Création d'un outil d'intégration continue

Rapport d'élève ingénieur
Stage de 2^{me} année
Filière F2 : Génie Logiciel et Systèmes Informatiques

Présenté par : **Rémi CHASSAGNOL**

Responsable ISIMA : Loïc YON
Responsable entreprise : Ludovic DESCOUT

Soutenance : 30/08/2023
Durée du stage : 5 mois

Campus des Cézeaux. 1 rue de la Chébarde. TSA 60125. 63178 Aubière CEDEX

Table des matières

Remerciements

Les remerciements !!

Table des figures

Résumé

Le super résumer !

Mots-clés : **C/C++**, **intégration continue**, **tests**, **systèmes embarqués**, **émulation**

Abstract

the amazing abstract

Keywords : **C/C++**, **continuous integration**, **testing**, **embeded system**, **emulation**

Introduction

Introduction

TODO : la partie sur la branche C principale très détailler puis on parlera de ce qui a été fait coté évent et Qt (qui suit globalement le même principe).

plan !

Première partie

Contexte du projet

1 CKsquare

CKsquare est une entreprise d'ingénierie, d'étude et de conseil spécialisée dans la conception de systèmes de paiement automatisés. La société, au départ nommée cbsquare, a été créée en 2003 par Emmanuel Bertrand et compte aujourd'hui plus de 30 employés. Au départ, l'entreprise se tourne vers le secteur des stations de lavage auto en créant une gamme de distributeurs de jetons. Par la suite, elle élargie sa collection de produits articulés autour de la monétique et se lance dans la conception de ses propres cartes électroniques.

L'entreprise conçoit des bornes principalement pour les stations de lavage auto ainsi que les laveries mais s'intéresse aussi à d'autres marchés comme l'hôtellerie. Un projet de casiers automatisés, qui a été présenté sur TF1, est aussi en train de se mettre en place. Ce nouveau projet est innovent et écologique, car il privilégiera les producteurs locaux et évitera les voyages en voiture pour se rendre dans les grandes surfaces. Ce projet permettra à l'entreprise de faire face au déclin des stations de lavage auto auxquelles on impose des restrictions à cause des sécheresses de plus en plus fréquentes. Aujourd'hui, plus de 40000 stations de lavage auto sont équipées de bornes CKsquare. On peut voir sur la figure 1 un exemple de produits conçues par l'entreprise.

FIGURE 1 – Produits CKsquare

CKsquare fait partie du groupe le Petit Poucet (LPP) composé de cinq sociétés qui travaillent en collaboration. Parmi ces entreprises, on compte la société M-Innov qui se charge de la conception et de l'installation des bornes et systèmes monétiques pour les aires de services, campings, parking ou hôtels. La société Mecasystem International, elle, se charge de la tôlerie et de la mécanique pour les bornes CKsquare et M-Innov. La société Ehrse, née au sein même de CKsquare, se charge des tests, du prémontage et du câblage des cartes électroniques. Enfin, il y a la société Logawin, société fille de CKsquare qui est une entreprise de développement informatique. La société Logawin est composée de deux pôles, Logawin France basé à Clermont-Ferrand et Logawin Tunisie basé à Tunis. Ces cinq sociétés travaillent en coopération ce qui permet au groupe CKsquare d'avoir la maitrise de la conception et de la fabrication de tous les composants des produits. On peut voir sur la figure 2 les logos des sociétés citées précédemment.

FIGURE 2 – Logos des membres du groupe LPP

L'objectif de la société CKsquare est de pouvoir fournir des produits configurables et adaptables aux besoins des différents clients. C'est pour cela que les bornes possèdent beaucoup d'options et que l'entreprise entretient un savoir faire quant à la gestion de la plupart des systèmes de paiements disponibles sur le marché. De plus, une équipe SAV reste à l'écoute du besoin des clients ce qui permet à l'entreprise de concevoir des solutions encore plus spécifiques et personnalisées.

Un des gros atouts de la société est son savoir faire concernant l'utilisation des cartes électroniques. En effet, presque toutes bornes CKsquare sont équipées de cartes électroniques beaucoup plus fiables et moins énergivores que des PC. Cependant, ces cartes doivent assurer beaucoup de fonctionnalités et gérer un grand nombre de composants ce qui pose de gros problèmes en terme d'optimisation du stockage. En plus, la gestion de systèmes de paiements bancaires implique encore plus de contraintes. Par exemple, la loi finance de 2016 (appliquée en 2018) a imposer la collection et la sauvegarde sécurisée des historiques de paiement.

2 Travail demandé

L'objectif du stage est de réaliser un outil servant à faire de l'intégration continue pour valider les fichiers de la partie commune du code utilisé sur les bornes. L'outil doit permettre l'écriture de tests pour valider le code et doit pouvoir être automatisé dans une pipeline Gitlab. L'intérêt pour l'entreprise est de pouvoir détecter un maximum de problèmes le plus tôt possible et de manière automatique pour ne pas envoyer du code non fonctionnel en production. Le code doit normalement s'exécuter sur une carte électronique qui gère différents composants. La carte et les composants électroniques ne seront pas accessibles dans la pipeline, il faudra donc simuler les composants pour pouvoir tester le code.

Le résultat final doit être un outil qui doit pouvoir être facilement réutilisable et adaptable. La documentation et la structure du code doit pouvoir permettre d'aisément modifier ou copier les différents éléments. Par exemple, il faudra que tous les composants simulés aient la même structure et que cette structure soit suffisamment simple et générique pour pouvoir être copiée pour la création d'un nouveau composant. Ici, le produit final n'est pas tellement le code en lui même mais plus la méthode employée pour mettre en place les tests.

À noter que l'objectif du projet n'est pas d'écrire des tests, la philosophie de l'entreprise est que ce sont

les développeurs qui tests leurs code. Des tests ont été écrits durant le stage mais ces derniers ont pour objectif de valider le bon fonctionnement des composants simulés et non celui du code de la société. Par contre, il faudra fournir une documentation complète décrivant comment tester les programmes. Cette documentation devra présenter le framework de test et décrire le fonctionnement des composants simulés ainsi que leur utilisation.

Enfin, concernant les limites du projet, l'objectif est de pouvoir tester les bibliothèques communes aux différents projets. Cela comprend le code commun de la partie C ainsi que le code commun de la partie Qt (C++). À noter qu'au niveau de la partie, il y aura deux branches à traiter, et ces dernière on un fonctionnement différent, il faudra donc adapter l'outil de tests pour les deux branches.

Maintenant que nous avons décrit les bases du projet, intéressons nous un peut plus en détail au problème posé.

3 Analyse du problème

Le but du stage est de créer un outil permettant de faire de l'intégration continue. De ce fait, cela va nécessiter de trouver un framework de test complet mais aussi suffisamment simple d'utilisation. En effet, les développeurs de l'entreprise ne sont pas nécessairement formé à l'utilisation de ce types d'outils. Concernant le framework, il faudra aussi que ce dernier soit simple à mettre en place dans une pipeline. Cela signifie que les frameworks disponibles dans les bases des gestionnaires de paquets seront privilégié.

Comme mentionné dans la partie précédente, l'outil sera utilisé pour tester du code normalement exécuté sur du matériel embarqué, il faudra donc un moyen de tester des fonctions qui interagissent avec le matériel électronique. Pour ce faire, il faudra simuler les interactions avec le matériel en créant des fonctions et des structures de données qui réagiront comme les composants électroniques. Par exemple, si une fonction doit modifier un registre sur une carte, il faut pouvoir émuler le registre pour vérifier que les bonnes modifications ont été apportées dessus. De plus, les composants électroniques sont interfacés avec différents protocoles. Il faudra donc comprendre le fonctionnent de ces protocoles ainsi que leur implémentation par l'entreprise pour pouvoir simuler une communication réaliste entre la carte et les composants. Le fait de devoir faire de la simulation pour les tests peut complexe si le framework choisi n'est pas adapté. Il faudra que le framework de test ne soit suffisamment flexible et offre un maximum d'outil pour tester les structures de données qui vont être utilisées dans la simulation.

Étant donné que les tests seront exécutés dans une pipeline (donc dans un docker), il faudra s'assurer que le code puisse compiler sous Linux. Ici, on ne pourra pas utiliser MPLAB (IDE fourni par Microship) et les compilateurs utilisés par l'entreprise à cause de la simulation. Le compilateur utilisé sera gcc et une partie des bibliothèques de Microship seront modifiées pour arriver à générer un exécutable.

Enfin, le plus gros défi de ce stage va être le fait de comprendre et d'utiliser le code de l'entreprise. En effet, la taille du code est assez conséquente et beaucoup de choses ont été réécrites par l'entreprise, notamment des bibliothèques du compilateur. Il faudra soigneusement choisir ce qui va être simulé et où va se placer la simulation car tout le code ne pourra pas certainement pas être testé. Par exemple, la partie concernant le protocole TCPIP est assez ancienne et suffisamment complexe pour que les développeurs de l'entreprise ne souhaitent pas la modifier ou la tester.

Deuxième partie

Réalisation et conception

1 Choix du framework de test

Le but du projet est de concevoir un outil permettant de tester du code, la première tâche à donc été de choisir un framework de test. Le framework de test constitue la base de l'outil, c'est donc un choix assez important. Dans cette partie nous traiterons de la procédure qui a été utilisée pour trouver et comparer des frameworks et des bibliothèques de test.

1.1 Les critères de comparaison

Étant donné le fait que le langage C est très utilisé, il y a beaucoup de choix quand aux différentes bibliothèques de test utilisables. Le premier travail a été de comparer bibliothèques et frameworks de tests existants. Une liste des frameworks disponibles sur Wikipédia [1] a permis de prendre connaissance des frameworks disponibles pour ensuite pouvoir faire plus de recherches. Ce travail de recherche à permis de faire un prés tri et d'éliminer les framework incomplets ou trop peu utilisés. Une fois la liste des meilleurs frameworks terminée, il a fallut trouver des critères pour comparer les frameworks.

Tout d'abord, l'équipe de développement souhaitait pouvoir tester à la fois du code **C** et du code **C++** pour certaines parties développées par l'équipe **Qt**. Ce critère était optionnel mais apprécié. À noter que lorsque l'on parle de pouvoir tester du code C++, cela ne prend pas seulement en compte le fait de pouvoir exécuter des fonctions basiques puisque c'est possible avec tous les frameworks C étant donné la compatibilité entre le C et le C++. Pour pouvoir tester du code C++, il faut aussi que le framework soit capable d'interagir avec les structures de donnés fournis par la bibliothèque standard de C++ ou encore de pouvoir traiter des exceptions. Étant donné ce critère, l'idée d'utiliser un framework écrit en C++ a été envisagé.

Un autre critère concerne la modernité et la facilité d'utilisation du framework. Cela peut sembler anodin mais l'écriture des tests est une tâche aussi longue que le développement. Pour ne pas perdre de temps, il est préférable que les tests soient le plus simple possible à mettre en place. De plus, les tests peuvent aussi servir de documentation, c'est donc un avantage non négligeable que d'avoir un framework qui permette d'écrire des tests simples, lisibles et compréhensibles. Enfin, ce critère impacte aussi le temps de conception de l'outil de test car le fait d'utiliser un framework trop complexe aurait nécessité la conception de fonctions et macros (pour réduire la complexité) et rallongé le temps d'écriture de la documentation. À noter que pour valider ce critère, la documentation des différents outils a aussi été étudiée, les frameworks devaient donc fournir une documentation suffisamment claire et précise permettant d'utiliser facilement toutes les fonctionnalités proposées.

Le critère le plus important est celui du statut du développement du framework. En effet, lorsque l'on souhaite utiliser un outil, une question importante à se poser est de savoir ce que l'on peut faire en cas de problème. Ici, il a fallut regarder la taille, la popularité et l'age des projets. En effet, plus un projet est populaire plus il sera facile de trouver de l'aide en cas de problème. De plus, les projet important ont souvent beaucoup plus de collaborateurs ce qui peut accélérer la corrections des bugs ou la vitesse de réponse au issue. Enfin la dates de dernières mis à jours ont aussi été répertoriées car là aussi, il est beaucoup plus simple de résoudre les problèmes sur un projet qui est encore activement maintenu.

D'autres critères ont permis de démarquer les frameworks comme par exemple le fait que les frame-

works fournissent des fonctionnalités supplémentaires comme l'export des résultats des tests dans différents format comme TAP ou XML (utile pour faire des rapports) ou encore des générateur de nombres pseudo aléatoires pour faire des tests avec des entrée aléatoires, ... Une autre fonctionnalité intéressante est que les frameworks exécutent les tests dans des threads séparer ce qui permet de tester des signaux ou encore de ne pas stopper tous les tests à cause d'une sortie erreur. Le fait que les frameworks utilisent beaucoup de macros à aussi été pris en compte car bien que ces dernières permettent de rendre le code beaucoup plus simple, elles peuvent aussi être source de problèmes (elles ont parfois un comportement non souhaité et elles sont très compliqué à déboguer). Le frameworks qui a été choisi possède ce défaut et nous verrons les problème que cela pose lorsque nous détaillerons ce framework.

1.2 Les frameworks

Dans cette section, nous allons faire une revue de tous les frameworks de tests étudiés pendant le début du stage. Une fois ceci fait, nous présenterons le choix final.

Check

Le premier framework de la liste est Check, il propose un interface simple pour l'écriture des tests, cependant, toute la mise en place des suites de tests est plus complexe mais peut être changée facilement. Check permet d'exécuter les tests dans des zones mémoires séparées ce qui permet de ne pas s'arrêter lors de l'émission d'un signal comme SIGSEV. La bibliothèque Check est disponible avec un paquet aptitude, ce qui le rend simple à installer. C'est aussi une bonne preuve de la popularité de cette bibliothèque. Check est assez complet et donne un rapport clair et facile à utiliser après l'exécution des tests. Le framework permet de construire une structure de tests classique où l'on groupe les tests dans des suites. Par contre, Check n'est pas compatible avec le C++.

Pour présenter les framework aux développeurs de la société, des exemples de codes ont été présentés pour permettre d'avoir une idée de comment le framework s'utilise. Sur le listing 4 de l'annexe B on peut voir un exemple d'utilisation de Check.

CUnit

Le second framework est CUnit, il est aussi disponible avec un paquet aptitude cependant. Le frameworks est assez complet et fourni beaucoup de fonctions d'assertion. CUnit permet de construire la même structure de tests que Check, et utilise des pointeurs de fonctions pour construire les suites. Pour chaque suite de tests, on peut fournir deux fonctions qui seront exécutées avant et après les tests pour permettre d'initialiser l'environnement de tests (ces fonctions sont généralement appelées **setup** et **teardown**). Ce framework ne sera pas compatible avec C++. Le framework CCPUnit est similaire à CUnit et permet aussi de tester du code C, cependant, il est nécessaire d'utiliser des classes C++ ce qui rend les tests plus complexes à mettre en place.

On peut voir sur le listing 5 de l'annexe B un extrait de code utilisant CUnit.

Criterion

Le framework suivant est Criterion qui est assez récent et aussi disponible avec un paquet aptitude. Il propose une interface très simple pour écrire des tests et des suites de tests. On peut facilement ajouter les fonctions de setup et teardown (pour les tests et les suites de tests) et les tests peuvent êtres paramétrés. Comme Check, les tests sont exécutés dans des zones mémoires séparées. On peut aussi facilement tester si un signal (comme SIGSEV) est émis ou pas. De plus, le framework propose beaucoup de macros

permettant de faire des assertions non seulement sur les types primitifs, mais aussi sur les tableaux. Enfin, Criterion possède aussi une interface C++.

À noter tout de même que la simplicité de l'interface de Criterion vient du fait que le frameworks utilise beaucoup les macros ce qui peut poser problème.

On peut voir sur le listing 6 de l'annexe B un example de tests écrits avec Criterion.

Minunit

Minunit est la plus simple des bibliothèques de tests trouvée. Elle ne se compose que d'un fichier d'entête. L'interface proposée est très simple mais très basique, elle permet seulement l'écriture de tests et de suites de tests. Cette bibliothèque est une collections de macros qui pourraient être utilisées pour créer un framework de test plus complet. La bibliothèque fourni aussi quelque fonctions d'assertion.

On peut voir sur le listing 7 de l'annexe B un exemple d'utilisation de Minunit.

Munit

Munit propose une interface plus complexe car cela nécessite d'utiliser des tableaux et des structures. Pour les tests, les fonctions de tests sont très simple à écrire et il y a la possibilités d'avoir différents types de retours. Pour chaque test, on peut associer les fonctions de setup et de teardown. Les tests peuvent aussi être paramétrés. Le framework propose aussi une interface en ligne de commande, il est donc possible de donner des paramètres au programme pour choisir quels tests sont exécutés. Il est aussi possible de construire une structure de test plus complexe cas on peut avoir des suites de suites de tests. Le framework propose aussi des fonctions de génération de nombres aléatoires.

On peut voir sur le listing 8 de l'annexe B comment s'utilise Munit.

Unity

Framework de test spécialisé pour les systèmes embarqué, léger et simple. Il ne permet pas de construire une structure de tests très complexe (seulement de simple tests) mais possède beaucoup de fonctions d'assertion. En terme d'interface, le framework propose une collection de macros simples et lisibles. Le framework fourni aussi un script pour faciliter la mise en place des tests (en revanche ce script est assez mauvais car il ne génère un runner que pour une seule suite).

Le framework possède beaucoup de fonctions d'assertions mais il ne possède pas beaucoup plus de fonctionnalités. Il ne sera pas utilisable pour tester du code qui utilise des fonctionnalités propres à c++ comme les exceptions par exemple.

On peut voir sur le listing 9 de l'annexe B un exemple de test utilisant Unity.

Tau

Très léger, le framework ne se compose que de fichiers d'entête. Il permet de construire une structure de tests avec des tests et des suites de tests. Les tests sont écrits en utilisant des macros, ce qui rend l'interface très simple. Tau permet facilement d'avoir plusieurs fichiers de tests en générant son propre main. Par contre, il ne permettra pas de tester les exceptions en C++.

On peut voir sur le listing 10 de l'annexe B un extrait de code utilisant Tau.

1.3 Le choix final

Le choix final s'est porté sur Criterion car le framework possède beaucoup d'avantage. Tout d'abord, il est le seul à être complètement compatible avec le C++ et ce, sans proposer une interface nécessitant

la mise en place de classe comme ce que l'on pourrais voir avec CPPUnit. De plus, ce framework est très simple d'utilisation, il permet d'écrire des tests clairs facilement et rapidement. Il propose aussi beaucoup de fonctionnalités comme la possibilité de générer des logs, les tests paramétrés, les théories (tests des vecteurs d'entrées et de sorties) et possède aussi une interface en ligne de commande permettant de passer des options en paramètres du programme.

Par contre, la simplicité d'utilisation de ce framework cache une grande complexité au niveau de son implémentation. Le framework a posé quelques problèmes mineurs dans la suite du projet. Tout d'abord, la simulation des composant nécessite l'appel de fonctions d'initialisation au préalable et cela aurait été pratique de le faire dans la fonction `main`. Comme expliqué dans les parties précédentes, le framework génère son propre point d'entré, cependant il est possible d'écrire la fonction `main` à la main si besoin. Le problème est que pour une raison inconnue, cela n'a pas marché lors des essais au début du projet (d'après le message d'erreur, le problème venait des threads). Ce n'est pas un problème grave car Criterion propose beaucoup d'outils qui permettent de mettre en place des solutions alternatives mais c'est tout de même un point important à noter. De plus, le fait que le framework utilise beaucoup de macros pose parfois problème car ces dernière peuvent avoir des comportements indésirables. Par exemple, il peut arriver qu'un test basique d'égalité avec la fonction d'assertion principale ne compile plus lorsque l'on échange les expressions de par et d'autre de l'opérateur `==`. Les macros peuvent être très utiles et très puissantes, elles permettent ici de rendre le code beaucoup plus lisible, cependant, il faut noter que les macros trop complexe sont très difficiles à écrire et à déboguer.

Le framework choisit possède donc beaucoup de qualité mais aussi quelques défauts. Au final, bien que ce framework ne soit pas parfait, il remplit bien son rôle et il propose beaucoup de fonctionnalités très utiles pour écrire les tests. De plus, il faut noter que ce framework est assez récent, il est donc normal qu'il y ai encore des défaut qui seront certainement corrigés au fil des années de développement.

2 Organisation d'un projet CKsquare

L'outil créé pendant ce stage a été testé sur un projet de l'entreprise. Cela a permis dans un premier temps de pouvoir voir et comprendre comment le code à tester fonctionnait puis ensuite cela à permis de vérifier la bonne intégration de l'outil dans le projet. Dans cette partie, nous détaillerons comment sont organisés les projets de CKsquare. Cela permettra une meilleur compréhension des choix qui ont été fait par la suite.

2.1 Organisation des fichiers

Dans cette partie, nous allons résumer l'organisation des fichiers dans un projet CKsquare puis nous traiterons le fonctionnement du code dans la partie suivante. On rappel que ce projet d'intégration continue ne concerne que la partie du code qui est commune à tous les projets. Elles fait parties des bibliothèques ajoutés aux projets sous la forme de sous module git.

Pour pouvoir penser le fonctionnement de l'outil de test à créer pendant ce stage, il était important de comprendre le fonctionnement général du projet. Avant de s'intéresser au fonctionnement, il faut comprendre comment le code est organisé. Cela permet deux choses, premièrement, cela permet de ne pas se perdre et de pouvoir facilement retrouver les fichiers. Pour comprendre le fonctionnement du code, il est important d'avoir un plan de l'organisation. Deuxièmement, il est aussi très important de comprendre comment s'est organisée l'entreprise pour pouvoir structurer le code de l'outil créé. En effet, l'organisation du projet de test doit suivre les principes de CKsquare pour rendre l'outil facile à utiliser

et à maintenir pour les développeurs de l'entreprise.

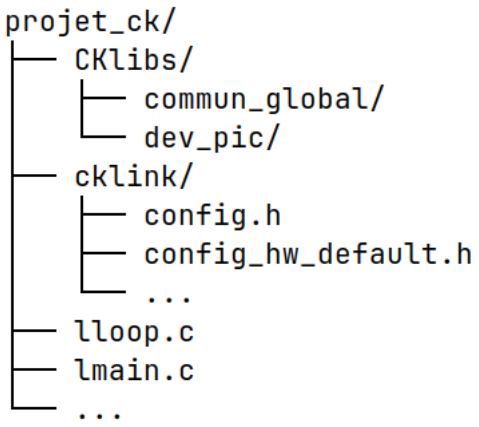

FIGURE 3 – Arborescence d'un projet CKsquare

Comme on peut le voir sur l'arborescence de la figure 3, chaque projet comporte une partie locale ou les noms de fichiers commencent par **l**. Cette partie contient tous les fichiers qui sont spécifiques au projet courant (fonction d'initialisation, fonction principale, ...). Chaque projet contient au moins deux fichiers de configuration qui sont des fichier d'entête contenant des constantes préprocesseur utilisées pour faire de la compilation conditionnelle. La gestion de la configuration s'est avérée complexe pour créer l'outil de test à cause du grand nombre d'options. Chaque projet contient aussi un répertoire `CKLibs` dans lequel se trouvent des sous module git. Sur le projet d'étude qui a permis la mise en place du projet de test, il y avait deux sous modules. Le premier sous module se nomme `commun_global` et il contient la partie du code qui est commune à tous les projet C, C++ et python. Le second sous module est `dev_pic`, ce dernier ne concerne que la partie C.

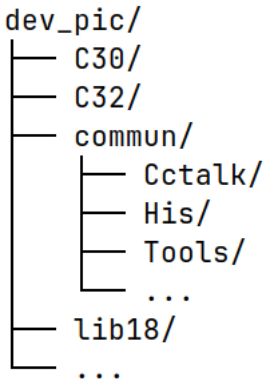

FIGURE 4 – Arborescence du répertoire dev_pic

La partie que nous allons traiter se trouve dans le répertoire `dev_pic` (aussi appelé codes communs). Comme on peut le voir sur la figure 4, ce répertoire possède un sous répertoire `commun` qui contient la partie du code à tester. À la racine des communs, se trouvent aussi une partie des bibliothèques des différents compilateurs. L'entreprise CKsquare a réécrit une grande partie de ces bibliothèques pour avoir plus de maitrise quand à la gestion des différents éléments. Ces bibliothèques ont été très utiles pour mettre en place une partie de l'émulation des composants dans le projet de test, nous traiterons cela plus loin dans ce rapport. Le reste des bibliothèques des microcontrôleurs a aussi été utilisé pour compiler avec gcc, là aussi, ce sera traité dans les prochaines sections.

Le répertoire `commun` s'organise en catégories. Par exemple, on retrouve le dossier `StorageDriver` qui contient les fichiers qui gèrent le stockage ou encore le dossier `Payment` qui contient tous ce qui concerne le paiement (types de paiement, accepteur de pièce, ...). L'objectif est de pouvoir faire des recherche par catégories et de garder les dépendance au plus proche des fichiers. Par exemple, plusieurs des sous répertoires de `commun` contiennent un dossier `Web` qui regroupe les dépendances web des fichiers de chaque catégories. On peut ainsi retrouver facilement les dépendance des fichiers.

Maintenant que nous avons expliqué l'organisation générale du projet, nous allons pouvoir traiter le fonctionnement de ce dernier.

Fonctionnement du projet

Le code commun comporte deux branches, une branche principale et une branche plus récente sur laquelle le code fonctionne avec un système d'évènement. La branche sur l'événementiel a été traitée en deuxième partie de stage.

Sur les deux branches, le code est géré par des machines à états. Elles sont utilisées autant du côté du C que de celui du C++. Les machines à état sont un outil très puissant est très pratique car elles permettent d'avoir un meilleur contrôle sur l'exécution du code et ont un fonctionnement assez intuitif. De plus, elles peuvent permettre de rendre le code plus clair en particulier quand il y a beaucoup de cas à traiter. Enfin, elles permettent de gérer les erreurs simplement en évitant la duplication de code.

Branche principale

Cette branche a été la première créée par les développeurs de la société, elle est utilisée sur la plupart des bornes. Le principe est que chaque fichier fournit une fonction `Control` (exemple : `BILLVALIDATOR_Control`). Ces fonctions sont des machines à états qui permettent de contrôler une fonctionnalité. Par exemple, pour un accepteur de billet, la fonction `Control` aura des états qui seront utilisés au début et qui vont permettre de faire de l'initialisation. Ces états peuvent faire des requêtes au composant pour récupérer sa configuration. La machine à états aura aussi un état servant à interroger le composant durant sont exécution, pour par exemple savoir si il y a un billet à traiter.

Les fonctions `Control` sont appelées par un gestionnaire (`ControlManager`). À l'initialisation du programme, toutes les fonctions `Control` sont ajoutées dans une liste et pour chacune d'entre elles, on précise la durée minimale entre deux appels. Ensuite, le gestionnaire est utilisé dans la boucle principale et va appeler toutes les fonctions qui ont été ajoutées lorsque c'est nécessaire. Par exemple, on peut ajouter la fonction de l'accepteur de billet à l'initialisation et spécifier que cette fonction soit appelée toutes les dix centièmes de seconde. Avec cette configuration, la fonction sera appelée par le gestionnaire et il y aura au minimum dix centièmes de seconde entre chaque appel.

Les fonctions `Control` vont être un élément très important à tester. Ces fonctions ont beaucoup servi lors des essais réalisé sur les fichiers pendant la création de l'outil. Elles sont très intéressantes car elles permettent de couvrir une grande quantifié de code et de tester la plupart des cas (pour le code de l'outil de test). À noter cependant que les machines à états sont très compliquées à tester puisqu'il ne faut normalement pas avoir accès aux états dans les tests [2]. Il faut donc être capable de manipuler l'environnement dans le code pour savoir dans quel états on se trouve et choisir l'état dans lequel se rendre. L'utilisation d'un débogueur a grandement facilité la mise en place des tests réalisés sur ces fonctions.

2.1.1 Branche événementiel

Cette branche a été créée pour simplifier le code, elle utilise un système mis en place sur la partie Qt (code C++) qui permet de déclencher des appels de fonctions en émettant des signaux. Le deuxième objectif de ce système est de limiter les appels de fonctions dans la boucle principale. En effet, sur l'autre branche, les fonctions `Control` des composants sont continuellement appelées par le `Contro Manager`, ce qui peut poser des problèmes de performance. Ici, les fonctions qui gèrent les composants ne sont appelées que lorsqu'un évènement est émit. Par exemple, lorsqu'une trame est envoyée, un évènement est émit pour signifier la fin de la communication. Cela aura pour effet de déclencher l'appel de la fonction de gestion du composants qui a envoyé la trame. Cette fonction est une machine à états qui passera dans un état d'attente de réponse (par exemple). Le défaut de ce système est qu'il nécessite l'utilisation de structures de données assez lourdes en mémoire, cependant, ce défaut est compensé par le fait que la taille du code diminue fortement.

Le fonctionnent de ce systèmes repose sur deux structures de données, les `SMs` et les `Connects`. `SM` signifie *State Machine*, comme son nom l'indique, cette structure permet de gérer les machines à états. La seconde structure de données permet de gérer les signaux. Ici, les fonctions `Contro` sont remplacer par des fonctions `SwitchControl` qui prennent en paramètre un `SM` qui va permettre la gestion de la machine à états dans la fonction. Les `SM` contiennent des pointeurs vers des données utiles pour les machines à états, comme des arguments ou encore une horloge spécifique à la machine. Le fichier des `SM` propose plusieurs fonctions pour naviguer entre les états des machines comme `SM_Go` qui permet d'aller directement dans un nouvel état de la machine. Il y a aussi la fonction `SM_Exit` qui fonctionne comme la fonction précédente sauf qu'ici, la machine à état s'arrête et attend qu'un signal soit émit pour se relancer. Le fichier propose aussi des fonctions permettant de gérer les horloges des machines à états, permettant par exemple de quitter une machine à états pendant un temps minimum donné et de revenir lorsque l'horloge aura émis son signal de fin. On a donc plusieurs façons de changer d'état dans une machine, on peut y aller directement ou attendre un signal comme celui émit à la fin d'un minuteur.

Pour que l'appel à une fonction soit déclenché à l'émission d'un signal, il faut connecter la fonction au signal. Pour ce faire, on utilise la structure des `Connects`. Un `Connect` est un signal qui est connecté à une fonction. Lors de la connections, on spécifie les arguments à donner à la fonctions qui est connectée. Lorsque l'on a créé un `Connect` et que ce dernier a été lié à une fonction, on peut utiliser une fonction `Emit`, qui va émettre le signal. Cela aura pour effet de déclencher l'appel de toutes les fonctions connectées au signal (immédiatement ou non). Ce système est très similaire à celui utilisé sur Qt en C++. L'implémentation de ce système en C utilise une liste chaînée de `Connects`. Lorsque l'on fait une connections, on créé un nouveau `Connect` en spécifiant le pointeur sur la tête de la liste des `Connects` liés à un signal. La fonction `Emit` prend en paramètre la tête le la liste, et va ajouter toute la liste à une liste de `Connects` à traiter (il y a deux chaînages). Ensuite, on a une fonction qui traite les signaux émit en appelant toutes les fonctions connectées avec les bon arguments.

À noter que les fonctions `SwitchControl` prennent aussi en paramètre une évènement. En effet, dans certains état, on ne souhaite pas que les machines soient lancées par n'importe quel signaux. Pour palier à ce problème, on entre toujours dans une machine à état avec une évènement, et ce dernier sera vérifié ou non en fonction de l'état dans lequel se trouve la machine.

Ce système est assez élégant et très intéressant d'un point de vue génie logiciel. Il permet vraiment de simplifier le code et il est souvent plus intuitif dans son utilisation que le système de `Control Manager`. Par contre, ce n'est pas un système simple à mettre en place en C.

3 Organisation du projet de tests

Dans cette partie, nous allons détailler la structure du projet de test. Cette structure s'appuie sur celle des projets de l'entreprise.

Il est important de noter que l'organisation du projet de test à changée. Au départ, le projet de test devait être un géré à part. Ce dernier devait être cloné dans la pipeline du code commun pour pouvoir tester les fichiers. Au final, il a été décidé d'intégrer complètement l'outil de test dans `dev_pic`. L'organisation des tests suis les principes expliqués dans la section 2.1. De ce fait, les tests sont stocké à proximité des fichiers testés. Sur la figure 5, on peut voir une représentation de l'arborescence des fichiers dans le projet commun.

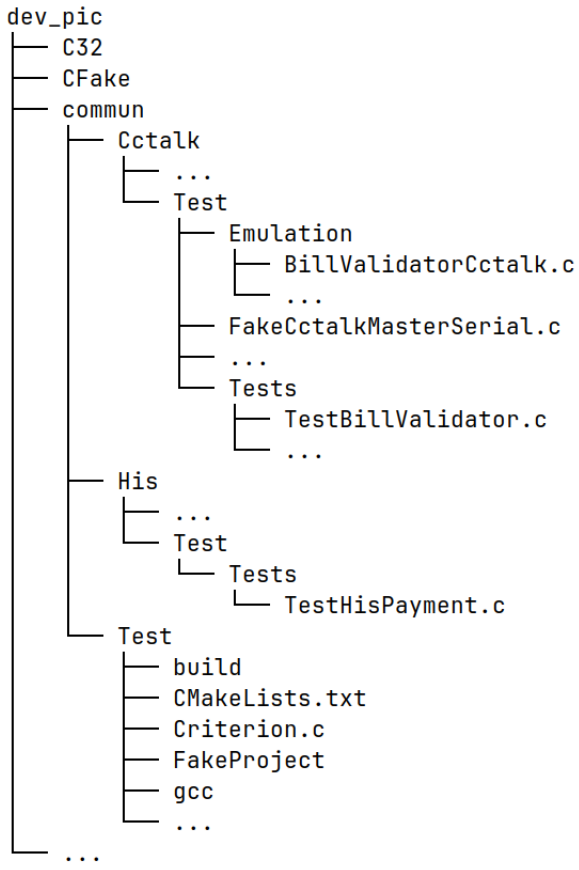

FIGURE 5 – Intégration des tests dans le code commun

Cette arborescence montre une partie du contenu du répertoire `dev_pic/commun`. Ici, on peut voir trois des catégories qui ont été mentionnées dans la section 2.1. Le répertoire `Test` a été ajouté lors de l'intégration du projet de test dans le projet commun, ce dernier contient les fichiers généraux spécifiques aux test. À noter qu'il y a un autre répertoire `CFake` qui lui se trouve à la racine du projet, au même niveau que `commun`. Ce dernier contient la simulation des interfaces pour le stockage. Étant donné que le stockage est un élément assez général, il a été mis à part. Comme on peut le voir ici, tous les fichiers ayant un lien entre eux sont stockés ensemble et il en est de même pour les tests.

Maintenant que nous avons définie une carte générale du projet, détaillons les parties les plus importantes.

3.1 Le répertoire Test principal

Comme expliqué précédemment, ce répertoire contient des fichiers globaux qui permettent de pouvoir compiler et exécuter les tests. Les différentes parties sont décrites dans les sections suivantes.

Faux projet

Ce fichier contient des versions modifiées des fichiers du projet local (ceux dont le nom est préfixé d'un `l`). Il contient aussi les fichiers de configuration. Pendant la création de l'outil, il s'est avéré qu'il y avait des dépendances entre les fichiers des communs et le projet local. Cela pose problème car depuis la pipeline, on a pas accès aux fichiers locaux puisqu'ils sont spécifiques aux différents projets. Une solution envisageable est le clonage d'un projet depuis la pipeline, cependant cette méthode possède deux gros défaut. Le premier réside dans le fait de devoir cloner un projet entier juste pour exécuter une pipeline, cela prend du temps et des ressources. Le deuxième défaut vient du fait que la configuration des tests n'est pas la même que celle du projet local. En effet, dans les tests, on ne peut pas compiler tous les fichier (car tout n'a pas été simulé et tout ne doit pas être testé) et on utilise une configuration qui permet de pouvoir tester plusieurs fonctionnements différents. De ce fait, il est impossible de compiler les tests en utilisant les fichiers originaux. C'est pour ces raisons qu'il a été décidé de récupérer seulement les fichiers nécessaire à la compilation des tests et de les modifier si besoin.

Gcc

Ce répertoire contient des versions modifié des fichiers du compilateur `c32` (normalement stockés dans le répertoire `C32` à la racine des communs). Là aussi, les fichiers sont modifiés pour que le code puisse compiler avec `gcc`. À noter que ce répertoire contient un fichier créé à partir de la bibliothèque `plib` qui définie toutes les variables globales normalement utilisées par les bibliothèques carte. Tous les fichiers contenus dans ce répertoire permettent donc de compiler le code sans avoir accès au bibliothèques du `PIC`.

Configuration de Criterion

Pour que les tests incluant l'émulation de composants fonctionnent, il faut pouvoir lancer et initialiser la simulation. Pour ce faire, on utilise un fichier qui fourni des fonctions d'initialisation et de terminaison globales qui peuvent être utilisées dans les test. La fonction d'initialisation permet de créer les threads utilisé pour les horloges comme nous le verrons dans la section 4 ainsi que d'initialiser toutes les variables nécssaire au bon fonctionnement de la simulation. Dans la fonction de terminaison, on termine simplement les threads des horloges. Dans ce fichier, on peut aussi ajouer d'autre fonctions spécifiques au framework.

3.2 Les répertoires de test secondaires

Comme dit précédemment, les développeurs de l'entreprise souhaitaient stocker les tests à proximité des fichiers testés. C'est pour cela que l'on a des répertoires de test secondaires dans chaques répertoire du projet. Dans les section suivant, nous allons décrire le contenu de ces dossiers.

3.2.1 Les fichiers modifiés

Certains tests nécessitent la création de versions modifies de certains des fichiers de l'entreprise pour permettre d'isoler des fonctionnalités du programme et faciliter les tests. Par exemple, les entrées et sorties sont gérées pas un fichier nommé `Ios.c`. Il est très compliqué de savoir si les fonctions de ce fichier sont appelées depuis les tests or parfois, on a besoin de tester si une action d'entré ou de sortie a été

réalisée. Une version modifiée de ce fichier a donc été créée pour les tests. Les fonctions de la version test du fichier mettent à jour des variables globales accessibles depuis les tests permettant ainsi de savoir si des actions d'entrée ou de sortie ont été réalisées. Les versions test des fichiers sont compilés à la place des fichiers originaux pour les tests. Cette solution est similaire au *mocking* (au niveau des fichier) qui est très utilisé avec les langages objets **??**.

3.2.2 Les fichiers de la simulation

Les sous-répertoires de test permettent aussi de stocker les fichiers qui contiennent les éléments nécessaire à la simulation des composants. Par exemple, les fichiers qui permettent de simuler les composants interfacés avec le protocole Cctalk sont stocker dans le répertoire `commun/Cctalk/Test`. Nous traiterons ce protocole dans la section 6.1.

3.2.3 Les tests

Pour les tests, on ajoute encore un nouveau sous-répertoires à l'arborescence. Chaque répertoire `Test` contient un sous-dossier `Tests` qui permet le stockage des fichiers de test.

3.3 Cmake

Dans cette partie nous allons voir comment est compilé le projet de test. Nous justifierons tout d'abord le choix de l'utilisation de CMake. Ensuite, nous détaillerons l'organisation et le fonctionnement de la configuration de cet outil.

Au début du projet, l'outil choisit pour compiler était Makefile. L'avantage de Makefile est qu'il est assez proche du script, ce qui donne une grande flexibilité car on défini toutes les commandes à la main. Le défaut de Makefile est qu'il peut très vite devenir peu lisible. De plus, sur de gros projet, mettre en place de la compilation séparée peut être très complexe, or cela était nécessaire pour le développement du projet étant donnée le nombre de fichier à compiler. L'outil de test à créer devait être suffisamment simple à utiliser et à modifier, la complexité croissante du Makefile à mesure que le projet avançait a pousser à l'utilisation de CMake.

CMake est un outil qui permet de générer un Makefile très complexe. Il permet de mettre en place de la compilation séparée sur de gros projets automatiquement. La configuration de CMake est assez simple et beaucoup plus lisible que celle de Makefile. De plus, CMake propose des fonctionnalités avancées comme la gestion automatique des bibliothèque ou encore **CTest**, qui permet d'automatiser l'exécution de tests. Détaillons à présent la configuration de cet outil.

Le fichier de configuration de CMake est le fichier `CMakeLists.txt` (il doit avoir exactement ce nom). Il faut un ficher de configuration par sous projet. Ici, il n'y a que le projet de tests alors ce fichier se trouve dans le répertoire `commun/Tests`.

La configuration comporte les six sections suivantes :

- **configuration du projet** : cette partie contient la configuration minimale de CMake. Ici, on spécifie la version minimale de CMake requise ainsi que le nom du projet. Cette partie comprend aussi l'ajout du framework Criterion à l'édition des liens ainsi que des options pour `gcc` comme l'option `-g` par exemple (débogage avec gdb).
- **jeux de tests** : dans cette partie, il y a plusieurs listes de fichiers tests (stockées dans des variables réutilisables plus loin). Il y a plusieurs jeux de tests car on souhaite générer plusieurs exécutables. Comme dit précédemment, il y a plusieurs configurations du projet à tester. Chaque exécutable correspond à une fonctionnalité qui nécessite une configuration particulière.
- **projet de test** : cette partie contient la liste des fichiers relatifs au projet de test.

- **projet commun** : ici on a une liste qui contient les fichiers du projet `dev_pic` qui sont à compilés avec tous les exécutables. Ce sont les fichiers dont tous les jeux de tests ont besoin.
- **exécutables** : dans cette partie, on génère les exécutables en spécifiant les bonnes listes de fichiers à compiler. De plus, pour chaque exécutable, on utilise une commande CMake qui permet de définir des constantes préprocesseur lors de la compilation (option `-D` de gcc). Cela permet de choisir les configurations du projet pour les tests.
- **tests** : dans cette section, on ajoute les exécutables à la liste des tests. Ces tests pourront être lancé par `CTest` après la compilation.

CMake permet donc de compiler les tests avec une configuration organisée et lisible en plus de fournir des outils facilitant l'exécution des tests. Dans la section suivant, nous parlerons des premiers composants qui ont été simulés durant le projet, les horloges.

4 Simulation des horloges

Le premier élément qui a été simulé était l'horloge principale du programme. Par la suite, c'est l'horloge Rtc qui a été simulée en suivant le même principe que pour l'horloge principale. Dans cette section nous allons traiter le fonctionnement de ces composants.

Au début du projet, quelques tests simples ont été réalisés sur certains fichiers, cependant, il est rapidement devenu évident que certaines fonctions n'allaient pas pouvoir être testées à cause de l'horloge. En effet, à plusieurs endroits dans le code, on met en place des temps d'attente et le programme est bloqué tant que le temps d'attente n'est pas passé. L'horloge est gérée dans le code à travers une variable globale `TIMER_Centieme` et cette dernière doit être incrémentée pour que le temps passe. Dans le cas contraire, le programme reste bloqué.

Le problème technique que pose la simulation des horloges est qu'il faut que ces dernières fonctionnent en même temps que le programme principale tourne. L'implémentation la plus simple consiste à incrémenter les variables d'horloge dans les tests à chaque fois que l'on sait qu'un temps d'attente est mis en place. Cette solution n'était pas suffisamment pratique et réaliste. Pour ce problème, il a été très rapidement décidé d'utiliser des threads. L'objectif était d'incrémenter les variables des horloges dans des fonctions simples s'exécutant dans un processus séparé en même temps que le programme principal.

Cette solution a été très simple et rapide à mettre en place étant donné que les processus n'avaient pas besoin d'être synchronies. Au final, les deux horloges fonctionnent sur le même principe. Pour chacun des fichiers on a une fonction `Start` qui permet de créer le thread. Cette fonction possède une sécurité qui fait que l'on peut faire autant d'appels que l'on souhaite, le thread n'est créé qu'une seule fois (cela rend l'utilisation plus simple dans les tests). À noter que dans le cas du Rtc, l'horloge démarre à la date du jour. Les fichier comportent aussi une fonction `Loop` qui est la fonction qui s'exécute dans le thread. De plus, des moyen d'interagir avec les horloges ont étaient ajoutés. Par exemple, à certains endroits du code, on met en place des temps d'attente relativement long. Pour éviter de bloquer le programme, chaque fichier propose une fonction `Wait` qui permet d'incrémenter le compteur manuellement. Il est aussi possible de mettre les horloges en pause et de les relancer. Enfin, il y a une fonction `Stop` qui permet de détruire les threads. Les fonctions `Start` et `Stop` sont appelées dans les fonctions d'initialisation et de terminaison globales décrites dans la section 3.1.

Les horloges représentaient donc une partie importante du projet car elles sont énormément utilisées dans le code et si les compteurs ne sont pas incrémentés, il devient impossible de tester les fonctions. La solution qui a été trouvée est très simple et assez réaliste en plus d'être assez pratique à utiliser dans les

tests. Dans la section suivante, nous traiterons le deuxième éléments qui a été simulé lors de la création de l'outil de test et qui propose une solution différente de celle utilisée avec les horloges.

5 Simulation du stockage

Une partie importante de l'émulation concerne le stockage et il y a plusieurs types composants à simuler, les registres, les eeproms, et la mémoire flash. L'émulation des périphériques de stockage est assez simple puisqu'il s'agit juste de tableaux de caractères non signés (codés sur 8 bits sur la plupart des machines). La partie complexe de l'émulation du stockage concerne l'interface qui permet d'interagir avec les périphériques. Il y a deux protocoles qui sont utilisés avec le stockage. Tout d'abord il y a le protocole I^2C qui est utilisé avec les registres et certaines eeproms. Ensuite il y a le protocole SPI qui est utilisé avec les eeproms et les mémoires flash. La simulation de ces deux interface a nécessité l'apprentissage des deux protocoles qui ne sont pas traité dans les cours de la filière 2 à l'ISIMA.

TODO : mettre des images des composants (montrer l'objet au delà du code est intéressant)

Dans cette partie nous allons voir comment a été réalisée l'émulation de l'interface permettant d'utiliser le stockage avec les différents protocoles.

5.1 Échec des threads

La première solution qui a été implémentée utilisait des **threads** de la même façon que les horloges. L'objectif été de pouvoir simuler les composants de sorte à ce qu'ils se comportent comme les composants réels installés sur la carte. Pour se faire, il été souhaitable que les composants simulés soient actif en même temps que la carte (représentée ici par le programme à tester) et c'est pour cela que les threads ont été utilisés. Le principe était que les composants étaient représentés par des machines à états qui bouclaient dans un état de base jusqu'à ce que le composant soit appelé (donc jusqu'à ce que le programme principale décide de lancer une communication en utilisant un des protocoles cités précédemment). Une fois le composant appelé, la machine à états permettait d'assurer la communication. Pour que les composants simulés s'exécutent en même temps que le programme principale, ils s'exécutaient dans des threads séparés.

Le problème des threads réside dans la synchronisation de ces derniers. Il y a différentes méthodes pour synchroniser des threads. Sur ce projet, il a au départ été utiliser des boucles infinies qui permettait de faire attendre les threads. Par exemple, le programme principale était stoppé par une boucle pour attendre que le les composants émulés s'exécutent et le débloque. Cette solution a été utilisée au départ car ce genre de boucles été déjà présentes dans l'implémentation de l'I^2C. Par la suite, cette solution s'est avérée complexe d'utilisation et peu élégante, les boucles ont donc étaient remplacées par des sémaphores. Au final, la synchronisation des threads est devenue trop compliquée et très peu fiable (l'exécution du programme ne donnait pas toujours les même résultats). L'objectif du projet étant de concevoir un outil qui soit facilement réutilisable, cette solution était trop complexe et donc pas adaptée. Il a donc été décidé de ne plus les utiliser les threads même si la nouvelle solution devait être moins pratique d'utilisation au niveau de l'écriture des tests.

5.2 Interface I^2C

Dans cette partie, nous allons voir le fonctionnement de l'émulation de l'interface I^2C du stockage. Nous commencerons par voir les principes du protocole I^2C puis nous détaillerons le fonctionnement de la simulation.

Le protocole I²C

Le protocole I²C (Inter Integrated Circuit) est un protocole série bidirectionnel en halfduplex. La plupart des connaissances sur l'I²C on été trouvées sur ce document [3] et de Wikipédia [4], le reste vient des développeurs de l'entreprise. Ce protocole fonctionne en mode maître et esclave. Par exemple, la carte électronique est considérée comme le maître et elle pilote les périphériques qui eux sont les esclaves. Un esclave ne prend jamais la parole, c'est au maître de démarrer la communication et l'esclave répond lorsque c'est nécessaire. La connexion est réalisée par l'intermédiaire de deux fils, SDA (Serial Data Line) qui permet de transmettre les données et SCL (Serial Clock Line) qui correspond à l'horloge, quand SCL est à 1, on envoi un bit sur SDA. Ces deux lignes sont bidirectionnelles. Le schéma de la figure 6 illustre le principe de l'I²C.

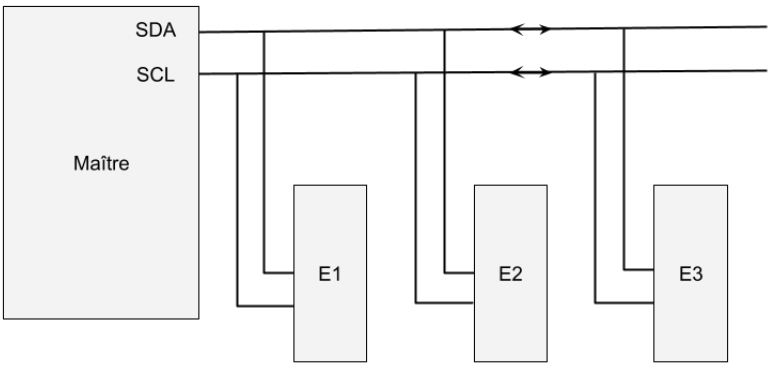

FIGURE 6 – Liaison maître/esclave (I²C)

Pour démarrer la communication, le maître envoi un octet que l'on appel *l'octet d'adresse*. Sur ce premier octet, les sept premiers bits corresponde à l'adresse du destinataire (l'esclave qui est appelé par le maître). Le dernier bit permet de spécifier le mode d'accès, s'il est à 0, le maître va envoyer des données à l'esclave. S'il est à 1, c'est l'esclave qui devra envoyer des données aux maître. Si on prend l'exemple du stockage, quand ce bit est à 0, la carte va envoyer des données à écrire, sinon, la carte souhaite lire les données écrites sur le composant. La suite de la communication va vairée en fonction de ce qui a été demandé par la carte.

Pour résumer, ce protocole fonctionne en mode maître et esclaves avec deux lignes, SDA qui permet de transmettre des données et SCL qui correspond à l'horloge. Lors d'une communication, la carte électronique (le maître) va envoyer un permet octet contenant l'adresse du composant (l'esclave) auquel elle s'adresse ainsi que le mode d'accès. Ensuite, l'esclave et le maître se transmettent des données jusqu'à la fin de la communication. Ce protocole possède d'autre subtilités qui ne seront pas détaillées ici car elles n'ont pas été utiles pour la réalisation de ce projet. Dans la section suivante nous allons aborder le fonctionnement de l'implémentation du protocole par CKsquare puis nous traiterons les différents éléments qui permettent de simuler l'interface I²C pour le stockage.

Émulation

Maintenant que nous avons abordé le fonctionnement du protocole I²C dans la première partie, intéressons nous à la façon dont l'interface a été simulée pour le stockage. Pour mettre en œuvre l'émulation, une étude du fonctionnement de l'implémentation du protocole par l'entreprise ainsi que son utilisation a été nécessaire. Pour ce faire, il a fallut s'intéresser à l'organisation des fichiers et aux fonctions qu'ils contiennent. À la demande des développeurs de l'entreprise, la simulation devait permettre de tester

tout le code. C'est pour cette raison qu'elle a été placée au plus proche des bibliothèques de Microship (fournisseurs des cartes) au lieu de simplement remplacer des fichiers. Il était obligatoire de comprendre comment l'entreprise utilise l'I²C car le protocole en lui même ne décrit que la façon dont les composants communiquent en non les données qu'ils envoient. Par exemple, ce protocole peut être utilisé avec plusieurs types de composants, pas seulement du stockage. Pour deux types de composants différents, la communication suivant le même protocole n'aura pas la même forme car ces derniers ne proposent pas les même fonctionnalités. La simulation mise en place est donc spécifique au stockage sur les cartes Microship.

Tout d'abord, commençons par nous intéresser au fonctionnement du code de l'entreprise. Le compilateur sur lequel se base les tests est le `c32`, ce sont donc les bibliothèques de ce compilateur qui seront utilisées pour mettre en place la simulation. Ces fichiers mettent à disposition plusieurs fonctions permettant de démarrer et stopper la communication ainsi que d'envoyer des données (les fonctions correspondent à toutes le fonctionnalités décrites par le protocole). La communication en elle même est gérée par d'autres fichiers qui eux font partie du code commun. Les fonctions définies dans les fichiers ont permis de comprendre comment sont utilisés les périphériques de stockage. À noter qu'ici, la ligne permettant de transmettre des données est modélisée par une variable (globale). Cette variable est gérée par la bibliothèque du PIC et est définie dans le fichier `PicVariables` mentionné dans la section 3.1.

Pour écrire sur un composant, la carte démarre la communication puis envoie l'adresse du périphérique de stockage auquel ele s'adresse avec le bit de mode à 0 (comme le demande le protocole). Ensuite, elle envoie une adresse qui va permettre de placer le curseur du composant. Cela va permettre de spécifier au composant à quelle adresse on souhaite écrire. Le nombre d'octets à envoyer pour le déplacement du curseur dépend du composant ciblé. Dans le cas d'un registre, on envoie un octet, dans le cas d'une eeprom 2 (les flash ne sont pas utilisées avec l'I²C). Enfin, la carte envoie les données à écrire sur le composant avant de terminer la communication. La lecture suis le même principe sauf qu'ici, on ne spécifie pas la position du curseur. La subtilité est que si on souhaite lire à une certaine adresse, il faut commencer par écrire, déplacer le curseur du composant puis redémarrer la communication en mode lecture.

Maintenant que nous savons comment utiliser le protocole, intéressons nous à la simulation. Comme dit précédemment, la simulation doit se positionnée au niveau des bibliothèques du PIC, la meilleur option était donc de modifier les fichiers de cette bibliothèque. Comme nous l'avons vu dans la section 5.1, la première solution utilisait une machine a états qui s'exécutait dans un thread. Pour réaliser la nouvelle solution, il a été décidé de reprendre le code de la machine à état et d'en faire une fonction `Control` similaire à ce que l'on peut trouver dans le code de CKsquare. Cette décision apporte deux avantages, le premier étant que le fonctionnement est très similaire à celui du code de l'entreprise, ce qui est intéressant étant donné le fait que l'outil est destiné à être utilisé par les développeurs de la société. Le deuxième avantage est le gain de temps du à la réutilisation du code de la solution précédente. Les appels à cette fonction `Control` ont étaient fait dans une version modifiée pour les tests du fichier de `C32` compilé à la place de l'original. Cela permet aux développeur des tests de ne pas avoir à se préoccuper de la simulation, de son point de vue, tout fonctionne comme sur la carte. On peut voir sur la figure 7 le diagramme représentant le fonctionnement de la machine à états de la simulation.

Pour utiliser cette simulation, il faut commencer par configurer les composants en modifiant la fonction d'initialisation. Dans cette fonction, on ajoute les périphériques de stockage à une liste en spécifiant leurs types (eeprom ou registre) ainsi leurs adresses (utilisée pour reconnaitre le périphérique choisi). Il faut aussi donner les indices des composants dans les tableaux mémoire. Comme dit précédent, la mémoire est simulée à l'aide de tableaux à deux dimensions. Ces tableaux correspondent à des listes de mémoires dont la taille varie en fonction du type du composant. Par exemple, il y a un tableau représentant la liste des mémoires pour les eeprom (dont la taille correspond à la taille d'une eeprom). Lors de la configuration, pour ajouter un composant, il faut lui attribuer une mémoire (un indice dans le tableau). Ensuite, le code

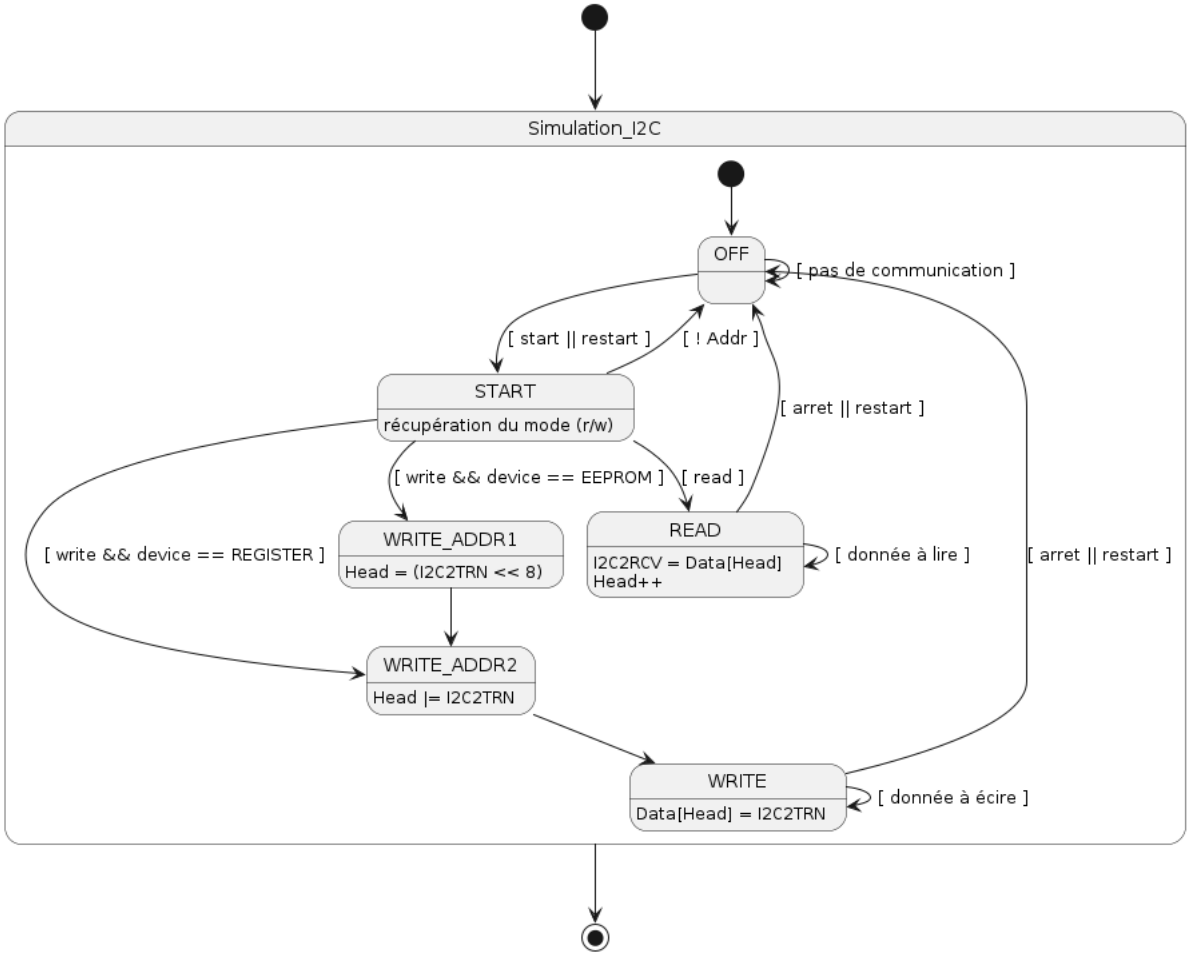

FIGURE 7 – Diagramme d'états de l'émulation de l'interface I²C du stockage

testé va interagir avec la machine à états de la simulation qui va utiliser la mémoire simulée. Pour vérifier le contenu de la mémoire après une opération, on peut utiliser les tableaux de la simulation auxquels on a accès depuis les tests.

TODO : faire un schéma

TODO : parler des graphs dans la doc

5.3 Interface SPI

Le second protocole utilisé avec le stockage est le protocole SPI. Comme pour le protocole I²C traité dans la partie précédente, nous commencerons par expliquer le fonctionnement du protocole puis nous détaillerons l'émulation. Ce protocole est très similaire à l'I²C et il en est de même pour le fonctionnement de la simulation, 'il y aura donc beaucoup d'analogies avec la partie précédente. Ce protocole est très similaire à l'I²C et il en est de mémé pour le fonctionnement de la simulation, il y aura donc beaucoup d'analogies avec la

L'étude de ce protocole à était faite à l'aide de deux documents partie précédente. 'étude de ce protocole à était faite à l'aide de deux documents. [5] et [6]. Le protocole SPI (Serial Peripheral Interface) est un protocole série en fullduplex. Là où l'I²C possédait seulement deux lignes, ici, il y en a au minimum quatre. Il y a tout d'abord SCLK qui permet de gérer l'horologe. Ensuite il y a MOSI (Master Out Slave In) et MISO (Master In Slave Out) qui permette d'envoyer des données. Ici, le fait d'utiliser deux lignes

permet au maître et à l'esclave d'échanger des données en même temps. La dernière ligne, $\overline{CS}$, permet, quand elle est à zéro, de sélectionner le composant avec lequel on souhaite échanger. Cette ligne est relié à un seul composant, il faut donc autant de lignes $\overline{CS}$ (Chip Select) que d'esclaves. Sur la figure 8, on peut voir un schéma représentant la liaison entre la carte et un composant en SPI.

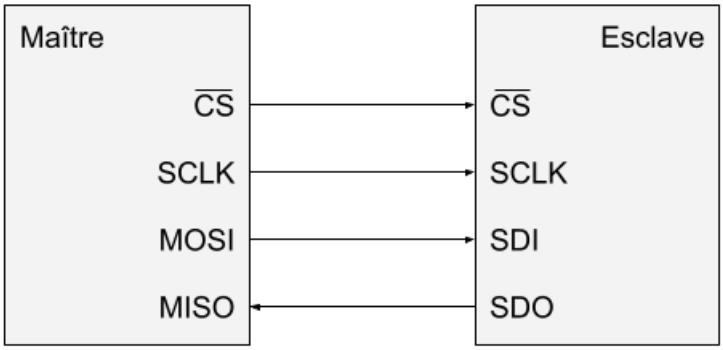

FIGURE 8 – Liaison maître/esclave (SPI)

Maintenant que nous avons vu les principes du protocole, intéressons à la simulation.

Émulation

Pour réaliser la simulation de l'interface SPI, la démarche a était la même que pour l'I²C. La mise en place d'une solution a été précédée d'une étude du fonctionnement du code permettant d'utiliser le protocole.

La gestion du protocole dans le code est très similaire à celle de l'I²C. Il y a un fichier dans le répertoire `C32` contenant les fonctions basiques du protocole. Là aussi, les lignes sont gérées à l'aide de variables globales définies dans le fichier `PicVariables`. Les fichiers plus haut niveau du code commun ont permis de designer la machine a états au départ utilisée dans la solution avec les threads puis réutilisée dans la nouvelle solution. Ici le fonctionnement est un peu plus complexe qu'avec l'I²C. Pour faire une écriture sur un composant, on commence par démarrer la communication puis on sélectionne le composant auquel on souhaite s'adresser. Ensuite, on envoi un code qui va permettre de choisir une action à réaliser. Parmi les actions possibles, il y a la lecture, l'écriture ou encore la possibilité de supprimer des données (erase). Il y a aussi un code permettant d'activer les droits d'écriture sur un composant. Il faut donc activer l'écriture à chaque fois que l'on souhaite modifier les données sur le stockage. Par exemple, pour faire une écriture sur un composant, on commence par sélectionner la cible. Ensuite, on envoi le code permettant d'activer l'écriture. Une fois fait, il faut redémarrer la communication puis envoyer le code permettant d'écrire. Avant de pouvoir envoyer les données à écrire, il faut déplacer le curseur du composant en envoyant une adresse sur deux où trois octets, les données serons ensuite écrites à partir de la nouvelle adresse du curseur. Lorsque l'on a fini d'envoyer les données, on arrête la communication en repassant $\overline{CS}$ à 1.

Comme pour lI²C, il y a une fonction `Control` qui est une machine à états qui reprend les étapes décrites dans le paragraphe précédent. Là aussi, pour utiliser la simulation, il faut modifier la fonction d'initialisation en ajoutant les composant de façon similaire à l'I²C. À noter que la simulation de la mémoire ne change pas et que l'on peut très bien configurer le même composant à la fois sur lI²C et le SPI.

5.4 Système d'évènements

À la fin de la conception de la simulation des interfaces I²C et SPI, il restait un problème important. Ce problème était que l'on ne pouvait pas savoir depuis les machines à états si les protocoles étaient utilisés correctement. Par exemple, si un utilisateur oubliait de redémarrer la communication après le déplacement du curseur pour une lecture I²C, il pouvait être très compliqué de trouver l'origine du problème. Cela est du à la rigidité des machines à états pour la simulation des interfaces de ces deux protocoles. Si les étapes ne sont pas respectées à la lettre et que les bonnes fonctions ne sont pas appelées dans le bon ordre le résultat est indéterminé. C'est un énorme problème étant donnée le fait que l'on ne sait plus d'où viennent les erreurs, du code ou de la simulation ? C'est une question qu'il est légitime de se poser étant donné la complexité de la simulation des interfaces du stockage.

Pour tenter de résoudre ce problème, un système d'évènement a été pensé. Ce système est loin d'être parfait mais il peut éviter une partie des erreurs. Le principe est que les machines à états de l'I²C et du SPI enregistrent des évènements au fur et à mesure de l'exécution. Une fois que l'on a terminer une action, on peut vérifier si les évènements enregistrés sont bon. Par exemple, on peut essayer de faire un certain nombre de lectures avec le SPI. Le fichier du SPI fournit une fonction qui permet de générer la liste des évènements attendus pour n lectures. Il suffit donc de comparer la suite d'évènements attendus à celle des évènements enregistrés pour savoir s'il y a un problème. Cela nécessite donc d'avoir un cas de test dédié qui fait la vérification des évènements pour chacune des actions possibles.

TODO : détailler le système d'évènements
TODO : CONCLUSION sur le stockage

6 Protocoles monétiques

Nous avons vu dans la partie précédente que les périphériques de stockage étaient interfacés avec les protocoles I²C et SPI qui sont deux protocoles assez courant dans les systèmes embarqués. La société CKsquare est spécialisée dans la monétique et utilise donc beaucoup de périphériques interfacés avec des protocoles spécifiques à ce domaine. Une grosse partie du stage a été passée à la simulation d'interfaces utilisant deux protocoles très connus dans le milieux de la monétique, le Cctalk et le MDB. Dans cette section, nous allons présenter ces deux protocoles ainsi la solution proposée pour l'outil de test.

6.1 L'interface Cctalk

Une partie des composants utilisés sur la carte ont été interfacés avec le protocole Cctalk. Dans cette section, nous commencerons par détailler les grands principes de ce protocole puis nous verrons son implémentation par l'entreprise. Enfin, nous expliquerons comment les périphériques Cctalk ont été simulés.

6.1.1 Le protocole

Le Cctalk est un protocole utilisé dans la monétique dont la première version a été réalisée en 1996 par la société `Coin Controls` basée en Angleterre [7]. Ce protocole est un protocole série en halfduplex créé pour être utilisé pour des communications à débit moyen. En 2010 a été ajouté la possibilité de chiffrer les trames (standard DES). Ce protocole est destiné à être utilisé pour assurer la communication entre différents composants reliés à une même carte.

Comme le SPI et l'I²C , le Cctalk fonctionne en mode maître et esclave. Ce standard utilise un système de d'entête pour transmettre des commandes, par exemple, l'entête 253 permet de demander son

adresse à un composant. Il y a en tout 256 entêtes différentes, numérotées de 0 à 255 ce qui permet la transmission d'une entête sur un octet. Les entêtes de 20 à 100 sont réservées pour les développeurs des applications utilisant le Cctalk, le reste est décrit dans la documentation [8]. Sur la figure 9 on peut voir la composition d'une trame Cctalk qui comprend six éléments. Le premier est l'adresse du composant ciblé par la trame (l'adresse du destinataire). Cette adresse est suivit du nombre d'octets de données qui vont être envoyées. Ce nombre étant transmis sur un octet, on ne pourra jamais transmettre plus de 255 octets de données. Ensuite, on envoi l'adresse de la source (le composant qui a envoyé la trame) puis l'entête. Une fois l'entête envoyée, on transmet les données puis le checksum qui va permettre de vérifier l'intégrité des données à la réception.

FIGURE 9 – Trame Cctalk

Prenons l'exemple de l'écran créé par la société. Ici, on souhaite afficher du texte sur l'écran, pour cela, on va utiliser l'entête 203 qui permet de contrôler ce composant [8]. Pour sélectionner l'action à réaliser, il faut utiliser une sous-entête. Dans le cas de l'écran, il y a cinq sous-entêtes permettant de réaliser des actions spécifiques à ce composant comme effacer l'affichage par exemple. Dans notre exemple, on souhaite envoyer du texte à afficher, pour ce faire, il faut utiliser la sous-entête 3. Ici, on va donc envoyer une trame dont l'entête sera 203 et les données seront la sous-entête 3 suivie du texte à afficher. L'écran doit normalement répondre par un acquittement (entête 0).

Une exemple plus complexe est celui du système vocal. Ici, on va demander au système vocal de lire un fichier audio. Là, il n'y a pas d'entête dédiée, la société à donc utilisé une des entêtes mise à disposition des développeur : l'entête 34. Cette entête ne sert que pour réaliser cette action, il n'y a donc pas de sous-entête à envoyer. La trame qui sera transmise au système vocal sera donc composée de l'entête 34 et des données requises à la lecture du fichier comme le nom du fichier à lire, le nombre de répétitions, ... Là aussi, le composant doit répondre par acquittement.

Maintenant que nous avons expliqué le fonctionnement général du protocole, nous allons nous intéresser au fonctionnement du code de l'entreprise.

6.1.2 Le code de l'entreprise

Dans le code de l'entreprise, le Cctalk est utilisé avec beaucoup de composants comme un accepteur de billets, un lecteur de cartes, mais aussi le système vocal de la borne ou encore l'écran tactile. Il y a un fichier pour chacun des composants interfacé en Cctalk et ces fichiers permettent de gérer la partie maître. Comme pour le stockage, la simulation a au départ été pensée pour se situer au plus proche des fichiers du compilateur. Cela a nécessiter une compréhension de toute la chaine d'instruction qui permet la transmission des trames.

Pour l'implémentation du Cctalk, il y a deux fichier principaux. Tout d'abord, il y a le fichier `CctalkMaster.c` qui permet de gérer la structure de données qui modélise les commandes. Ce fichier possède aussi une fonction `Control` qui permet de gérer les *Core Commands*, qui sont des commandes généralement exécutées à l'initialisation et qui permettent d'obtenir des informations sur les composant (constructeur, numéro de version, ...). Ensuite, il y a le fichier `CctalkMasterSerial.c` qui gère la communication. Ce second fichier sert de pont entre le code haut niveau des communs et le code bas niveau des

bibliothèques du PIC. Pour l'envoi des trames, le fichier `CctalkMasterSerial.c` utilise deux machines à états, une *haut niveau* et une autre *bas niveau*. La machine à états *bas niveau* communique directement avec la bibliothèque de la carte, elle se charge de la transmission et de la réception des données (envoi d'une trame et réception de la réponse). La machine à états *haut niveau* va permettre l'envoi de différents types de trames. En effet, les trames Cctalk transmises par la machine *bas niveau* sont limités à 255 octets de données. Pour palier à ce problème, la machine à états *haut niveau* permet d'envoyer des trames étendues. Le principe est de décomposer la trame à envoyer en plusieurs sous trames. Pour que le composant sache que la trame est une trame étendue, il faut envoyer deux trames supplémentaires avec une entête de début et une entête de fin. L'envoi d'une commande se fait donc de la façon suivante : on commence par appeler la fonction d'envoi de commande qui va ajouter la nouvelle commande à une liste chaînée. Ensuite, le `Control Manager` va appeler la machine à états *haut niveau* qui va utiliser l'autre machine à état pour envoyer une ou plusieurs trames en fonction du type de trame à transmettre (simple ou étendue).

L'implémentation par CKsquare du Cctalk propose beaucoup d'autres fonctionnalités qui n'ont pas été traitées lors du stage. Par exemple, pour limiter le *polling* (interrogation d'un composant) sur certains composants comme l'écran tactile, les développeurs ont mis en place un système permettant à la carte de passer en mode esclave. L'interface du Cctalk est aussi utilisée avec le TCPIP. En effet, le Cctalk est fait pour assurer la communication entre les composants reliés à une même carte. Il peut arriver que l'on ai besoin de lier des composants qui ne sont pas directement connectés et pour ce faire on utilise le protocole TCPIP qui est le plus adapté. Le code de la société propose une abstraction qui permet de gérer tous les composants en utilisant seulement l'interface du Cctalk. Pour rendre cela possible, le code utilise des drivers qui fonctionnent à l'aide de pointeurs de fonctions.

6.1.3 Émulation des composants

Comme expliqué dans la partie précédente, le répertoire `Cctalk` contient les fichiers qui permettent de gérer la partie maître pour chacun des composants interfacés en Cctalk. Pour cette simulation, il fallait mettre en place du code permettant de simuler le comportement des esclaves (donc des composants) lors de la réception de trame. L'objectif des composants simulés est simplement de répondre aux commandes Cctalk qu'ils reçoivent. Ces commandes peuvent être automatiques ou paramétrées dans les tests. Un effet de bord souhaitable pour la simulation est aussi la récupération de la trame reçue, de sorte à ce que l'on puisse vérifier que les bonnes commandes sont envoyées dans les tests.

Pour simuler les composants, il a été décidé d'utiliser un fichier par composant. Lors du développement, il a été jugé plus simple et plus sage d'adapter les principes utilisés lors de la simulation du stockage. Cette décision a pour objectif d'apporter un maximum de cohérence dans le fonctionnement de l'outil, cela pourra faciliter la maintenance ainsi que les modifications. De plus, les tests réalisés sur le stockage ont prouvé le bon fonctionnement de la solution. Enfin, pour faciliter la copie, les fichiers de la simulation adoptent une forme standard que nous détaillerons plus loin. La documentation met même à disposition un patron qu'il suffit de copier et de modifier pour simuler un nouveau composant.

Comme pour le stockage, la simulation devait se situer au plus proche des bibliothèques du compilateur. Cette approche a permis d'avoir une simulation très réaliste qui pousse à l'utilisation du tous le code créé pour gérer le Cctalk. C'est la première solution qui a été mise en place et qui a nécessité la compréhension de toute l'implémentation du protocole. Bien qu'il soit intéressant d'avoir une simulation réaliste permettant de faire des tests d'intégration qui valident toute la chaine d'instruction, cette solution était relativement complexe et poussait les tests à dépendre de l'entièreté du code. Cela n'est pas souhaitable dans le cas où l'on veut faire des tests unitaires sur un fichier sans se préoccuper du reste de l'implémentation. Cela est intéressant car il permet de mieux repérer l'origine des problèmes car le code

testé est isolé, de ce fait, les échecs des tests sont dus uniquement aux erreurs dans le fichier testé et non aux erreurs dans le reste du code. C'est pour cette raison qu'une deuxième solution à été mise en place, permettant ainsi de faire à la fois des tests unitaires et des tests d'intégrations. Pour mettre en place cette solution, une version modifiée du fichier `CctalkMasterSerial.c` a été crée, dans cette version, les machines à états sont supprimées. La sauvegarde de la trame reçue ainsi que l'envoi de la réponse du composant est géré directement depuis la fonction qui permet d'envoyer les commandes. De ce fait, quand un élément du code testé envoie une commande Cctalk, cette dernière est enregistrée et la réponse est directement envoyée. On ne passe donc plus par les machines à états, ni par les fichier du compilateur, le code testé est donc bien isolé. Il est possible de passer d'une implémentation de la simulation à une autre en définissant une constante préprocesseur. À noter que c'est deux solutions ont été pensées pour être interchangeables, de sorte à ce que les mêmes tests fonctionnent peu importe l'implémentation de la simulation.

Comme nous l'avons mentionné dans un précédent paragraphe, tous les fichiers de la simulation ont la même forme. Après plusieurs tests, une *forme standard* pour la simulation a été crée. Chaque fichier se compose des trois fonctions suivantes :

- `Control` : Cette fonctions est utilisée avec la simulation bas niveau qui est la première solution crée. C'est une machine qui se compose de trois états. Dans le premier état, la machine fait la vérification de l'adresse sur la commande envoyée sur le bus par le code testé. Si cette adresse ne correspond pas à l'adresse du composant simulé, la machine s'arrête. Dans le cas contraire, la machine commence par récupérer la trame sur le bus pour que cette dernière soit accessible depuis les tests. Ensuite, elle envoi un écho qui permet de spécifier au maître que la trame a bien été reçu. Enfin, la machine à état envoi la réponse au maître en fonctions des données reçues, en utilisant les fonction du compilateur.
- `Run` : Cette fonction permet de gérer la fonction décrite dans le point précédent. Elle fait une suite d'appels à la machine à états haut niveau de `CctalkMasterSerial.c` pour déclencher l'envoi d'une commande avec les fonctions des bibliothèques du compilateur. Ensuite, elle fait appel à la fonction `Control` qui récupère la commande du maître et envoie l'écho et la réponse. Enfin, elle fait une autre suite d'appels à la machine haut niveau pour enclencher la récupération de la réponse. La fonction `Run` est à utiliser dans les tests à chaque fois qu'une commande doit être envoyée, elle permet gérer la simulation d'une communication entre la carte et un composant. À noter que cette fonction n'est utile qu'avec la version bas niveau (version où l'on test tout le code), cependant, il existe une version vide de cette fonction qui permet de ne pas avoir à modifier les tests quand on passe de la version bas niveau à la version haut niveau.
- `HandleResponse` : Cette dernière fonction est utilisée avec la version haut niveau. Comme expliqué précédemment, dans cette version, tout est géré depuis la fonction `CmdSnd` de la version modifiée pour les tests du fichier `CctalkMasterSerial.c`. Ici, il n'y a qu'un seul fichier pour tout les composants ce qui pose problème pour les réponses prés définies. Pour pouvoir paramétrer des réponses pour plusieurs composants, on passe par un pointeur de fonction qui pointe sur la fonction `HandleResponse` du composant testé. Cette fonction se compose juste d'un `switch` qui va envoyer une réponse en fonction des données reçues. À l'initialisation de chaque test, il faut affecter l'adresse d'une fonction `HandleResponse` à la variable globale `HANDLE_RESPONSE`.

TODO : code en annexe

6.1.4 Exemple de test

TODO : conclusion sur le cctalk

6.2 L'interface MDB

Dans la partie précédente nous avons traité le fonctionnement du protocole Cctalk et nous avons aussi vu comment les composants Cctalk ont été simuler pour faire fonctionner les tests. Dans cette partie, nous allons traiter un autre protocole utilisé avec d'autre composants, le protocole MDB. À noter que le fonctionnement du MDB dans le code de CKsquare ainsi que celui des composants simulés est très similaire au Cctalk, il y aura donc beaucoup de parallèles fait avec la partie précédente.

6.2.1 Le protocole

Le protocole MDB a été créé dans les années 1980 par la société *CoinCo* et a au départ été très utilisé dans les distributeurs automatiques de *Coca-Cola* [9]. Le protocole est devenu *open-source* en 1992 et la *National Automatic Merchandising Association* (NAMA) a réalisé la première version du standard en 1995.

Comme le Cctalk, le protocole MDB fonctionne en mode maître et esclave où la carte électronique est le maître qui contrôle les périphériques qui eux sont les esclaves. Pour que le maître puisse transmettre des ordres aux esclaves, le MDB utilise un système de commandes et de sous-commandes.

Comme le précise la documentation du protocole [10], la transmission des informations avec le MDB se fait sur neuf bits. Parmi ces neuf bits, il y a huit bits qui permettent de transmettre des données et le dernier bit et un bit de mode. Lorsque le bit de mode est à 0, les huit premiers bits contiennent des données. S'il est à 1, alors les huit premiers bits peuvent contenir l'adresse d'un composant cible (l'adresse du composant auquel s'adresse le maître), une commande spéciale ou un checksum. Les commandes spéciales sont ACK (acquittement), NAK (non acquittement) et RET (demande de répétition). Un octet d'adresse contient généralement aussi une commande : un **ou** logique est fait entre l'adresse du composant et la commande à envoyer. À noter que le standard impose des adresses spécifiques pour chaque type de périphérique. Par exemple, un accepteur de billet doit avoir pour adresse `0x30`. Les commandes envoyées avec l'adresse permettent de donner des ordres aux périphériques et chaque type de périphérique possède son propre jeux de commandes. Les commandes correspondent à des catégories assez génériques, par exemple pour l'accepteur de billet, `0x03` permet d'interroger le composant (pour savoir s'il y a un billet a été accepté par exemple). Ensuite, pour spécifier une action à réaliser on envoi à nouveau neuf bits avec le bit de mode à 0 et une sous-commande dans les données. On peut envoyer plus de données si besoin mais les sous-commandes doivent toujours apparaitre au début.

Traitons l'exemple du début de la validation d'une vente simple avec un périphérique de paiement sans contactes. Ce scénario est décrit dans la documentation [10, p. 169]. Ici, la carte va commencer par interroger le composant en utilisant la commande *poll* (code `0x02`). Les systèmes de paiement sans contactes ont pour adresse `0x10` ou `0x60`, ici, la première trame sera donc composée de neuf bits. Le bit de mode sera à 1 car on souhaite interroger un composant, et pour ce faire il faut envoyer son adresse sur le bus. La valeur des huit bits suivant est le résultat d'un ou logique entre la commande `0x02` et l'adresse du composant donc `0x12` ou `0x62`. Ensuite, le composant va renvoyer neuf bits où le bit de mode sera à 0 et la huit bits suivant contiendront la commande `0x03`, ce qui permet de démarrer une session (échange de trames). Une fois la session démarrée, la carte va demander s'il y a une ventre à traiter, pour ce faire, on utilise la commande *Vend* (`0x03`) et cette dernière s'accompagne d'une sous-commande *Vend Request* (`0x00`). On peut noter ici que pour un même code, la commande correspondante sera différente en fonction de l'émetteur de la trame (maître ou esclave). La nouvelle trame sera donc composée de 18 bits : `0x113` (bit de mode à 1, adresse du composant `0x10` et commande `0x03`) `0x000` (bit de mode à 0 et sous-commande `0x00`). Ensuite le périphérique doit répondre par la commande *ACK*, donc ici, le bit de mode est à 1 et les huit bits suivants sont nuls. Comme mentionné plus haut, le reste du scénario est décrit dans la documentation du MDB.

Maintenant que nous avons décrit les bases du protocole, intéressons nous au code de l'entreprise. On pourra ensuite étudier le fonctionnement de la simulation des composants.

6.2.2 Le code de l'entreprise

L'implémentation du MDB suis le même principe que celle du Cctalk sauf qu'ici, il n'y a pas plusieurs types de trames à gérer, le code est donc plus simple. Le protocole est géré par le fichier `MDBApi.c` qui contient la fonction `CmdSnd` qui permet d'enregistrer les commandes à envoyer. Ce fichier contient aussi une fonction `Control` qui correspond à la machine à état principale. Comme pour le Cctalk, la fonction `Control` interagir avec les fonctions du fichier `serial.c` qui fait partie des bibliothèques du compilateur. La fonction commence dans un premier temps par envoyer les différentes parties de la trame (adresse, données puis checksum), puis attend que la réponse soit disponible.

Lorsque l'on souhaite envoyer une trame, on utilise la fonction `CmdSnd`, qui va marquer la commande qu'elle reçoit en paramètre comme étant prête à être envoyée. Il n'y a pas de liste chaînées ici, toutes les commandes sont stockées dans un tableau et elles sont affectée aux différents composants lorsqu'ils sont enregistré (a l'initialisation du programme). Toutes les commandes sont accessibles depuis le fichier `MDBApi.c` et elles sont simplement marquées pour être envoyées. Lorsque le `ControManager` appelle la fonction `Control` du fichier, cette dernière transmet toutes les commandes marquées et récupère toutes les commandes.

6.2.3 Émulation des composants

La simulation des composants interfacés en MDB a été réalisée dans la continuité de ce qui a été fait pour le Cctalk. Là aussi, on a un fichier par composant, et les fichiers ont une forme similaire.

Comme le Cctalk, la simulation MDB est disponible en deux versions, une *haut niveau* qui utilise une version modifiée du fichier `MDBApi.c` et une version *bas niveau* qui passe par toute la chaine d'instructions jusqu'aux bibliothèques du compilateur. La grosse différence avec le Cctalk en terme d'utilisation est que la simulation ne propose pas de réponses prés-définies aux commandes envoyées par la carte. En effet, comme il peut y avoir beaucoup de réponses différentes pour une même commande, il a été décidé que les réponses des composants devaient être configurées à chaque fois. Cela signifie que dans les tests, il faut configurer une réponse avant l'envoi d'une commande. On a donc une grande maitrise depuis les tests du fait qu'il est très simple de tester toutes les possibilités dans plusieurs tests. La forme des fichiers des composants simulés est aussi similaire à celle des fichiers du Cctalk. Il y a une fonction `Control` qui est utilisée avec la version *bas niveau* et qui a la même utilité que la fonction du Cctalk. Là aussi, il y a une fonction `Run` qui comme pour le Cctalk permet de lancer la simulation du composant depuis les tests lorsque la version *bas niveau* est utilisée. Comme il n'y a pas de réponse prés-définies, il n'y a pas de fonction `HandleResponse`. L'obligation de configurer toutes les réponses aux commandes rend certains tests complexe à écrire, c'est pour cela qu'un système de scénarios a été ajouté au MDB, ce système est décrit dans la section 6.2.4.

TODO : code en annexe

6.2.4 Système de scénarios

Comme dit précédemment, les réponses aux commandes étant trop complexes, la simulation des composants MDB ne propose pas de moyen d'avoir des réponses prés définies comme avec le Cctalk. Par contre, ici, la documentation fournit des scénarios qui décrivent des échanges de trames entre le maître et les composants. Pour pouvoir vérifier si ces échanges de trames sont possibles avec le code de l'entreprise, un système de scénarios a été mis en place. Ce système permet de définir une suite d'envoi de commande par le maître et de réponse par l'esclave. À chaque fois que le composant simulé reçoit une trame, il

vérifie que cette dernière correspond à celle prévue par le scénarios avant de transmettre la réponse. Si la trame reçue par l'esclave ne correspond pas à ce qui été prévue par le scénario, on fait échouer le test avec un `exit(1)`. À noter que seul le test sur le scénario joué échoue et les tests suivants sont exécutés correctement car avec notre framework, chaque test s'exécute dans son propre thread. Cela simplifie grandement l'écriture des tests puisqu'il n'y a pas besoin de paramétrer les réponses du composant ni même de faire des assertions puisque si le test passe c'est que le scénarios a été joué correctement, sinon, c'est qu'il a planté. Ce système de scénario est donc un moyen très simple de valider le code CKsquare pour un type de périphérique.

L'implémentation de ce système est très simple, les scénarios sont composés d'étapes et chaque étape est modélisée par une structure. Les étapes sont stockées dans un tableau global. Lorsque l'on souhaite jouer un scénario, on appel la fonction *play* correspondante au début du test. Cette dernière fait un enregistrement d'étapes dans le tableau global. Ensuite, l'envoi réponses et la vérification des commande envoyées par la carte est automatique, il suffit juste d'appeler les fonctions du code pour passer par toutes les étapes du scénario. À noter que la plupart du temps, il faut initialiser les machines à états au préalable car les étapes de démarrage ne sont pas prises en comptes dans les scénarios.

TODO : exemple de scénario et de test
TODO : code en annexe

6.2.5 Exemple de test

7 Les historiques

Dans cette section, nous allons nous intéresser à la gestion des historiques (historique de paiement, ...). Ils sont stokés en mémoire sur la carte dans une base de données circulaire et ils sont régulièrement sauvegardé sur un serveur. C'est un élément important car la sauvegarde des historiques est juridiquement obligatoire quand ils concernent la monétique (loi finance de 2016). Dans cette partie, nous commencerons, dans un premier temps, par détailler le fonctionnement de la structure de données et des tests qui ont été fait dessus. Dans un second temps, nous traiterons l'envoi des historiques sur le serveur de CKsquare.

7.1 Les CDBs

Tous les historiques sont stockés dans un base de donnée circulaire (CDB signifie *Circular Data Base*). La première partie des tests concernant les historiques va porter sur la validation du bon fonctionnement de cette base. Un point important concernant cette structure est que les CDB sont stockées en mémoire (généralement sur des eeproms). Les tests qui vont être réalisés sur cette structure vont permettre de pleinement utiliser l'émulation du stockage. Ils vont donc constituer un exemple intéressant qui pourra être décrit dans la documentation.

Comme mentionné dans le paragraphe précédent, un CDB est une une base de donnée circulaire. Un CDB est une structure qui se compose d'un tableau, d'un compteur d'éléments, d'une tête et d'une queue. La tête correspond à la première donnée insérée et la queue à la dernière. Les CDB sont sauvegardés en mémoire sur des eeproms ou des flashs en fonction de la configuration. Il y a une seule interface (fichier d'entête) pour deux implémentation différentes utilisant les deux types de périphériques de stockage et on peut passer d'une implémentation à une autre avec la configuration. Il est important de pouvoir tester les deux implémentations, par contre, étant donné que les fonctions ont le même nom, on ne peut pas compiler tous les fichiers en même temps (une interface et deux implémentations). C'est pour cette raison que l'on génère deux exécutables différents, où chaque exécutable est compilé avec une configuration

différente. Comme expliqué dans la section 3.3, pour passer d'une configuration à une autre, on utilise des constantes préprocesseurs définies à la compilation. Il y aura donc aussi deux tests différents, un pour les flash et un pour les eeproms.

Traitons à présent un exemple de test sur un CDB dont le code est visible sur le listing 1. Ici, on test l'insertion d'éléments dans une flash. Pour réaliser ce test, on utilise plusieurs fonctions annexes. La fonction `TESTCDB_IndexToAddrRelative` prend en paramètre un CDB, un indice et une référence sur une adresse. Elle permet de récupérer l'adresse de l'élément du CDB à l'indice donnée dans la flash. Pour ce test, on utilise des historiques de paiement et on utilise la fonction `TESTCDB_RandomBufferGenerate` pour générer un élément aléatoire à insérer dans le CDB. Dans un premier temps, on insère un élément dans la base et on regarde si les données ont bien été écrites dans la flash. Ensuite, on vérifie que les données du CDB sont aussi écrites au début de la mémoire. En effet, lorsque l'on redémarre la carte, on a besoin de récupérer l'état de la base de données (le nombre d'élément stockés, ...). Pour pouvoir faire cela, la structure qui permet de gérer le CDB est écrite en début de mémoire (`CDB->Control`). Une fois que l'on vérifié qu'une insertion dans le CDB fonctionnait, on teste s'il est possible de remplir la base en ajoutant le maximum d'éléments à l'aide d'une boucle. À noter qu'à chaque fois, on vérifie que l'élément (toujours généré aléatoirement) est bien écrit dans la mémoire. À la fin, on test que le CDB est bien plein.

```
Test(CDB_Test, TESTCDB_Add, .init = TESTCDB_Setup, .fini = TESTCDB_Teardown)
{
    uchar Buff[sizeof(TITEM_PAYMENT)];
    ulong BeginingAddr = ((TCDBONFLASH_CONTROL * )CDB->StorageData)->NextAddr + 1;
    TFLASH_ADDR Addr;
    int Cpts;

    TESTCDB_IndexToAddrRelative(CDB,0,&Addr);
    TESTCDB_RandomBufferGenerate(Buff,sizeof(TITEM_PAYMENT));
    cr_assert(true == CDB_Add(CDB ,Buff));
    // le cdb a bien été écrit dans la flash
    cr_assert(eq(u8[sizeof(TITEM_PAYMENT)],TESTMEM_FLASHM[0] + Addr,Buff));
    // on vérifie que le control est bien écrit au début
    CDB->Control.CounterChange--;
    cr_assert(eq(u8[sizeof(CDB->Control)],
                 TESTMEM_FLASHM[0] + BeginingAddr,
                 (uchar*) &CDB->Control));
    CDB->Control.CounterChange++;

    /* remplissage de la base */
    for (Cpts = 1; Cpts < CDB->RecordMax; ++Cpts) {
        TESTCDB_RandomBufferGenerate(Buff,sizeof(TITEM_PAYMENT));
        cr_assert(true == CDB_Add(CDB ,Buff));
        TESTCDB_IndexToAddrRelative(CDB,Cpts,&Addr);
        cr_assert(eq(u8[sizeof(TITEM_PAYMENT)],TESTMEM_FLASHM[0] + Addr,Buff));
    }
    cr_assert(true == CDB_IsFull(CDB));
}
```

Listing 1 – Test d'insertion dans un CDB

Ce test permet de valider l'insertion d'éléments dans un CDB et il y a d'autres tests qui permettent de valider d'autres fonctionnalités ces bases de données comme la suppression par exemple. Ici, on peut voir que la simulation des périphériques de stockage fonctionne bien et que cette dernière est assez simple à utiliser dans les tests. On voit bien que l'on peut aisément vérifier si la mémoire a été modifiée en

utilisant les outils du framework. Ce sont des résultats positifs qui montrent la viabilité de l'outil de test.

Maintenant que nous avons abordé le fonctionnement des CDBs ainsi que les tests qui sont associés à cette structure de données, nous allons nous intéresser au test de la sauvegarde des historiques sur le serveur.

7.2 Sauvegarde des historiques

Dans la section précédente, nous avons traité l'utilité et le fonctionnement des CDB, étudions le problème de l'envoi des historiques sur le serveur. À noter que c'est un problème concernant l'envoi des historiques de paiement sur le *CKWash* (serveur de CKsquare) qui a donné aux développeurs l'idée de ce stage.

Pour envoyer les historiques sur le serveur, le `ControlManager` va régulièrement faire appel à une fonction `Control` qui va envoyer tous les nouveau historiques sur le *CKWash* et les supprimer des CDB. Cette fonction se trouve dans le fichier `CksproHisSend.c` Pour transmettre les historiques, on utilise un protocole créé par la société qui est le protocole **CKspro**. Ici, seule une version haut niveau de la simulation à été mise en place car le code qui permet de gérer la transmission des données utilise directement des éléments bas niveau comme le TCPIP. Ces éléments étant relativement complexes et assez peu utilisés dans le code, ils n'ont pas été traités durant le stage car cela aurait pris trop de temps.

Pour pouvoir tester l'envoi des commandes, une version modifiée du fichier chargé de la gestion du protocole CKspro a été créée. Comme pour le Cctalk et le Mdb, la fonction `Control` du fichier n'est pas utilisée et la fonction `CmdSnd` a été modifiée. Cette dernière récupère les commandes envoyées et les sauvegarde dans une variable globale accessible depuis les tests. Pour tester si les commandes sont bien envoyées, on place des éléments dans un CDB et on appelle la fonction `Control` du fichier `CksproHisSend.c` qui va envoyer tous les éléments enregistrés dans la base. On peut ensuite tester si les commandes envoyées correspondent aux données qui avaient été enregistrées en vérifiant que les identifiants stockés dans les trames sont les mêmes que ceux des éléments dans CDB. À noter qu'ici, un test assez réaliste a été écrit pour vérifier que l'outil permet bien de créer un test capable de détecter le problème qui a pousser la société à créer ce sujet de stage. Le problème été survenu plusieurs fois à cause de mauvaises manipulations de git, ce qui aurait pu être détecté facilement avec une pipeline. Il venait d'une erreur de configuration qui faisait que dans la fonction `Control` de l'envoi des historiques, une fonction était appelée à la place d'une autre. Ce problème était assez difficile à détecter avec de simples tests puisqu'il apparaissait uniquement lorsque le CDB était remplit plusieurs fois (problème d'index dans le CDB). Pour détecter ce problème, on remplit le CDB, on envoie tous les historiques et on refait ces opérations en vérifiant à chaque fois que tous les éléments sont envoyés. Le test fonctionne bien et sont code est disponible à l'annexe **??**.

Les tests sur l'envoi des historiques sont relativement complexes dans le sens où ils doivent utiliser plusieurs des éléments mis en place par l'outil de test. En effet, ici, le test doit utiliser la version modifiée du fichier `CKspro.c` et les modifications apportées à ce fichier suivent les mêmes principes que pour le Cctalk et le MDB. En plus de la modification de fichier pour isoler les éléments du code testé, les tests sur l'envoi d'historiques utilisent les CDB et donc la simulation du stockage. Le fait d'avoir un résultat positif sur un test qui utilise une grande partie du code et autant des éléments mis en place par l'outil est très positif. Cela permet donc de valider en grande partie les techniques et méthodes employées durant le stage. Enfin, un autre point intéressant concerne la difficulté d'écriture des tests. En effet, les tests ne sont pas triviaux à écrire, cependant, leur complexité vient plus de celle du code testé que de l'outil.

L'envoi des historiques était un élément important à tester et c'est le dernier élément qui a été traité

avant de mettre en place la pipeline. Dans la section suivante, nous traiterons la configuration de la pipeline pour que les tests soient exécutés de façon automatique à chaque nouvel envoi de code sur Gitlab.

8 Mise en place de la pipeline

Dans cette section nous allons voir comment a été mise en place la pipeline Gitlab.

La configuration de la pipeline contient deux *jobs*, un *job* de compilation et un *job* de test. Il y a une contrainte de précédence entre la compilation et les tests.

Pour la compilation, on commence par installer le nécessaire pour compiler avec le gestionnaire de paquets de la pipeline (*Aptitude*). Cela comprend les outils comme `gcc` et `cmake` mais aussi le framework de test. Ici, on utilise l'image docker par défaut de Gitlab (`ruby:3.1`) car il n'y a pas beaucoup de paquets à installer mais il serait possible d'utiliser une image plus complète ou d'en créer une. À noter que cela se fait dans une section à part (`before_script`), ce qui permet de ne pas copier un partie de la configuration pour tous les *jobs* et permet aussi de spécifier à Gitlab de garder les éléments en cache pour ne pas tout télécharger à chaque fois. Le projet `dev_pic` est dépendant du projet `commun_global` qui comme expliqué en début de rapport correspond au code partagé entre tous les projets. Pour accéder à ce projet depuis la pipeline, il y a deux solution, la première consiste à utiliser un sous module git. Le défaut de cette méthode est qu'elle nécessite la gestion d'un sous module dans les communs. De plus, cette solution n'est pas très logique et complexifierait énormément la gestion du projet car `dev_pic` est déjà un sous module des projet de l'entreprise. La seconde méthode est beaucoup plus simple puisqu'elle consiste juste en le clonage du projet `commun_global` dans la pipeline. Pour pouvoir cloner un projet privé, on utilise un *job token* accessible depuis la configuration à l'aide d'une variable. Une fois que l'on a téléchargé toutes les dépendances du projet, on met en place CMake et on compile les tests. Après la compilation, les exécutables se trouvent dans le répertoire `buil/bin`, pour pouvoir y accéder depuis l'autre *job*, on déclare le répertoire `build` comme artefact ce qui permettra de le réutiliser. On peut voir la configuration sur le listing 2.

```
before_script:
  - apt-get update
  - apt install -y libcriterion-dev cmake gcc

build-job:       # This job runs in the build stage, which runs first.
  stage: build
  script:
    - cd commun/Test/
    - git clone https://gitlab-ci-token:${CI_JOB_TOKEN}@git.cksquare.fr/cklibs/commun_global.git
    - mkdir build
    - cmake -S . -B build
    - cd build
    - make
  artifacts:
    paths:
      - commun/Test/build
```

Listing 2 – .gitlab-ci.yml : build job

Dans le second *job*, on va lancer les tests. On se rend dans le répertoire de build (qui était sauvegardé dans un artefact) et on exécute les les tests en utilisant la commande `ctest`. `CTest` est un exécutable installé avec CMake, il permet de gérer les tests de façon automatique. On peut voir sur le listing 3 que

CTest est utilisé avec plusieurs paramètres. Le premier permet de faire afficher les erreurs générées par le framework de test car avec l'affichage par défaut, on ne peut pas savoir quel test a échoué et pourquoi. Ensuite, on utilise l'option `-T` pour spécifier deux actions à réaliser. Tout d'abord on souhaite que CTest exécute les tests puis ensuite on lui demande de calculer le pourcentage de couverture des tests. Ceci se fait à l'aide d'un outil installer par défaut avec gcc : `gcov`. Cet outil permet de savoir le pourcentage de lignes de codes compilées qui sont exécutées par les tests. Cela permet d'avoir une idée de la qualité des tests car il est intéressant de maximiser la couverture des tests pour être sûr que tout le code est testé.

```
launch-test-job:   # This job runs in the test stage.
  stage: test    # It only starts when the job in the build stage completes successfully.
  script:
    - cd commun/Test/build
    - ctest --output-on-failure -T Test -T Coverage
  needs:
    - job: build-job
      artifacts: true
  coverage: '/Percentage Coverage: \d+\.\d+/'
```

Listing 3 – .gitlab-ci.yml : test job

La création de cette pipeline a clos la première partie du stage qui concernait la création d'un outil de test sur la branche principale du code C. Le travail suivant a consisté en l'adaptation de l'outil sur la branche évènementiel qui est l'autre branche du projet `dev_pic` que nous avons détaillé dans la section 2.1.1. Dans la partie suivante nous allons nous intéresser aux modifications apportées à l'outil pour que ce dernier fonctionne sur la seconde branche.

9 Les tests sur la branche de l'événementiel

Dans cette section, nous allons traiter l'intégration de l'outil de test dans la seconde branche. À noter que cette section sera courte étant donné le fait qu'il n'y a pas eu de nouvel élément créé pour l'événementiel et qu'il n'y a eu que très peu de modifications apportées au code de l'outil.

Compilation

Avant de commencer à intégrer l'outil de test, il a fallut mettre en place une première configuration de CMake capable de compiler une partie du code. L'objectif était de comprendre le fonctionnement du code et de compiler un maximum de fichier pour intégrer l'outil plus facilement. Cette tâche a pris beaucoup de temps car tous les fichiers présents ne pouvaient pas être compiler, or cela n'était mentionné nul part. Cependant, le fait de faire cela en amont a grandement facilité le reste du travail.

Lorsque les premiers fichiers de tests ont étaient ajoutés, un problème très intéressant a été rencontré. Comme expliqué dans la section 2.1.1, on utilise une fonction `Connect` pour connecter un évènement à une fonction `SwitchControl`.

TODO : détailler les paramètres de la fonction et expliquer le problème de l'évalutation des arguments avec gcc.

Le stockage

Le premier élément qui a été ajouté sur la nouvelle branche est le stockage. Étant donné que ce dernier est placé très bas niveau dans le code, il n'y a pas eu de modifications dessus. En effet, la partie évènementiel change le fonctionnement du code haut niveau mais les fichiers du compilateur eu ne sont

pas modifiés. Le stockage étant une partie assez importante de l'outil de test, le fait que tout fonctionne sans modifications a permis de gagner beaucoup de temps.

Cctalk et MDB

La partie qui a pris le plus de temps à adapté a été celle sur les protocoles monétiques. Cette partie a été complexe a adapté car le code a beaucoup changé entre les branches que ça soit pour les versions haut et bas niveau. Ici, les fonction `Run` et `Control` ont été adaptées pour pouvoir fonctionner avec le `Sate Manager`.

TODO : détailler les modif avec exemple de code

À noter qu'ici, il y a eu beaucoup moins d'éléments simulés que sur la branche principale. En effet, cette branche est secondaire et elle n'est pas nécessairement à jour car tous les composants ne sont pas utilisés. De ce fait, la plupart des composants Cctalk et MDB qui étaient présents sur l'autre branche ne l'étaient pas ici.

Les historiques

Comme dit dans la section 7.1, les historiques sont stockés dans la mémoire de la carte. Étant donné que le stockage n'a pas nécessité de modifications sur ce branche, toute la partie concernant les CDBs était presque fonctionnelle. Ici, des modifications ont été apportées au niveau du code de l'entreprise ainsi qu'au niveau des tests. En effet, la branche n'étant pas à jours, des éléments qui étaient utilisés sur la branche principale n'existaient pas, ils ont donc étaient supprimé. Dans le code de l'entreprise, il manquait aussi des conditions préprocesseurs qui étaient utilisaient sur l'autre branche pour compiler, ces dernières ont donc été rajoutées.

L'intégration de l'outil de test sur la branche événementiel a marqué la fin de la partie C du stage. Le travail a avancé plus vite que prévu et cela a permis d'aborder la partie C++. Dans la partie suivante, nous allons voir l'implémentation de l'outil de test en C++ ainsi que la mise en place de la pipeline sur le code commun Qt.

10 Le projet de test sur la partie Qt

Dans cette section, nous allons traiter l'implication de l'outil de test en C++. L'objectif est de pouvoir intégrer l'outil dans le code commun Qt.

10.1 Nouveau framework

- Premier framework compatible cpp pour rien - collaboration avec samuel qui trouvé un framework plus adapté à Qt

10.2 Le stockage

Une partie qui avait pris beaucoup de temps côté C était le stockage. Pour rappel, l'implémentation du stockage avait nécessiter la simulation des périphériques de stockage ainsi que des deux interface utilisée pour interagir avec ces composant. Pour la partie C++, il n'y a pas eu besoin de simulation car le stockage se fait à l'aide de simples fichier. En effet, les cartes électroniques utilisées avec Qt sont beaucoup plus haut niveau et intègrent un système d'exploitation (Linux) capable de gérer un système de fichier. Ici, on

utilise l'objet `QFile` pour écrire dans des fichiers au format binaire. Dans les tests, il suffit simplement d'aller lire dans les fichiers pour vérifier si les données sont bien stockées.

À noter qu'il peut être intéressant de ne pas utiliser l'objet `QFile` dans les tests mais de plutôt opter pour les outils de la bibliothèque standard. Cela a pour objectif de détecter les mauvaises manipulations des outils de Qt. En effet, comme c'est le développeur du code qui écrit les tests, si ce dernier manipule mal les outils de Qt, il y a de fortes chance pour que ça soit aussi le cas dans les tests. Pour éviter ce problème, on ne réutilise pas les même outils dans le code et dans les tests. De plus, cela pourra permettre la détection de problème avec Qt si jamais il y en a.

10.3 Les protocoles monétiques

- une seule version - mocking pour tester les fichiers bas niveaux

10.4 Le problème de l'encapsulation

- fonctions control privées et actionnées avec des évènements timer - solution avec des wait pas idéale - mocking pas toujours possible ou trop complexe - pas de solution pour controler les timers - solution du define pas utilisée mais envisagée - fonctionnement du Méta Object Compiler pas évident au départ

10.5 La mise en place de la pipeline

La configuration de la pipeline pour les commun Qt suis le même principe que celle du C. On a deux jobs, le premier permet de compiler les tests et le second les exécutes. Les tests sont toujours compilés avec CMake (qui utilise g++ au lieu de gcc) et exécutés avec CTest.

Bien que les deux configuration soient similaires, il y a tout de même une grande différence avec le C. En effet, ici, on doit compiler du code fonctionnant avec la bibliothèque Qt, cela nécessite donc de l'installer. Le problème est que Qt est une bibliothèque assez lourde, comportant un grand nombre de paquets. Le fait de devoir télécharger toutes les dépendance au début de chaque job comme on le faisait pour le C prend beaucoup plus de temps et consomme aussi beaucoup plus de bande passante. Pour économiser les ressources il est préférable d'utiliser une image docker déjà configurée et de lancer la pipeline dedans. Ce type d'images existes et certaines sont disponibles sur docker hub mais elles ne sont généralement pas complètes ou pas à jours. La solution employée déjà employée sur une autre pipeline Qt était `darkmattercoder/qt-build`. Le problème est que ce répertoire n'est plus maintenu et il n'était pas souhaitable d'utiliser une solution open-source plus mise à jour. Au final, il a été décidé de créer un nouveau projet se comportant d'une pipeline ayant pour rôle de compiler des images docker et de les envoyer sur le registre de Gitlab. Cette pipeline va créer une image docker avec une installation complète de Qt5 qui sera utilisée dans la pipeline du code commun Qt. Ce projet est très intéressant car il pourra permettre à l'entreprise de créer automatiquement d'autres images docker pour d'autres projets.

11 Déroulement du projet

Dans cette dernière section, nous allons nous intéresser aux méthodes de trivial et aux outils utilisés durant ce stage. Nous ferrons ensuite la comparaison entre le planning prévisionnel et le planning réel.

11.1 Les outils utilisés

Dans cette partie, nous allons faire une revue des outils utilisés pendant le stage. Les outils les plus utilisés ne seront pas détaillé, par contre, cette partie a pour but de mettre en lumière les avantages qu'il y a à connaitre et configurer ses outils de travail.

11.1.1 Gestion de version

Pour la réalisation du projet, le gestionnaire de version utilisé été Git. À noter que la société n'utilise Git que depuis un ans, le gestionnaire de version qu'ils utilisaient avant été SVN.

L'avantage de Git est qu'il est assez simple d'utilisation mais propose tout de même des fonctionnalités très complexes. De plus, par rapport à SVN, Git est décentralisé et permet de faire des branches ce qui a été très utile pour éviter que le projet de tests pose problème.

La philosophie de travail utilisé a consisté en une version réadapté de Gitflow. Le principe de Gitflow est d'avoir une branche `master` qui va contenir les versions du projet. Ensuite, il y a une branche `dev` qui est utilisée pendant le développement. La branche `dev` contient du code fonctionnel mais qui n'est pas encore déployé sur `master`. Entre les branches `master` et `dev` il doit normalement y avoir une branche `release` qui contient des prés-versions mais elle n'a pas été utile ici. Enfin, à partir de la branche `dev`, on créer des branches de `features` sur lesquelles on développe les nouvelles fonctionnalités. Étant donné que le travail se faisait seul, le fait d'utiliser Gitflow avec autant de branche n'était pas vraiment nécessaire. Cependant, cette méthode permet d'être très organisé. Cela permet par exemple de toujours avoir du code fonctionnel présentable à un collègue. De plus, cette organisation des branches permet de ne pas se perdre et de ne jamais caser du code fonctionnel. L'avantage des branches est que l'on peut facilement faire des tests pour savoir si une solution est réalisable.

À noter que l'équipe de développement de CKsquare utilise l'application **gitahead**. Cette application n'a pas été utilisée pendant le développement du projet de tests. Ici, c'est l'interface en ligne de commande de Git qui a été privilégiée ainsi que l'utilisation du plugin **fugitive** sur neovim.

11.1.2 GDB

La taille et la complexité du code a nécessité l'utilisation d'un déboguer. Les déboguer sont des outils très utiles pour comprendre comment un code s'exécute. L'outil qui a été utilisé est GDB (GNU Debugger) qui est compatible avec beaucoup de langages dont le C et C++.

GDB est un outil très puissant, comme la plupart des débogueurs, il permet de suivre l'exécution du code, d'évaluer des expressions, d'interagir avec le code en modifiant les valeurs des variables de façon interactive pendant une session ou encore d'inspecter le contenu des registres (utilisés dans le code assembleur généré par le compilateur). Cet outil propose aussi des fonctionnalités très avancées qui n'ont pas été utilisées durant le stage mais qui sont tout de même intéressantes à noter. GDB permet la création de checkpoints (commande `checkpoint`) qui permettent la sauvegarde de l'état du code pendant l'exécution pour pouvoir recharger cet état sans relancer tout le programme. Il permet aussi d'enregistrer une suite d'instructions pour pouvoir faire de l'exécution inversée et revenir en arrière dans l'exécution du code (commandes `record` et `reverse-step`). Il propose aussi un interpréteur et un module python intégré et il est possible d'utiliser python pour réécrire des commandes. Par exemple, on peut réécrire la fonction d'affichage lorsque l'on souhaite afficher des structures de données complexes. À noter également que GDB a aussi été très utile pour faire fonctionner la solution utilisant des threads décrite dans la section 5.1 en aidant à suivre l'exécution du programme dans les différents threads.

GDB est aussi un outil configurable et sa configuration intègre un langage de script qui permet de créer des commandes personnalisées. Comme on peut le voir sur cet article [11], il est possible de faire

des choses assez complexes avec la configuration de GDB comme créer une commande pour afficher un arbre binaire.

Bien que GDB soit à l'origine fait pour être utilisé dans le terminal, il existe beaucoup d'interface compatibles. Tout d'abord il existe des interfaces graphiques comme `ddd` qui a été créé pour permettre la visualisation de structures des données. GDB est aussi intégré dans des éditeurs de textes, par exemple, *vim* et *emacs* possèdes des intégrations de GDB permettant de déboguer directement depuis l'éditeur. Enfin, certains des serveur DAP comme `cpptools`, utilisent aussi GDB. Lors de ce stage, c'est d'abord l'interface TUI par défaut du programme qui a été utilisée (accompagnée de macros aidant à la mise en place de breakpoints). Par la suite, l'utilisation de `Termdebug` a été privilégiée. C'est une interface proposée par l'éditeur `(neo)vim` qui permet une utilisation très simple du débogueur une fois configurée. Sur la figure **??**, on peut voir à quoi ressemble l'interface Termdebug. Le code exécuté se trouve à gauche et sur la droite il y a une fenêtre montrant les affichages dans le terminal (sortie standard) et une fenêtre permettant d'interagir avec GDB pour exécuter des commandes complexes.

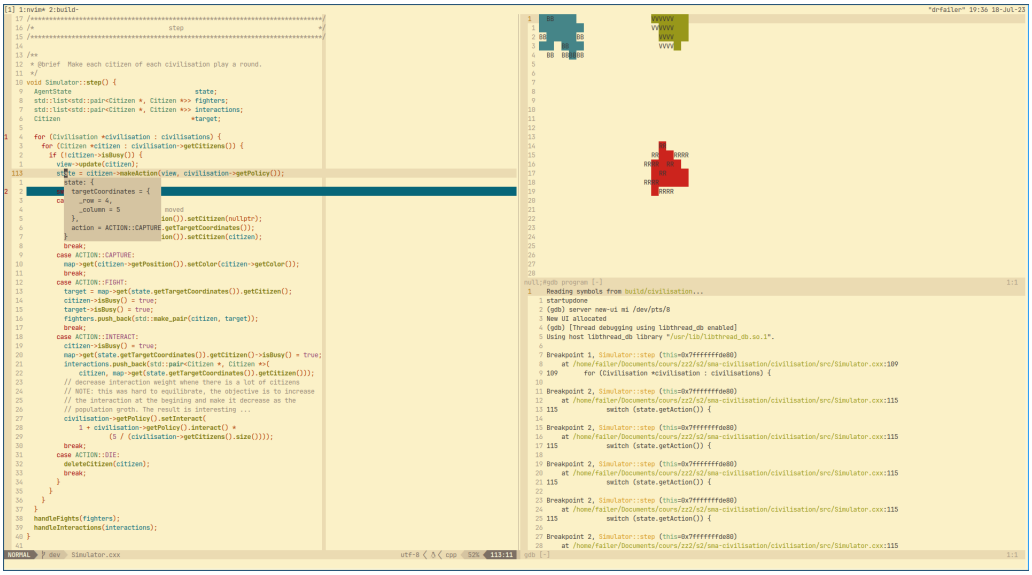

FIGURE 10 – Termdebug : GDB intégré à (neo)vim

GDB a permis de gagner énormément de temps pendant le développement de l'outil de test et c'est malheureusement un outil assez sous-estimé à l'école. Certains développeurs comme John Carmack considèrent l'utilisation d'un débogueur indispensable pour comprendre le fonctionnement d'un code. GDB n'est pas simple d'utilisation mais c'est un outil léger puissant et flexible qu'il est très intéressant de connaitre.

11.1.3 IDE

TODO : nvim -> vim-fugitive, clangd, l'intégration de gdb ... Présentation rapide de ma configuration.

11.2 Rédaction de la documentation

11.3 Planification des tâches

Les tâches prévues lors du stage étaient les suivantes. Au départ, il était prévu de faire le choix du framework de tests puis de faire une revue des éléments du code à tester. Ensuite, il était prévu de faire des tests sur des fichiers simples pour s'habituer au code de l'entreprise et aussi permettre de compiler une première partie du projet. Une fois le projet pris en mains, l'objectif était de s'attaquer aux grandes parties qui étaient tout d'abord le stockage, puis le Cctalk et le MDB. Les historiques devaient être testés à la fin du fait de leur dépendance au stockage. Il était aussi prévu de réaliser de la documentation en parallèle tout au long du projet.

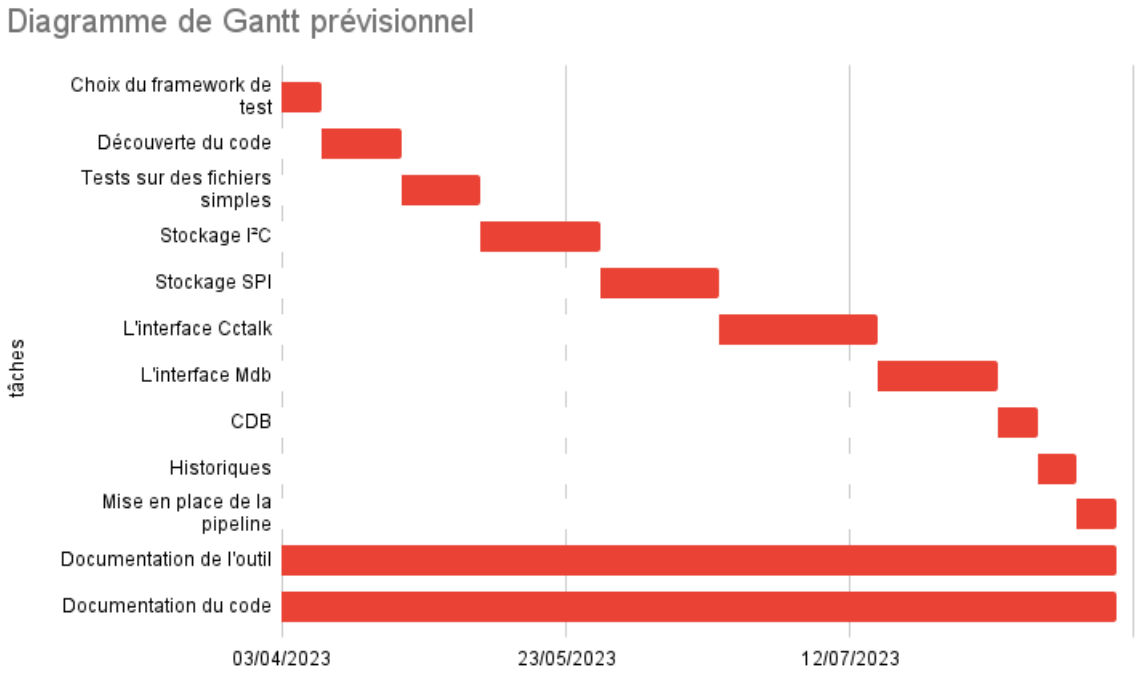

FIGURE 11 – Diagramme de Gantt prévisionnel

L'organisation de ces tâche a été faite sans connaissance de la complexité du code. Certains des composants de l'outil final ont été refait plusieurs fois car les résultats ne satisfaisaient pas les critère nécessaire à leur validation. C'est le cas du stockage où au départ il avait été fait le choix d'utiliser des threads, un choix qui a été remis en cause par la suite. Un autre exemple serait celui du Cctalk dont certaines parties ont été réécrites suite à l'implémentation de l'émulation des composants MDB pour ajouter plus de cohérence. Malgré le fait que certains éléments ont été refaits, le développement a pris moins de temps que prévu. Les tâches ont été surévaluées du fait du manque de connaissances sur certains éléments comme les protocoles par exemples.

NOTE : diagramme actuel

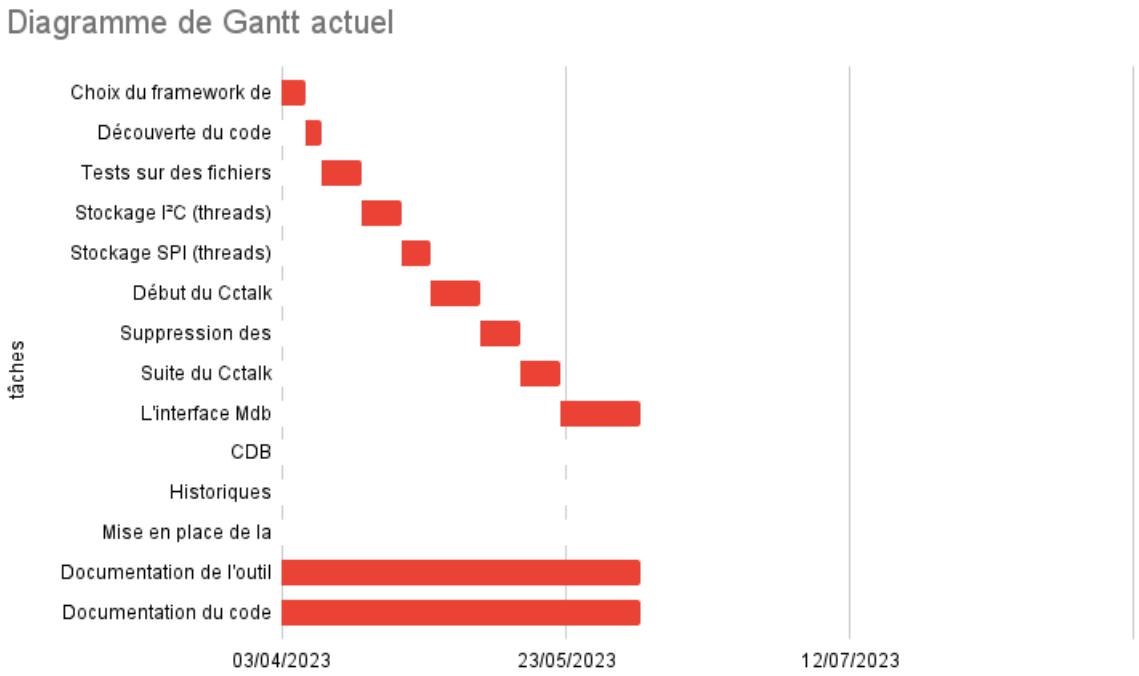

FIGURE 12 – Diagramme de Gantt actuel.

Troisième partie

Résultats et discussions

Dans cette dernière partie, nous commencerons par présenter l'outil final dans toutes ces version (C et C++) puis nous discuterons des perspectives du projet. Enfin, avant de conclure, nous parlerons brièvement de l'inclusion du projet dans le développement durable.

1 L'outil final

Cette section va présenter le résultat final, tout d'abord sur la partie C puis sur la partie Qt.

Concernant la partie C, l'outil final à été complètement intégré au code commun sur les deux branches. Tout d'abord, l'outil met en place un framework de test complet et simple d'utilisation : `Criterion`. L'utilisation framework a été détaillée dans la documentation rédigée durant le stage pour compléter celle du framework. Une configuration de CMake a aussi été mise en place pour simplifier la compilation. Cette configuration utilise CTest qui permet beaucoup d'opérations sur les tests comme le calcule de la couverture (lancement automatique de `gcov`) ou encore l'analyse de fuites mémoire (lancement automatique de `valgrind`). L'outil propose aussi un système permettant de simuler les interfaces des composants électroniques utilisés dans les projets de l'entreprise. Cela comprend les périphériques de stockage (registres, eeproms et flash) utilisés en I^2C et en SPI mais aussi d'autre périphériques interfacés en Cctalk et en MDB qui sont des protocoles spécialisés dans la monétique. Pour chacun des composants, la documentation détaille la méthode employée pour la simulation de sorte à ce que l'on puisse facilement simuler de nouveaux composants. La simulation des protocoles monétiques proposes deux niveaux de simulation pour les tests. Une version où des fichiers ont été modifiés pour isoler les éléments testés du reste du code et une autre version qui passe par toute la chaine d'instructions jusqu'aux fichiers du compilateur. Cela permet de faire à la fois des tests unitaires mais aussi des tests d'intégration où l'on test tout le code. La méthode consistant à créer des versions modifiées des fichiers a beaucoup été utilisée pour rendre les tests plus simples ou pour changer du code qui ne pouvait pas être compilé. Au final, il est bien possible d'écrire des tests où le code testé interagir avec les composants simulés. Depuis ces test, on peut accéder aux données que les composants reçoivent pour vérifier que cela correspond à ce qui était attendu. Enfin, la pipeline permettant d'automatiser les tests sur Gitlab a bien été mise en place et cette dernière permet même de calculer le pourcentage de couverture des tests. Cela n'était pas demandé au départ et permet aux développeur d'avoir une idée de la qualité des tests.

Description de l'outil final réalisé à la fin du stage.

TODO : Rétrospective sur le choix du framework (problème macros, ...).

2 Discussion et perspectives

Regard sur ce qui a été fait et comment cela a été fait (parler de l'échec des threads, manque de bibliographie, ...). Parler de ce qui reste à faire ou a améliorer.

3 Développement durable

Dans cette section, nous allons voir en quoi ce simple projet de test peut s'inscrire dans une démarche de développement durable.

Tout d'abord, l'objectif du stage était la mise en place d'un outil permettant l'automatisation de tests sur du code s'exécutant sur des cartes électroniques. Les cartes électroniques sont ensuite utilisées sur des bornes installées dans divers endroits, en France comme à l'étranger. Les problèmes sur ce genre de dispositifs peuvent avoir un impacte non négligeable sur l'environnement. En effet, lorsqu'il y a un problème au niveau du code, cela nécessite de mettre les cartes à jour. Dans le meilleur des cas, cela est possible en utilisant le système de mise à jour automatique créé par la société. Cependant, cela va consommer beaucoup de bande passante car il va falloir envoyer via internet un nouveau binaire ou une nouvelle configuration les cartes. Dans le pire des cas, lorsque la borne est trop vielle ou qu'elle n'est pas connectée à internet, il faut faire intervenir un technicien qui devra se déplacer en voiture pour mettre jour la borne manuellement. Le fait de détecter les problèmes au plus tôt avec des tests automatisés peut grandement limiter les risques de problèmes par la suite et donc l'obligation de mettre à jour les bornes.

Ensuite, l'outil créé ne permet pas seulement de faire des tests, il permet aussi de faire des statistiques au niveau du code. Par exemple, l'analyse de la couverture des tests peut non seulement donner une idée de la qualité des tests mais aussi sur la qualité du code. Si on cherche à faire des tests les plus complets possibles mais que le pourcentage de couverture restes bas, cela peut être indicateur du fait qu'il y a du code inutile. Le fait d'enlever les éléments inutiles dans un programme permet d'un part de gagner en simplicité et donc de limiter les risques d'erreurs. D'autre part, cela aide à limiter la taille des exécutable et donc à consommer moins de mémoire. Étant donné que les programme sont de plus en plus consommateur de mémoire, le fait d'optimiser cette consommation peut permettre d'exécuter le code sur des cartes plus ancienne et d'économiser la mémoire sur les nouvelles cartes augmentant ainsi leurs durées de vie. Étant donné le fait que la fabrication de cartes électroniques coûte cher en métaux rares, le fait de pouvoir utiliser d'anciennes cartes permet d'économiser les ressources naturelles. En plus de cela, il reste peu d'entreprise qui utilisent encore des cartes électroniques à cause de la complexité du code. Maintenant ce sont les PC qui sont privilégiés même dans l'embarque alors que ces derniers sont moins fiables et plus consommateurs en énergie et en métaux rares. L'optimisation du code donc est cruciale pour pouvoir continuer à utiliser de simples cartes électroniques.

Enfin, la mise en place de l'outil s'est aussi fait dans une démarche environnementale. Par exemple, lors de la mise en place de la pipeline des commun C++, il s'est avéré qu'il faillait télécharger et installer toutes les bibliothèques à chaque exécution. Cela n'était pas souhaitable étant donné la taille des paquets à installer, surtout lorsque l'on considère le fait que les pipeline peuvent être exécuté plusieurs dizaines de fois par jours. C'est pour cela que la création d'une image docker prés-configurée a été automatisée, de sorte à ce que l'on ait plus rien à télécharger au lancement de la pipeline des communs. Cette image est compilée bien moins souvent et permet d'économiser du temps et de la bande passante au lancement de la pipeline Qt. Le temps gagner permet aussi de libérer les ressources du serveur Gitlab plus rapidement pour économiser de l'énergie.

Ce projet n'est donc à priori aucunement lié au développement durable, il permet cependant de manière indirecte de limiter les consommations de ressource. De plus, le projet à était fait de manière responsable de sorte à polluer le moins possible.

Conclusion

CONCLUSION

4 Résumé biblio (WARN : section temporaire)

Test affichage biblio :

— [1] : liste des frameworks sur Wikipédia
— [12] : définition fixture Wikipédia
— [2] : tester des machines à états
— [3] : présentation du protocole I^2C
— [5] et [6] : présentation du protocole SPI
— [13] : test de systèmes embarqués en utilisant de la simulation. Très différent du projet mais il y a des point intéressants.
— [14] : simulation pour CI sur systèmes embarqués
— [15] : intégration continue sur les systèmes embarqués.
— [16] : livre sur les frameworks de tests (p 42 : mocking)
— [7], [8],[17] : trois premières parties de la doc cctalk
— [10] : doc mdb

Bibliographie

[3] J. Mankar, C. Darode, K. Trivedi, M. Kanoje et P. Shahare, "Review of I2C protocol," *International Journal of Research in Advent Technology*, t. 2, n° 1, 2014. adresse : https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=314537daa1f601f83044b25b68e2af6c8f331f3f.

[5] P. Dhaker, "Introduction to SPI interface," *Analog Dialogue*, t. 52, n° 3, p. 49-53, 2018. adresse : https://b2.sisoog.com/file/zmedia/dex/cd38acc402d52de93a241ca2fcb833b5_introduction-to-spi-interface.pdf.

[6] L. L. Li, J. Y. He, Y. P. Zhao et J. H. Yang, "Design of microcontroller standard SPI interface," in *Applied Mechanics and Materials*, Trans Tech Publ, t. 618, 2014, p. 563-568. adresse : https://www.researchgate.net/profile/Jianhong-Yang-2/publication/286761932_Design_of_Microcontroller_Standard_SPI_Interface/links/58bfe7eba6fdcc63d6d1bb37/Design-of-Microcontroller-Standard-SPI-Interface.pdf.

[7] C. P. Solutions, *ccTalk Serial Communication Protocol - Generic Specification - Issue 4.7 (partie 1)*. "Crane Payment Solutions", 2013. adresse : https://cctalktutorial.files.wordpress.com/2017/10/cctalkpart1v4-7.pdf.

[8] C. P. Solutions, *ccTalk Serial Communication Protocol - Generic Specification - Issue 4.7 (partie 2)*. "Crane Payment Solutions", 2013. adresse : http://www.coinoperatorshop.com/media/products/manual/cctalk/cctalk44-2.pdf.

[10] E. Technical Members of NAMA et EVMMA, *Multi-Drop Bus / Internal Communication Protocol*. "National Automatic Merchandising Association", 2011. adresse : https://www.ccv.eu/wp-content/uploads/2018/05/mdb_interface_specification.pdf.

[13] J. Engblom, G. Girard et B. Werner, "Testing Embedded Software using Simulated Hardware," jan. 2006. adresse : https://hal.science/hal-02270472v1/file/7B4_J.Engblom_Virtutech%20%281%29.pdf.

[14] J. Engblom, "Continuous integration for embedded systems using simulation," in *Embedded World 2015 Congress*, 2015, p. 18. adresse : http://www.engbloms.se/publications/engblom-ci-ew2015.pdf.

[15] T. Mårtensson, D. Ståhl et J. Bosch, "Continuous integration applied to software-intensive embedded systems–problems and experiences," in *Product-Focused Software Process Improvement : 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*, Springer, 2016, p. 448-457. adresse : https://www.researchgate.net/profile/Torvald-Martensson/publication/309706302_Continuous_Integration_Applied_to_Software-Intensive_Embedded_Systems_-_Problems_and_Experiences/links/5beb0d4e4585150b2bb4d803/Continuous-Integration-Applied-to-Software-Intensive-Embedded-Systems-Problems-and-Experiences.pdf.

[16] P. Hamill, *Unit test frameworks : tools for high-quality software development.* " O'Reilly Media, Inc.", 2004. adresse : https://books.google.fr/books?hl=fr&lr=&id=2ksvdhhnWQsC&oi=fnd&pg=PT7&dq=c+unit+test+framework&ots=AM5ZXbSUG4&sig=jPl1QzC0OSPfnBQi6IKXWLPeNzo&redir_esc=y#v=onepage&q=c%20unit%20test%20framework&f=false.

[17] C. P. Solutions, *ccTalk Serial Communication Protocol - Generic Specification - Issue 4.7 (partie 3)*. "Crane Payment Solutions", 2013. adresse : http://www.coinoperatorshop.com/media/products/manual/cctalk/cctalk44-3.pdf.

Webographie

[1] Wikipedia contributors, *List of unit testing frameworks — Wikipedia, The Free Encyclopedia*, [Online ; accessed 29-May-2023], 2023. adresse : `https://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=1143492860` (visité le 03/03/2023).

[2] M. Jones, *TESTING STATE MACHINES*, 2009. adresse : `https://accu.org/journals/overload/17/90/jones_1548/` (visité le 03/03/2023).

[4] Wikipédia, *I2C — Wikipédia, l'encyclopédie libre*, [En ligne ; Page disponible le 13-octobre-2022], 2022. adresse : `http://fr.wikipedia.org/w/index.php?title=I2C&oldid=197726464`.

[11] K. Elphinstone. "GDB - Init File." (2020), adresse : `https://www.cse.unsw.edu.au/~learn/debugging/modules/gdb_init_file/`.

[12] Wikipedia contributors, *Test fixture — Wikipedia, The Free Encyclopedia*, [Online ; accessed 3-June-2023], 2023. adresse : `https://en.wikipedia.org/w/index.php?title=Test_fixture&oldid=1152280894`.

Glossaire

DES (Data Encryption Standard) algorithme de chiffrement symétrique.. 21

fullduplex tous les éléments peuvent parler en même temps.. 19

halfduplex deux émeteurs ne transmettent jamais des données en même temps.. 17, 21

PIC (Peripheral Interface Controller) famille de microcontrôleurs de l'entreprise Microship.. 18, 23

protocole série protocole dont les éléments d'informations se succèdent sur une seule voie entre deux points.. 17, 19, 21

sous module git biblithèques qui on leur propre répertoire distant, elles sont stockées dans un projet à part sur Gitlab.. 8, 9, 30

Table des annexes

A Extraits de codes pour les frameworks

Check

```c
#include <check.h>

START_TEST(test_name)
{
  ck_assert(1 == 1);
  ck_assert_msg(2 == 2, "Should be a success");
}
END_TEST

Suite *simple_suite(void) {
  Suite *s;
  TCase *tc_core;
  s = suite_create("suite name");
  tc = tcase_create("test case name");
  tcase_add_test(tc_core, test_name); // adding tests
  suite_add_tcase(s, tc_core); // create the suite

  return s;
}

int main(void) {
  int number_failed;
  Suite *s;
  SRunner *sr;
  s = simple_suite();
  sr = srunner_create(s);
  srunner_run_all(sr, CK_NORMAL);
  number_failed = srunner_ntests_failed(sr);
  srunner_free(sr);
  return (number_failed == 0) ? 0 : 1;
}
```

Listing 4 – Check : Exemple simple

CUnit

```c
/*************************************************************************/
/*                                tests                                  */
/*************************************************************************/

void test_function(void) {
  CU_ASSERT(0 == 0);
}

/*************************************************************************/
/*                            setup & teardown                           */
/*************************************************************************/

int init_suite(void) {
  return 0; // -1 for error
}

int clean_suite(void) {
  return 0; // -1 for error
}

/*************************************************************************/
/*                          lancement des tests                          */
/*************************************************************************/

int main(void)
{
  CU_pSuite pSuite = NULL;
  /* initialize the CUnit test registry */
  if (CUE_SUCCESS != CU_initialize_registry())
    return CU_get_error();
  /* add a suite to the registry */
  pSuite = CU_add_suite("suite name", init_suite, clean_suite);
  if (NULL == pSuite) {
    CU_cleanup_registry();
    return CU_get_error();
  }
  /* Adding to the test suite */
  if ((NULL == CU_add_test(pSuite, "description", test_function)))
  {
    CU_cleanup_registry();
    return CU_get_error();
  }
  /* Run all tests using the CUnit Basic interface */
  CU_basic_set_mode(CU_BRM_VERBOSE);
  /* Run tests */
  CU_basic_run_tests();
  /* withdraw error number (for returning to pipeline) */
  unsigned int nb_errors = CU_get_number_of_suites_failed();
  /* registry cleanup */
  CU_cleanup_registry();
  return 0;
}
```

Listing 5 – CUnit : exemple simple

Criterion

```c
// test basique
Test(suite_name, test_name) {
  cr_assert(1 == 1);
}

// avec des fonctions de setup et teardown
Test(suite_name, test_name, .init = setup_function, .fini = teardown_function) {
  unsigned char Expected[3] = {1, 2, 3};
  unsigned char Founded[3] = {1, 2, 3};
  cr_assert(eq(u8[3], Founded, Expected));
}

// This test will pass
Test(sample, passing, .signal = SIGSEGV) {
    int *ptr = NULL;
    *ptr = 42;
}
```

Listing 6 – Criterion : exemple simple

Minunit

```c
MU_TEST(test_name) {
  mu_check(0 == 0); // ce test doit échouer
}

MU_TEST_SUITE(suite_name) {
  MU_RUN_TEST(test_name);
}

int main(void)
{
  MU_RUN_SUITE(test_suite);
  MU_REPORT();
  return MU_EXIT_CODE;
}
```

Listing 7 – Minunit : exemple simple

Munit

```c
MunitResult test_function() {
  munit_assert_true(0 == 0);
  return MUNIT_OK;
}

/*********************************************************************************/
/*                                  test setup                                   */
/*********************************************************************************/

// setup all the tests
MunitTest tests[] = {
  {
    "test_name",
    test_function,
    NULL,                    // setup function
    NULL,                    // teardown function
    MUNIT_TEST_OPTION_NONE, // options
    NULL,                    // test parameters
  },
  { NULL, NULL, NULL, NULL, MUNIT_TEST_OPTION_NONE, NULL } // end of the tests list
};

/*********************************************************************************/
/*                               test suite setup                                */
/*********************************************************************************/

static const MunitSuite suite = {
  "simple-test",
  tests,
  NULL, // no sub-suites
  1,    // iterations (utile avec les générateur de random number)
  MUNIT_SUITE_OPTION_NONE // no options
};

/*********************************************************************************/
/*                                     main                                      */
/*********************************************************************************/

int main(void)
{
  return munit_suite_main(&suite, NULL, 0, NULL);
}
```

Listing 8 – Munit : exemple simple

Unity

```c
#include "unity.h"
#include "file_to_test.h"

void setUp(void) {
    // set stuff up here
}

void tearDown(void) {
    // clean stuff up here
}

void test_function(void) {
    //test stuff
}

// not needed when using generate_test_runner.rb
int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_function);
    return UNITY_END();
}
```

Listing 9 – Unity : exemple simple

Tau

```c
#include <tau/tau.h>

TEST(foo, bar1) {
    int a = 42;
    int b = 13;
    CHECK_GE(a, b); // pass :)
    CHECK_LE(b, 8); // fail - Test suite not aborted
}

TEST(foo, bar2) {
    char* a = "foo";
    char* b = "foobar";
    REQUIRE_STREQ(a, a); // pass :)
    REQUIRE_STREQ(a, b); // fail - Test suite aborted
}

TAU_MAIN() // sets up Tau (+ main function)
```

Listing 10 – Tau : exemple simple

B Test sur l'envoi des historiques sur le serveur

```
Test(CKSPROHISSND_Tests, TESTCKSPROHISSND_CdbFill,
     .init = TESTCKSPROHISSND_Setup, .fini = TESTCKSPROHISSND_Teardown)
{
    TEEPROM_ADDR Addr;
    uchar HisElts[20][sizeof(TITEM_PAYMENT)] = {0};
    int HisEltsCount = 0;
    int Cpts;

    // 1. remplire le cdb
    for (Cpts = 0; Cpts < History->RecordMax - 1; ++Cpts) {
        CDB_IndexToAddr(History,History->Control.Count,&Addr);
        TESTCKSPROHISSND_RandomItemGenerate();
        // sauvegarde des éléments enregistrés
        memcpy(HisElts[HisEltsCount++],
               &TESTCKSPROHISSND_ItemHisPayment,sizeof(TITEM_PAYMENT));
        cr_assert(true == CDB_Add(History,(uchar*)
                                  &TESTCKSPROHISSND_ItemHisPayment));
        cr_assert(eq(u8[sizeof(TITEM_PAYMENT)],
                     TESTMEM_EEPROMM[0] + Addr,
                     (uchar*) &TESTCKSPROHISSND_ItemHisPayment));
        // vérification du début de l'eeprom
        cr_assert(eq(u8[sizeof(History->Control)],
                     TESTMEM_EEPROMM[0] + History->Store.DataAddr + 8 + 1,
                     (uchar*) &History->Control));
    }

    // init CKSPROHISSND_Control
    CKSPROHISSND_Control();
    TESTTIMER_Wait(2001);
    CKSPROHISSND_Control();
    CKSPROHISSND_Control(); // on arrive dans l'état 2


    // 2. traiter l'historique
    HisEltsCount = 0;
    for (Cpts = 0; Cpts < History->RecordMax - 1; ++Cpts) {
        CKSPROHISSND_Control();
        cr_assert(true == TESTCKSPRO_CmdSent);
        TESTCKSPRO_CmdSent = false;
        // on récupère l'élément que l'on est sensé envoyer et on vérifie l'envoi
        memcpy(&TESTCKSPROHISSND_ItemHisPayment,
               HisElts[HisEltsCount++], sizeof(TITEM_PAYMENT));
        cr_assert(true == TESTCKSPROHISSND_CmdCheck());
        CKSPROHISSND_Control();
```

```
    }

    // 3. re-remplir le cdb
    HisEltsCount = 0;
    for (Cpts = 0; Cpts < History->RecordMax - 1; ++Cpts) {
        CDB_IndexToAddr(History, History->Control.Count, &Addr);
        TESTCKSPROHISSND_RandomItemGenerate();
        // sauvegarde des éléments enregistrés
        memcpy(HisElts[HisEltsCount++],
               &TESTCKSPROHISSND_ItemHisPayment,sizeof(TITEM_PAYMENT));
        cr_assert(true == CDB_Add(History,
                                  (uchar*) &TESTCKSPROHISSND_ItemHisPayment));
        cr_assert(eq(u8[sizeof(TITEM_PAYMENT)],
                     TESTMEM_EEPROMM[0] + Addr,
                     (uchar*) &TESTCKSPROHISSND_ItemHisPayment));
        // vérification du début de l'eeprom
        cr_assert(eq(u8[sizeof(History->Control)],
                     TESTMEM_EEPROMM[0] + History->Store.DataAddr + 8 + 1,
                     (uchar*) &History->Control));
    }

    // 4. traiter à nouveau l'historique
    HisEltsCount = 0;
    for (Cpts = 0; Cpts < History->RecordMax - 1; ++Cpts) {
        CKSPROHISSND_Control();
        cr_assert(true == TESTCKSPRO_CmdSent);
        TESTCKSPRO_CmdSent = false;
        // on récupère l'élément que l'on est sensé envoyer et on vérifie
        // l'envoi
        memcpy(&TESTCKSPROHISSND_ItemHisPayment,
               HisElts[HisEltsCount++],sizeof(TITEM_PAYMENT));
        cr_assert(true == TESTCKSPROHISSND_CmdCheck());
        CKSPROHISSND_Control();
    }
}
```