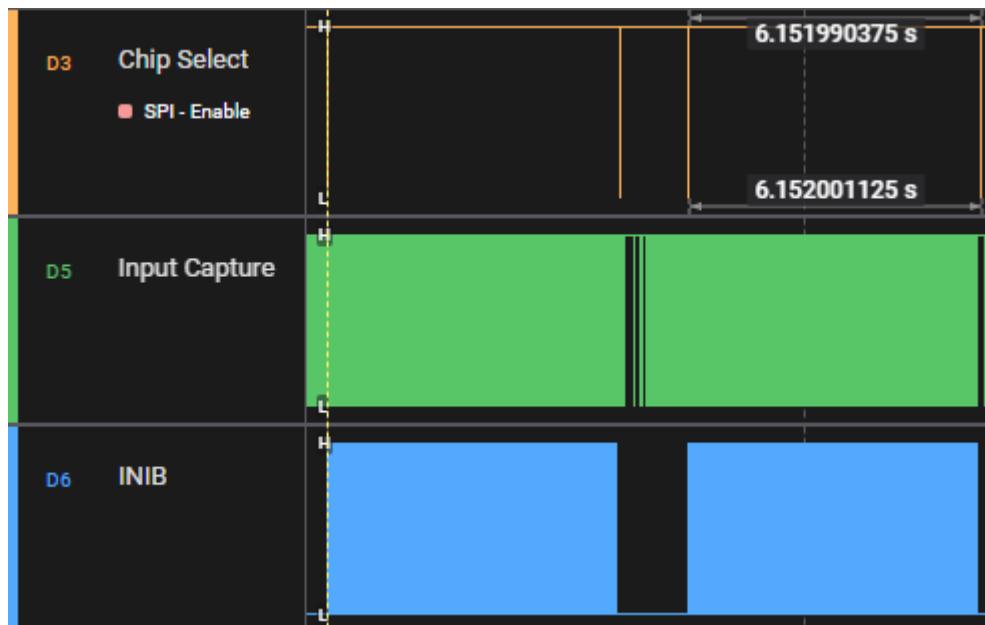


3.15.3 - Tests sur machine

Avant d'opérer les tests sur machine il faut tout d'abord mettre en place la carte de test avec la solution de remplacement. Des réglages sont à faire sur la carte à l'aide de potentiomètres. On relève alors les valeurs de la carte présente sur la machine que l'on reporte ensuite sur la nôtre une fois montée.

Chronologiquement, les tests sur adjust 2D ont été réalisés en premier. Adjust 2D est une fonctionnalité de réglage de la carte, il faut donc la faire fonctionner avant de lancer un comptage 2D avec du sang. Une fois l'environnement de test prêt, on connecte l'analyseur logique et on passe les tubes de sang dans la machine.

Figure n°37 : Analyse du comptage 2D sur machine



On observe ici des comptages d'une durée de six secondes. Les douze tranches sont effectuées en deux temps selon un numéro de comptage. Grâce aux impulsions émises sur INIB, on visualise bien la séparation entre le premier et le second comptage.

En superposant le Chip Select, on observe le protocole qui se répète deux fois : Envoi de la commande de comptage -> Le comptage opère.. -> Lecture du statut.

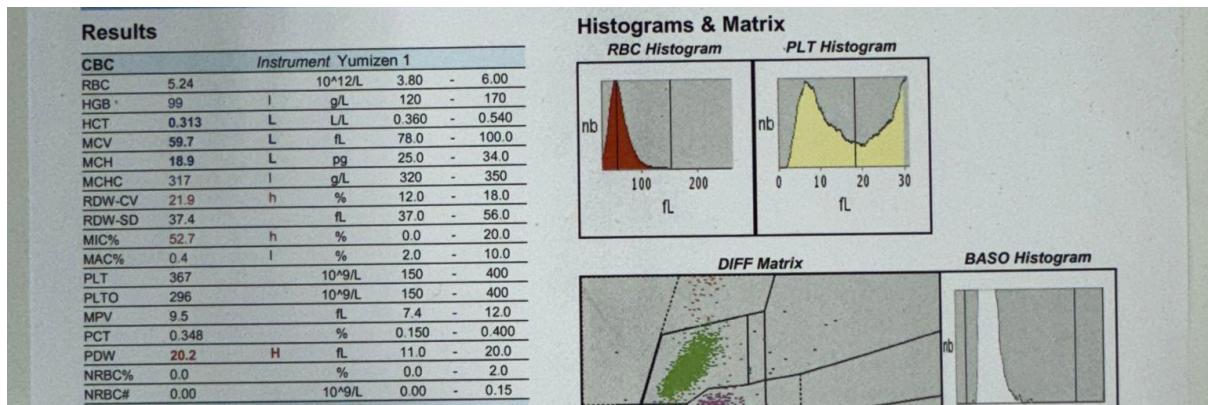
Figure n° 38: Versions de test

COUNT2D_V1	07/03/2025 16:48
COUNT2D_V2	10/03/2025 14:21
COUNT2D_VF	13/03/2025 10:18

3.15.4 - Résultats sur machine

Pour tester le bon fonctionnement de notre solution de remplacement, on teste les résultats obtenus en les comparants à des résultats de machines fonctionnelles. On récupère alors les fiches patient en même temps que les tubes de sang que l'on analyse (cf [annexe](#)). Les résultats apparaissent sous forme de courbes :

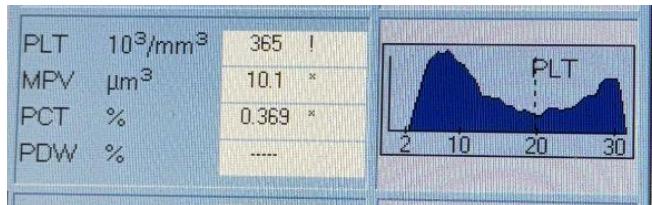
Figure n°39 : Résultats d'un Yumizen



On s'intéresse à 2 valeurs ici : PLT qui est le nombre de Plaquettes ainsi que MPV qui est le Volume Moyen Plaquettaire. La courbe quant à elle est représentée en jaune, c'est l'histogramme PLT.

Le Pentra 80 fournit lui aussi de nombreux résultats sur son interface graphique (cf [annexe](#)).

Figure n°40 : Résultats du Pentra 80 avec la solution de remplacement



On observe que la courbe ressemble à celle du Yumizen. Le nombre de Plaquettes est cohérent (Yumizen = 367 | Pentra 80 = 365). Le volume plaquettaire moyen est lui aussi cohérent à 0,6 prêt (Yumizen = 9,5 | Pentra 80 = 10,1).

À noter que la répétabilité d'une mesure se fait sur plusieurs passages dans un laps de temps le plus court possible. Les mesures sont sensibles à la température par exemple, les résultats sont donc très satisfaisants.

D'autres résultats sont disponibles en [annexe](#).

3.16 - ADJUST 2D

Étant donné que le processus de comptage 2D a été documenté en détail, des raccourcis seront pris quant à l'explication du fonctionnement du mode ADJUST.

3.16.1 - Fonctionnement ADJUST 2D

La fonctionnalité ADJUST 2D est une fonctionnalité destinée aux techniciens amenés à régler la machine.

On ne passe pas du sang lors de ce comptage mais un liquide contenant des billes de latex représentant les cellules. Un liquide est disponible pour chaque voie (les cellules ne faisant pas toutes la même taille). On utilise ici le liquide pour les PLT.

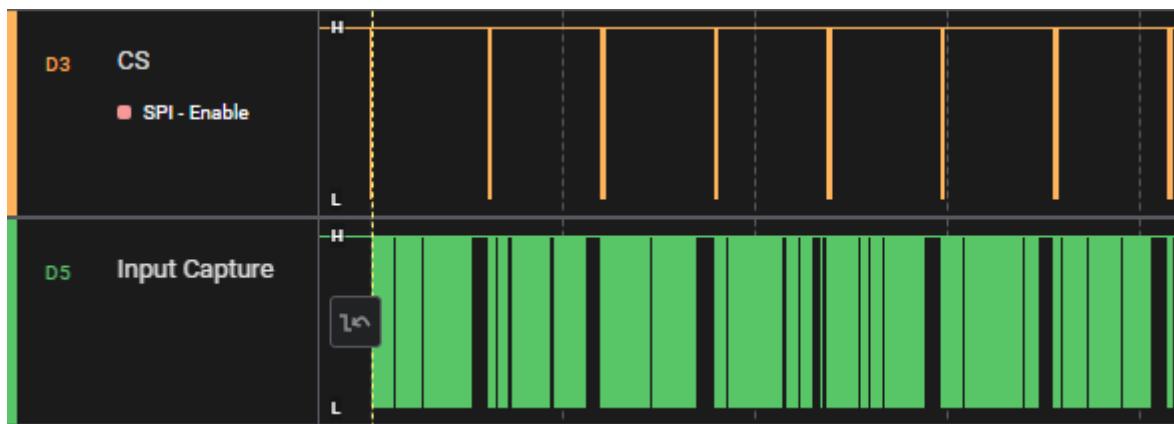
L'avantage d'utiliser des billes de latex est qu'elles ont toutes la même circonférence. On vient donc régler un gain via un potentiomètre pendant la mesure, ce qui nous permet d'avoir une hauteur d'impulsion de référence et ainsi calibrer l'appareil.

Concernant le protocole, le maître lance un comptage 2D sur 30 tranches de 0.5 secondes chacune :

- Chaque fois qu'une tranche est prête, le STM lève l'interruption hardware
- Le maître récupère alors les résultats disponibles
- La mémoire est effacée côté STM
- Le STM continue ce protocole jusqu'à ce que les tranches soient terminées.

En effet, pour avoir le gain en temps réel affiché à l'écran, le résultat des tranches doit être récupéré périodiquement à intervalle réduit (toutes les 0.5 secondes).

On peut observer l'interruption hardware levée à chaque fin de tranche suivie du bloc d'octets transférés:

Figure n°41 : Interruption hardware levée sous ADJUST 2D**Figure n°42 : Visualisation de plusieurs tranches de comptage**

En superposant le *Chip Select*, on observe que plusieurs tranches sont réalisées consécutivement et qu'à chaque tranche, un paquet est envoyé sur le bus SPI.

3.16.2 - Code ADJUST 2D

Ce qui diffère du comptage 2D basique est dans la gestion du statut d'acquisition et l'envoi de données. ADJUST 2D se contente de ne récupérer que la somme des éléments en fonction de leur hauteur (tableau de 12 par 256 shorts). Les autres sommes ne sont pas réalisées.

L'envoi des données est donc moins conséquent, une seule tranche est envoyée et non 12. Chaque paquet fait donc 512 octets.

On peut observer une gestion de statut un peu particulier en sortie de fonction :

```
// if not finished -> we send him back in BUSY
    if (cnt2D.slicesRemaining > 0)
        return BUSY;
    return FREE;
```

En effet, comme l'envoi des résultats ne se fait pas à la fin du comptage, s'il reste des tranches à compter, le STM ne peut pas se permettre d'être libre et donc prêt à exécuter de nouvelles commandes. On le renvoie donc en BUSY pour bloquer ce comportement.

Le maître a cependant besoin de lire les résultats pendant le comptage 2D, le STM ne peut donc pas rester en permanence avec un statut BUSY sinon, lorsque le maître demandera les résultats, le STM refusera la demande.

Cette particularité est gérée dans la fonction de callback associée au Timer de comptage lorsqu'une tranche se termine :

```
static void endOfSlice_ADJUST2D()
{
    // Last slice's Status + Interrupt will be set by the count2D function exiting
    if (cnt2D.slicesRemaining > 0) {
        // change status to allow communicating commands to perform
        ACQU_setStatus(
            ACKNOWLEDGE | cnt2D.channel << 2 | cnt2D.countNumber
        );
        // notice Master that the slice is ready
        BSP_setHardwareIT();
    }
}
```

Le premier if a été ajouté suite à un comportement survenu lors de la phase de test. Étant donné que le comptage 2D et ADJUST 2D partagent la même fonction, l'interruption hardware ainsi que le statut étaient gérés deux fois sur la dernière tranche. Cela cause un déphasage sur le bus SPI faisant reboot la carte mère (envoi de RESET). Grâce à ce 'if', sur la dernière tranche seulement, l'interruption hardware et le statut sont mis à jour en sortie de fonction principale du comptage 2D.

Revenons-en à la gestion du statut en fin de tranche sur le mode ADJUST. En mettant le statut à jour, cela permet de ne plus être BUSY, ainsi, lorsque le maître demande les résultats, le STM les lui envoie puis retourne en BUSY.

À noter que la gestion du statut ici est de l'ordre de la gestion des erreurs. En effet, étant donné que le maître a lancé un mode ADJUST, il attend de son côté la récupération des 30 tranches de comptage et n'est pas supposé envoyer une autre commande avant la fin du comptage.

4 - Apprentissages critiques mobilisés

Grâce à ce Stage, de nombreuses compétences ont été renforcées d'un point de vue de l'IUT :

- En termes de conception, test et intégration d'une solution, ce stage regroupe tous ces aspects. En effet, réaliser la portabilité d'un code existant vers une nouvelle technologie nécessite de concevoir une application avec des contraintes logicielles et fonctionnelles.

Dans le cadre de faire évoluer une application, j'ai pu renforcer mes compétences en adoptant de bonnes pratiques de conception / programmation, ainsi que par la validation et l'intégration de la solution développée durant mon stage.

- En termes d'optimisation d'application, dans la mesure où l'application est développée sur un microcontrôleur avec la contrainte de devoir coller à un existant en plus de contraintes matérielles, l'optimisation est partout.

Que ce soit dans les pratiques de développement avec une allocation mémoire consciente ou bien dans les tests de performance réalisés pour optimiser les algorithmes d'inversion d'octets, l'amélioration des performances d'un programme dans un contexte contraint est plus que vérifiée !

- Pour ce qui est d'une application communicante, ce n'est certes pas une application concernant le réseau. En revanche, on y développe bien une communication inter-microcontrôleur avec un protocole de communication selon des normes bien définies.

- En ce qui concerne la conduite de projet, j'ai non seulement identifié les critères de faisabilité, mais j'ai aussi pu planifier mes activités pour suivre une démarche bien définie.

- Enfin, en termes de collaboration au sein d'une équipe informatique, j'ai travaillé de pair avec mon maître de stage, mais j'ai aussi travaillé main dans la main avec les électroniciens qui m'ont aidé sur des thématiques concernant leur métier.

J'ai pu grâce à ce stage, travailler en équipe certes, mais surtout, travailler avec plusieurs corps de métiers dans le cadre de la réalisation d'un projet.

5 - Méthodologie et organisation du projet

L'entreprise fonctionne sur une base de ressources attribuées par métiers. Pour la planification des tâches plusieurs outils sont utilisés, des diagrammes de Gantt, la suite Microsoft pour de nombreux usages comme la planification des réunions à court terme, les prises de notes collectives lors de réunions..

J'ai pu réaliser ma propre planification avec l'outil de mon choix approuvé par mon maître de stage, excalidraw. Cet exercice fût compliqué dans la mesure où je ne suis pas habitué à développer de l'embarqué. Je n'avais qu'une vague idée du temps que pouvaient me prendre les tâches à effectuer. ([cf Planification-Stage](#))

Sofiane Megdoud, responsable métier, a ensuite pris le temps avec moi de planifier mes activités avec son expertise sur le métier. ([cf Planification_gantt](#)).

Dans l'ensemble, la méthodologie que j'ai suivie se résume en sept étapes :

❖ Conception

Dans un premier temps, je décortique le problème et réfléchi à une manière efficace d'y remédier en analysant l'existant, dessinant des schéma au brouillon. J'essaye d'articuler mes idées jusqu'à ce qu'elles deviennent réalisables.

❖ Fonctions utilitaires

Dans un second temps, je développe des fonctions que j'appelle les 'utilitaires' qui vont me permettre de développer par la suite mes fonctionnalités. Elles sont la plupart du temps génériques et opèrent des actions essentielles dont je pourrai avoir besoin.

❖ Test des utilitaires

Je procède ensuite aux tests de ces fonctions. Dans le monde de l'embarqué nous travaillons avec des machines et des cartes. Nous possédons de multiples outils pour effectuer ces tests:

- Le débogueur pas à pas pour tester notre application, notamment les valeurs des variables, le bon comportement des interruptions..
- Des outils pour tester le hardware comme un oscilloscope, un GBF (Générateur d'impulsion), une sonde salae (analyseur logique).

Ces tests me permettent d'être sûr que les fonctions développées ainsi

que les outils utilisés sont opérationnels et que le développement du protocole ne rencontrera pas de problèmes à ce niveau là.

❖ Développement du protocole

Une fois la conception et les tests d'utilitaires faits, il ne me reste plus qu'à développer ce qui a été décidé précédemment. J'en profite pour documenter mon code, cela me permet de me l'approprier. Je me rends parfois compte de problèmes de conceptions lors de cette étape. Elle me sert d'étape de validation pour la suite.

❖ Tests d'intégrations

Une fois le protocole développé, j'effectue ensuite les tests d'intégration. Ce sont des tests sur table et donc j'utilise les mêmes outils que pour les tests d'utilitaires. A la différence des premiers tests, ils vérifient que le microcontrôleur répond au comportement attendu.

❖ Peaufinage de la fonctionnalité

Enfin, je repasse le code en revue pour le nettoyer et finir de documenter s'il y a des précisions à apporter ou des idées futures à noter pour les prochaines fonctionnalités.

❖ Commit

Après chaque fonctionnalité, je commit le code et sa documentation associée sur les dépôts pour ne pas perdre trace de ce que je viens de produire. J'essaye la plupart du temps de commit des versions stables pour ne pas versionner du code qui ne fonctionne pas.

6 - Conclusion

Le stage au sein de HORIBA Médical a été une expérience formatrice pour moi. Elle m'a permis d'apprendre de nouvelles technologies sous un aspect différent tout en approfondissant les connaissances acquises au sein de l'IUT.

6.1 - Bilan du travail réalisé

Le projet de faire fonctionner la voie plaquette a été un succès. J'ai pu sortir une version stable, documentée qui a donné des résultats cohérents. J'ai réussi à produire du code propre, optimisé et simple à reprendre pour la suite.

Les deux principaux objectifs de faire fonctionner la voie PLT ainsi que de proposer un code documenté ont été atteints.

Il reste cependant des améliorations, comme toujours. En rédigeant ce rapport, je me suis rendu compte de deux ou trois incohérences dans le code, notamment sur le traitement des paramètres reçus sur le bus SPI. J'ai pris des raccourcis qui vont très certainement bloquer le fonctionnement sur d'autres machines. J'ai déjà réfléchi à une solution de mon côté.

Le fait de tout reprendre avec énormément de recul m'a permis d'avoir une approche différente du sujet. Avec le temps, depuis que la version stable est sortie, de nouvelles idées me sont venues concernant la conception du code.

6.2 - Perspectives

Le développement de la solution de remplacement est loin d'être terminé. Comme précisé précédemment, quatre voies sont présentes et nous n'avons pu en développer qu'une seule. Il reste donc encore trois voies :

- GB pour les globules rouges, cette voie est partiellement développée dans la voie plaquette étant donné qu'elles utilisent le même canal d'acquisition.
- BASO et LMNE pour les globules blancs. Ces deux voies nécessitent d'être entièrement développées, d'autant plus que la voie LMNE comporte un autre type de comptage par voie optique qui est plus complexe à mettre en place.

Aujourd'hui, nous avons testé la solution de remplacement du HC11 sur le Pentra 80. Le HC11 est aussi présent sur d'autres machines au sein d'HORIBA dont une qui comporte une cinquième voie. Le remplacement du HC11 n'est donc clairement pas terminé mais ce projet est sur la bonne voie !

6.3 - Bilan des acquis techniques

Mettre un pied dans l'embarqué m'a permis de découvrir énormément de technologies. Cependant, j'ai certes découvert de nombreuses choses au cours de mon stage mais j'ai surtout pu mettre en pratique les compétences acquises durant mon cursus. Que ce soit en termes de développement C, de conception logicielle ou via l'utilisation du gestionnaire de version Git.

Le fait de développer sous interruptions entraîne un challenge que je n'avais encore jamais rencontré au sein de l'IUT. Cela mène à des conceptions différentes mais tout autant complexes.

Travailler sur des microcontrôleurs force l'optimisation. En effet, le fait d'être restreint par le matériel m'a permis de développer une approche consciente quant à l'allocation mémoire de mon code et l'optimisation de mes fonctions.

Les principaux acquis techniques du stages sont les suivants :

- Développer du bas niveau, savoir s'interfacer avec une machine en C.
- Optimiser la gestion de la mémoire ainsi que le fonctionnement du code.
- Concevoir en prenant en compte les contraintes logicielles mais aussi les contraintes matérielles.
- Développer sur STM32.
- Maîtriser le développement avec registres.
- Appréhender un environnement électronique ainsi que souder des composants sur une carte.

Tous ces acquis s'articulent autour du bas niveau. À partir du moment où l'on développe à même la machine, une maîtrise totale des instructions est nécessaire.

6.4 - Bilan de l'expérience en entreprise

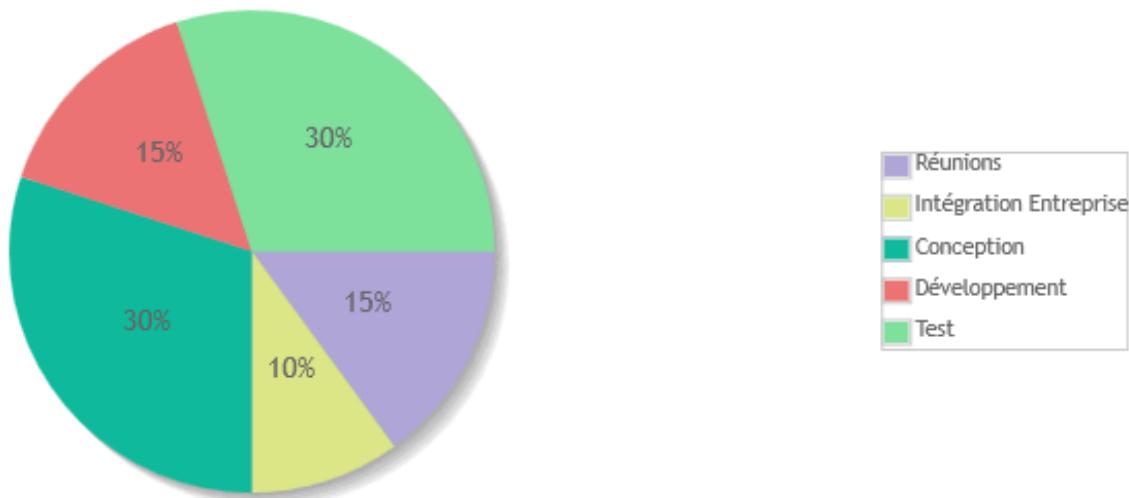
Cette immersion dans l'entreprise représente pour moi le premier pas idéal avant l'alternance qui est demandée en troisième année, que j'aurai probablement la chance d'effectuer dans cette même entreprise afin de me permettre d'approfondir ce que j'ai pu commencer.

J'ai pu aborder plusieurs concepts caractéristiques de l'entreprise :

- Appréhender un projet possédant un existant riche pour ensuite travailler dessus et pouvoir l'améliorer
- Intégrer les conseils de mon maître de stage qui a plus de 30 ans d'expérience dans le domaine et qui est littéralement une encyclopédie en ce qui concerne les technologies anciennes.
- Travailler dans une équipe de développeurs ainsi qu'avec un autre corps de métier : les électroniciens, pour mener à bien le projet. L'entraide fût salvatrice par moment.

A noter que le travail en entreprise ne se limite pas à développer du code :

Figure n°43 : Activités en entreprise



Enfin, j'ai pu goûter au rythme et aux exigences horaires du monde du travail, ainsi qu'à la flexibilité et à la confiance (plages horaires d'arrivée et de sortie) offerte par une entreprise comme HORIBA.

6.5 - Bilan de l'expérience personnelle

En termes d'expérience, je ne pouvais pas espérer mieux qu'un stage dans le développement embarqué. Je m'étais déjà pris d'intérêt pour ce domaine avant d'obtenir ce stage, je suis maintenant convaincu de vouloir continuer sur cette lancée. Le rapport à la machine est bien trop captivant pour que je n'en fasse pas l'objet de ma carrière en tant que développeur.

J'ai découvert le monde du travail depuis cinq ans maintenant, mais seulement pour assurer ma subsistance si je puis dire. Aujourd'hui, grâce à ce stage, j'ai pu découvrir ce que c'était que de travailler par passion, ne pas compter les heures, se lever le matin et avoir la certitude que l'on va apprendre de nouvelles choses.

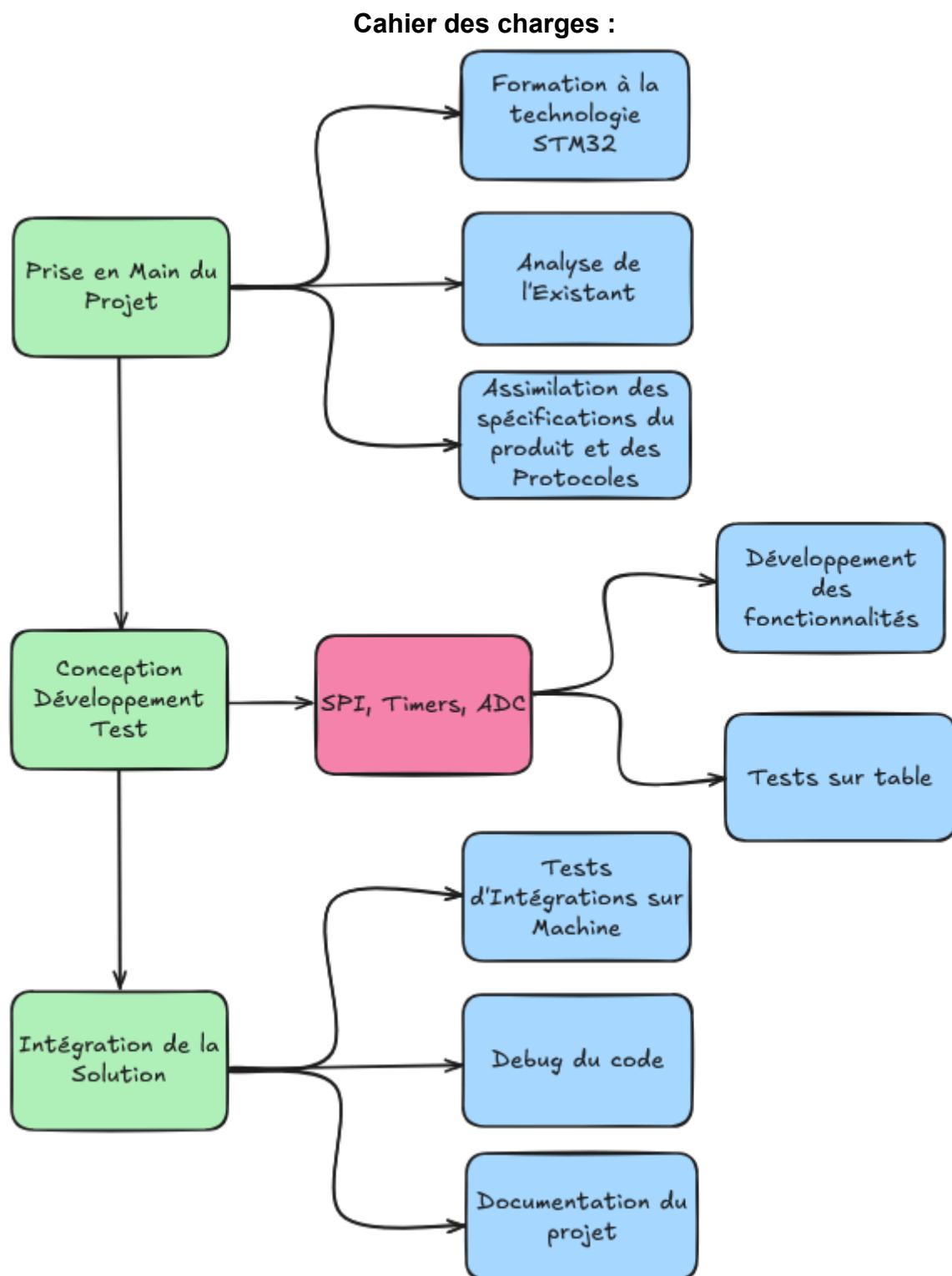
Alors, la section des remerciements est déjà passée mais je tiens sincèrement à remercier une seconde fois l'équipe de développeurs embarqués comme l'équipe d'électroniciens qui ont su m'accueillir chaleureusement au sein de l'entreprise.
Merci.

Vu par Christophe DOMERGUE le [26 mars 2025]

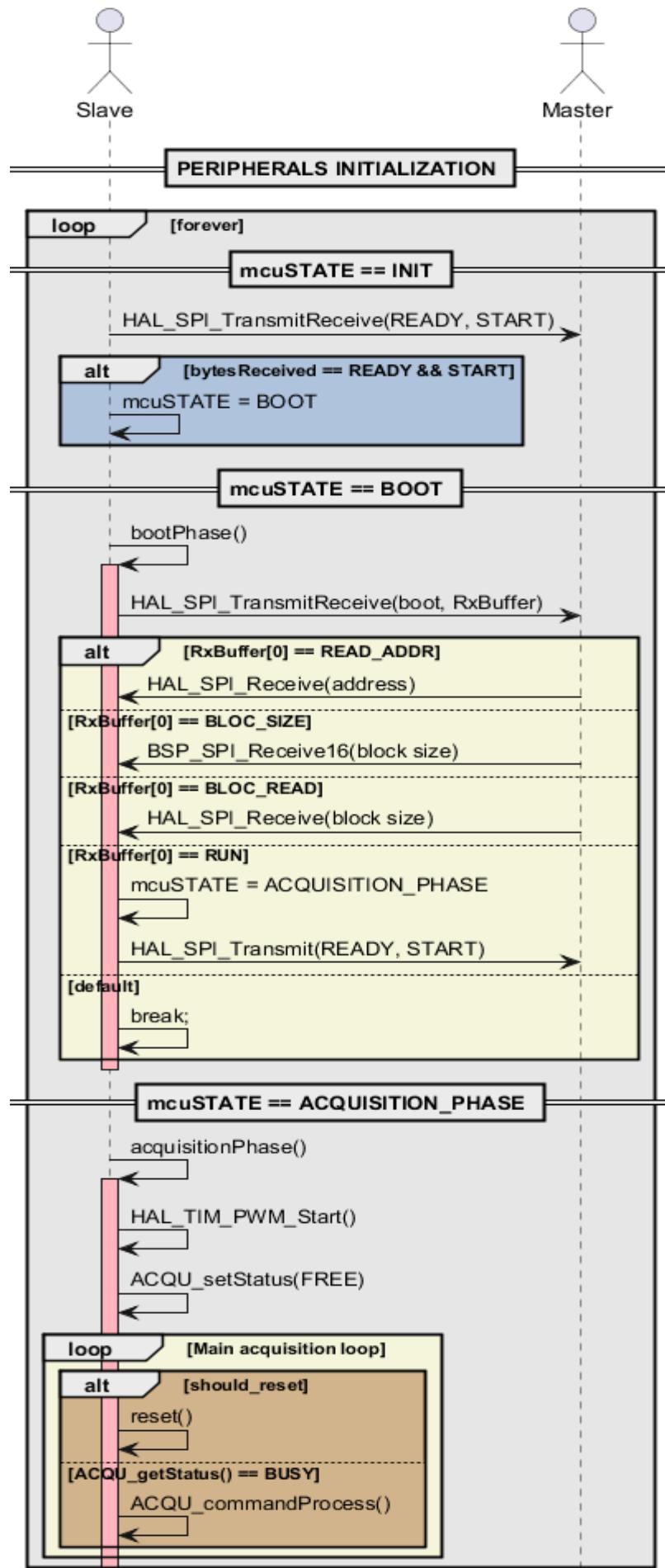
7 - Bibliographie

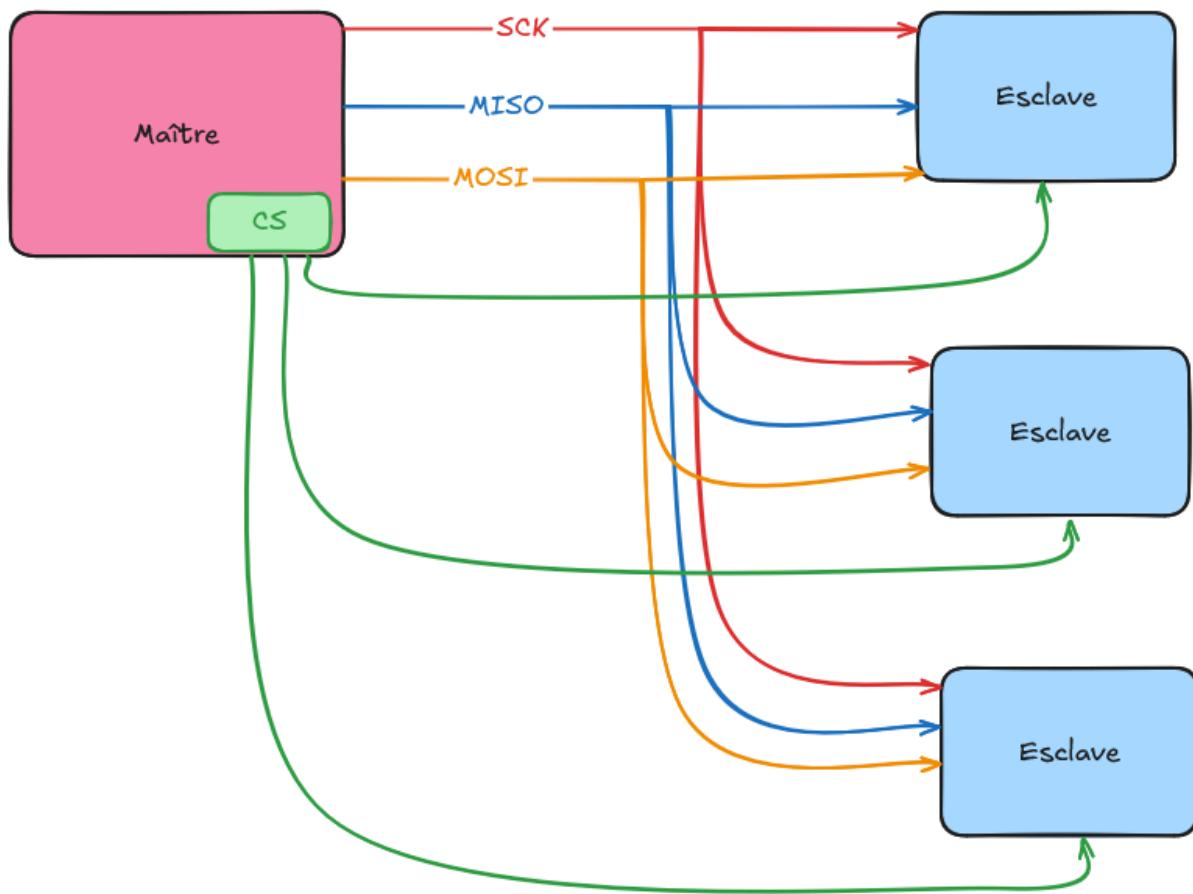
- [1] STMicroelectronics, "ST Community," [En ligne]. Disponible: <https://community.st.com/>. [Consulté le: 20 mars 2025].
- [2] Stack Overflow, "Questions," [En ligne]. Disponible: <https://stackoverflow.com/questions>. [Consulté le: 20 mars 2025].
- [3] Pomad, "Tutorials STM32," [En ligne]. Disponible: <https://pomad.fr/tutorials/stm32>. [Consulté le: 20 mars 2025].
- [4] DeepBlue Embedded, "DeepBlue Embedded," [En ligne]. Disponible: <https://deepbluembedded.com/>. [Consulté le: 20 mars 2025].
- [5] Gladir, "ARM Reference," [En ligne]. Disponible: <https://www.gadir.com/CODER/ARM/reference.htm>. [Consulté le: 20 mars 2025].
- [6] Nordic Semiconductor, "Nucleo G0B1RE Documentation," [En ligne]. Disponible: https://docs.nordicsemi.com/bundle/ncs-2.5.2/page/zephyr/boards/arm/nucleo_g0b1re/doc/index.html. [Consulté le: 20 mars 2025].
- [7] STMicroelectronics, "STM32Cube HAL and low-layer drivers" STMicroelectronics, 2024. [En ligne]. Disponible: <https://www.st.com/en/embedded-software/stm32cube-embedded-software.html>. [Consulté le: 20 mars 2025].

8 - Annexes



Architecture logicielle : Machine à état



Agencement du bus SPI :

Contenu de la librairie endian :

```

1④ /*
2 * endian.h
3 *
4 * Created on: Feb 4, 2025
5 * Author: Kelian.Michiel
6 */
7
8 #ifndef BSP_INC_ENDIAN_H_
9 #define BSP_INC_ENDIAN_H_
10 #include "stm32g0xx.h"
11 #include "main.h"
12
13④ /*
14 * Conversion Functions
15 */
16 void ENDIAN_convert16          (uint8_t *buffer);
17 void ENDIAN_convert32          (uint8_t *buffer);
18
19 void ENDIAN_convertMultiple16  (uint8_t *buffer, uint8_t *idx, uint8_t nbIndex);
20 void ENDIAN_convertMultiple32  (uint8_t *buffer, uint8_t *idx, uint8_t nbIndex);
21
22 void ENDIAN_convertTab16       (uint8_t *buffer, uint8_t size);
23 void ENDIAN_convertTab32       (uint8_t *buffer, uint8_t size);
24
25#endif /* BSP_INC_ENDIAN_H_ */
26

```

Contenu de la librairie bsp pour le SPI :

```

14④ /*
15 * SPI Functions with Endianness concerns. (@endian.c)
16 */
17 void BSP_SPI_Transmit16        (SPI_HandleTypeDef *hspi, uint8_t *pShort);
18 void BSP_SPI_Transmit32        (SPI_HandleTypeDef *hspi, uint8_t *pInt);
19
20 void BSP_SPI_Receive16         (SPI_HandleTypeDef *hspi, uint8_t *pShort);
21 void BSP_SPI_Receive32         (SPI_HandleTypeDef *hspi, uint8_t *pInt);
22
23 void BSP_SPI_TransmitReceive16 (SPI_HandleTypeDef *hspi, uint8_t *pTxShort, uint8_t *pRxShort);
24 void BSP_SPI_TransmitReceive32 (SPI_HandleTypeDef *hspi, uint8_t *pTxInt, uint8_t *pRxInt);
25
26 void BSP_SPI_TxFlush          (SPI_HandleTypeDef *hspi);

```

Environnement de laboratoire



Fiche patient :

PATIENT REPORT **HORIBA** Medical **HELO**
HORIBA End-to-end Laboratory Organisation

Informations

Last name	PATIENT	Sample ID	03130903460411803
First name	NAME 03130903460411803	Patient ID	03130903460411803
Birthdate		Collection Date	13/03/2025 09:04
Sex		Validation Date	13/03/2025 09:05
Ward	No ward	Validation User	ruleResult
Prescriber	No prescriber	Rack - Pos	041180 - 3
		Request ID	03130903460411803

Results

CBC	Instrument	Yumizen 1		
RBC	4.89	10 ¹² /L	3.80	- 6.00
HGB	157	g/L	120	- 170
HCT	0.458	L/L	0.360	- 0.540
MCV	93.8	fL	78.0	- 100.0
MCH	32.1	pg	25.0	- 34.0
MCHC	343	g/L	320	- 350
RDW-CV	14.0	%	12.0	- 18.0
RDW-SD	43.2	fL	37.0	- 56.0
MIC%	0.9	%	0.0	- 20.0
MAC%	3.3	%	2.0	- 10.0
PLT	308	10 ⁹ /L	150	- 400
MPV	7.9	fL	7.4	- 12.0
PCT	0.243	%	0.150	- 0.400
PDW	12.4	fL	11.0	- 20.0
NRBC%	0.0	%	0.0	- 2.0
NRBC#	0.00	10 ⁹ /L	0.00	- 0.15
DIFF	Instrument	Yumizen 1		
WBC	6.30	10 ⁹ /L	3.50	- 10.00
TNC	6.30	10 ⁹ /L	3.50	- 10.00
LYM%	40.2	%	15.0	- 45.0
LYM#	2.53	10 ⁹ /L	1.00	- 4.00
MON%	14.1	h	4.0	- 13.0
MON#	0.88	h	0.20	- 0.80
NEU%	40.3	%	40.0	- 75.0
NEU#	2.55	10 ⁹ /L	1.50	- 7.00
EOS%	4.6	%	0.5	- 7.0
EOS#	0.29	10 ⁹ /L	0.00	- 0.50
BAS%	0.5	%	0.0	- 2.0
BAS#	0.03	10 ⁹ /L	0.00	- 0.20
IMG%	0.2	%	0.0	- 2.0
IMG#	0.01	10 ⁹ /L	0.00	- 999.90
IMM%	0.1	%	0.0	- 0.5
IMM#	0.01	10 ⁹ /L	0.00	- 0.10
IML%	0.0	%	0.0	- 0.2
IML#	0.00	10 ⁹ /L	0.00	- 0.05
LIC%	0.3	%	0.0	- 3.0
LIC#	0.02	10 ⁹ /L	0.00	- 0.20
ALY%	0.9	%	0.0	- 2.5
ALY#	0.06	10 ⁹ /L	0.00	- 0.25

Histograms & Matrix

RBC Histogram

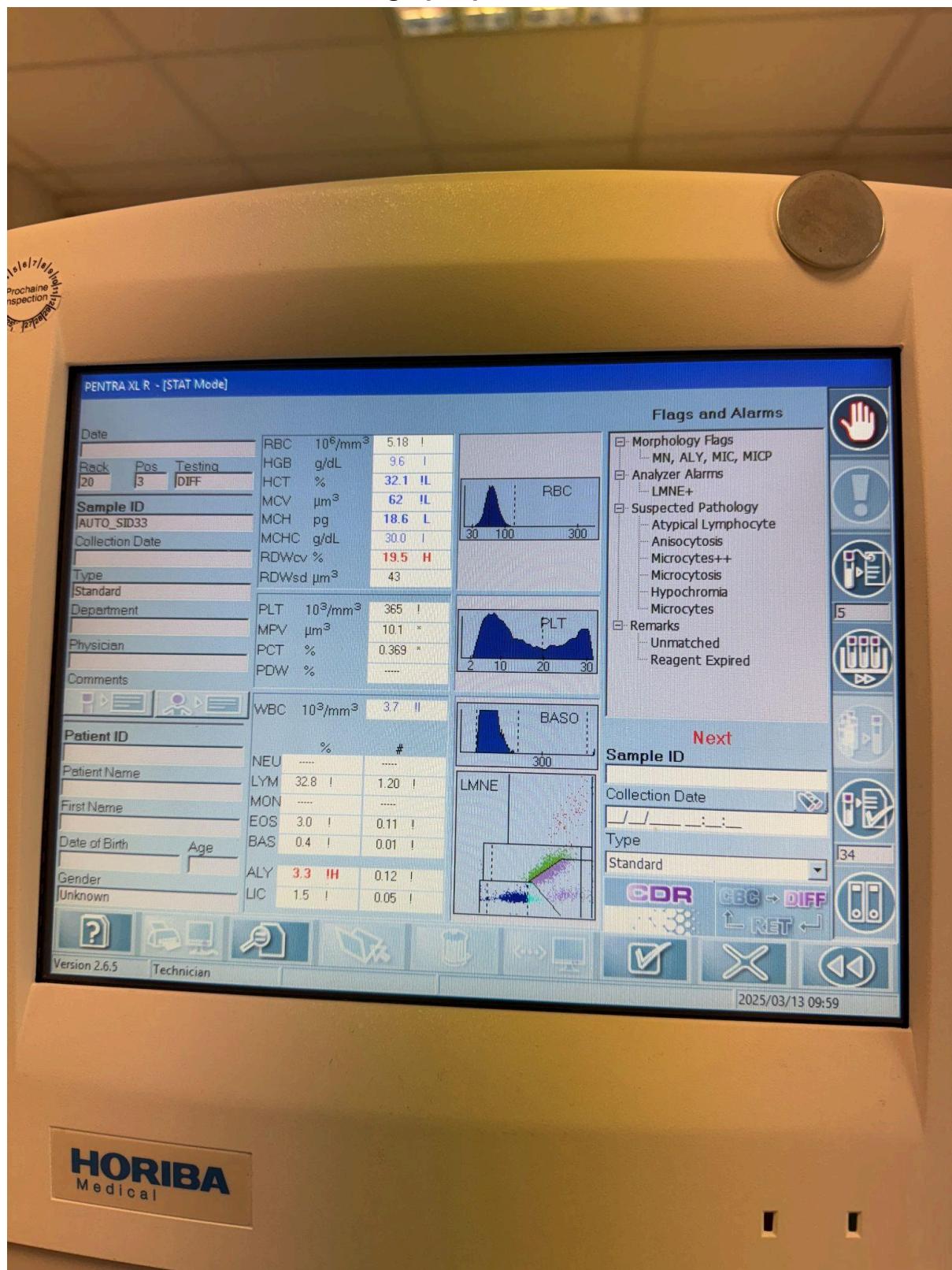
PLT Histogram

DIFF Matrix

BASO Histogram

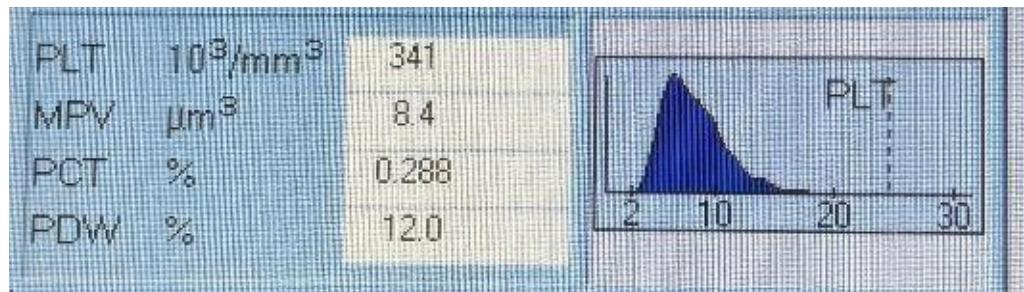
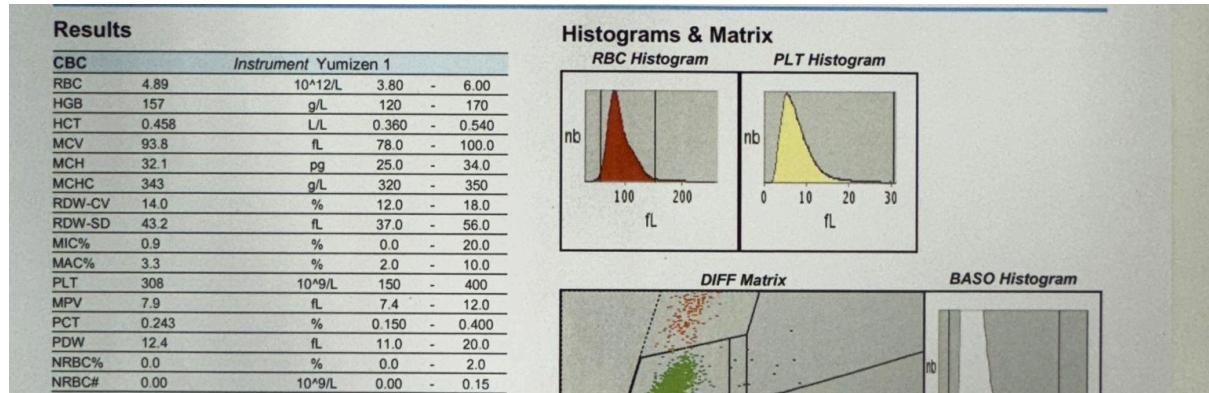
Comments:(1)

Interface graphique du Pentra 80 :



Comparaison de résultats Yumizen/Pentra 80 sous solution de remplacement :

Résultat n°1 : niveau de plaquettes HAUT



Résultat n°2 : niveau de plaquettes BAS

