

République Tunisienne
Ministère de l'Enseignement Supérieur
Et de la Recherche Scientifique

Université de Carthage

Institut Nationale des Sciences
Appliquées et de la Technologie



Cycle Ingénieur :
Informatique Industrielle
Et Automatique

Option :
Systèmes Embarqué

Rapport de Stage

Présenté à

**L'Institut Nationale des Sciences Appliquées
et de la Technologie**

Stage d'initiation en :

Informatique Industrielle et Automatique

Option :

Systèmes Embarqué

Par

Nourchene BARGAOUI

REALISATION D'UNE PLATEFORME DE
DEVELOPPEMENT POUR LA CARTE MENZU BASEE SUR
LE LANGAGE RUST

Entreprise d'accueil :

Sofiatech
A Onetech company

Mr.	Nader LADHARI	Encadrant
-----	---------------	-----------

Dédicaces

Avec gratitude, amour et joie, je dédie ce modeste travail,

A mes chers parents

Qui m'ont été toujours une source d'amour, d'affection et de force

*Qui m'ont appris à travailler avec enthousiasme qu'ils ont su me communiquer depuis mon
jeune âge*

Je me souviendrai toujours avec gratitude des sacrifices énormes consentis par vous

*Puisse Dieu vous accorder santé, bonheur et longue vie et faire en sorte que jamais je ne vous
déçoive*

A ma grande famille

Je cite en particulier mes grands-parents pour leurs soutiens et encouragements

A mes ami(e)s

*En leur souhaitant le succès dans leurs vies aussi bien professionnelles que familiales Ce
travail est accompli grâce à votre support et votre encouragement.*

Remerciements

C'est avec un grand plaisir que je réserve ces quelques lignes en signe de gratitude et de reconnaissance à tous ceux qui ont contribué à l'élaboration de mon projet au sein de la Société
Sofia Technologies

*Je tiens à remercier vivement **Mr. Nader LADHARI**, Directeur des Plans et Programmes de SOFIA Technologies, qui a accepté aimablement de m'encadrer et n'a jamais hésité à consentir les efforts pour m'orienter.*

*Je remercie tous les collaborateurs et le cadre administratif de SOFIA Technologies pour leur accueil chaleureux et spécialement **Mr. Imed JEDDIDI** de m'avoir accueilli au sein de ses équipes.*

*Un remerciement très particulier à **Mr. Bacem KAABIA**, Développeur système embarqué chez SOFIA Technologies, qui s'est investi sans compter pour que ce projet soit de qualité. J'ai appris à apprécier sa capacité d'écoute et sa rigueur scientifique en travaillant à ses côtés.*

Table des matières

Dédicaces.....	II
<i>Remerciements</i>	III
Table des matières.....	IV
Liste des figures.....	VIII
Liste des Tableaux	IX
Liste des abréviations	X
Introduction générale.....	XI
Chapitre 1 : Contexte générale	1
I. Introduction.....	2
II. Présentation de l'entreprise.....	2
II.1 SOFIA Technologies	2
II.1.1 Introduction.....	2
II.1.2 Les départements	3
II.1.3 Organigramme de l'entreprise.....	3
II.1.4 Les structures hiérarchiques et fonctionnelles.....	3
II.2 Les produits de Sofia	4
II.3 Les autres filiales de Sofia	5
III. Problématique et solution envisagée.....	7
IV. Présentation de projet et objectifs visés par l'intervention	8
V. Méthodologie de travail.....	8
V.1 Choix de modèle d'organisation	8
V.2 Vue d'ensemble des activités de projet.....	9
VI. Travail demandé.....	9
VII. Conclusion	10
Chapitre 2 : Familiarisation avec l'équipement matériel et généralité sur le langage Rust	11
I. Introduction.....	12
II. Familiarisation avec la carte électronique embarquée Menzu	12
II.1 Définition	12

II.2 Caractéristiques de la carte.....	12
II.3 Les différentes composants (BSP).....	13
II.4 Les microcontrôleurs STM32L4xx :	14
II.4.1 Interface SPI (Serial Peripheral interface).....	15
II.4.2 Interface I2C (Inter-Integrated Circuit).....	16
II.4.3 Interface UART (Universal Asynchronous Receiver Transmitter)	16
III. Généralités sur le langage de programmation (RUST).....	17
III.1. L’historique de Rust.....	17
III.2. Définition	17
III.3 Avantages et limites :	18
III.4 Les concepts de programmation RUST.....	19
III.4.1 Gestion de mémoire (Pile et Tas).....	19
III.4.2 La sûreté par défaut.....	19
III.4.3 Inférence de type (type inference).....	20
III.4.4 Variables et mutabilité	20
III.4.5 Partage et propriété (Ownership).....	21
III.4.6 Références et emprunts	22
III.4.7 Durée de vie (Lifetime)	22
IV. Conclusion.....	23
Chapitre 3 : Installation de l’environnement et création d’un projet Rust embarqué.....	24
I. Introduction.....	25
II. Environnement logiciel.....	25
II.1 Configuration de l’environnement Rust sous Windows	25
II.1.1 La chaîne de compilation Rustup.....	25
II.1.2 RLS et composants	26
II.1.4 Vérification de l’installation	27
II.1.5 Les commandes de cargo.....	28
II.2 Choix de l’IDE : l’Eclipse corrodion 2018.....	28
II.2.1 Installation de l’Eclipse	29

II.2.2 Configuration des préférences de l'Eclipse	29
II.2.3 Installation et configuration de plug-in Eclipse de Débogage OpenOcd.....	29
II.2.4 Utilisation du plug-in OpenOCD.....	30
II.3 Limitation de débogage de Rust sous Eclipse	31
II.4 La solution proposée	31
III. Création d'un projet Rust embarqué	32
III.1 Arborescence exemplaire	32
IV. Conclusion.....	35
Chapitre 4 : Développement ET Tests	36
I. Introduction.....	37
II. Développement d'un exemple : Clignotement de LED de la carte stm32f429	37
II.1 Création de projet.....	37
II.2 Développement et code : src/main.rs.....	39
II.2.1 Dépendances et inclusion des « Crates ».....	39
II.2.2 Configuration de GPIO de la carte.....	41
II.2.3 Configuration des pins des leds	41
II.2.4 Configuration de l'horloge	41
II.3 Implémentation et test	42
III.1 Arborescence retenue.....	43
III.2 Développement des modules.....	44
III.2.1 Développement de module LED	44
III.2.2 Développement de module Bouton.....	47
III.2.3 Développement de module Relais	49
III.2.4 Développement de module Timer	50
III.2.5 Développement de module I2C (Inter-Integrated Circuit)	52
III.2.6 Développement de module SPI (Serial Peripheral Interface).....	53
III.2.7 Développement de module UART	54
IV. Conclusion.....	56
Conclusion et perspectives.....	57



Bibliographie	59
Résumé	60
Abstract	60

Liste des figures

Figure 1: Logo de Sofia Holding	2
Figure 2: Organigramme de l'entreprise	3
Figure 3: La structure hiérarchique de l'entreprise	4
Figure 4: Logo de la solution phyt'eau	4
Figure 5: Logo de la plateforme Senya	5
Figure 6: La vue de face de la carte Menzu	5
Figure 7: Les filiales de Sofia Holding.....	6
Figure 8: Cycle en V	8
Figure 9: La vue de face de la carte Menzu	12
Figure 10: Les différents composants de la carte.....	14
Figure 11: Le diagramme de l'interface SPI.....	15
Figure 12: Diagramme de l'interface I2C	16
Figure 13: Le diagramme de l'interface UART	17
Figure 14: Logo de langage RUST	18
Figure 15: Rapport sécurité / contrôle des différents langages de programmation.....	19
Figure 16: Exemple d'inférence de type.....	20
Figure 17: Exemple d'une création d'une variable immuable	20
Figure 18: Résultat de compilation	21
Figure 19: Exemple de Références et emprunts	22
Figure 20: Exemple montrant la Durée de vie.....	23
Figure 21: Lien de l'installation de Rustup.....	25
Figure 22: Choix de l'option de l'installation de Rust stable	26
Figure 23: La commande de demande de version de compilateur rustc.....	27
Figure 24: La commande de demande de version de cargo.....	28
Figure 25: La version de l'Eclipse	28
Figure 26: Spécification des chemins d'accès de dossier Rust	29
Figure 27: Spécification de chemin d'accès de dossier l'OpenOCD	30
Figure 28: Création de fichier *.rs	31
Figure 29: L'exécutable dans une nouvelle application C/C++	32
Figure 30: Création d'un nouveau projet Rust.....	Erreur ! Signet non défini.
Figure 31: Le package du fichier cargo.toml	33
Figure 32: Les dépendances de projet	33
Figure 33: La carte STM32F429	37
Figure 34: Le contenu du fichier cargo.toml.....	38
Figure 35: Le contenu du fichier memory.x.....	38

Figure 36: Le fichier de configuration.....	39
Figure 37: Les caisses externes importées	40
Figure 38: La caisse dédiée pour la carte stm32f429	40
Figure 39: Le clignotement des LEDs.....	43
Figure 40: Architecture générale de projet.....	44
Figure 41: Organigramme de l'algorithme de module LED	46
Figure 42: Clignotement de LED1 rouge.....	47
Figure 43: Clignotement de LED2 vert	47
Figure 44: Organigramme de l'algorithme de module Bouton	48
Figure 45: Test de bouton	49
Figure 46 : Organigramme de l'algorithme de module relais	50
Figure 47: Organigramme de l'algorithme Timer	51
Figure 48: Compilation de module Timer.....	52
Figure 49: Organigramme de l'algorithme I2C.....	53
Figure 50: Organigramme de l'algorithme SPI.....	54
Figure 51: Organigramme de l'algorithme UART	55

Liste des Tableaux

Tableau 1: Les microcontrôleurs supportés par la carte Menzu	13
Tableau 2: Les interfaces de communications supportées par la carte.....	13
Tableau 3: Composants de la carte Menzu	14

Liste des abréviations

EDA	Electronic Design Automation
PCB	Printed Circuit Board
LED	Light-Emitting Diode
IDE	Integrated Development
I2C	Inter-Integrated Circuit
SPI	Serial Peripheral interface
UART	Universal Asynchronous Receiver Transmitter
BSP	Board Support Package
HAL	Hardware Abstraction Layer
IOT	Internet Of Thing
LAN	Local Area Network
WLAN	Wireless Local Area Network
CPU	Central processing Unit
USB	Bus Serie Universel
SCLK	Serial Clock
MOSI	Master Output Slave Input
MISO	Master Input Slave Output
SS	Slave Select
RISC	Reduced Instruction set computing
SDA	Serial Data Line
SCL	Serial Clock Line
TCP/IP	Transmission Control Protocol - Internet Protocol
TOML	Tomnuident, Langage Minimal
GPIO	General Purpose Input/Output
RCC	Reset and Control Clock

Introduction générale

Les systèmes embarqués sont en forte croissance, de plus en plus populaire ces dernières années. Les domaines les utilisant étant de plus en plus nombreux et certains requérant une très forte sécurité, la sécurisation des systèmes embarqués est donc devenue aujourd'hui un axe majeur de recherche.

Il faut noter que plusieurs langages de programmation se veulent dédiés à l'embarquer. Des langages proches de la machine comme le langage C et dans une moindre mesure le C++ sont utilisés. Mais plusieurs chercheurs en sécurité démontrent comment les bugs des langages de programmation peuvent créer des failles de sécurité dans les applications. Même en développant dans les règles de l'art de la sécurité, un développeur peut introduire des vulnérabilités dans ses programmes. Car **les langages de programmation** sont eux-mêmes remplis de failles.

C'est dans ce cadre que se situe le sujet de notre projet effectué au sein de l'entreprise Sofia Technologies. Ce projet consiste à créer une plateforme de développement sur la carte Menzu basée sur le langage Rust qui nous offre des fonctionnalités qui le distingue aux autres langages traditionnels non sécurisés, et qui améliore la fiabilité des logiciels systèmes.

Le présent rapport comporte quatre chapitres qui illustrent toute la démarche de notre travail. Ils sont organisés comme suit :

- **Dans le premier chapitre** intitulé « Contexte général », nous allons présenter l'organisme d'accueil, le cadre du projet, la problématique et la méthodologie qui va être adoptée pour réaliser notre projet et on le termine par la fixation des tâches à réaliser.
- **Le deuxième chapitre** intitulé « Familiarisation avec l'équipement matériel et généralité sur le langage Rust » porte sur deux axes principaux, à savoir : la présentation de la carte Menzu par la description de ses différentes fonctionnalités, L'introduction des spécificités de langage de programmation Rust.
- **Le troisième chapitre** intitulé « Installation de l'environnement et création d'un projet Rust embarqué ». Nous allons décrire le cadre de réalisation de notre projet à savoir l'environnement matériel, logiciel, les différentes techniques utilisées et enfin la création d'un projet embarqué.

- **Le quatrième chapitre intitulé** « Développement et test ». Nous allons le dédier pour le développement des modules puis leurs implémentations et leurs tests sur la carte Menzu.
- **Enfin**, nous clôturons ce rapport par une conclusion générale dans laquelle nous donnons un résumé de notre travail ainsi que quelques perspectives.

CHAPITRE 1 : CONTEXTE GÉNÉRALE

I. Introduction

Dans ce chapitre, nous commencerons par présenter l'organisme d'accueil, au sein duquel nous avons élaboré notre projet, Ensuite, nous présenterons le cadre et la méthodologie adoptée pour accomplir sa gestion. Et nous finissons par fixer les tâches à réaliser pour atteindre nos objectifs.

II. Présentation de l'entreprise

SOFIA Holding est un groupe international fondé en 2016, basé à Tunis et à Paris. Travaille dans le domaine de l'énergie, la sante, de la technologie de l'information et de l'agriculture intelligente Il regroupe 5 filiales ; Sofia technologies, Sofia Europa, Sofia Academy, IFarming et Sater Solar.



Figure 1: Logo de Sofia Holding

II.1 SOFIA Technologies

II.1.1 Introduction

Sofia technologies est experte dans la sécurité informatique et assure que leurs logiciels sont conformes aux normes afin de garantir l'intégrité des données et des informations échangées.

Grâce à ses outils EDA , l'équipe Sofia Technologies est capable de réaliser tout type de système électronique, de la phase de recherche à la validation finale en passant par les étapes de conception, simulation, intégration, suivi de production et de fabrication.

II.1.2 Les départements

Les différents départements de Sofia Technologies sont :

- Département système embarqué : Ce département est chargé de la création et de la conception de logiciels et d'applications intégrés en fonction des besoins du projet. Il est composé environ de 25 personnes. C'est le département dont je fais partie pendant mon stage pour réaliser le projet.
- Département des systèmes électroniques : Composé de 5 employés, ce département est spécialisé dans l'analyse des spécifications, la conception et la simulation, le prototypage (PCB et PCBA), les tests (évolutifs et fonctionnels) et l'industrialisation.
- Département des systèmes d'information : Experts en industrialisation, configuration, optimisation et développement d'applications serveur.

II.1.3 Organigramme de l'entreprise

Le diagramme présenté dans la Figure2 montre les différents départements au sein de la société.

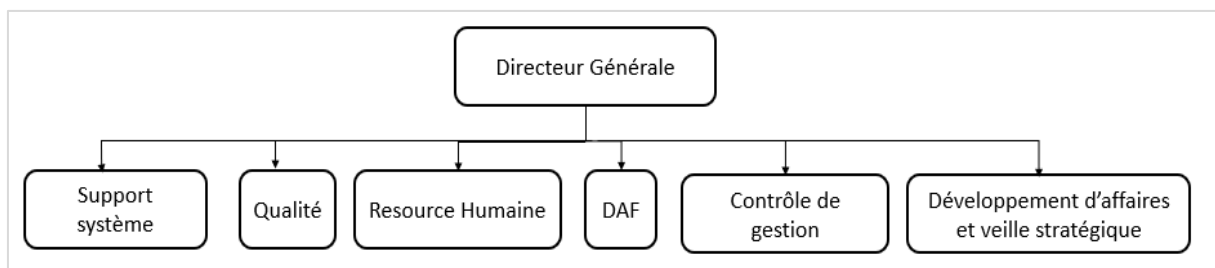


Figure 2: Organigramme de l'entreprise

II.1.4 Les structures hiérarchiques et fonctionnelles

Cet organigramme schématise les relations hiérarchiques dans l'entreprise, c'est un outil interne à l'entreprise mais aussi pour les relations avec les partenaires extérieurs. Il détaille les postes de chacun.

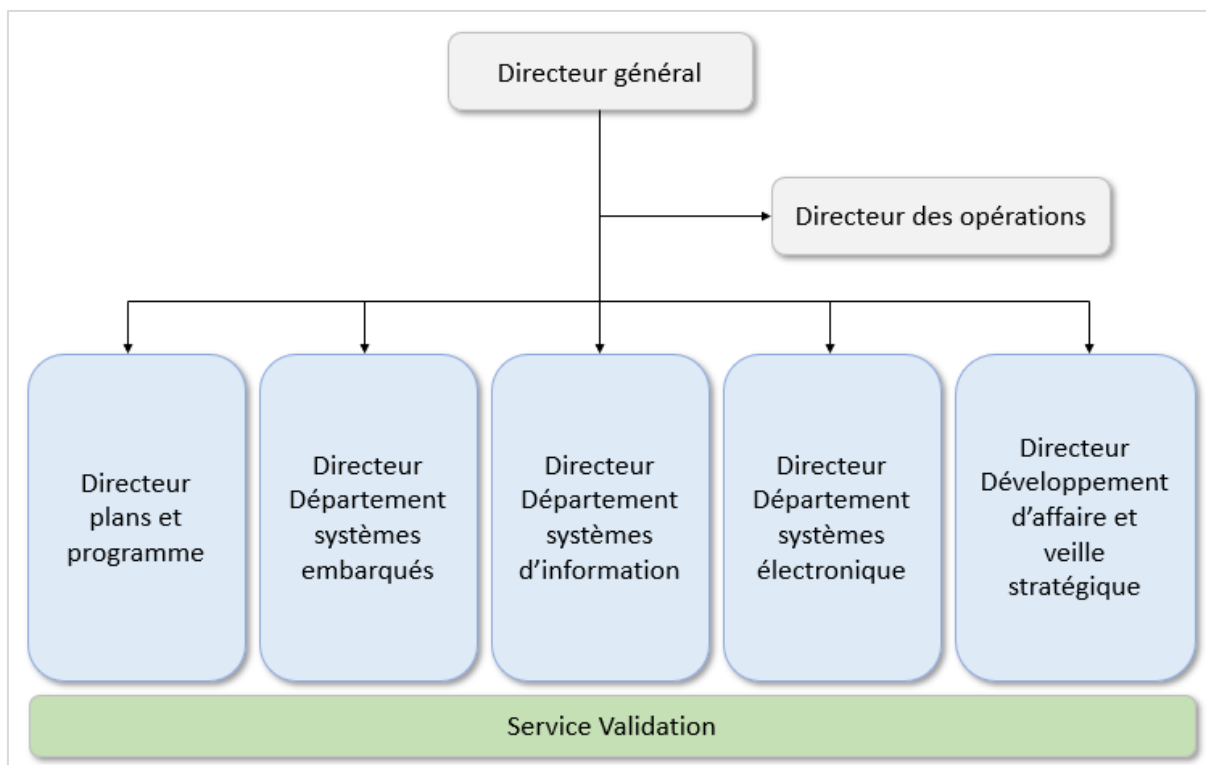


Figure 3: La structure hiérarchique de l'entreprise

II.2 Les produits de Sofia

Sofia Technologies a livré des projets innovants dans plusieurs secteurs d'activités :

– Phyt'eau :

« Phyt'Eau » est une solution innovante pour l'irrigation de précision basée sur des algorithmes développés par nos chercheurs permettant la simulation des besoins en eau d'une culture et la prédiction des doses d'irrigation en fonction de la culture, des conditions climatiques et de son stade de développement.



Figure 4: Logo de la solution phyt'eau

– **SENYA :**

Est une plateforme par les agronomes pour les agriculteurs permettant de réunir la gestion technique agronomique et la gestion financière et facilitant, ainsi, la planification, la surveillance et l'enregistrement de toutes les activités agricoles durant la saison de culture.



Figure 5: Logo de la plateforme Senya

– **Menzu :**

Menzu est une plateforme de développement universelle dédiée aux solutions connectées. Développée par les ingénieurs de SOFIA TECHNOLOGIES, elle a été conçue pour accompagner les clients et partenaires dans la transformation digitale de leurs processus industriels et répondre ainsi aux exigences de l'Industrie 4.0.

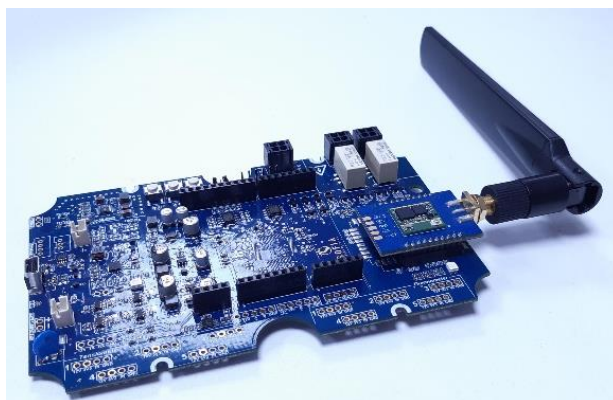


Figure 6: La vue de face de la carte Menzu

II.3 Les autres filiales de Sofia

Les autres filiales de Sofia Holding sont :

- **IFarming** : Start-up Tunisienne appartient au groupe Sofia Holding créée en 2017 et basée à Tunis. La société œuvre dans le domaine de l'agriculture, l'environnement et l'agro-alimentaire. IFarming a développé des applications web et mobile basées sur l'IoT et l'intelligence artificielle.

- **Sofia Europa:** Elle est spécialisée dans le secteur d'activité de l'édition de logiciels applicatifs.
- **Sofia Academy :** Est un centre de formation spécialisé dans les cours industriels et technologiques et qui fournit également des formations en compétences non techniques. Tous ses cours sont donnés par des professionnels hautement qualifiés.
- **Sater Solar:** Ayant pour ambition d'avoir une place dans l'histoire de la révolution énergétique en Tunisie, spécialisé dans la commercialisation et l'installation des panneaux photovoltaïques.

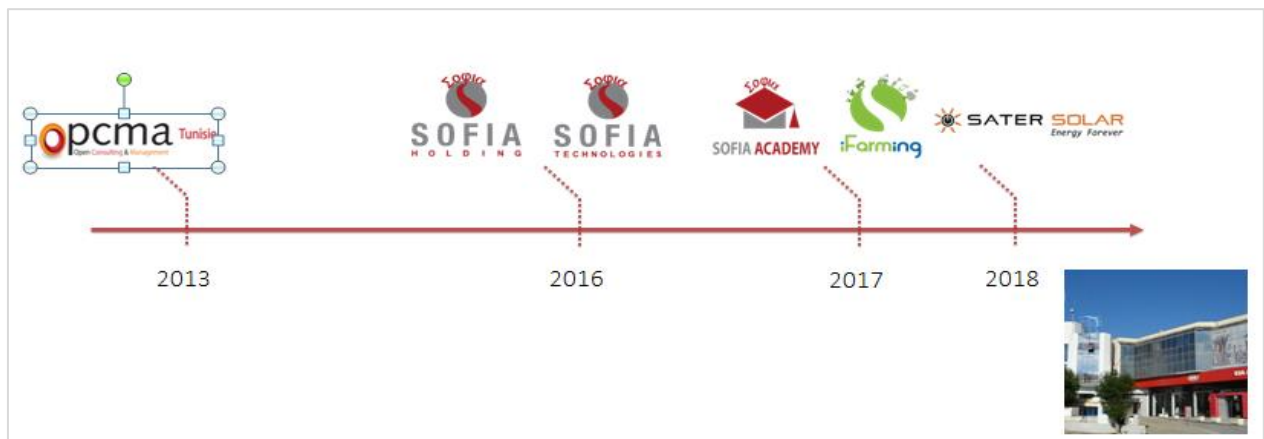


Figure 7: Les filiales de Sofia Holding

III. Problématique et solution envisagée

Certes le langage C a connu le succès grâce à la liberté nouvelle qu'il offrait aux programmeurs. Il produit des programmes rapides et une large collection d'APIs est disponible. Il reste le meilleur outil pour la programmation proche du hardware et s'utilise même pour des applications.

Mais comme tout langage, il avait plusieurs inconvénients comme le problème de sécurité « safety » telle que les bugs graves qui peuvent être causés par un simple manque d'attention du développeur, tel le dépassement de mémoire qui constitue une faille de sécurité informatique exploitable par les logiciels malveillants, ainsi la faible productivité du langage par rapport aux langages plus récents.

Pour pallier à l'insécurité du langage C, plusieurs langages prétendent à le remplacer comme le C++, JAVA et D, mais ils n'ont pas réussi, alors que RUST est un nouveau langage de programmation système qui offre une alternative pratique et sûre à C.

Rust est unique en son genre car il renforce la sécurité sans temps d'exécution, et surtout sans les frais généraux liés au ramassage des ordures (garbage collector). La sécurité à coût zéro est remarquable en soi.

Cependant, le système de type linéaire de Rust offre des fonctionnalités qui ne peuvent pas être mises en œuvre efficacement dans des langages traditionnels, sûrs ou non sécurisés, et qui améliorent considérablement la sécurité et la fiabilité des logiciels système.

Pour cette raison, SOFIA Technologies a proposé ce sujet dans le but d'introduire le nouvel langage RUST qui aide leurs développeurs à concevoir des applications ultra-rapides et sécurisées.

IV. Présentation de projet et objectifs visés par l'intervention

Dans le cadre de l'amélioration de la qualité de développement des produits logiciels, le département embarqué propose d'implémenter des drivers pour la carte Menzu de microcontrôleur « STM32L4xx » qui est l'une des produits de l'entreprise.

L'objectif est de gérer la communication entre les différents périphériques informatiques d'une façon sécurisée avec le langage de programmation RUST qui est connue par leur performance et leur fiabilité afin de concevoir des applications ultra-rapides et sécurisées.

V. Méthodologie de travail

Toute démarche de développement nécessite un modèle de développement définissant une méthode qui doit fournir des résultats fiables. Une telle méthode doit décrire une modélisation efficace et complète du système logiciel.

Le cycle de vie de notre projet comprend toutes les étapes depuis sa conception et sa réalisation jusqu'à sa mise en œuvre. L'objectif d'un tel découpage est de permettre de définir des jalons intermédiaires permettant la validation du développement du projet et la vérification de son processus de développement.

V.1 Choix de modèle d'organisation

Nous avons opté pour le modèle de **cycle de vie en V**. Ce choix revient au fait que ce cycle est le plus efficace avec son principe de travail qui nécessite la vérification de chaque étape et la possibilité de corriger les fautes avant de se lancer vers l'étape suivante.

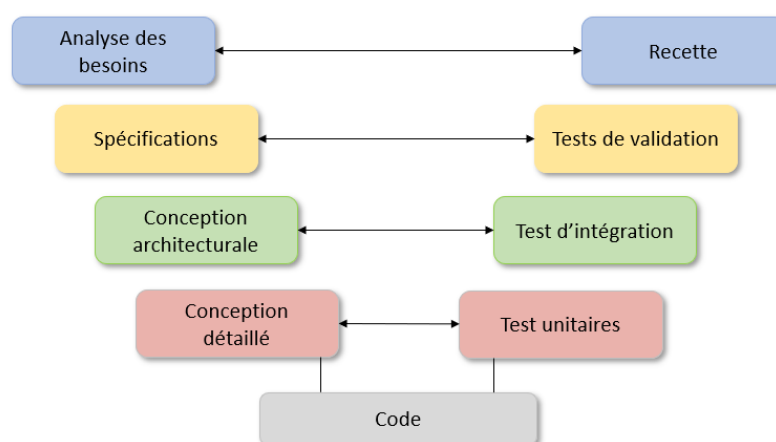


Figure 8: Cycle en V

V.2 Vue d'ensemble des activités de projet

Le cycle de vie du projet se traduit comme suit :

Une première série d'étapes, le flux descendant, vise à détailler le produit jusqu'à sa réalisation. Il comprend l'expression des besoins, l'analyse, la conception générale et détaillée, puis la mise en œuvre ou le développement qu'il s'agit d'une traduction des fonctionnalités définies dans la phase de conception en langage de programmation. Le flux ascendant assemble le produit en vérifiant sa qualité et performance.

VI. Travail demandé

Le présent travail vise à créer un ensemble de modules développés en langage Rust pour la carte Menzu.

Afin d'aboutir aux objectifs finaux du projet, nous avons fixé ces passages que nous jugeons nécessaires :

- Maîtriser les nouveaux concepts de programmation de Rust, ainsi que l'architecture et les fonctionnalités de la carte Menzu,
- Préparation de l'environnement de travail (l'IDE Eclipse et Rust) sur Windows,
- Conception de l'architecture de projet Rust,
- Développement des modules LEDs, Boutons, Relais, Timer, I2C, SPI, UART pour la carte Menzu,
- Implémenter et tester les modules sur la carte.

VII. Conclusion

Dans ce chapitre, tout d'abord nous avons présenté l'organisme d'accueil. Par la suite, nous avons posé la problématique qui nous a amené à la proposition de l'adoption du nouvel langage de programmation Rust. Puis nous avons décrit la méthodologie de travail adoptée qui est le cycle en V. Finalement nous avons fixé les tâches à effectuer.

CHAPITRE 2 :FAMILIARISATION AVEC L'ÉQUIPEMENT MATÉRIEL ET GÉNÉRALITÉ SUR LE LANGAGE RUST

I. Introduction

La phase préliminaire représente le point de départ pour tout projet informatique ce qui nous permet de comprendre les principaux concepts autour desquels tourne notre projet. Pour cela, nous allons consacrer ce chapitre pour la présentation de la carte embarquée Menzu ainsi que ses fonctionnalités et les spécificités du langage de programmation Rust.

II. Familiarisation avec la carte électronique embarquée Menzu

II.1 Définition

La carte Menzu illustrée dans la Figure 9 est conçue et fabriquée entièrement par la société Sofia technologies, C'est une plateforme de développement universelle dédiée aux mondes de l'IoT. Cette carte permet de surveiller plusieurs paramètres environnementaux impliquant une grande gamme d'applications. [2]

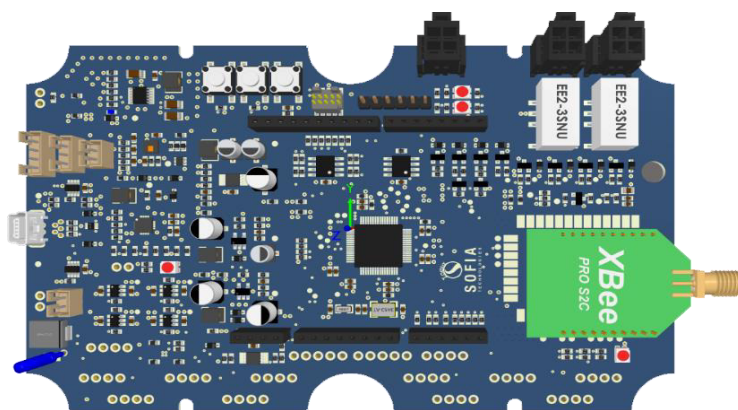


Figure 9: La vue de face de la carte Menzu

II.2 Caractéristiques de la carte

- La diversité d'application

Menzu peut être utilisé dans divers domaines et applications, allant de programmes simples tels que l'acquisition et l'envoi de signaux ou l'affichage de données, à des programmes plus complexes tels que l'agriculture intelligente et la gestion intelligente et stations météo.

- Microcontrôleurs supportés

L'un des grands avantages caractérisant la carte embarquée Menzu c'est qu'elle supporte les deux grandes familles de microcontrôleurs STM32 et ATMEL SAMD21. Actuellement, STM32L0, STM32L4, SAMD21-J17 et SAMD21-J18 sont les CPUs supportées par la carte Menzu.

Tableau 1: Les microcontrôleurs supportés par la carte Menzu

<i>Microcontrollers</i>	<i>Mémoire (en KO)</i>
STM32L0	192
STM32L4	512
Atmel-J17	256
Atmel-J18	512

- Options d'alimentation

Menzu a une gestion intelligente de l'alimentation, elle est alimentée par une batterie rechargeable au phosphate de lithium respectueuse de l'environnement. La batterie pourrait être chargée avec les panneaux photovoltaïques ou avec l'énergie du secteur.

Les batteries sont protégées contre les surtensions pour éviter toute surcharge ou tout endommagement.

- Connectivité, interfaces de communication séries et RF

Menzu est compatible avec les réseaux IOT : LoRaWAN et SigFox. Elle supporte aussi des modules de connectivité permettant un accès à internet ou réseaux locaux, par exemple ATWIN1500 (module Wifi) et SIM900(module GPRS). Les interfaces de communication séries disponibles sont RS485, I2C, SPI et UART.

Tableau 2: Les interfaces de communications supportées par la carte

<i>Communication</i>	<i>Modules et interface supportées</i>
LAN / WLAN	WiFi and GPRS
Communications series	I2C, SPI, UART, RS-458
Communications RF	Zigbee, Z-wave, BLE, Wireless M-Bus, LoRa

II.3 Les différentes composants (BSP)

La Figure 10 présente les différents composants de la carte Menzu que nous allons utiliser dans le développement de nos modules.

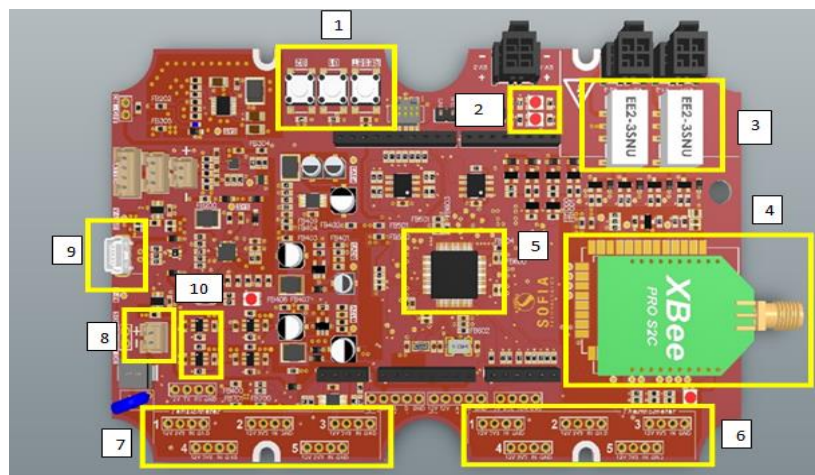


Figure 10: Les différents composants de la carte

Le Tableau 3 présente les noms des différents éléments intégrés dans la carte leurs nombres ainsi que leurs types

Tableau 3: Composants de la carte Menzu

Référence	Composants	Nombre	Type
1	Boutons poussoir	3	
2	Leds bicouleurs	2	LED RG
3	Relais	2	EE2-3SNU
4	Support Module ZigBee	1	S2D
5	Microcontrôleur (CPU)	1	STM32/SAMD21
6 - 7	Capteurs Analogiques	10	Thermomètre(6) Tensiomètre(7)
8	Port d'Alimentation	1	
9	Port USB	1	
10	Capteur de courant	1	INA

II.4 Les microcontrôleurs STM32L4xx :

La série L4 de STM32 est une série des microcontrôleurs 32 bits ultra basse puissance, basée sur le cœur ARM Cortex-M4 RISC haute performance fonctionnant à une fréquence allant jusqu'à 80MHz. Le cœur Cortex-M4 dispose d'une unité à virgule flottante (FPU) simple précision, qui prend en charge toutes les instructions de traitement de données ARM simple précision et les types de données. Elle met également en œuvre un ensemble complet d'instructions DSP et une unité de protection de mémoire (MPU) qui améliore la sécurité des applications.

Ce microcontrôleur dispose également des interfaces de communication standard et avancées.

- Trois I2C
- Trois SPI
- Trois USART, deux UART et un UART basse consommation.

II.4.1 Interface SPI (Serial Peripheral interface)

Comme la Figure 11 montre un esclave est connecté à un périphérique maître. L'horloge est générée par le dispositif maître pour la synchronisation du transfert de données. Il est également possible de connecter plusieurs périphériques esclaves avec un seul maître pour la communication.

L'interface SPI fonctionne en mode semi-duplex ou duplex intégral. SPI est la forme abrégée d'interface de périphérique série.[3]

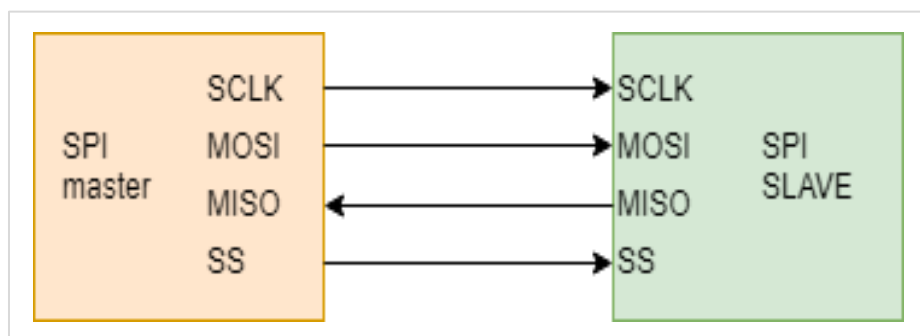


Figure 11: Le diagramme de l'interface SPI

Cette interface comporte quatre lignes principales :

- MOSI (*Master Output Slave Input*) : Il est utilisé pour transférer des données d'un périphérique maître à un périphérique esclave.
- MISO (*Master Input Slave Output*) : Il est utilisé pour transférer des données d'un périphérique esclave à un périphérique maître.
- SCLK (*Serial Clock*) : sortie d'horloge du maître et utilisée pour la synchronisation.
- SS (*Slave Select*) : Il est utilisé par le périphérique maître pour sélectionner un esclave parmi plusieurs esclaves. Il insère un signal actif bas pour sélectionner le périphérique esclave particulier.

II.4.2 Interface I2C (Inter-Integrated Circuit)

I2C c'est un bus de connexion de données série à faible vitesse et à deux fils utilisés dans les circuits intégrés. Il est utilisé pour transmettre des signaux entre des circuits intégrés montés sur la même carte.

Il n'utilise que deux lignes entre plusieurs maîtres et plusieurs esclaves (**figure 12**) :

- SDA(*Serial Data*)
- SCL(*Serial Clock*)

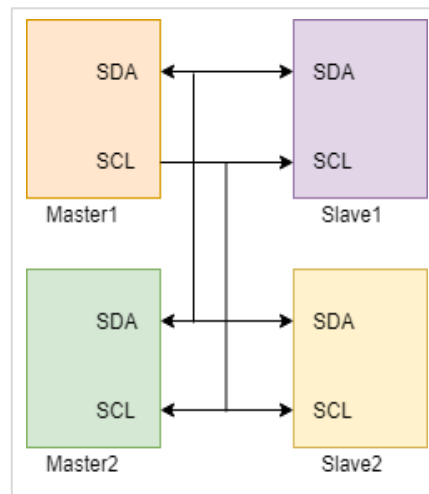


Figure 12: Diagramme de l'interface I2C

II.4.3 Interface UART(Universal Asynchronous Receiver Transmitter)

Il s'agit du principe le plus simple de liaison série. Cette liaison série fut l'une des premières utilisées pour communiquer avec différents périphériques à l'aide de deux fils uniquement, tel que le clavier, un terminal (écran), etc. Le fonctionnement de base, soit la transmission série des bits d'information, reste actuel. Toutes les transmissions à haut débit récente utilisent ce principe de transfert en série.

Il utilise le bit de début, le bit d'arrêt, le bit de parité dans son format de base pour le formatage des données. Le bit de parité facilite la détection d'erreur sur un bit.

De plus UART génère une horloge en interne et la synchronise avec le flux de données à l'aide de la transition du bit de démarrage

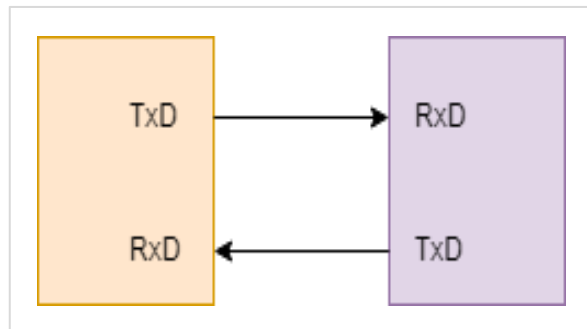


Figure 13: Le diagramme de l'interface UART

III. Généralités sur le langage de programmation (RUST)

III.1.L'historique de Rust

En 2006 Graydon Hoare, un développeur chez Mozilla, commence un projet personnel, un nouveau langage de programmation qu'il va nommer *Rust*. Trois ans plus tard, le langage est jugé suffisamment mature pour intéresser Mozilla, qui décide de le prendre sous son aile. Confrontés aux problèmes de sécurité du C++, notamment au niveau de la gestion de la mémoire, les ingénieurs en charge du développement souhaitaient utiliser un langage de programmation à la fois sécurisé, aux performances comparables au C++, et capable d'intégrer la gestion de la concurrence.

III.2. Définition

Rust est un langage de programmation système, compilé et multi paradigme. C'est un croisement entre langage impératif (C), objet (C++), fonctionnel (Ocaml) et concurrent (Erlang). Il s'inspire des recherches en théories des langages de ces dernières années et des langages de programmation les plus populaires afin d'atteindre trois objectifs : rapidité, sécurité (en mémoire notamment) et concurrent (partage des données sécurisé entre tâches).

Le langage RUST est créé pour rivaliser avec C++. Ainsi il été choisi afin d'assurer les mêmes performances que C++ mais avec un degré de sécurité plus important. Rust est inspiré du C++ mais aussi de la plupart des langages fonctionnels comme Haskell.

Donc il est amené à être plus rapide que C++ sachant qu'il l'est déjà à niveau de sécurité égale. Ainsi il peut compter sur une base de plus de 2000 bénévoles cherchant à améliorer le langage.



Figure 14: Logo de langage RUST

III.3 Avantages et limites :

➤ Les Avantages :

Rust est extrêmement rapide et utilise peu de mémoire : sans moteur d'exécution, il peut alimenter des services critiques en termes de performance, s'exécute sur des périphériques intégrés et s'intègre facilement à d'autre langage.

Ainsi Rust est très fiable il comporte des caractéristiques qui garantissent la sécurité de la mémoire et les threads, et il permet d'éliminer de nombreux bugs au moment de la compilation.

De plus, il possède une excellente documentation, un compilateur convivial avec des messages d'erreur utiles et un outillage de premier ordre, un gestionnaire de paquets intégré et un outil de construction, une prise en charge intelligente du multi-éditeur et des inspections de type, etc.

➤ Les Limites :

Malgré toutes ses qualités, Rust présente aussi des inconvénients qui déplaisent tout autant à ses fans qu'à ceux qui le découvrent.

Partant du fait que Rust est encore jeune et sa version 1.0 ne remonte qu'à 2015 et il y a encore pas mal de choses à ajuster comme les opérations asynchrones. Par exemple, qui ne sont pas bien représentées dans la syntaxe.

De plus il est difficile d'appréhender ses métaphores comme la propriété (ownership), et l'emprunt lorsqu'on les découvre et il nécessite plus du temps à le bien maîtriser et par conséquent il y aura moins de développeur.

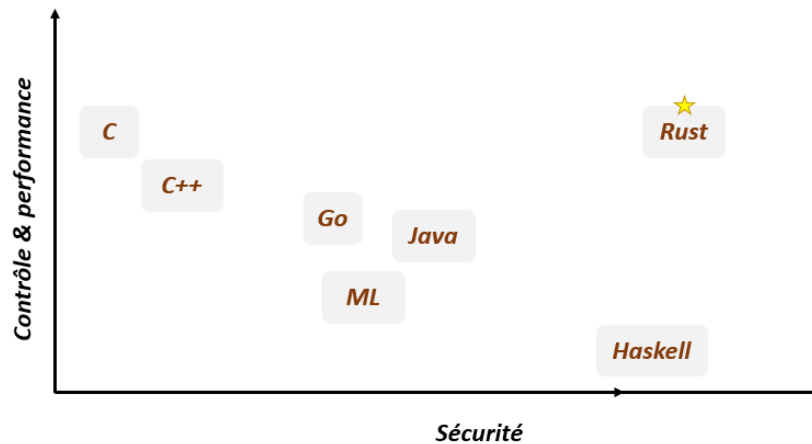


Figure 15: Rapport sécurité / contrôle des différents langages de programmation

III.4 Les concepts de programmation RUST

Rust repose sur des concepts connus et éprouvés et n'intègre pas de nouveaux concepts et non testés. Ces concepts ont été empruntés à des langages de programmation existants et assemblés dans un seul langage, parmi ces concepts on cite la mutabilité, Partage et propriété, Références...[4]

III.4.1 Gestion de mémoire (Pile et File)

Rust donne la possibilité de choisir la gestion de la mémoire adaptée au programme (pile, file) via l'utilisation de pointeurs intelligents, aussi la gestion de la concurrence intégrée dans le langage.

III.4.2 La sûreté par défaut

Cette sûreté est assurée par le typage statique sans conversion implicites et la validation statique d'accès mémoire par le compilateur.

Le compilateur de langage à typage statique détecte les erreurs de types avant que le programme ne soit exécuté (on obtient ainsi la sûreté du typage). Ainsi il peut tirer parti de l'information sur les types pour réaliser certaines optimisations du code.

Par exemple la tentative d'additionner un entier avec une chaîne de caractères.

III.4.3 Inférence de type (type inference)

Le moteur d'inférence de type est assez intelligent. Il examine comment la variable est utilisée par la suite pour déduire son type.

Exemple :

```
fn main() -> ! {  
    let i = 10 ; /* c'est un entier visiblement */  
    let max = 10i32 ;  
    if ( i max) {  
        println("i est inférieur à max"); /* max est un entier de 32 bits,  
                                           donc le compilateur en déduit que  
                                           i est aussi */  
    }  
}
```

Figure 16: Exemple d'inférence de type

III.4.4 Variables et mutabilité

Dans Rust, les variables sont par défaut immuables. C'est l'un des nombreux rappels que Rust nous propose d'écrire notre code de manière à tirer parti de la sécurité et de la facilité d'accès simultanés offerts par Rust. Cependant, il est toujours possible de rendre les variables mutables. Lorsqu'une variable est immuable, une fois qu'une valeur est liée à un nom, vous ne pouvez pas modifier cette valeur.

```
fn main() {  
    /* Creation d'un variable x immuable */  
    let x = 5;  
    println!("The value of x is: {}", x);  
    /* Modifier la valeur de x */  
    x = 6;  
    println!("The value of x is: {}", x);  
}
```

Figure 17: Exemple d'une création d'une variable immuable

La compilation de ce code provoque un message d'erreur parce que nous avons essayé d'attribuer une deuxième valeur à la variable immuable `x`.


```
error[E0384]: cannot assign twice to immutable variable `x`
--> src/main.rs:4:5
  |
2 |     let x = 5;
  |         - first assignment to `x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

Figure 18: Résultat de compilation

Nous pouvons les rendre mutables en ajoutant **mut** devant le nom de la variable (x). En plus de permettre à cette valeur de changer, mut transmet l'intention aux futurs lecteurs du code en indiquant que d'autres parties du code modifieront cette valeur de variable.

III.4.5 Partage et propriété (Ownership)

La propriété est la caractéristique unique de Rust et permet à Rust de garantir la sécurité de la mémoire sans avoir besoin d'un récupérateur de place. Par conséquent, il est important de comprendre le fonctionnement de la propriété à Rust, ainsi que de plusieurs fonctionnalités connexes : l'emprunt (*borrowing*), les tranches(*slices*)et la façon dont Rust dispose les données en mémoire.

En développement nous avons gardé des règles à l'esprit pendant que nous travaillons à travers le projet qui les illustre :

- Chaque valeur dans Rust a une variable appelée son propriétaire.
- Il ne peut y avoir qu'un seul propriétaire à la fois.
- Lorsque le propriétaire sort de la portée, la valeur est supprimée.

Prendre possession de chaque fonction, puis en revenir à chaque fonction est un peu fastidieux. Et si on veut laisser une fonction utiliser une valeur sans en prendre possession ? C'est assez énervant que tout ce que nous transmettons doit être renvoyé si nous voulons l'utiliser à nouveau, en plus des données résultant du corps de la fonction que nous pourrions également vouloir renvoyer.

Mais c'est trop de cérémonie et beaucoup de travail pour un concept qui devrait être commun. Heureusement pour nous, Rust possède une fonctionnalité pour ce concept, appelée références.

III.4.6 Références et emprunts

Rust remédie à un système d'emprunts créant en quelque sorte des mutex chargés de limiter l'accès à une ressource et ainsi éviter les risques d'écriture simultanée. L'emprunt fera respecter ces trois règles :

- Une (ou plusieurs) variable peut emprunter la ressource en lecture. (Référence immuable)
- Un seul et seulement un, pointeur doit disposer d'un accès en écriture sur la ressource c'est-à-dire une seule référence mutable.
- Vous ne pouvez pas accéder à la ressource en lecture et en écriture en même temps par exemple :

```
fn main() {  
    let s1 = String::from("hello");  
  
    /* Fait la référence à la valeur de s1 */  
    let len = calculate_length(&s1);  
  
    println!("The length of '{}' is {}. ", s1, len);  
}  
  
fn calculate_length(s: &String) -> usize {  
    s.len()  
}
```

S sort du champ de l'application mais comme il ne possède pas ce à quoi il fait référence, rien ne se passe

Figure 19: Exemple de Références et emprunts

III.4.7 Durée de vie (Lifetime)

Chaque référence dans Rust a une durée de vie, c'est-à-dire la portée pour laquelle cette référence est valide. Le compilateur Rust a un vérificateur d'emprunt qui compare les étendues pour déterminer si tous les emprunts sont valides. Rust apporte également beaucoup d'autres fonctionnalités qui sont très importantes dans la programmation système comme la généricité avec les traits, le filtrage par motif, et les pointeurs intelligents.

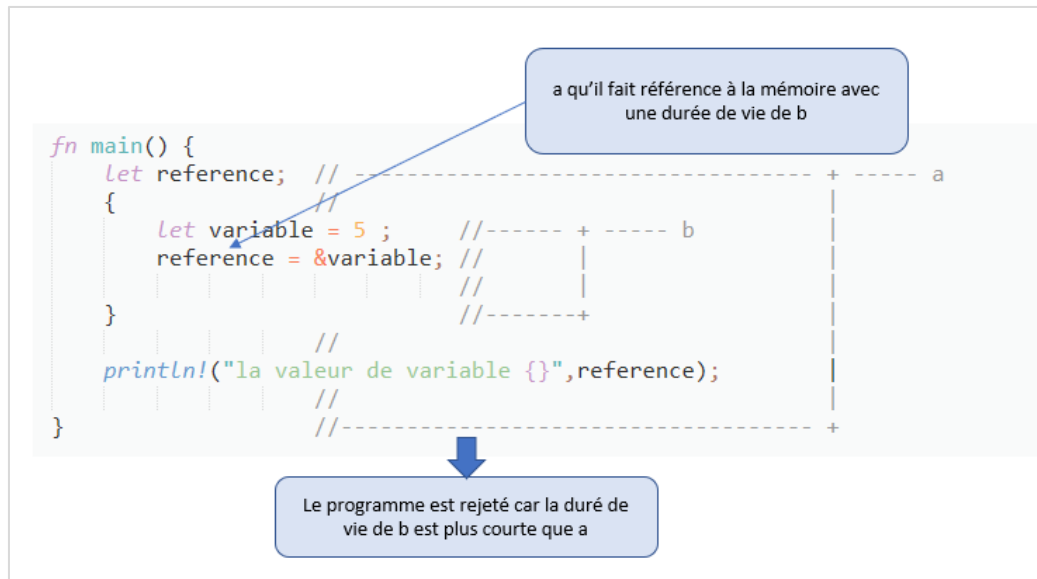


Figure 20: Exemple montrant la Durée de vie

IV. Conclusion

À travers cette étude préliminaire, nous avons non seulement présenté un aperçu général du projet concerné, mais également défini les éléments clés qui nous guideront vers sa réalisation.

La préparation de l'environnement matériel et logiciel de projet sera l'objectif du prochain chapitre.

CHAPITRE 3 :INSTALLATION DE L'ENVIRONNEMENT ET CRÉATION D'UN PROJET RUST EMBARQUÉ

I. Introduction

Après la phase préliminaire du travail demandé, nous entamons la réalisation du projet. Cette tâche a comme but l'achèvement d'un système répondant aux exigences citées au niveau de travail demandé.

Afin de réussir cette tâche, il est tout d'abord nécessaire de préparer l'environnement de travail pour la création des projets embarqués.

II. Environnement logiciel

II.1 Configuration de l'environnement Rust sous Windows

II.1.1 La chaîne de compilation Rustup

En système embarqué, une chaîne de compilation (Toolchain) désigne l'ensemble des paquets utilisés dans le processus de compilation d'un programme pour un processeur donné. Le compilateur n'est qu'un élément de cette chaîne, laquelle varie selon l'architecture matérielle cible [1].

Rustup est un multiplexeur de chaînes de compilation qui fournit des mécanismes permettant d'installer, gérer et changer facilement de nombreuses chaînes d'outils actifs en reconfigurant le comportement de ces propriétés.

Pour installer Rustup sous Windows, Nous avons accédé à la page officiel <https://www.rust-lang.org/tools/install> et suivre les instructions d'installation. À un moment donné, un message expliquant que l'installation nécessite des outils de génération C ++ pour Visual Studio 2013 ou version ultérieure.

Le moyen le plus simple d'acquérir les outils de génération consiste à installer les outils de génération de Visual Studio 2019.

A screenshot of a text box with a light gray background. The text inside reads: "To install Rust, download and run rustup-init.exe then follow the onscreen instructions." The text "rustup-init.exe" is highlighted in blue.

To install Rust, download and run
rustup-init.exe
then follow the onscreen instructions.

Figure 21: Lien de l'installation de Rustup

Ensuite, nous choisissons l'installation par défaut (option 1). Comme indiqué dans la capture d'écran, il installera une Rust stable pour MSVC ABI.

```
Current installation options:

  default host triple: x86_64-pc-windows-msvc
  default toolchain: stable
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
_
```

Figure 22: Choix de l'option de l'installation de Rust stable

▪ Mettre à jour Rustup

Avec Rustup installé, la majeure partie de la gestion de la chaîne d'outils est effectuée en ligne de commande avec la commande Rustup. Tout d'abord, permet de tout mettre à jour pour être sûr

```
> rustup update
```

Ensuite, nous ajoutons la version Nightly du compilateur pour essayer des fonctionnalités expérimentales et pour pouvoir compiler un outil utile.

```
> rustup install nightly
```

II.1.2 RLS et composants

RLS est essentiel et constitue le composant principal, mais il a ses propres exigences : le code source Rust, la documentation et le composant d'analyse.

Fondamentalement, RLS permet la complétion automatique, les info-bulles de documentation et les fonctionnalités d'édition de code associées à tout IDE prenant en charge le protocole Microsoft Language Server.

C'est en quelque sorte l'équivalent de C++ IntelliSense ou de Visual Assist.

```
> rustup default nightly  
> rustup component add rust-src  
> rustup component add rust-docs  
> rustup component add rust-analysis  
> rustup component add rls-preview  
> rustup update
```

II.1.3 Rustfmt

Rustfmt formatera automatiquement le code conformément à la norme de style de code actuelle. Opposé à des langages tels que C ou C++ où chaque base de code a son propre style, Rust (et pour l'essentiel, les langages les plus récents) applique un style standard au code.

Pour l'installer, il suffit d'utiliser Cargo :

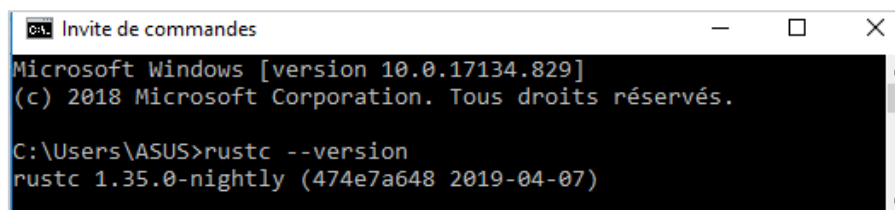
```
> cargo install rustfmt
```

II.1.4 Vérification de l'installation

Pour vérifier si Rust est correctement installé, nous ouvrons un Shell et nous entrons cette ligne :

```
$ rustc --version
```

Le résultat affiche le numéro de la version, le hachage de validation et la date de validation de la dernière version stable publiée dans le format suivant :



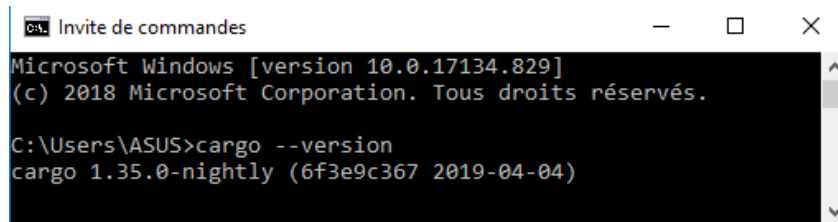
```
Microsoft Windows [version 10.0.17134.829]  
(c) 2018 Microsoft Corporation. Tous droits réservés.  
  
C:\Users\ASUS>rustc --version  
rustc 1.35.0-nightly (474e7a648 2019-04-07)
```

Figure 23: La commande de demande de version de compilateur rustc

Cargo est un outil de construction utilisé pour gérer nos packages et projets Rust.

La plupart des développeurs utilisent cet outil pour gérer leurs projets Rust, car Cargo gère des nombreuses tâches, telles que la construction de notre code, le téléchargement des

bibliothèques dont dépendent notre code et la construction de ces bibliothèques. Pour nous assurer que ceci est également présent sur notre ordinateur, nous exécutons :



```
Microsoft Windows [version 10.0.17134.829]
(c) 2018 Microsoft Corporation. Tous droits réservés.

C:\Users\ASUS>cargo --version
cargo 1.35.0-nightly (6f3e9c367 2019-04-04)
```

Figure 24: La commande de demande de version de cargo

II.1.5 Les commandes de cargo

- Cargo build : Cette commande compile le paquet.
- Cargo check : Cette commande vérifie les erreurs d'un paquet local et de toutes ses dépendances.
- Cargo clean : Supprime les artefacts générés par Cargo dans le passé.
- Cargo run : Exécute un binaire ou un exemple du package local.

II.2 Choix de l'IDE : l'Eclipse corrosion 2018

Eclipse IDE est un environnement de développement de bureau très populaire téléchargé plus de 2 millions de fois par mois. Il constitue l'environnement de développement essentiel de plus de 4 millions d'utilisateurs actifs. La version de corrosion peut être utilisée pour créer, exécuter et emballer des applications Rust avec l'intégralité de l'IDE Eclipse.

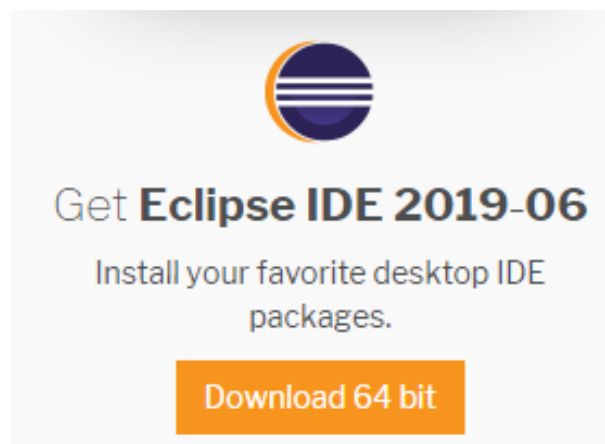


Figure 25: La version de l'Eclipse

II.2.1 Installation de l'Eclipse

Corrosion est un plug-in de développement Rust pour l'IDE Eclipse, offrant une expérience d'édition enrichie grâce à l'intégration avec Langage Rust et Cargo. Disponible au téléchargement sous forme de package Eclipse complet.

On télécharge le package Eclipse IDE pour les développeurs de Rust. Avec l'installation de plugins requis, c'est le moyen le plus simple de commencer à développer des projets Rust. Java doit être installée pour exécuter cet IDE.

II.2.2 Configuration des préférences de l'Eclipse

Lorsque nous accédons aux préférences nous cliquons sur Rust pour spécifier le chemin d'accès :

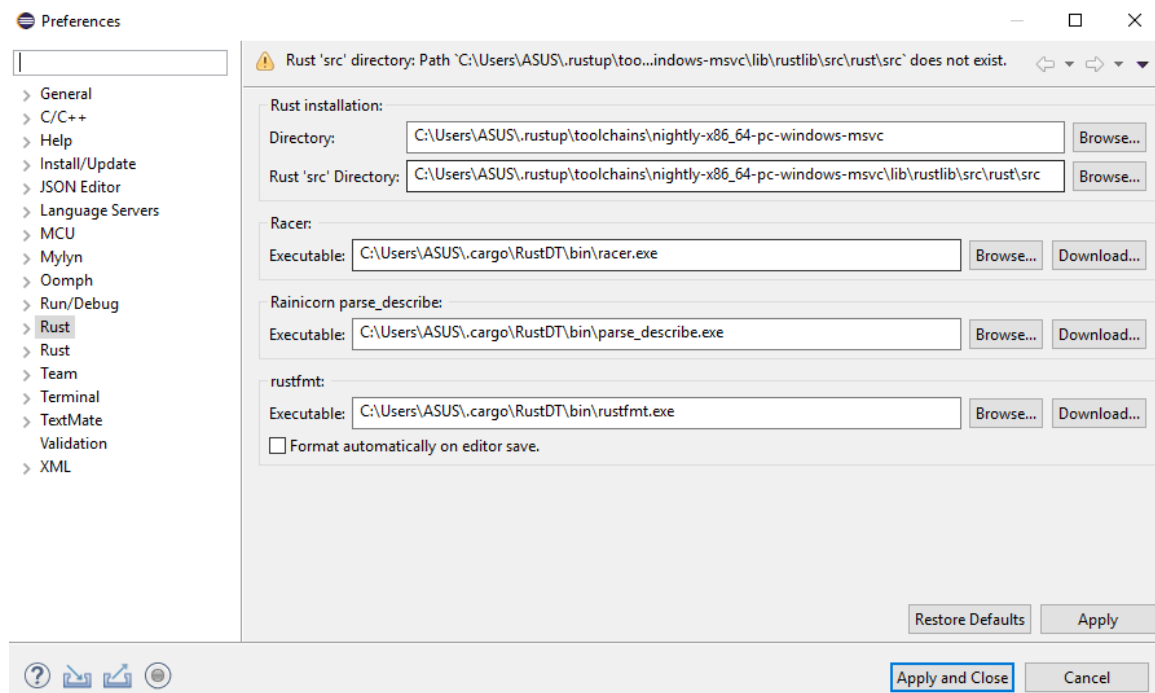


Figure 26: Spécification des chemins d'accès de dossier Rust

II.2.3 Installation et configuration de plug-in Eclipse de Débogage OpenOcd

Par défaut, Eclipse prend en charge OpenOCD via le plug-in GDB Hardware Debugging qui lance OpenOCD.

GDB ne peut pas communiquer directement avec le matériel de débogage ST-Link de la carte de développement MENZU de microcontrôleur STM32L4A6. Il a besoin d'un traducteur, OpenOCD(Open On-Chip) est ce traducteur.

OpenOCD: Est un programme qui s'exécute sur l'ordinateur portable / PC et qui traduit le protocole de débogage distant basé sur TCP / IP de GDB et le protocole basé sur USB de ST-Link et il effectue également d'autres tâches importantes dans le cadre de la traduction du débogage du microcontrôleur ARM Cortex-M sur notre carte de développement comme :

- La manipulation des points d'arrêts (breakpoints) / points d'observation
- La lecture et l'écriture des registres de la CPU
- La détection quand le CPU a été arrêté pour un événement de débogage
- La poursuite de l'exécution de la CPU après qu'un événement de débogage a été rencontré

II.2.4 Utilisation du plug-in OpenOCD

Toutes les plateformes, GNU Eclipse OpenOCD, sont publiées sous forme d'archives portables. Les versions Windows de GNU MCU Eclipse Openocd sont compressées sous forme de fichiers ZIP, nous avons téléchargé la version :

gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-win64.zip.

Avant de commencer à travailler avec OpenOCD, il faut définir le chemin d'accès au dossier contenant les exécutables de OpenOCD.

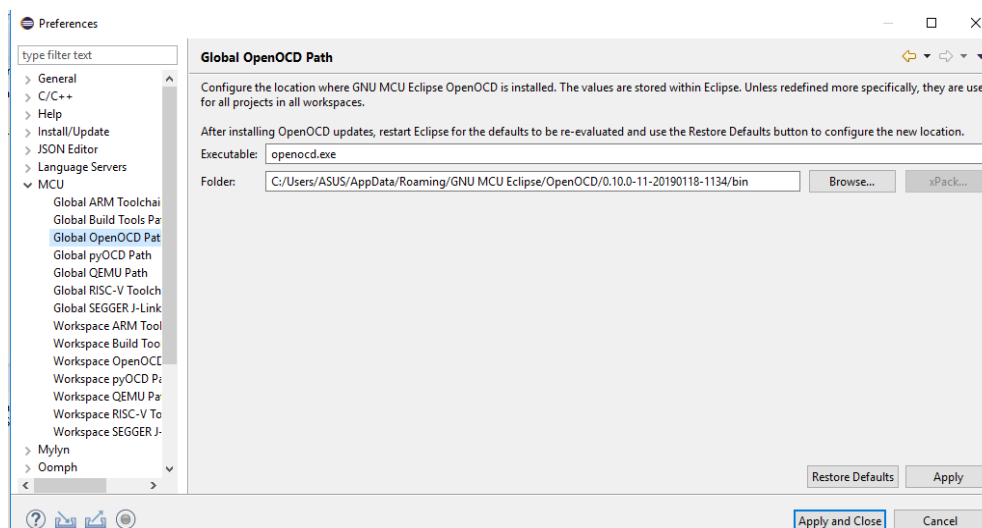


Figure 27: Spécification de chemin d'accès de dossier l'OpenOCD

II.3 Limitation de débogage de Rust sous Eclipse

L'Eclipse est un outil très puissant et prisé et ce qui a lui rendu plus populaire auprès des développeurs c'est sa modularité. Encore, grâce à un modèle de développement libre, chacun peut développer ses propres modules et les intégrer à sa plateforme Eclipse. Mais aussi il connut des limitations, avec l'Eclipse corrosion nous ne pouvons pas debugger un projet Rust.

II.4 La solution proposée

Tout d'abord nous avons besoin d'une instance Eclipse avec les outils de développement c/c++ installés. Puis nous devons dire à Eclipse que les fichiers .rs sont des fichiers sources valides à afficher dans le débogueur et nous ajoutons le modèle *.rs comme fichier.

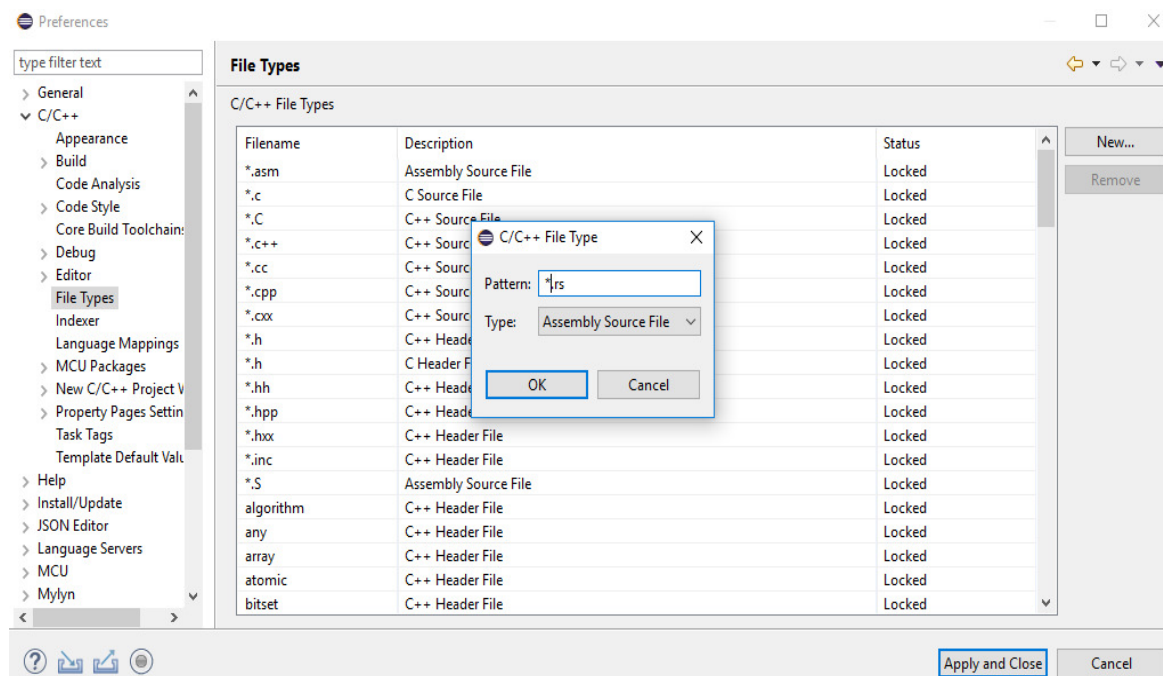


Figure 28: Création de fichier *.rs

Afin de compiler un programme avec les informations de débogage, nous utilisons la commande `rustc-zextra-debug-info./*.rs`. Cela produira un exécutable dans le même répertoire que le fichier source. Pour l'exécuter sous Eclipse, nous ouvrons le **Debug Configuration** menu et nous créons une nouvelle application C/C++.

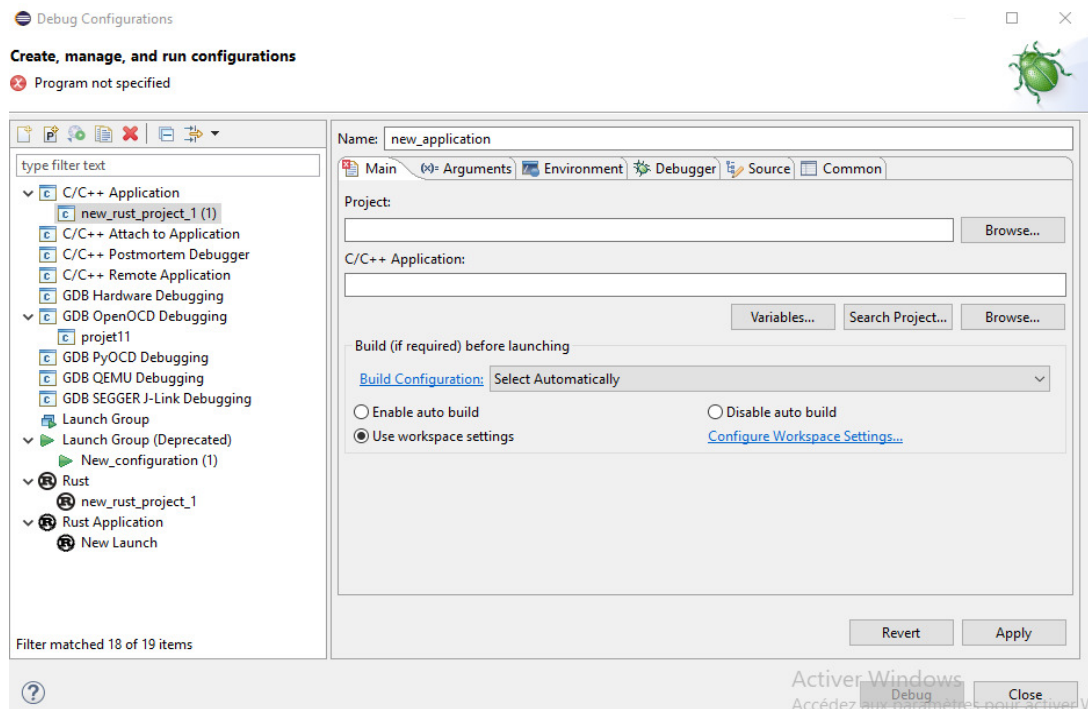


Figure 29: L'exécutable dans une nouvelle application C/C++

III. Création d'un projet Rust embarqué

III.1 Arborescence exemplaire

Cargo est un gestionnaire des projets Rust qui utilise des conventions pour le placement des fichiers afin de faciliter la plongée dans un nouveau package. L'exécution de la commande **cargo new** génère deux fichiers et un répertoire pour nous : un fichier **Cargo.toml** et un répertoire **src** contenant un fichier **main.rs**. Donc, l'arborescence d'un projet Rust embarqué va être présentée comme suit :

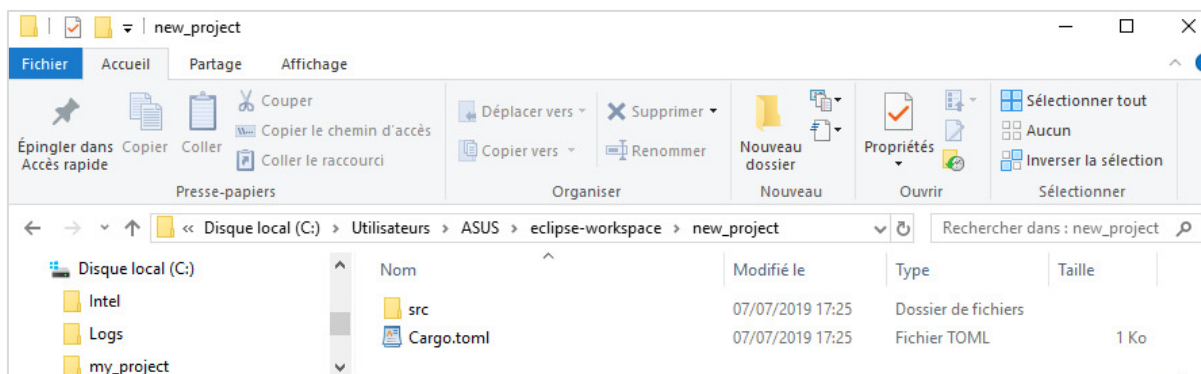
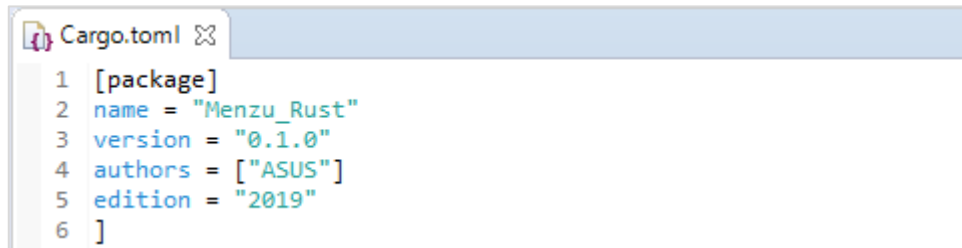


Figure 30: Création d'un nouveau projet Rust

Pour un projet Rust dédié pour un système embarqué, nous avons chargé d'ajouter des fichiers supplémentaires qui sont **Build.rs**, **Memory .x**, **Openocd.cfg** et **Openocd.gdb**.

- **Cargo.toml** :

Ce fichier est dans le format TOML (Tomnuident, Langage Minimal), qui est le format de configuration de Cargo.



```
1 [package]
2 name = "Menzu_Rust"
3 version = "0.1.0"
4 authors = ["ASUS"]
5 edition = "2019"
6 ]
```

Figure 31: Le package du fichier cargo.toml

[Package] est un en-tête de section indiquant que les instructions suivantes configurent un package. Les quatre lignes suivantes définissent les informations de configuration qu'avec Cargo doit compiler le code : le nom, la version, l'auteur et l'édition de Rust utilisé.

[dependencies] est le début d'une section dans laquelle nous allons répertorier les dépendances de notre projet. Dans Rust, les paquets de code sont appelés des **Crates**.



```
8 [dependencies]
9 cortex-m-rt = "0.6.8"
10 cortex-m-semihosting = "0.3.3"
11 panic-semihosting = "0.5.2"
12 cortex-m = "0.5.8"
13 nb = "0.1.1"
14 stm32l4 = "0.7.0"
```

Figure 32: Les dépendances de projet

- **Cargo.lock**

Contient des informations exactes sur les dépendances. Il est géré par Cargo et ne doit pas être modifié manuellement.

- **Build.rs :**

Nous créons ce fichier à la racine de notre caisse. Cargo détectera l'existence de ce fichier, puis le compilera et l'exécutera avant que le reste de la crate ne soit construit. Ceci est utilisé pour générer du code au moment de la compilation.

- **Memory.x :**

C'est un fichier qui fournit les informations sur le périphérique à l'éditeur de liens. Les informations principales que ce fichier doit fournir sont la structure de la mémoire du périphérique sous la forme de la commande MEMORY

Après avoir terminé la création de projet nous allons passer à la partie de débogage

- **OpenOCD :**

OpenOCD nécessite deux fichiers d'interface préconfigurés pour chaque adaptateur de débogage utilisé.

- **Openocd.cfg :** Dans ce fichier on indique l'interface de la carte.
- **Openocd.gdb :** OpenOCD soutient la gdb, cela permet au serveur distant GDB d'envoyer des informations à GDB.

IV. Conclusion

Tout le long de ce chapitre, nous avons abordé la phase de l'installation de l'environnement logiciel qui va nous permet de créer une arborescence complète de projet de développement des modules.

CHAPITRE 4 : DÉVELOPPEMENT ET TESTS

I. Introduction

Le développement des projets embarqués sur des cartes électroniques nécessite l'existence de tous les fichiers que nous avons cités dans le chapitre précédent.

Pour nous familiariser au développement des projets nous allons opter pour un exemple de clignotement des LEDs sur la carte STM32L429_disc. Ensuite nous allons réaliser le développement des différents modules mentionnés au niveau de la partie de travail demandé.

II. Développement d'un exemple : Clignotement de LED de la carte stm32f429

Pour se familiariser de l'environnement de Rust et de l'Eclipse nous avons commencé par tester le clignotement de LED de la carte stm32f429 car c'est une carte standard et toutes ses crates sont disponible sur le site crates.io.

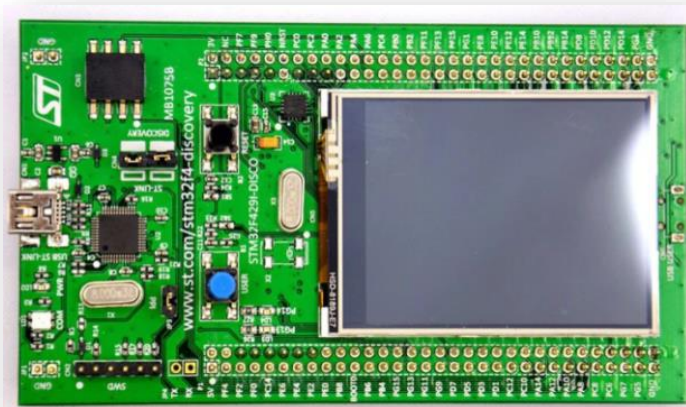


Figure 33: La carte STM32F429

II.1 Création de projet

Tout d'abord, nous initialisons un nouveau projet Cargo en exécutant les commandes suivantes :

```
>cargo init stm32f429-blink
```

```
>cd stm32f429-blink
```

Ensuite, nous modifions le fichier **Cargo.toml** comme suit pour inclure les bibliothèques requises :



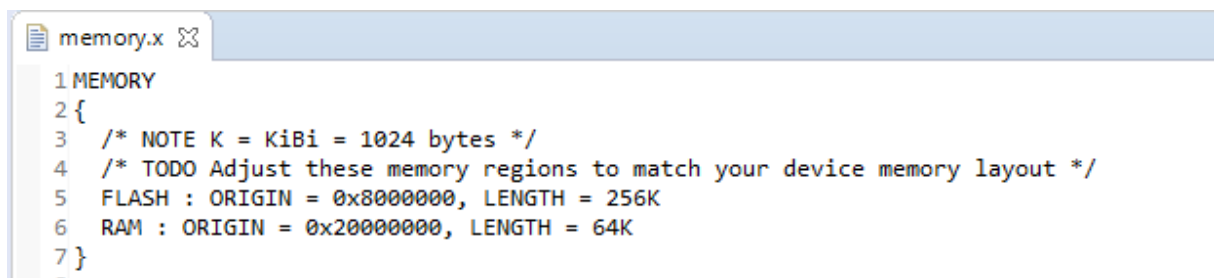
```

1 [package]
2 name = "STM32F4_Blink"
3 version = "0.1.0"
4 authors = ["ASUS"]
5 edition = "2019"
6
7 [dependencies.cast]
8 version = "0.2.2"
9 default-features = false
10
11 [dependencies.stm32f429i-disc]
12 version = "0.2.0"
13
14 [dependencies.embedded-hal]
15 version = "0.2.1"
16
17 [dependencies.cortex-m]
18 version = "0.5.7"
19
20 [dependencies.cortex-m-rt]
21 version = "0.6.4"
22
23 [dependencies.panic-abort]
24 version = "0.3.1"
25

```

Figure 34: Le contenu du fichier cargo.toml

La caisse cortex-m-rt nécessite également un fichier **memory.x**, qui spécifie la configuration de la mémoire de la carte. Il suffit donc de créer le fichier memory.x avec le contenu suivant :



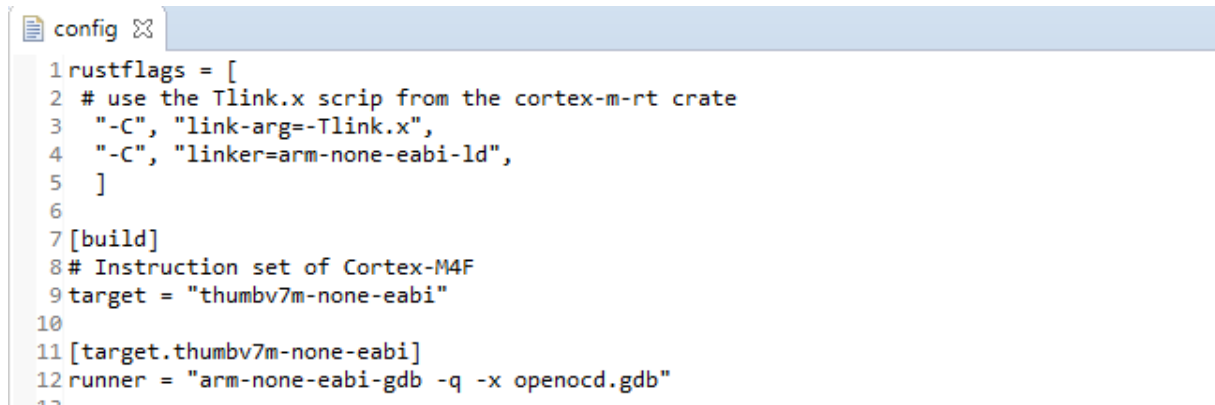
```

1 MEMORY
2 {
3     /* NOTE K = KiBi = 1024 bytes */
4     /* TODO Adjust these memory regions to match your device memory layout */
5     FLASH : ORIGIN = 0x8000000, LENGTH = 256K
6     RAM : ORIGIN = 0x20000000, LENGTH = 64K
7 }
8

```

Figure 35: Le contenu du fichier memory.x

Nous pouvons maintenant compiler le programme en utilisant cargo rustc --target thumbv7m-none-eabi -C link-arg = -Tlink.x, mais ce serait un peu gênant. Par conséquent, nous créons le fichier .cargo / config (et le dossier .cargo) avec le contenu suivant (basé sur ce fichier):



```
1 rustflags = [
2   # use the Tlink.x scrip from the cortex-m-rt crate
3   "-C", "link-arg=-Tlink.x",
4   "-C", "linker=arm-none-eabi-ld",
5 ]
6
7 [build]
8 # Instruction set of Cortex-M4F
9 target = "thumbv7m-none-eabi"
10
11 [target.thumbv7m-none-eabi]
12 runner = "arm-none-eabi-gdb -q -x openocd.gdb"
```

Figure 36: Le fichier de configuration

Nous avons utilisé thumbv7m-none-eabi qui couvre le noyau Cortex-M4F de la carte stm32f4.

Maintenant le projet est disposé d'être compilé et génère comme résultat un fichier d'extension elf mais le code n'est pas encore développé.

II.2 Développement et code : src/main.rs

II.2.1 Dépendances et inclusion des « Crates »

Nous commençons par définir les caisses externes (crates) qui sont des déclarations qui spécifient les dépendances ensuite elles seront liées à la portée en tant qu'identificateur fourni dans la déclaration.

```
#![no_main]
#![no_std]

extern crate cortex_m;
extern crate cortex_m_rt;
extern crate panic_abort;

extern crate stm32f429i_disc as board;
extern crate embedded_hal as hal;
```

Figure 37: Les caisses externes importées

#! [no_std] est un attribut au niveau de la crate qui indique que la crate se liera à la crate principale au lieu de la crate standard. La crate libcore est à son tour un sous-ensemble de la crate std ne tenant pas compte de la plate-forme, qui ne fait aucune hypothèse concernant le système sur lequel le programme sera exécuté.

#! [no_main] indique que ce programme n'utilisera pas l'interface main standard utilisée par la plupart des programmes Rust. La raison principale d'aller avec no_main est que l'utilisation de l'interface main dans le no_std contexte nécessite Nightly.

▪ Les crates externes

Cortex_m : La crate qui fournit une API pour utiliser les fonctionnalités communes à tous les microcontrôleurs Cortex-M.

Cortex_m_rt : La crate qui démarre le périphérique, initialise la mémoire vive puis appelle la main. Il fait tout cela implicitement.

Panic_abort : c'est une panique qui provoque l'exécution de l'instruction d'abandon.

Stm32f429i_disc : c'est une crate qui contient un package de support de carte de base pour la carte de microcontrôleur stm32f429i_disc permettant d'écrire des firmwares en utilisant le langage Rust.

stm32f429-hal

crates.io v0.0.0

HAL for the STM32F429 family of microcontrollers, forked from the one for STM32F30x

Documentation Repository

↓ All-Time: 588

↓ Recent: 100

Figure 38: La caisse dédiée pour la carte stm32f429

Embedded_hal : Une couche d'abstraction matérielle (HAL) pour les systèmes embarqués.

Il s'agit d'un ensemble de caractéristiques qui définissent les contrats d'implémentation entre les **implémentations**, les **pilotes** et les **applications (ou firmwares) de HAL**.

Ces contrats incluent à la fois des fonctionnalités (par exemple, si un trait est implémenté pour un certain type, l'**implémentation de HAL** fournit une certaine capacité) et des méthodes.

II.2.2 Configuration de GPIO de la carte

Nous activons le port G de la carte sur lequel il existe les pins des LEDs.

```
#[entry]
fn main() -> ! {
    if let (Some(p), Some(cp)) = (stm32::Peripherals::take(), Peripherals::take()) {
        let gpiog = p.GPIOG.split();
```

II.2.3 Configuration des pins des LEDs

Nous configurons les deux LEDs vert et orange sur les pins pg13 et pg14 de port G.

```
// Configure LED outputs
let mut leds = Leds {
    green: gpiog.pg13.into_push_pull_output(),
    orange: gpiog.pg14.into_push_pull_output(),
};
```

II.2.4 Configuration de l'horloge

Nous configurons le registre rcc (Reset and clock control) de l'horloge et nous réglons la fréquence à 168MHZ :

```
// Constrain clock registers
let rcc = p.RCC.constrain();

// Configure clock to 168 MHz (i.e. the maximum) and freeze it
let clocks = rcc.cfgr.freeze();
let clocks = rcc.cfgr.sysclk(168.mhz()).freeze();
```

Puis nous définissons un quantum du temps DELAY durant lequel les LEDs s'allument ou s'éteindraient :

```
//let mut delay = Delay::new(cp.SYST, clocks);
let mut delay = Delay::new(cp.SYST, clocks);
```

II.3 Implémentation et test

Maintenant notre projet est disposé d'être compilé, la commande de compilation est **cargo build** qui compile les packages locaux et toutes leurs dépendances :

Ensuite pour faire connecter la carte avec l'ordinateur on a besoin d'un débogueur ST-LINK/V2 qui est un programmeur en circuit pour les familles de microcontrôleurs STM8 et STM32.

Les interfaces SWIM (Single Wire Interface Module) et SWD (JTAG / serial Wire Debugging) sont utilisées pour communiquer avec tout microcontrôleur STM8 ou STM32 situé sur une carte d'application.

```
C:\Users\Kaabia\Rust_examples\stm32f429-blink>cargo build
Finished dev [unoptimized + debuginfo] target(s) in 0.18s

C:\Users\Kaabia\Rust_examples\stm32f429-blink>openocd
GNU MCU Eclipse 64-bits Open On-Chip Debugger 0.10.0+dev-00404-g20463c28 (2018-01-23-12:30)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v17 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.776736
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Enfin, nous exécutons la commande *openocd* pour assurer la communication entre le pc et la carte :

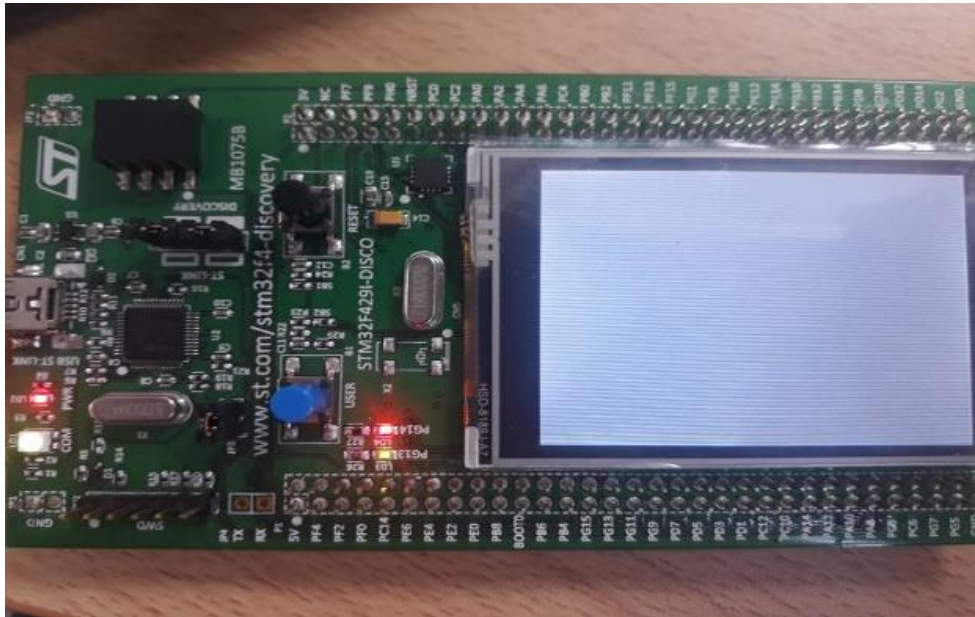


Figure 39: Le clignotement des LEDs

III. Développement de projet

III.1 Arborescence retenue

Maintenant, Pour le développement des modules avec le langage Rust qui sont par analogie des drivers en langage C, nous allons présenter l'architecture globale de notre projet.

Tout d'abord, nous avons choisi la carte à utiliser pour le développement des modules qui est la carte Menzu de microcontrôleur STM32L4xx puisqu'elle dispose des crates dans le site docs.rs pour qu'on puisse développer nos modules contrairement à la carte Menzu de microcontrôleur atsamdj17 ou bien atsamdj18 qui sont en cours de développement

L'architecture de projet est présentée comme suit :

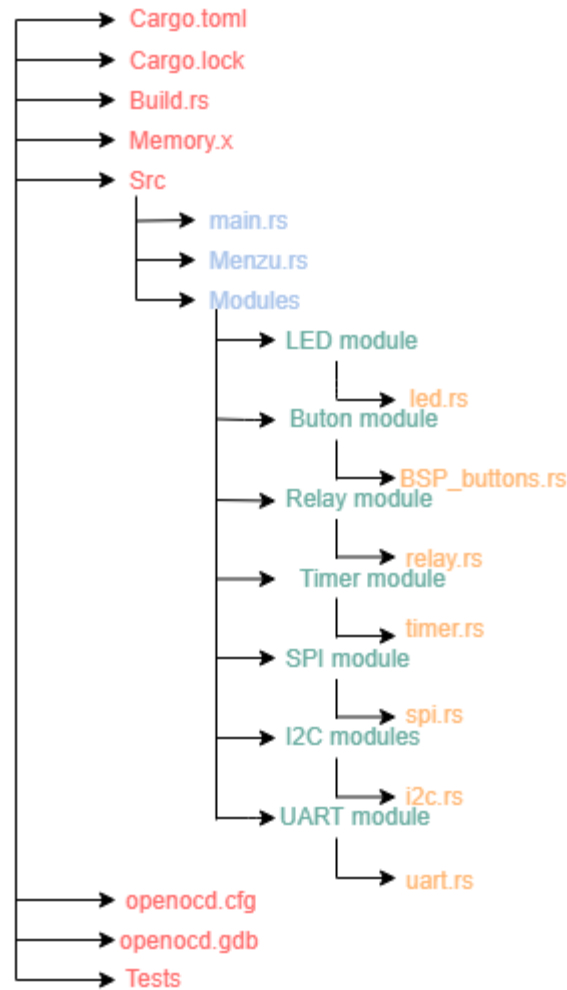


Figure 40: Architecture générale de projet

III.2 Développement des modules

III.2.1 Développement de module LED

Pour le développement de module LED nous allons développer des structures de base que nous devons obligatoirement les intégrer dans le module : **Menzu** et **led**

➤ La structure BspLed

Cette structure est déclarée pour configurer les pins des deux LEDs bi couleurs (vert et rouge) qui existent dans la carte Menzu. Elle comporte quatre champs tous de type `output<pushpull>` : (`leds_pin_led1_g`, `leds_pin_led1_r`, `leds_pin_led2_g`, `leds_pin_led2_g`).


```
17  /*----- Module -----*/
18  pub struct BspLed{
19
20      pub leds_pin_led1_g : PC7<Output<PushPull>>,
21      pub leds_pin_led1_r : PC6<Output<PushPull>>,
22      pub leds_pin_led2_g : PB14<Output<PushPull>>,
23      pub leds_pin_led2_r : PB13<Output<PushPull>>,
```

Puis on implémente dans la structure la méthode (trait en Rust) *led_init()* avec l'option **impl** qui a comme type de retour *BspLed* pour configurer les pins (pc7 et pc6) de port C et les pins (pb13 et pb14) de port B en tant que sortie pushpull puis les registres *moder* et *otyper* passent à la fonction *into_push_pull_output()* afin de configurer les ports :

```
impl BspLed {
    pub fn led_init( mut ar : MenzuStruct) -> BspLed {
        {
            BspLed {
```

➤ La structure MenzuStruct

Cette structure comporte trois champs (clocks, mygpiob, mygpior) pour configurer toutes les horloges dans le système. Puis on implémente dans la structure la méthode *system_init()* dans laquelle on initialise l'horloge et on acquiert les ports C et B :

```
/*----- Module -----*/

pub struct MenzuStruct {
    pub myclock : Stm32_HAL::rcc::Clocks,
    pub mygpiob : Stm32_HAL::gpio::gpiob::Parts,
    pub mygpior : Stm32_HAL::gpio::gpior::Parts,
}
```

Puis On intègre les deux structures **led** et **menzu** dans le programme principal par « **mod** » :

```
/* Include modules */
mod Menzu;
mod Led;
```

Et on configure un minuteur système pour clignoter les LEDS toutes les secondes :

```
let mut Delay = SystemDelay::new(CortexPeriph.SYST, config.myclock);
```

Cet organigramme explique l'algorithme de la fonction main de module LED :

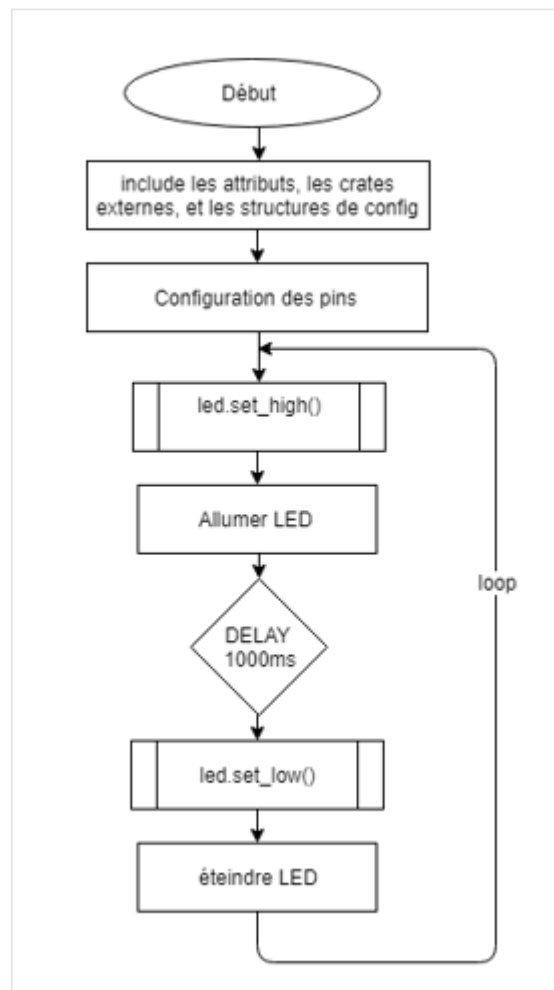


Figure 41: Organigramme de l'algorithme de module LED

➤ Compilation et Débogage de module

Pour la compilation de module LED on exécute la commande **cargo build** :

```

C:\Users\Kaabia\Rust_examples\SVN_Ons_Work\Examples\blink>cargo build
Compiling stm32l4_blink v0.1.0 (C:\Users\Kaabia\Rust_examples\SVN_Ons_Work\Examples\blink)
Finished dev [unoptimized + debuginfo] target(s) in 2.88s
  
```

Puis on connecte la carte par le débogueur ST-LINK/V2 et exécute la commande **openocd** :

```

C:\Users\Kaabia\Rust_examples\SVN_Ons_Work\Examples\blink>openocd
GNU MCU Eclipse 64-bits Open On-Chip Debugger 0.10.0+dev-00404-g20463c28 (2018-01-23-12:30)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 500 kHz
adapter_nsrst_delay: 100
  
```

➤ **Test :**

La Figure 42 montre le clignotement de la LED_r :



Figure 42: Clignotement de LED1 rouge

La Figure 43 montre le clignotement de la LED_g



Figure 43: Clignotement de LED2 vert

III.2.2 Développement de module Bouton

➤ **Structure BSP_Bouttons**

Le développement de module « *BOUTON* » nécessite les deux structures de modules « *LED* » et une autre structure *BouttonsStruct* pour la configuration des pins des boutons sur GPIOB

Cette structure contient deux champs (*button_pin_button_1*, *button_pin_button_2*) de types *Input<PullUp>* :

```
/*----- Module -----*/
pub struct ButtonsStruct {
    pub button_pin_button_1: PB15<Input<PullUp>>,
    pub button_pin_button_2: PB12<Input<PullUp>>,
}
```

Puis on implémente sous cette structure la méthode `Buttons_Init` qui a comme type de retour `ButtonsStruct` pour configurer les pins (PB15 et PB12) de port B en tant que sortie pullup et les registres `moder` et `pupdr` passent à la fonction `into _pull_up_input()` afin de configurer le port :

```
impl ButtonsStruct {
    pub fn Buttons_Init(mut ar: MenzuStruct) -> ButtonsStruct {
    {
```

➤ Organigramme

Cet organigramme nous permet d'expliquer l'algorithme de la fonction main de module boutons :

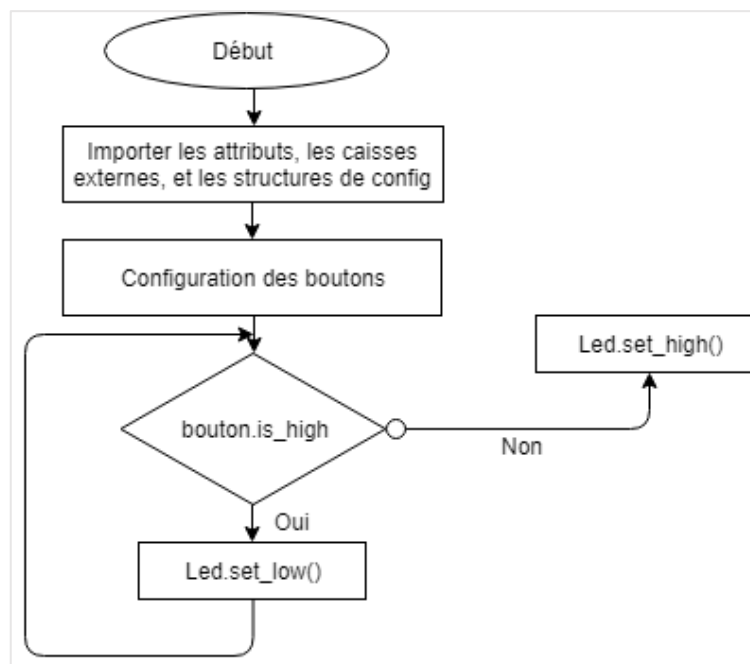


Figure 44: Organigramme de l'algorithme de module Bouton

➤ Compilation

Pour la compilation du module, il faut taper la commande **cargo build** :

```
C:\Users\Kaabia\Rust_examples\SVN_Ons_Work\Examples\button>cargo build
Compiling typenum v1.10.0
Compiling semver-parser v0.7.0
Compiling proc-macro2 v0.4.30
Compiling unicode-xid v0.1.0
Compiling rand_core v0.4.0
Compiling syn v0.15.34
Compiling stable_deref_trait v1.1.1
Compiling vcell v0.1.0
Compiling cortex-m v0.6.0
```

➤ Test de boutons

Nous avons vérifié le fonctionnement de bouton par l'allumage de la LED :



Figure 45: Test de bouton

III.2.3 Développement de module Relais

➤ Structure BSP_Relay

Le développement de module « *RELAIS* » nécessite deux structures : La structure *Menzu* de la configuration des horloges et des GPIOs et une deuxième structure « *BSP_Relay* ». Cette dernière dédiée pour la configuration des pins des deux Relais sur les ports (C et B), elle comporte quatre champs de type `output<pushpull>` :

```

/*----- Module -----*/

pub struct BspRelay {
    pub relay1_pin_set: PB4<Output<PushPull>>,
    pub relay1_pin_reset: PB5<Output<PushPull>>,
    pub relay2_pin_set: PB6<Output<PushPull>>,
    pub relay2_pin_reset: PB7<Output<PushPull>>,
}

```

➤ Organigramme de l'algorithme de module relais

Dans l'organigramme de l'algorithme de module relais nous avons défini le corps de la fonction main pour le fonctionnement des relais.

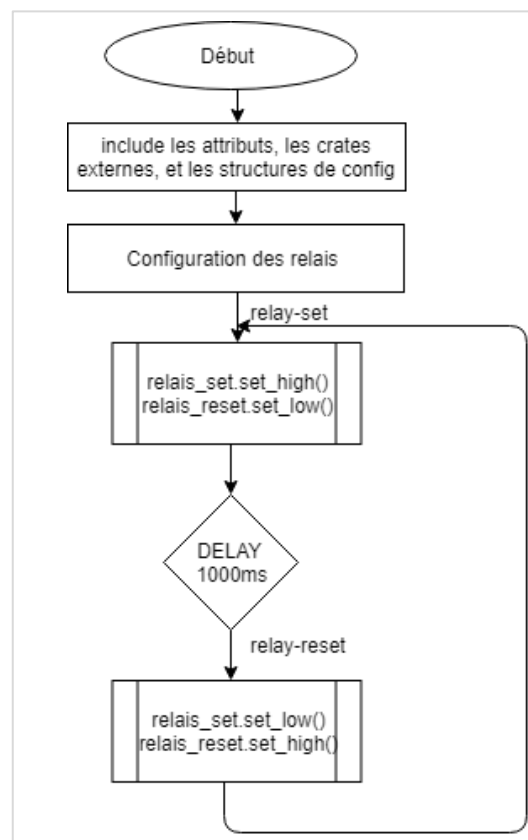


Figure 46 : Organigramme de l'algorithme de module relais

III.2.4 Développement de module Timer

Le développement de module « *TIMER* » nécessite la structure de configuration de l'horloge de système et une configuration de Timer 7 pour générer des demandes d'interruption pour une raison claire :

```
let mut timer = Timer::tim7(Peripherals.TIM7, 1.hz(), clocks, &mut rcc.apb1r1);
timer.listen(Event::Timeout);
```

Après une certaine fréquence il sort de la boucle et entre dans une interruption :

```
#[interrupt]
fn TIM7() {
}
```

#[interrupt] est un attribut permettant de déclarer les gestionnaires d'interruptions. Dans cette interruption on va déclarer une tâche à réaliser dans chaque interruption.

- **Organigramme de l'algorithme de module Timer**

Pour le test de module Timer nous avons inséré un compteur dans l'attribut interruption qui incrémente 1 lorsqu'il entre dans une interruption :

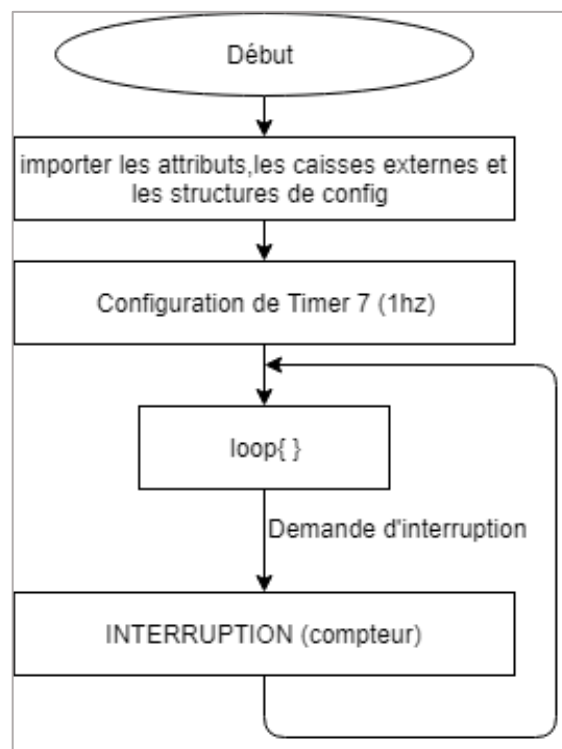


Figure 47: Organigramme de l'algorithme Timer

- **Compilation**

L'exécution de la commande **Cargo run** nous permet de déboguer le module et montrer l'incréméntation de compteur après 1hz :

```
Info : halted: PC: 0x08003dde
*** Timer Tutorial ***
1
2
3
4
5
6
7
8
9
10
11
12
```

Figure 48: Compilation de module Timer

III.2.5 Développement de module I2C (Inter-Integrated Circuit)

Pour lire ou écrire des données à partir des capteurs de température (slave) nous devons acquérir deux choses : l'adresse du composant qui doit être unique sur le bus dans notre cas l'adresse de capteur (BMP280_ADDRESS_I2C= 0x76), et le ou les octets de données à lire ou à écrire :

```
const BMP280_ADDRESS_I2C: u8 = 0x76;
```

Nous avons configuré le périphérique I2C1 en conjonction avec les pins *pa9* de port A pour (SCL) et *pa10* de même port pour (SDA) en tant que sortie output et les registres *moder* et *otyper* passent à la fonction *into_open_drain_output()* afin de configurer le GPIOA.

La ligne d'horloge (SCL) détermine la rapidité avec laquelle les données peuvent être échangées. On la fixe sur une fréquence 100KHz (mode standard).

Avec la fonction *read()* nous allons lire le buffer à partir de registre interne de capteur de température sur 16 bit.

- **Organigramme**

La Figure 49 suivante montre l'algorithme de la fonction main de module I2C :

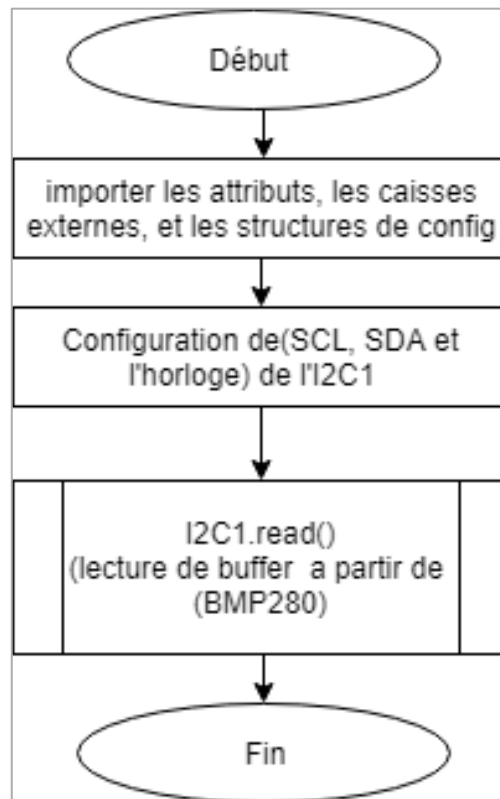


Figure 49: Organigramme de l'algorithme I2C

III.2.6 Développement de module SPI (Serial Peripheral Interface)

Le bus SPI est un bus série assurant une communication en full-duplex synchrone. Il est multipoint avec un maître communiquant avec plusieurs esclaves.

Pour une communication SPI nous devons configurer le bit de poids fort CPOL (ClockPolarity) et le bit de poids faible CPHA (Clock Phase) :

```

/// SPI mode
pub const MODE: Mode = Mode {
    phase: Phase::CaptureOnFirstTransition,
    polarity: Polarity::IdleLow,
};
  
```

Puis nous configurons le pin pb0 de NSS (slave selector) sur le GPIOB et les pins pa6, pa7 et pa5 respectivement de (MISO, MOSI, SCK) sur le GPIOA. Ensuite nous configurons le périphérique SPI1 :

```
let mut spi = Spi::spi1(
    p.SPI1,
    (sck, miso, mosi),
    MODE,
    // 1.mhz(),
    100.khz(),
    clocks,
    &mut rcc.apb2,
);
```

• Organigramme de l'algorithme de module SPI

L'organigramme suivant explique la fonction main de la communication SPI :

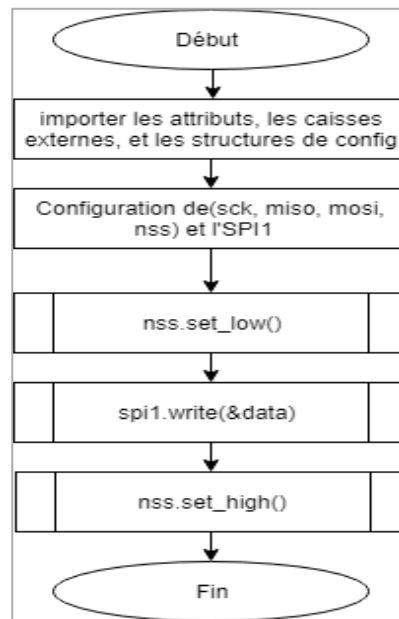


Figure 50: Organigramme de l'algorithme SPI

III.2.7 Développement de module UART

Une liaison série consiste à envoyer une information bit après bit avec un délai entre chacun. Le bus *Universal Asynchronous Receiver Transmitter* (UART) est une implémentation de ce principe.

Le microcontrôleur STM32L4 dispose d'un périphérique USART full-duplex. Une communication bidirectionnelle via l'USART nécessite au moins deux broches :

- Une broche, notée RX, pour recevoir les bits, configurée sur le pin pa2 sur le GPIOA
- Une broche, notée TX, pour transmettre les bits, configurée sur le pin pa3 sur le GPIOB.

Quand l'émetteur est désactivé, la broche prend l'état imposée par la configuration du port, alors que si l'émetteur est activé et que rien n'est à transmettre, la broche TX est dans l'état haut.

Par la suite, nous allons configurer le périphérique USART2 en émetteur avec un *baud rate* de 9600 bps.

- **Organigramme de l'algorithme de module UART**

Nous montrerons le fonctionnement de l'interface UART par l'organigramme suivant de la Figure 50

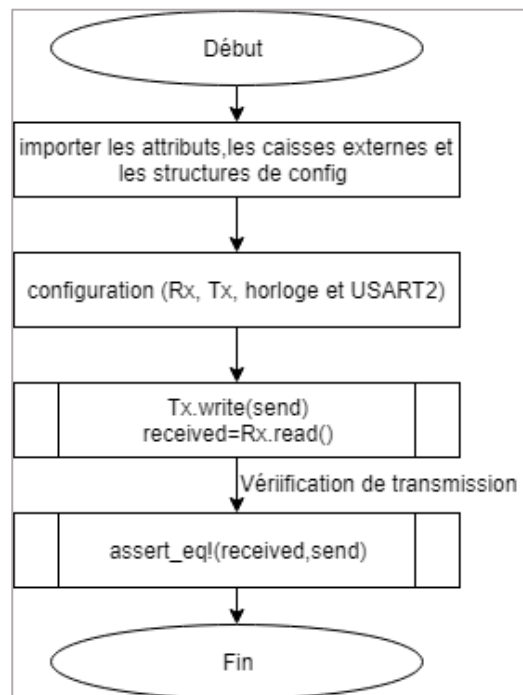


Figure 51: Organigramme de l'algorithme UART

IV. Conclusion

Ce chapitre, est dédié principalement pour la partie réalisation de projet. En premier lieu nous avons commencé par l'exemple de clignotement des LEDs de la carte STM32F429. Ensuite, nous avons passé au développement des modules, puis l'intégration et le test des modules développés sur la carte Menzu.

Nous présentons, dans la conclusion générale, un résumé du travail effectué, quelques perspectives ainsi que nos acquis sur le plan personnel et professionnel.

Conclusion et perspectives

Dans le cadre de notre projet, nous avons développé des modules (LEDs, Boutons, Relais, Timer, I2C, SPI, UART) destinés au microcontrôleur STM32L4 de la carte Menzu conçu par la société Sofia Technologies. Par la suite, nous avons entamé la compilation et l'implémentation de ces modules développés sur la carte.

Notre point de départ était la récolte des informations nécessaire sur le cadre général du projet qui nous a permis de comprendre les principaux concepts autour desquels tourne notre projet. Ceci nous a garanti la découverte de langage Rust afin de mieux comprendre la problématique et le travail demandé.

La partie suivante était consacrée à la préparation de l'environnement de travail qui nous a permis de créer une architecture globale de notre projet.

La dernière partie de notre projet représente la phase la plus importante, dans laquelle nous avons développé nos modules et les implémentés sur la carte en moyennant les ressources matérielles, logicielles et technologiques affectées au projet.

Le travail en équipe et la notion de collaboration régnant au sein de la société SOFIA Technologies ont facilité notre intégration et notre reconnaissance des procédures de travail et de gestion de projet. Ainsi, les relations inter stagiaires, ingénieurs, chefs d'équipe, managers et directeurs est enrichissantes et conviviales. Le respect entre les membres d'équipe était favorable pour aboutir aux résultats souhaités.

Du point de vue personnel, cette expérience professionnelle nous a garanti l'intégration dans une équipe bien organisée. Ce qui nous a appris la gestion de projet d'une façon méthodique et organisée ainsi que la manière de confronter les problèmes techniques rencontrés.

Ce cadre nous a permis l'application des connaissances techniques acquises toute au long du cursus universitaire à l'Ecole Nationale d'Electronique et des Télécommunications de Sfax. Éventuellement, il nous a permis aussi d'acquérir et d'approfondir des connaissances pratiques et même théoriques.

Malgré nos efforts énormes, notre travail ne peut pas être parfait, nous avons toujours une liste infinie d'idées à développer pour améliorer notre performance.

Donc en tant que perspectives éventuelles, nous pouvons suggérer :

- Développement de ces modules sur la carte Menzu de microcontrôleur ATSAM21J17 et ATSAM21J18 puisque ses crates sont en cours de développement.
- Développement d'autres modules comme le GPS, Bluetooth, Bmp qui fonctionnent par les modules déjà développés. Par exemple le GPS possède une interface de communication série uart.
- Développer ces modules avec la crate Cortex-m et non pas les crates qui fournissent des abstractions de types HAL qui sont disponibles dans embedded-hal afin de s'approcher du Hardware ainsi pour qu'on puisse développer sur n'importe quel microcontrôleur.
- Création d'une application qui a une tâche bien déterminée qui teste tous ces modules à la fois. Et pour améliorer les performances de cette application il faut ajouter un RTOS qui utilise des ordonnanceurs spécialisés afin de fournir des systèmes temps réel les outils et les primitives nécessaires pour produire un comportement temps réel souhaité dans le système final.

Maintenant les RTOS en Rust sont en cours de développement le seul Rtos qui est disponible est le tock qui est un operating system (os) intégré conçu pour exécuter plusieurs applications simultanées et méfiantes sur des plates-formes intégrées basées sur Cortex-M. La conception de Tock est centrée sur la protection, à la fois d'applications potentiellement malveillantes et de pilotes de périphériques.

Bibliographie

- [1] Pierre Ficheux et Eric Bénard, Linux embarqué : Nouvelle étude de cas, Eyrolles, 4ème édition « La chaîne de compilation croisée », Mai 2012.
- [2] Sirine Othman « Guide de démarrage SOFIOS_PFA », Janvier 2019.
- [3] <https://www.rfwireless-world.com/Terminology/UART-vs-SPI-vs-I2C.html>
- [4] Steve Klabnik et Carol Nichols « Le langage de programmation RUST 1.31.0 » Février 2018.

Résumé

Le présent rapport synthétise le travail effectué dans le cadre d'un projet en systèmes électronique et embarqué au sein de l'entreprise SOFIA Technologies. L'objectif de ce travail est le développement des modules qui permettent la communication entre les différents périphériques informatiques d'une façon sécurisée avec le langage de programmation Rust. Nous avons réalisé ce projet avec l'ambition de répondre aux besoins fonctionnels et non-fonctionnels d'un utilisateur final, l'impatience d'apprendre des technologies récentes tel RUST, MENZU, et l'envie de s'adapter à une méthodologie de travail professionnel au sein d'une équipe.

Mots clés : Menzu, langage Rust, Eclipse

Abstract

This report summarizes the work done as part of the graduation project for obtaining the national diploma of engineer in electronic systems within SOFIA Technologies. The objective of this work is the development of modules that allow communication between different peripherals in a secure way with the Rust programming language. We realized this project with the ambition to answer the functional and non-functional needs of an end user, the impatience to learn recent technologies such as RUST, MENZU, and the desire to adapt to a methodology of professional work within team.

Keywords : Menzu, langage Rust, Eclipse