الجمهورية الجزائرية الديمقراطية الشعبية
République Algérienne Démocratique et Populaire
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Département : Génie électrique Et Informatique Industrielle

Mémoire de projet de fin d'études

Pour l'obtention du diplôme d'Ingénieur d'état en **Électronique**

Spécialité

Systèmes Embarqués

Thème

Etude et conception d'une passerelle USB CAN/CANFD, Application à la voiture connectée.

Réalisé par
BOUSHAKI Mohamed Yacine
DAOUDI Hadjer

Les membres de Jury :

Président	Mme. HABANI Lamia	M.A.A	ENST
Promoteur	Mme. SAADI Hadjer	M.C.A	ENSTICP
Examinateur	Mme. BOUCHAMA Samira	M.C.B	ENST
Examinateur	M. BOUSSOUFA Ahmed	M.A.A	ENST
Co-Promoteur	M. ZAOUI Abdelhalim	PROF	ENST

Année universitaire 2022-2023

Dédicace

Je dedié ce travail

A mes parents

Pour l'éducation et les sacrifices, car sans eux je ne serais rien, et ils sont la raison qui me pousse à me surpasser pour les rendre un jour fiers de moi.

A mes deux petites soeurs

Pour leur soutien inconditionnel et leurs encouragements qui ont été d'une valeur inestimable pour moi.

À mes grands-parents paternels, que leurs âmes reposent en paix, et mes grands-parents maternels.

À mes professeurs

Pour leurs enseignements rigoureux et le partage de leurs connaissances.

A mes amis

Yazid, Noro et Belkacem, et à mes camarades de promotion, pour leur soutien et leurs encouragements dans les moments difficiles.

À mon binôme

Pour son aide et son soutien durant toutes les étapes de notre travail.

Mohamed Yacine.

Dédicace

A mes très chers parents,

Pour l'éducation, l'amour, les sacrifices et le soutien inébranlable.

A mes sœurs et mon frère,

Pour l'encouragement, le soutient et les conseils précieux tout au long de mes études.

A mes amis,

Pour leurs aides et supports dans les moments difficiles.

A mon binôme,

Pour son travail, son soutien moral, sa patience et sa compréension tout au long de ce projet.

Enfin, à moi-même.

Hadjer.

Remerciements

Nous tenons tout d'abord à remercier Allah le Tout-Puissant de nous avoir accordé la force, la persévérance et la sagesse nécessaires pour pouvoir achever ce mémoire avec succès. Nous sommes également profondément reconnaissants envers nos parents qui ont été une véritable source inépuisable de soutien et d'encouragement tout au long de ce parcours.

Leur amour inconditionnel, leurs conseils avisés et leurs sacrifices ont été essentiels pour notre développement académique et personnel. Leur présence constante et leur confiance en nous nous ont motivés à donner le meilleur de nous-mêmes.

Nous remercions Mr. BELLOULA Abdelmalek, Mme. SAADI Hadjer et Mr. ZAOUI Abdelhamid d'avoir accepté de nous encadrer et pour toute l'aide qu'ils nous ont apportée, ainsi que pour leur disponibilité et leurs précieux conseils sans lesquels nous n'aurions jamais pu aboutir à ce résultat.

Nous exprimons également notre gratitude envers le jury honorable qui a examiné et valorisé notre travail, ainsi qu'envers le cadre administratif du département de génie électrique de l'École Nationale Supérieure de Technologie.

Enfin, nous tenons à remercier chaleureusement tous ceux qui ont contribué de près ou de loin à l'élaboration de ce mémoire. Leurs conseils, leurs commentaires constructifs et leurs idées ont enrichi notre travail et ont contribué à sa qualité.

Table des matières

Liste des figures **i**

Liste des tableaux **iv**

Liste des abréviations **v**

Introduction générale **1**

I Généralités sur le Bus CAN, CAN-FD et l'Analyseur CAN **3**

I.1 Bus CAN et ses principales caractéristiques 4

I.1.1 Spécifications des protocoles liés au CAN 4

I.1.2 Protocole CAN dans le modèle OSI 5

I.1.3 Caractéristique physique de bus CAN 6

I.1.4 Représentation des Trames CAN 8

I.2 Bus CAN-FD 14

I.2.1 Caractéristiques du bus CAN-FD 15

I.2.2 Trames du bus CAN-FD 15

I.2.3 Différence entre le protocole CAN-FD et le CAN 2.0 16

I.3 Analyseur CAN 18

I.4 Protocole USB 19

I.4.1 Description physique 19

I.4.2 Topologie 20

I.4.3 Représentation des trames USB 20

I.4.4 Les types de paquets 21

I.4.5 Type de trames USB 21

I.5 Domaines d'utilisation 22

I.5.1 Bus CAN 22

I.5.2 Bus CAN FD 22

I.5.3 Analyseur CAN 23

I.5.4 Protocole USB 23

II Conception et Réalisation du Système de Passerelle CAN-USB **25**

II.0.1 Synoptique global du système 26

II.1 Les microcontrôleurs STM32 27

II.1.1 Choix de microcontrôleurs STM32 27

II.2 Contrôleurs CAN 29
II.3 Résistances de terminaison 31
II.4 Ecran LCD 31
II.5 Principe de fonctionnement du circuit et principe de la communication . . 32

III Mise en Œuvre, Résultats Expérimentaux et Interprétations 49
III.1 Description du matériel et de l'environnement logiciel du travail 50
III.2 Photographie du système. 51
III.3 Déscription des étapes de tests 53
III.3.1 Test de transmission avec protocole CAN 53
III.3.2 Test de transmissions avec protocole CAN-FD 57
III.3.3 Transmission CAN et CAN FD 66
III.3.4 Diagnostic OBD 71
III.3.5 Affichage sur le LCD 71

Conclusion générale 74

Bibliographie 76

Annexes A 77

Annexes B 85

Liste des figures

I.1 Couches OSI appliquées au CAN. 5
I.2 Structure physique d'un réseau CAN. 6
I.3 Structure de la trame de données. 9
I.4 Trame de requête (Version 2.0 A). 10
I.5 Structure de la trame d'erreur [3]. 10
I.6 Trame de surcharge. 11
I.7 Période d'inter-trame [3]. 11
I.8 Technique de bit stuffing. 12
I.9 Structure du temps de bit nominal. 12
I.10 Exemple d'arbitrage [2]. 14
I.11 Format de la trame CAN-FD [8]. 15
I.12 Différence entre la trame du CAN et CAN-FD [8]. 17
I.13 Description de bus USB [11]. 19
I.14 Constitution du PID [13]. 20

II.1 Schéma synoptique global de l'analyseur conçu. 26
II.2 Schéma électrique global de l'analyseur conçu. 26
II.3 Brochage du contrôleur CAN, le MCP2551 [17]. 29
II.4 Branchement du contrôleur MCP2551. 30
II.5 Branchement de résistances de terminaison. 31
II.6 Schéma électrique de l'afficheur LCD. 32
II.7 Configuration d'horloge. 33
II.8 Configuration de FDCAN. 35
II.9 Configuration de l'USB. 35
II.10 Configuration du Timer. 36
II.11 Configuration du SPI. 36
II.12 Configuration des pins PE9, PE10, PE15. 37
II.13 Configuration des pins pour les boutons. 37
II.14 Configuration finale des pins (Transmission). 37
II.15 Configuration finale des pins (Réception). 38
II.16 Déclaration des bibliothèques. 39
II.17 Déclaration des variables. 39
II.18 La variable hfdcan1. 40
II.19 Configuration de la variable sFilterConfig. 40
II.20 Configuration des structures TxHeader. 41

II.21 Code pour la transmission. . . . 41
II.22 Code de l'interruption du bouton K1. . . . 42
II.23 Code pour le changement de protocole. . . . 42
II.24 Déclaration des bibliothèques. . . . 43
II.25 Déclaration des variables. . . . 43
II.26 Déclaration des Fonctions. . . . 44
II.27 Code de la fonction « HAL_FDACN_RXFifo0Callback ». . . . 45
II.28 Déclaration des IDs. . . . 45
II.29 Code de traitement de données. . . . 46
II.30 Initialisation du LCD 46
II.31 Les fonctions utilisées pour afficher le chronogramme. . . . 47
II.32 organigramme de fonctionnement de l'analyseur. . . . 48

III.1 Carte STM32F446RE. . . . 51
III.2 Périphériques de la STM32F446RE. . . . 51
III.3 Bloc de transmission et de réception. . . . 52
III.4 Bloc de transmission. . . . 52
III.5 Bloc d'affichage. . . . 52
III.6 LCD pour l'affichage de débit et de protocole de transmission. . . . 53
III.7 Sysème final. . . . 53
III.8 Circuit du premier Test. . . . 54
III.9 Configuration de la STM32F446RE. . . . 54
III.10 Configuration de l'horloge. . . . 55
III.11 Déclaration des variables. . . . 55
III.12 Fonction d'interruption pour la transmission. . . . 56
III.13 Fonction d'interruption pour la réception. . . . 56
III.14 Vérification des données reçues. . . . 56
III.15 Organigramme de la transmission CAN. . . . 57
III.16 Configuration de l'horloge. . . . 58
III.17 Configuration du périphérique FD CAN. . . . 59
III.18 Les variables de transmission. . . . 59
III.19 Configuration du filtre. . . . 60
III.20 FDCAN_TxHeaderTypeDef. . . . 60
III.21 Code de transmission. . . . 61
III.22 Code de transmission. . . . 61
III.23 Code de réceptions. . . . 62
III.24 Résultats de réception des différentes données de la carte STM. . . . 62
III.25 Organigramme de la transmission avec buffer. . . . 63
III.26 Configuration du FDCAN 64
III.27 Configuration des variables de transmissions. . . . 64
III.28 Code pour la transmission. . . . 64
III.29 Code pour la réception. . . . 65

III.30 Résultats de la réception. 65
III.31 Organigramme de la transmission avec interruption. 66
III.32 Configuration du FD CAN. 67
III.33 Code pour le changement de débit. 68
III.34 Code pour le changement de protocole. 69
III.35 Résultats du test. 69
III.36 Organigramme du basculement. 70
III.37 Fonction OBD pour le diagnostic. 71
III.38 Fonction pour l'affichage LCD 72
III.39 Résultats du diagnostic 73

A.1 Diagramme d'Etat des erreurs du bus CAN 80
A.2 Paquet de début de trame [13]. 81
A.3 Les paquets jetons [13]. 81
A.4 Les paquets de données [13]. 82
A.5 Les paquets d'Etat [13]. 82
A.6 Communication USB [13]. 83
A.7 Le connecteur de diagnostic OBD [18]. 84

B.1 Caractéristiques des familles STM [18] 86
B.2 Schéma bloc du contrôleur MCP2551 [19] 87
B.3 Le module PCF8574. 89
B.4 Configurations du bus I2C 90

Liste des tableaux

I.1 Différences entre le bus CAN H-speed et CAN L-speed [3]. 7
I.2 Débit en fonction de la longueur [4]. 7
I.3 Durée des différents segments [6]. 13
I.4 Différence entre taille de donnée du CAN et CAN-FD [8]. 17
I.5 Différence entre le bus CAN et CAN-FD [8]. 18

II.1 Description du brochage du contrôleur CAN, le MCP2551 [17]. 30
II.2 Description du brochage du LCD [**?**] 32

B.1 Caractéristiques des familles STM [18] 85
B.2 Caractéristiques de l'afficheur LCD [19] 88
B.3 Le rôle des broche de l'afficheur LCD.. 89

Liste des abréviations

ACK	Acknowledgement
ADDR	Address
BRS	Bite Rate Switching
CAN	Controller Area Network
CAN FD	Controller Area Network Flexible Data-Rate
CRC	Cyclic Redundancy Code
DLC	Data Length Content
EDL	Extended Data Length
ENDP	Endpoint
EOP	End of Packet
ESI	Error State Indicator
FPU	Floating Point Unit
I2C	Inter-Integrated Circuit
IDE	Identifier Extension Bit
ISO	International Organization for Standardization
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LIN	Local Interconnect Network
LLC	Logic Link Control
MAC	Medium Access Control
MDI	Medium Dependent Interface
NRZ	Non Return to Zero
OBD	On Board Diagnostic
PID	Process IDentifier
PLS	Physical Signalling
PMA	Physical Medium Access
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RRS	Remote Request Substitution
RTR	Remote Transmission Request
SOF	Start Of Frame
SPI	Serial Peripheral Interface

SRAM	Static Random Access Memory
SWD	Serial Wire Debug
TFT	Thin-Film Transistor
TT-CAN	Time Triggered-CAN
USB	Universal Serial Bus

Introduction Générale

L'industrie automobile a connu une évolution significative au fil des dernières décennies, tant au niveau des technologies embarquées que des protocoles de communication utilisés, l'un des protocoles les plus couramment utilisés dans ce domaine est le CAN (Controller Area Network) qui permet la transmission fiable de données entre les différents systèmes électroniques d'un véhicule. Historiquement cette industrie a largement adopté ce protocole dans les années 1980, à cette époque, il s'agissait d'une solution révolutionnaire pour la communication entre les différents composants électroniques dans les véhicules.

Les avancées technologiques ont permis l'intégration de systèmes tels que l'ABS (Antiblockiersystem), les systèmes d'aide à la conduite et de stationnement, ainsi que les régulateurs de vitesse. Ces technologies offrent une meilleure sécurité, une conduite plus confortable et une expérience utilisateur améliorée. Cependant, pour gérer l'ensemble de ces fonctionnalités, il est indispensable d'avoir un calculateur central capable de coordonner et de synchroniser les différents signaux provenant des multiples capteurs et systèmes électroniques.

Avec l'augmentation de la complexité de ces systèmes embarqués et l'introduction sans cesse de nouveaux capteurs et actionneurs. La demande en termes de bande passante et de flexibilité de communication a augmenté. Ceci a poussé le protocole CAN a évolué vers une version améliorée qui est le protocole CAN FD (Controller Area Network Flexible Data-Rate) au début des années 2010, ce dernier présente des avantages significatifs par rapport à la version classique en terme de taux et de capacité de transmission.

Avec cette évolution des protocoles, il est devenu essentiel pour les ingénieurs et les développeurs d'avoir des outils adaptés pour analyser et diagnostiquer les réseaux CAN dans les véhicules. C'est ainsi que les tout premiers analyseurs CAN ont été développé, ce sont des outils essentiels dans le développement et la maintenance des systèmes électroniques des véhicules, ils permettent de capturer, d'enregistrer et d'analyser les messages échangés sur le réseau CAN, ce qui facilite le débogage des problèmes de communication, la validation des protocoles et la compréhension globale du fonctionnement des systèmes électroniques.

Par ailleurs avec l'introduction du protocole CAN FD, de nombreux analyseurs existants deviennent incompatibles avec cette nouvelle norme, ce qui a créé un besoin croissant de mettre en place des solutions capables de traiter à la fois les données CAN et CANFD.

Notre but est de fournir une solution polyvalente et efficace qui permet aux ingénieurs et développeurs de bénéficier des avantages du protocole CAN FD tout en maintenant la compatibilité avec le protocole CAN existant

Le concept d'une passerelle CAN/USB combinée à un analyseur vient de répondre à ce besoin. Ce nouvel outil est conçu pour permettre la réception, l'analyse et le diagnostic des signaux CAN et CAN FD. La solution combinée offre une interface USB conviviale pour la connexion aux ordinateurs, permettant ainsi une analyse approfondie des données et des fonctionnalités de débogage et de diagnostic avancées.

Ce mémoire est décomposé en trois chapitres. Le premier chapitre est consacré aux généralités du bus CAN et CAN FD et leurs principales différences, ainsi qu'à l'introduction des analyseurs CAN et leurs domaines d'utilisations.

Le deuxième chapitre introduit et détaille la conception et la réalisation de la passerelle/analyseur avec les différents composants utilisés.

Le troisième chapitre mets en exergue les différents tests et résultats expérimentaux obtenu suite à la conception et l'implémentation de notre systéme, tout en mettant l'accent sur leurs interpretations.

Chapitre I

Généralités sur le Bus CAN, CAN-FD et l'Analyseur CAN

Introduction

Durant les dernières décennies le monde de l'industrie a connu de très grands changements et avancements, ayant connu l'émergence et l'incorporation de l'électronique dans l'industrie qui se caractérise par l'utilisation de différents protocoles de communication série entre les systèmes utilisés dans de nombreuses applications industrielles, tels que le bus RS232, le RS485, le SPI (Serial Peripheral Interface), l'I2C (Inter-Integrated Circuit), le LIN (Local Interconnect Network) et le CAN.

Dans ce chapitre, nous explorons les concepts fondamentaux et les différents types de trames des bus CAN et CAN FD, ainsi que de l'analyseur CAN.

I.1 Bus CAN et ses principales caractéristiques

Le CAN, est un bus de communication série multi-maîtres dont les spécifications de conception de base exigent une vitesse élevée, une grande immunité contre le bruit et des fonctions de détection d'erreurs. C'est un bus qui supporte des systèmes temps réel avec un haut niveau de fiabilité [1].

Le protocole de réseau CAN a été défini pour fournir une communication déterministe dans des systèmes distribués complexes avec les caractéristiques et les capacités suivantes :

- Un haut niveau de fiabilité,
- Souplesse de configuration,
- Détection et signalisation des erreurs avec retransmission automatique des messages altérés,
- Détection des défaillances permanentes dans les nœuds et arrêt automatique pour isoler le nœud défectueux,
- Hiérarchisation des messages,
- Temps de latence maximum garanti,
- Accès multi-maîtres au bus avec synchronisation temporelle.

I.1.1 Spécifications des protocoles liés au CAN

Le protocole CAN a été normalisé par l'ISO (International Standardisation Organisation), selon deux normes : l'ISO 11519 pour un débit allant jusqu'à 125 kbit/s et l'ISO 11898 pour les débits allant jusqu'à 1 Mbit/s. Par la suite, ces normes ont été fusionnées dans la norme ISO 11898, cette dernière comporte trois parties : ISO 11898-1, ISO 11898-2, ISO 1898-3 [2].

- La version 1.1 de la spécification est la première version publiée de l'ensemble du protocole CAN, suivie par les versions : 1.2, 2.0A et 2.0B, qui sont toutes compatibles. Les modules CAN des différentes versions peuvent fonctionner dans un seul réseau.
- La version CAN 2.0B introduit des trames avec un identificateur 29 bits appelées « trame étendue » tandis que la spécification CAN 2.0A identique à la version 1.2, introduit des « trames standards » qui possèdent un identificateur de 11 bits

I.1.2 Protocole CAN dans le modèle OSI

La norme ISO décrit l'architecture du CAN selon les couches du modèle OSI (Open Systems Interconnection) de normalisation/Interconnexion des systèmes ouverts (ISO/OSI)1. Elle spécifie que le CAN fonctionne dans des parties des couches : physique et de liaison comme illustre la Figure I.1 [2, 3].

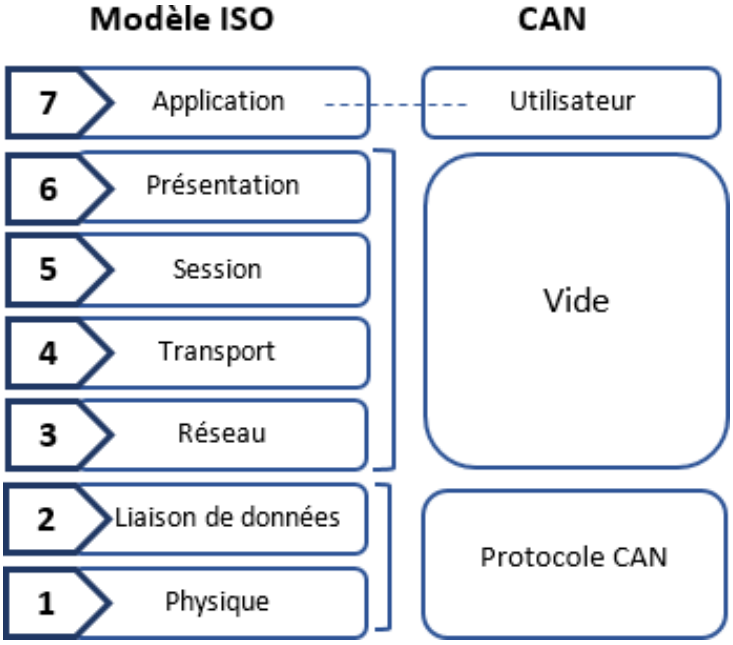


FIGURE I.1 : Couches OSI appliquées au CAN.

Ces couches du modèle OSI sont décrites comme suit :

- **Couche physique :** Responsable de contrôler la manière dont le signal est transmis et chargée d'assurer la transmission physique des bits entre les nœuds conformément à toutes les caractéristiques électriques et électroniques du système.

  Elle se subdivise en trois sous-couches PLS (Physical Signalling), PMA (Physical Medium Access), MDI (Medium Dependent Interface).

- **Couche liaison de données :** Comporte deux sous couches différentes, la LLC (Logic Link Control) et la MAC (Medium Access Control) qui représente le cœur du protocole CAN.

- **Couche réseau :** Gère l'adressage et le routage des données.
- **Couche transport :** Assure le transfert fiable des informations d'un point à un autre.
- **Couche session :** Organise les échanges de données.
- **Couche présentation :** Met en forme les informations pour une meilleure exploitation par les applications.
- **Couche application :** Spécifiée par l'utilisateur.

Les couches : réseau, transport, session, et présentation sont vides dans le protocole CAN.

I.1.3 Caractéristique physique de bus CAN

A. Structure du réseau CAN

La couche physique du bus CAN est composée d'une paire filaire différentielle a deux fils reliant tous les nœuds et formant une topologie en bus, permettant de réduire le câblage et d'insérer un nœud dans l'architecture sans avoir à déclarer celui-ci. Des résistances dites de terminaison sont installées aux deux extrémités de la paire filaire. Un nœud requiert également pour son fonctionnement un contrôleur CAN, l'interfaçage entre ce dernier et la paire filaire se fait par un transmetteur CAN. La Figure I.2 illustre la structure physique de réseau CAN [4].

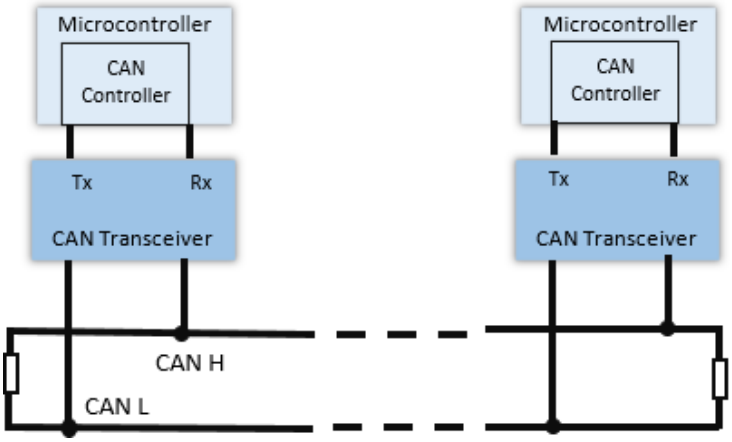


FIGURE I.2 : Structure physique d'un réseau CAN.

B. Support de transmission

La transmission de données entre les différents nœuds d'un réseau CAN est effectuée en série par une paire filaire différentielle qui se compose de deux fils [3] : CAN L (LOW), CAN H (HIGH).

Lorsqu'il y a un bit récessif, la valeur de la tension est la même dans les deux fils environ 2V, dans l'état dominant la différence de tension est autour de 2V. On distingue deux types de transmission possible, dont le Tableau I.1 résume les différences entre eux :

- Transmission en bus CAN low speed,
- Transmission en bus CAN high speed.

Tableau I.1 : Différences entre le bus CAN H-speed et CAN L-speed [3].

Paramètres	CAN low speed	CAN high speed
Débit	125 Kb/s	125 Kb/s à 1 Mb/s
Nombre de nœuds sur le bus	2 à 20	2 à 30
Courant de sortie (mode émission)	≥1 mA sur 2,2 kΩ	25 à 50 mA sur 60Ω
Niveau dominant	CAN H = 4 V CAN L = 1 V	CAN H = 3,5 V CAN L = 1,5 V
Niveau récessif	CAN H = 1,75V CAN L = 3,25	CAN H = 2,5 V CAN L = 2,5 V
Caractéristique du câble	30 pF entre les câbles de ligne	2*120Ω
Tensions d'alimentation	5V	5V

C. Longueur du bus et débit

La longueur du bus dépend de l'amplitude du signal qui varie en fonction de la résistance du câble et de l'impédance. Elle dépend également du délai de propagation sur les lignes physiques du bus. Le débit augmente lorsque la longueur du bus diminue. Le Tableau I.2 donne le changement de débit en fonction de la longueur de bus [1].

Tableau I.2 : Débit en fonction de la longueur [4].

Débit	Longueur
1Mbit/s	40 m
500 Kbits/s	100 m
100 Kbits/s	500 m
20Kbits/s	1000 m

D. Transmetteur CAN

Le Transmetteur CAN transforme la représentation électrique du bit utilisé par le contrôleur à celle du bus. En tant qu'émetteur, il protège le contrôleur contre les surcharges. En tant que récepteur il protège le comparateur d'entrée de la puce du contrôleur

contre les tensions excessives sur les lignes du bus. En outre, il détecte les erreurs de bus telles que les ruptures de ligne et les court-circuit [1, 5].

E. Contrôleur CAN

Les contrôleurs CAN permettent de gérer le bus de manière hardware. On peut trouver deux types de contrôleur, les contrôleurs qui se connectent aux microcontrôleurs via une liaison SPI, tels que le Microchip MCP2515 et les contrôleurs intégrés aux microcontrôleurs qui contiennent un module CAN reliés au transmetteur par des pins TX CAN et RX CAN [4].

I.1.4 Représentation des Trames CAN

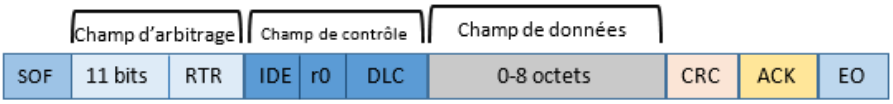
Le bus CAN comporte quatre types de trames spécifiques qui permettent le transfert des messages : la trame de données, la trame de requête, la trame de surcharge et la trame d'erreurs. Chaque trame permet de fournir une information spécifique, mais les données du message sont transportées seulement par la trame de données. L'information par une trame peut prendre deux états différents, "récessifs" et "dominants" (0 est généralement associé à l'état "dominant"). Ces différentes trames sont subdivisées en plusieurs champs permettant de définir tous les paramètres nécessaires de transmission.

1. Trame de donnée (Data Frame)

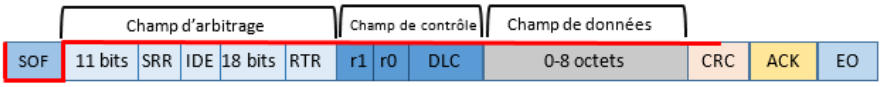
La trame de données est utilisée pour transporter les informations d'un émetteur vers un ou plusieurs récepteurs. Ces trames se divisent en sept champs comme illustre la Figure I.3 décrits ci-après [1, 2].

- **Début de trame (Start Of Frame) :** Un seul bit dominant qui indique le début d'une trame de données ou d'une trame de requête.
- **Champ d'arbitrage (Arbitration field) :** Ce champ contient un identificateur de 11 bits pour une trame standard et un identificateur de 29 bits pour une trame étendue. Il représente l'adresse logique et la priorité du message. L'identificateur est suivi par un bit RTR (Remote Transmission Request) pour distinguer une trame de données d'une trame de requête, il est dominant pour les trames de données et récessif pour les trames de requête.
- **Champ du contrôle (Control field) :** Composé de 6 bits. Le premier bit représente l'IDE (Identifier Extension bit) pour une trame standard, il est à l'état bas (dominant) et le bit suivant r0 est réservé. Dans une trame étendue, les deux bits sont réservés (r1 et r0). Les quatre derniers bits du champ de contrôle définissent la longueur du contenu des données (Data Length Content, ou DLC) en octets.

- **Champ de données (Data field) :** Ayant une longueur de 0 à 8 octets (0 à 64 bits) où chaque octet est transmis avec le bit de poids fort en premiers. Cette longueur est déterminée lors de l'analyse du champ de contrôle.
- **Champ de CRC (CRC Field) :** Composé de 16 bits, les 15 premiers bits représentent la somme de contrôle des bits précédents de la trame CRC (Cyclic Redundancy Code) et le dernier bit est un délimiteur de fin de champ de CRC (bit toujours récessif).
- **Champ d'acquittement (Ack field) :** Le champ ACK (Acknowledgement) se compose de deux bits. Le premier pour enregistrer l'acquittement des récepteurs (ACK slot) dans le cas où aucune erreur n'a été détectée par un nœud, ce dernier émet un bit dominant sinon il émet une trame d'erreur. L'autre bit est un délimiteur d'acquittement qui doit toujours être récessif.
- **Fin du trame (End of frame) :** Ce champ marque la fin de la trame par une séquence de 7 bits récessifs.



(a) Trame standard 2.0 A



(b) Trame étendue 2.0 B

FIGURE I.3 : Structure de la trame de données.

2. Trame de requête (Remote Frame)

Une trame de requête a le même format qu'une trame de données avec les caractéristiques suivantes [1, 2].

- Le champ de données est vide comme le montre la Figure I.4,
- Le bit RTR du champ d'arbitrage est toujours récessif. Dans le cas ou deux nœuds émettent chacun une trame de même identificateur, l'arbitrage de bit RTR va donner la priorité à la trame de données,
- Le champ Identificateur est utilisé pour indiquer l'identificateur du message demandé,

- Le champ DLC indique la longueur des données du message demandé (et non celle du message transmis).

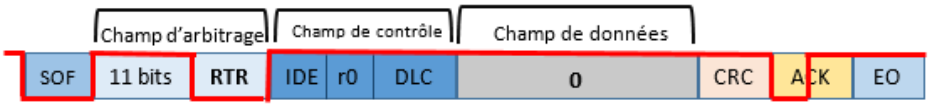
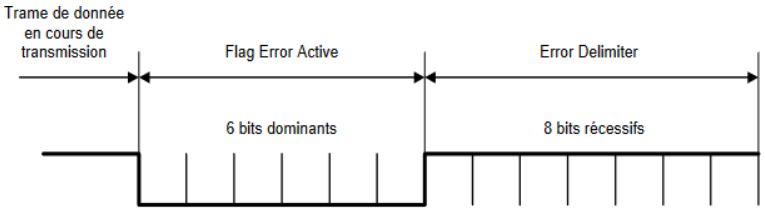


FIGURE I.4 : Trame de requête (Version 2.0 A).

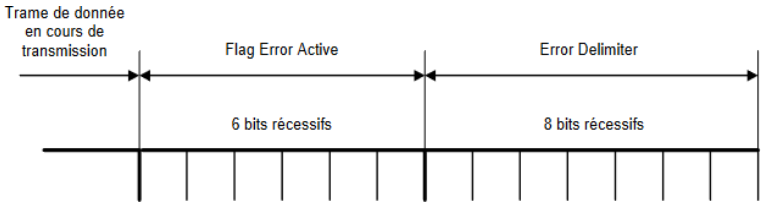
3. Trame d'erreur (Error Frame)

Elle présente le résultat de la signalisation d'une erreur de transmission sur le bus par une station et n'est pas considérée comme une trame réelle, elle est constituée de deux champs : le drapeau d'erreur et le délimiteur de champ.

Quand un nœud détecte une erreur, il l'a signale en envoyant un drapeau d'erreur ou Flag, qui est constitué de deux types de drapeaux comme le montre la Figure I.5, chacun de 6 bits, dominants et récessif dans l'ordre : actifs (Active Error Flag) et passifs (Passive Error Flag) [1, 2]. Le délimiteur d'erreur est composé de 8 bits récessif.



(a) Trame d'erreur active



(b) Trame d'erreur passive

FIGURE I.5 : Structure de la trame d'erreur [3].

4. Trame de surcharge (Overload Frame)

Qui a pour but de demander un laps de temps supplémentaire pour accepter la prochaine trame de donnée ou de requête. Elle se compose de deux champs comme illustre la Figure I.6 : le drapeau de surcharge (Overload Frame) avec 6 bits dominants, le délimiteur de surcharge (Overload Delimiter) avec 8 bits récessifs. Seulement deux flags de surcharges

consécutives sont autorisées sur le bus (pas plus de 12 bits dominants consécutifs émis) [1, 2].

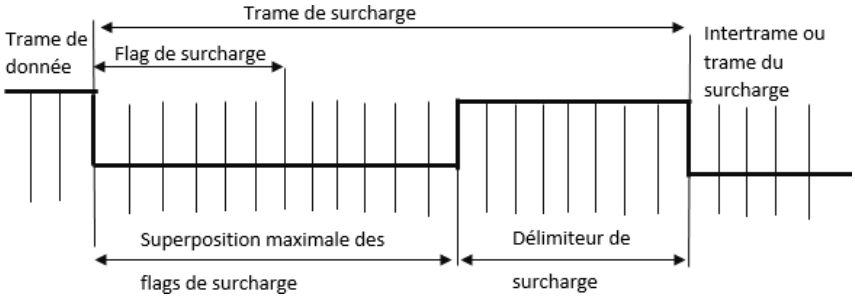


FIGURE I.6 : Trame de surcharge.

5. Inter-trame (Inter Frame Space IFS)

Sépare les trames de données ou les trames de requêtes entre elles par un espace d'une suite de bits récessifs. Elle est composée de trois champs [1, 3] comme l'illustre la Figure I.7 :

- Le champ d'intermission : composé de 3 bits récessifs consécutifs, interdit la transmission des trames et le gestionnaire de protocole signale les conditions de surcharge,
- Le champ de Bus Idle : le bus est inutilisé (au repos) et aucune trame ne circule,
- Le champ de suspension de transmission : ce champ est émis par un nœud lorsque celui-ci envoie une trame d'erreur passive.

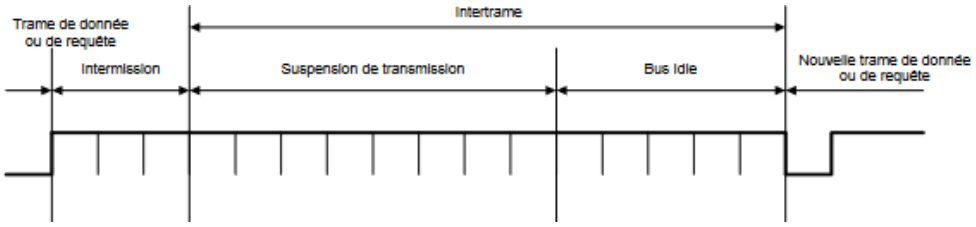


FIGURE I.7 : Période d'inter-trame [3].

6. Codage de bits

Le codage utilisé dans le bus CAN est de type NRZ (Non Return to Zero) qui garantit un nombre minimal de transitions et une grande résistance aux perturbations externes[2].

7. Bit Stuffing

Consiste à insérer un bit de polarité inverse dès qu'on émet 5 bits consécutifs de même polarité (Figure I.8), pour casser des chaînes trop importantes de bits identiques, le

récepteur reconnaîtra une séquence de cinq bits consécutifs et rejettera le bit ajouté avant le traitement du contenu de la trame. On obtient ainsi un grand nombre de transition ce qui permet de faciliter la synchronisation par les nœuds. Cette technique est uniquement activée sur les champs de SOF, d'arbitrage, de contrôle, de CRC (délimiteur exclu) dans les trames de données et les trames de requête, et désactivée dans les trames d'erreur ou les trames de surcharge [1].

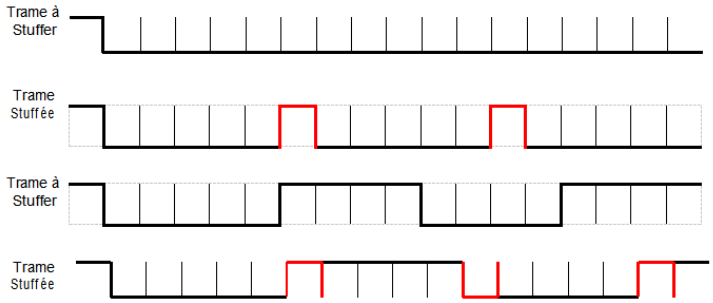


FIGURE I.8 : Technique de bit stuffing.

8. Nominal bit time

Représente la durée du bit sur le bus qui correspond à une période d'horloge. Tous les nœuds reliés sur le bus doivent avoir le même temps de bit nominal pour pouvoir émettre et recevoir les informations d'une façon correcte [1]. Selon la spécification CAN, le nominal bit time est divisé en quatre segments comme l'illustre la Figure I.9 :

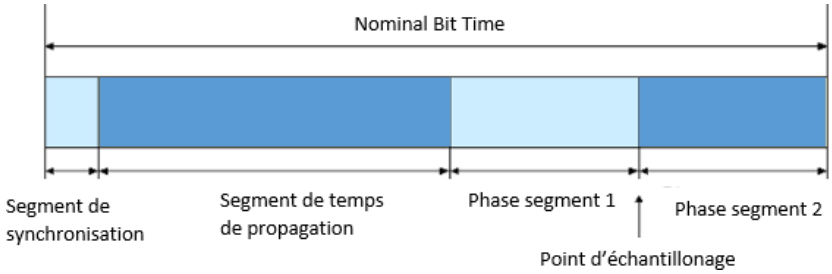


FIGURE I.9 : Structure du temps de bit nominal.

- Le segment de synchronisation (SYNC_SEG) : assure les transitions de 1 à 0 ou de 0 à 1 pour synchroniser les horloges de différents nœuds,
- Le segment de propagation (PROP_SEG) : utilisé pour compenser le temps de propagation (physiques) dans le réseau CAN,
- Le segment de phase buffer phase 1 et phase 2 (PHASE_SEG1 et 2) : utilisé pour compenser les erreurs de phase lors des transitions. Ils peuvent être allongés ou raccourcis pour resynchroniser la position de SYNCH SEG par rapport au front du bit suivant,

- Le point d'échantillonnage ou simple point : échantillonne la valeur du bit sur le bus.

Chaque segment se compose d'un certain nombre d'une unité du temps prédéfinie, reconnue sur le bus CAN par le « Time Quantum ». Voir Tableau I.3.

Time Quantum : Le Time Quantum est une unité de temps obtenue à partir de l'oscillateur interne en divisant l'horloge d'oscillateur en plusieurs périodes. La période d'horloge de l'oscillateur est appelée minimum Time Quantum. Il y a entre 8 et 25 Time Quantum par bit.

$$TIME_QUANTUM = m \times MINIMUM_TIME_QUANTUM \tag{I.1}$$

Tel que le m est la valeur du pré-diviseur. Lorsque le nombre de Time Quantum par segment est important, la synchronisation des nœuds sur le bus est plus précise.

Tableau I.3 : Durée des différents segments [6].

Segment	Durée en Time Quantum
SYNC_SEG	1
PROP_SEG	1 à 8
PHASE_SEG1	1 à 8
PHASE_SEG1	1 à 8

9. Arbitrage

L'attribution de droits d'accès sur le bus CAN est basée sur la méthode d'arbitrage bit à bit non destructive, lorsque plusieurs nœuds tentent simultanément d'émettre une trame sur le bus, ils comparent bit a bit leur message avec celui des messages concurrents, le concurrent de priorité supérieur est automatiquement sélectionné. Pour déterminer la priorité de messages.

Le bus CAN utilise la méthode d'accès CSMA/CR, deux nœuds commencent une transmission en même temps sur un bus libre. Tout au long de la trame, chaque nœud relié via sans transmetteur la valeur du bus pour chaque bit transmis, au début les deux nœuds envoient leur bit dominant de début du trame SOF, ensuite le MSB de l'identificateur est envoyé, Si un nœud transmet un bit récessif lit un bit dominant, il arrête la transmission. Ci-après la Figure I.10 répresente un exemple d'arbitration entre deux nœuds [2].

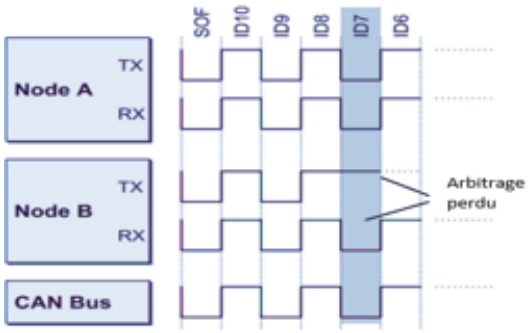


FIGURE I.10 : Exemple d'arbitrage [2].

10. Traitement des erreurs

Lors de transmission d'une trame par un nœud, plusieurs erreurs peuvent se produire et troubler le bon fonctionnement du réseau. Pour cela, le protocole CAN prévoit des mécanismes de détection des erreurs, de signalisation et d'autodiagnostics. Après la détection d'une erreur, le message est rejeté et sera retransmis automatiquement [1, 2].

Il existe plusieurs type d'erreur et chacune d'entre-elle demande une gestion et un traitement de recouvrement bien spécial. Voir Annexe A.

I.2 Bus CAN-FD

Le protocole CAN avait atteint ses limites et une évolution et amélioration de ses caractéristiques était nécessaire pour continuer à exister dans le domaine des véhicules automobiles et le monde industriel.

Avec l'évolution de l'industrie, le CAN-FD a émergé et il est présenté comme le successeur du CAN, le CAN-FD est en réalité une expansion du protocole CAN qui permet un contrôle en temps réel des données, avec une grande flexibilité et sécurité par rapport à son prédécesseur.

Il a été développé par la société BOSCH tout comme le protocole CAN et introduit sous forme de la norme ISO 11898-1 : 2015, parfait pour le domaine industriel et la communication embarquée.

Les véhicules actuels disposent de nombreux équipements électroniques communicants pour diverses tâches afin d'améliorer le confort, la sécurité, diminuer la pollution des véhicules et de rendre, le plus agréable possible, l'expérience du consommateur. Parmi ses fonctions on peut citer : l'aide à la conduite, l'assistance au freinage, le contrôle de trajectoire, le positionnement du véhicule dans son environnement, tout cela dans le but d'anticiper, d'alerter et d'assister le conducteur dans sa conduite

Cela fait que le CAN n'était clairement plus suffisant vu son débit limité à 1Mb/s et une trame de 8 octets maximum, les constructeurs se sont retrouvés obligés à utiliser

plusieurs bus CAN pour assimiler tout cela, mais ce n'était pas suffisant avec toutes ces flux de données ; certains de ses bus été surchargés, ce qui mettrait en péril tous le côté temps réel du système [6, 7].

I.2.1 Caractéristiques du bus CAN-FD

Parmi les caractéristiques les plus importantes du CAN-FD on peut retrouver [7, 8] :

- Comptabilité avec le protocole CAN, selon la norme ISO 11898-1, le CAN-FD est capable de recevoir et d'envoyer des données via le protocole CAN,
- Un traitement des erreurs nettement amélioré avec un CRC de 21 bits,
- Possibilité d'inculquer un ordre de priorité au message,
- Une configuration flexible,
- Un système de communication multi-maître,
- Une réception multicast avec une synchronisation temporelle,
- Permet de distinguer entre une erreur temporaire et la défaillance d'un nœud et la possibilité d'exclure automatiquement le nœud défaillant.

Les données sont envoyées selon un format fixe avec des longueurs variables mais limité, lorsque le bus est libre n'importe quel nœud connecté peut commencer une nouvelle transmission.

I.2.2 Trames du bus CAN-FD

Les trames de données dans le CAN-FD sont divisées selon 3 phases comme illustre la Figure I.11 [8] :

- Première phase d'arbitrage,
- Phase de donnée,
- Deuxième phase arbitrage.

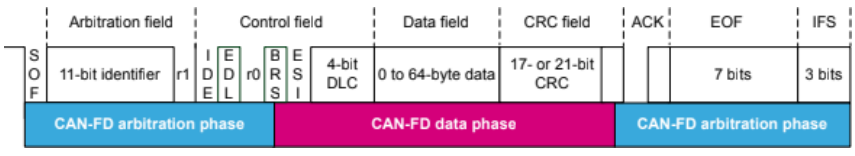


FIGURE I.11 : Format de la trame CAN-FD [8].

La première phase d'arbitrage du CAN-FD contient :

- SOF (Start Of Frame) un bit récessif pour indiquer le début d'une transmission,
- Un champ de 11 ou 29 bits spécifie l'ID du nœud et d'autres bits pour le format et la vitesse.

La phase de donnée contient :

- Le DLC qui indique la longueur de la donnée en octet,
- La donnée à transmettre,
- Le CRC pour le traitement des erreurs,
- Un bit dominant.

La seconde phase d'arbitrage est constituée de :

- Le bit ACK transmis par le récepteur,
- EOF (End Of Frame) 7 bits dominant pour indiquer la fin de la transmission. Durant la partie IFS l'inter-trame aucune donnée n'est transmissent, ce sont 3 bits récessifs successifs.

I.2.3 Différence entre le protocole CAN-FD et le CAN 2.0

L'apparition du CAN-FD est due à la demande du marché d'avoir un plus haut débit et une plus grande vitesse de transmission, chose faite avec la possibilité d'envoyer jusqu'à 64 octets dans une seule trame et la possibilité de changer la vitesse de transmission jusqu'à 8 fois plus vite durant la phase de donnée et de rebasculer vers une vitesse normale de 1Mb/s durant les deux phases d'arbitrage [8, 9]. Les différences majeures entre les 2 protocoles sont illustrées dans la Figure I.12 suivante. Les 2 protocoles commencent par un bit récessif SOF pour débuter la transmission et enchaîne par l'envoie de l'ID sur 11 bits pour le format standard ou 29 bits pour l'étendu, après cela les premières divergences apparaissent :

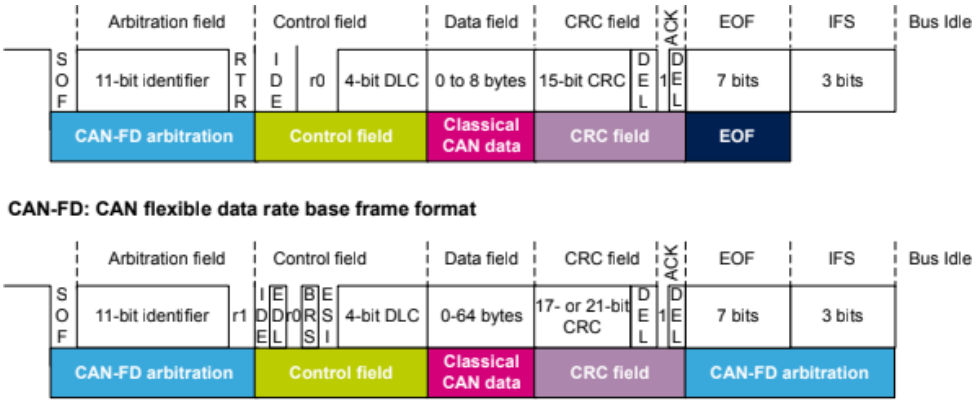


FIGURE I.12 : Différence entre la trame du CAN et CAN-FD [8].

- Le CAN 2.0 envoie le bit RTR pour spécifie quel genre de trame si c'est une trame de donnée ou de requête,
- Le CAN-FD quant à lui envoie toujours un bit dominant RRS (Remote Request Substitution) qui est un bit réserve car ce protocole n'émet que des trames de données.

Autre différence pour la CAN-FD dans le champ de contrôle, trois nouveaux bits sont ajoutés :

- Le bit EDL (Extended Data Length) qui est récessif pour signifie que c'est le protocole CAN- FD, dans le CAN classique ce bit et dominant c'est le R0 un bit réservé,
- Le bit BRS (Bite Rate Switching) ce bit permet de savoir s'il y aura deux débits différents pour la transmission un pour l'arbitrage et un pour la donnée,
- Le bit ESI (Error State Indicator) indique si la trame est en mode erreur active ou passive.

En outre la taille de donnée à envoyer passant de 0 à 8 octets dans le CAN 2.0 et de 0 à 64 octets pour le CAN-FD (Tableau I.4).

TABLEAU I.4 : Différence entre taille de donnée du CAN et CAN-FD [8].

DLC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CAN 2.0	0	1	2	3	4	5	6	7	8	8	8	8	8	8	8	8
CAN-FD	0	1	2	3	4	5	6	7	8	12	16	20	24	32	48	64

Avec une plus grande quantité de données transmises, le débit est alors augmenté, mais pas aussi la possibilité d'erreurs est également augmentée, alors pour pouvoir s'assure de la bonne transmission des données et la détection des erreurs, le CAN-FD dispose d'un

mécanisme CRC à base d'un polynôme de 17ème degré (16 bits) pour toute transmission de 16 octets maximum et d'un CRC de 21 bits pour tous les transmissions entre 16 et 64 octets. Le Tableau I.5 résume la différence entre les différentes caractéristiques des deux bus, CAN et CAN-FD.

Tableau I.5 : Différence entre le bus CAN et CAN-FD [8].

Caractéristiques	CAN 2.0	CAN-FD
Comptabilité	Ne supporte pas Support	CAN 2.0 A/B
Débit maximum (Mbit/s)	Débit de trame : jusqu'a 1	Débit d'arbitration : jusqu'à 1s Débit de donnes : jusqu'à 8
Code de champ de DLC (4 bits)	Code de 0 à 8	Code de 0 a 64
Bits de données maximum en un seul message	8 bites de données	64 bites de données
Support de BRS	Oui	Non
support de EDL	Oui	Non
support de ASI	Oui	Non
Code de vérification des bits CRC	Bits non inclus dans CRC	Bits inclus dans CRC
Support de Trame de requête	Oui	Non

Pour pouvoir recevoir, lire et décoder les différentes trames émissent à l'aide du protocole CAN, CAN-FD, nous utiliserons un dispositif très spécifique pour cette fonctionnalité qui est « l'analyseur ».

I.3 Analyseur CAN

L'analyseur est un outil de débogage et d'analyse, utilisé pour capturer et analyser les messages sur le bus, il est composé de l'interface CAN et de l'outil logiciel qui permet un accès simple aux dispositifs et à leurs objets.

Pour analyser les trames qui transitent sur un bus de données, on peut utiliser un oscilloscope numérique, certains oscilloscopes disposant d'une interface CAN qui montre directement la trame décodée qu'ils capturent. De façon plus conviviale et moins onéreuse, on peut utiliser un PC muni d'une interface matérielle, par exemple un adaptateur CAN-USB (Universal Serial Bus) et d'un logiciel d'analyse de réseaux.

L'analyseur CAN comprend un hardware analyseur de bus CAN, logiciel de l'analyseur de bus CAN, CD du logiciel de l'analyseur de bus CAN (comprend trois composants : firmware pour le PIC18F2550 (fichier hexadécimal), firmware pour le PIC18F2680 (fichier

hexadécimal), l'interface graphique PC de l'analyseur de bus CAN), mini-câble USB pour connecter l'analyseur de bus CAN au PC [10].

I.4 Protocole USB

Le protocole USB (Universal Serial Bus) est un standard de communication utilisé pour connecter des périphériques électroniques à un ordinateur ou à d'autres appareils compatibles. Il est largement utilisé dans l'industrie de l'informatique et de l'électronique grand public en raison de sa simplicité d'utilisation et de sa compatibilité avec de nombreux types d'appareils.

Le protocole USB est défini pour les communication entre un hôte par exemple, un ordinateur et un ou plusieurs périphériques USB. Il en existe plusieurs versions, les plus courantes étant : USB 1.1, USB 2.0, USB 3.0, USB 3.1, USB 3.2 et USB 4.0.

Chaque version a apporté des améliorations en termes de débit, de puissance électrique et de fonctionnalités [11].

I.4.1 Description physique

Le bus USB utilise 4 fils ainsi que des connecteurs très particulier. Deux fils pour l'alimentation et deux autres fils pour le transfert de données comme le montre la Figure I.13 [11]. Il existe différentes vitesses de transmission de données :

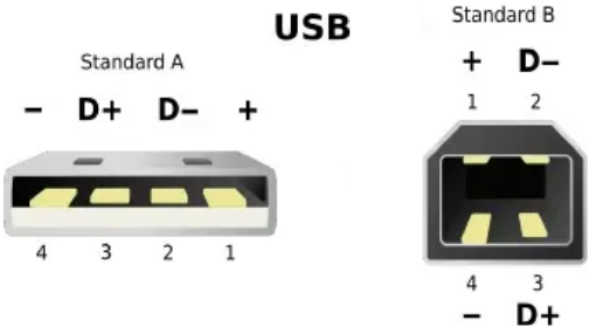


FIGURE I.13 : Description de bus USB [11].

- USB basse vitesse : avec une vitesse de transmission de 1.5Mbps, utilisé avec les périphériques qui demande une faible bande passante,
- USB pleine vitesse : avec une vitesse de 12Mbps utilisé avec des périphériques tel que : des imprimantes, scanners...etc. Ce mode qui a été utilisé pour la transmission de la passerelle CAN/USB,
- USB haute vitesse : avec une vitesse qui atteint les 480MBps utilisé pour un transfert de données importants tel que : des disques dur, caméras aux résolutions.

I.4.2 Topologie

Le protocole USB utilise un système de bus en étoile, où un hôte est connecté à plusieurs périphériques via un concentrateur USB. L'hôte assume le rôle de contrôleur et est responsable de la gestion de la communication avec les périphériques connectés. Chaque périphérique possède un identifiant unique appelé "adresse" et peut recevoir des commandes et envoyer des données à l'hôte.

La communication entre l'hôte et les périphériques USB se fait par l'intermédiaire de "paquets" de données. Les paquets sont des unités de transmission de données qui contiennent à la fois des informations de contrôle par exemple, des commandes de configuration et des données elles-mêmes. Le protocole USB définit également des protocoles de transfert spécifiques [12], tels que :

- Le transfert en mode "bulk" qui permet un transfert de données en vrac,
- Le transfert en mode "interrupt" qui lui permet un transfert de données à intervalles réguliers,
- Le transfert en mode "isochrone" qui lui est un transfert de données en temps réel.

I.4.3 Représentation des trames USB

Les paquets se composent de différents champs spécifiques avec une représentation du un bit LSB [13], ces différents champs sont les suivants :

- **Sync (Synchronisation)** : tous les paquets doivent commencer avec ce champ de 8 bits pour une basse et pleine vitesse et de 32 bits pour une haute vitesse, son rôle est de permettre la synchronisation des horloges d'entremetteur et du récepteur,
- **PID (Process IDentifier)** : Ce champ est utile pour l'identification du type de paquet envoyé,
  Le PID est constitué de 4 bits mais pour s'assure de son envoie on ajoute 4 bits complémentaires pour avoir un format comme indique dans la figure suivante :

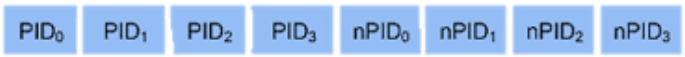


FIGURE I.14 : Constitution du PID [13].

- **ADDR (Address)** : Ce champ détermine le périphérique concerné par la communication, il est composé de 7 bits donc un total de 127 périphériques connectés avec l'adresse 0 qui n'est pas attribuées,

- **ENDP (Endpoint)** : Ce champ est composé de 4 bits pour un totale de 16 terminaison possibles, il permet d'identifier l'extrémité spécifique du périphérique concerné, cela représente le point d'entrée ou de sortie des données sur le périphérique,
- **CRC** : Pour le traitement des erreurs, tous les paquets ont un CRC de 5 bits sauf les paquets de données qui ont un CRC de 16 bits,
- **EOP (End of Packet)** : Ce champ permet de signaler la fin du paquet il est constitué d'un zéro 0.

I.4.4 Les types de paquets

L'USB comporte 4 types de paquets différents [13] :

- Paquet de début de trame (Start Of Frame) SOF,
- Paquet Jeton (Token),
- Paquet Data,
- Paquet d'Etat.

Ce protocole est géré par un hôte donc c'est lui qui initié la transaction avec le premier paquet SOF, ensuite le paquet Jeton pour décrire ce qui va suivre après lui et le mode de communication lecture ou écriture, ainsi que l'adresse du périphérique. Après cela on aura le paquet de data qui transporte les données utiles, et il est suivi par le paquet d'état (handshaking) pour valider la réception de données er rapporter d'éventuelles erreurs.

I.4.5 Type de trames USB

Dans le protocole USB les données sont transférées sous forme de trames qui sont des unités de transmission, il existe plusieurs types de trames utilisées, chacune ayant un rôle spécifique dans la communication [13]. Voici les principaux types de trames :

- Trame de synchronisation (Sync Frame),
- Trame de donnée (Data Frame) ,
- Trame de gestion de bus (Bus Management Frame),
- Trame de token (Token Frame),
- Trame de réponse (Handshake Frame),
- Trame de prévention des erreurs (Error Frame),

Ces différents types de trames sont utilisés en séquence pour permettre la communication fiable entre l'hôte et les périphériques USB. Ils sont encapsulés dans des paquets de données et transmis sur le bus. La structure et le contenu des trames sont définis par les spécifications du protocole USB, garantissant ainsi une communication cohérente et compatible entre les appareils USB.

I.5 Domaines d'utilisation

I.5.1 Bus CAN

Le bus CAN est un type de réseau de communication largement utilisé dans divers domaines industriels grâce à sa communication fiable et robuste entre différents composants électroniques d'un système on le retrouve dans [9, 14] :

- L'automobile pour connecter différents sous-systèmes tels que les moteurs, les transmissions, les freins, la suspension, etc. Il permet également la communication entre les systèmes de divertissement et de navigation,
- L'aérospatiale pour interconnecter différents sous-systèmes tels que les systèmes de commande de vol, les systèmes de communication, etc,
- L'énergie, il est utilisé dans les équipements de production d'énergie tels que les éoliennes et les panneaux solaires pour interconnecter différents sous-systèmes tels que les capteurs, les systèmes de surveillance et de commande,
- L'automatisation industrielle on le retrouve dans les machines de production et les robots industriels pour interconnecter différents sous-systèmes tels que les capteurs, les actionneurs et les systèmes de commande.

I.5.2 Bus CAN FD

Principalement utilisé dans des systèmes nécessitant des taux de transfert de données bien plus élevés que ceux du bus CAN traditionnel et nécessiteux d'une plus grande vitesse de transmission [9, 14] tels que :

- Les systèmes de contrôle moteur avancés dans l'industrie automobile tels que les systèmes de conduite assistée, les systèmes de navigation,
- Les systèmes de vision pour connecter différents sous-systèmes tels que les caméras, les microphones, les haut-parleurs, les capteurs de mouvement et les capteurs de température. Les données collectées par ces sous-systèmes sont ensuite traitées par un système de traitement d'image pour fournir des informations en temps réel sur l'environnement surveillé, mais aussi pour des systèmes de vision pour des véhicules autonomes et des drones,

- Les systèmes de contrôle de processus industriels à base de caméras et de capteurs pour assurer le contrôle, la sécurité et la fiabilité du processus,
- Les équipements médicaux modernes tels que les scanners IRM et les systèmes de monitoring utilisent également le protocole CANFD pour communiquer,
- Transport en commun tels que les trains et les bus utilisent également le protocole CAN FD pour communiquer entre les différents sous-systèmes tels que les moteurs, les freins, les systèmes de contrôle de traction, etc,
- Équipements de construction comme les engins de chantier et les équipements lourds tels que les excavatrices et les bulldozers utilisent également le protocole,
- Ascenseur et monte-charge, contrôle d'accès, commande d'éclairage.

I.5.3 Analyseur CAN

Principalement utilisé dans les domaines suivants [9, 14] :

- L'automobile, pour diagnostiquer résoudre les problèmes de communication entre différents sous-systèmes dans une voiture,
- L'aérospatiale pour surveiller et diagnostiquer les systèmes de communication des avions,
- L'énergie pour surveiller et diagnostiquer les systèmes de production d'énergie,
- L'automatisation industrielle pour diagnostiquer et résoudre les problèmes de communication entre les différents sous-systèmes dans les machines de production et les robots.

I.5.4 Protocole USB

Le protocole USB est utilisé dans de nombreux domaines et secteurs pour connecter et communiquer avec différents types de périphériques électroniques. Voici quelques-uns des principaux domaines d'utilisation du protocole USB [13] :

- Informatique grand public : Le protocole USB est largement utilisé dans les ordinateurs personnels, les ordinateurs portables et les tablettes pour connecter des périphériques tels que les claviers, les souris, les imprimantes, les scanners, les webcams, les haut-parleurs, les disques durs externes, les clés USB, etc.
- Téléphonie mobile : Les smartphones et les tablettes utilisent souvent des connexions USB pour charger leur batterie (via des câbles USB) et pour transférer des données entre l'appareil et l'ordinateur (par exemple, pour la synchronisation de données, le transfert de fichiers, etc.).

- Audio et vidéo : De nombreux périphériques audio et vidéo, tels que les haut-parleurs, les casques, les microphones, les caméras, les lecteurs MP3, les enregistreurs vidéo, les moniteurs, les téléviseurs et les systèmes de divertissement domestiques, utilisent le protocole USB pour la transmission de données et l'alimentation électrique.
- Automatisation industrielle : Le protocole USB est utilisé dans l'automatisation industrielle pour connecter des capteurs, des actionneurs, des contrôleurs et d'autres périphériques aux systèmes de contrôle et de supervision.
- Médical : Les dispositifs médicaux, tels que les scanners médicaux, les moniteurs de santé, les appareils de diagnostic, les pompes à perfusion, les électrocardiographes, peuvent être connectés à des ordinateurs et à des systèmes de gestion des dossiers médicaux via des ports USB pour le transfert de données et la configuration.

Conclusion

En conclusion de ce chapitre, nous avons pu observer l'évolution de la technologie de communication du bus CAN qui fut initialement conçu pour répondre aux besoins des applications automobiles, mais qui a su s'adapter et toucher de nombreux autres domaines tels que l'industrie, l'aéronautique et les systèmes embarqués.

Malgré l'évolution des besoins en termes de débits de transmission et de flexibilité dans la gestion des données, l'émergence du protocole CAN FD a marqué un tournant pour les systèmes complexes et les applications nécessitant une communication rapide et fiable.

Ainsi en prenant en compte toutes les spécificités de ces protocoles il devient plus aisé pour nous de choisir les composants appropriés pour pouvoir démarrer la réalisation de notre circuit.

Chapitre II

Conception et Réalisation du Système de Passerelle CAN-USB

Introduction

Dans ce chapitre nous nous focalisons sur la conception et la réalisation d'un analyseur CAN et d'une passerelle USB/CAN. Pour réaliser cela, nous allons utiliser deux cartes STM32H743vit6, une pour envoyer les données et simuler une machine ou véhicule et la deuxième qui sert à récupérer et à traiter et diagnostiquer ces données

II.0.1 Synoptique global du système

La Figure II.1 illustre la présentation générale du système ou le schéma de principe de l'analyseur que nous avons conçu et réalisé avec tous les composants utilisés et dont le schéma électrique est détaillé en Figure II.2, et les détails de chaque partie seront expliqué dans des parties à part dans un ordre de conception hiérarchique, bloc par bloc.

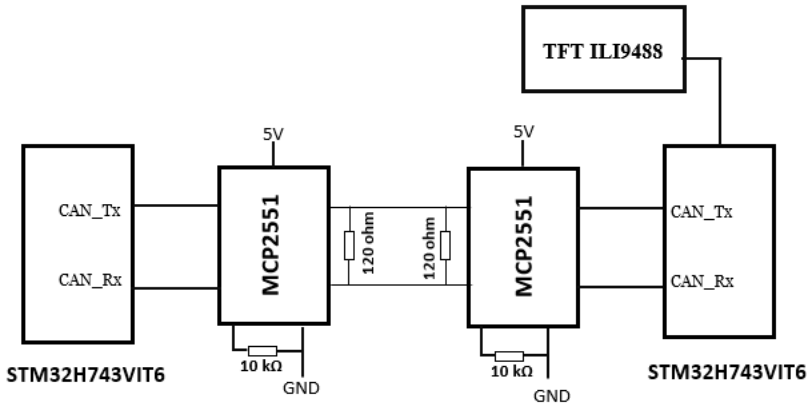


FIGURE II.1 : Schéma synoptique global de l'analyseur conçu.

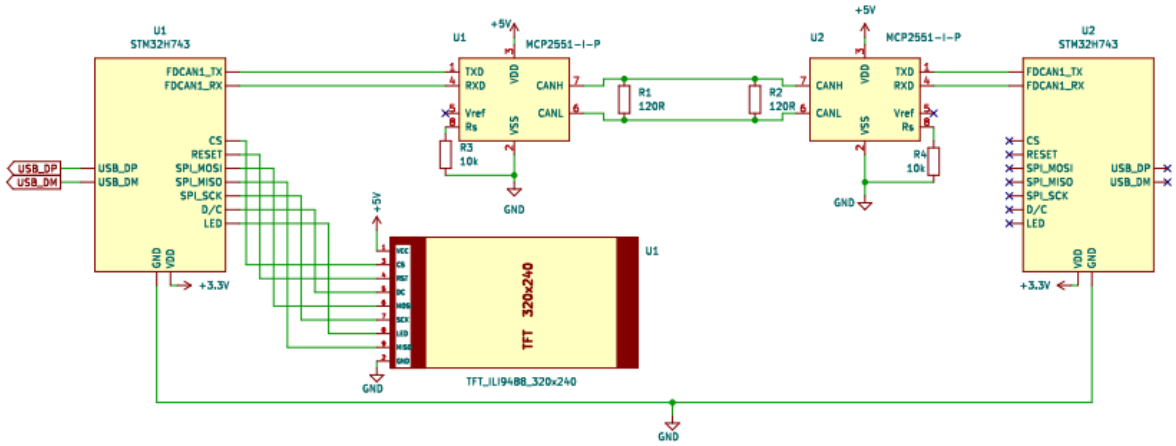


FIGURE II.2 : Schéma électrique global de l'analyseur conçu.

Pour la réalisation de la partie hardware les deux cartes STM sont relies entre elles, créant ainsi un bus et avant chacune d'elles un contrôleurs CAN et deux résistances de lignes seront placés. La carte de réception est aussi relie à un afficheur LCD pour pouvoir afficher un chronogramme et représenter les données reçues. Tandis que la partie software est conçue à l'aide des différents éditeurs STM comme le STM32CubeIDE et STM32CubeMX.

II.1 Les microcontrôleurs STM32

STM32 est une famille de microcontrôleurs 32 bits réalisés par STMicroelectronics et dégroupées en dix-sept sous-familles, ayant chacune ses propres caractéristiques, mais sont toutes basées sur le cœur du processeur Arm Cortex-M. Chaque microcontrôleur se compose d'une RAM (Random Access Memory) statique, d'un cœur du processeur, de la mémoire flash et de divers autres périphériques [15].

Les microcontrôleurs STM32 sont utilisés dans diverses applications. Ils offrent une combinaison de hautes performances, un fonctionnement à faible consommation d'énergie et à basse tension.

La programmation des microcontrôleurs STM32 est généralement réalisée à l'aide du langage de programmation C ou C++ ; Ces programmes, pour bien fonctionner, sont développés dans un environnement très riche, composé d'un large choix d'outils et de logiciels, notamment l'outil de configuration graphique STM32CubeMX, l'environnement de développement intégré STM32CubeIDE, ainsi qu'une série de bibliothèques et de middleware [15].

II.1.1 Choix de microcontrôleurs STM32

Le STM32H743VIT6 est un microcontrôleur fait partie de la série des microcontrôleurs STM32H7 STMicroelectronics, ils sont basés sur le cœur Arm Cortex-M7 32-bit RISC (Reduced Instruction Set Computer) cadencés à 480MHz, et intègrent des mémoires embarquées à grande vitesse, une mémoire Flash et une mémoire RAM, ainsi qu'une large gamme d'E/S et de périphériques intégrés qui facilitent l'interface avec d'autres appareils. De plus, le STM32H743VIT6 est un microcontrôleur très performant qui convient à une variété d'applications qui nécessitent une forte puissance et des périphériques avancés [15, 16], parmi ses caractéristiques on peut citer [16]. Voir Annexe B

- **Cœur :** Cœur Arm Cortex-M7 avec FPU (Floating Point Unit) double précision et cache L1 : 16 Ko de données et 16 Ko de cache d'instructions, fréquence qui peut aller jusqu'à 480MHz.
- **Mémoires :** Une mémoire flash qui peut aller jusqu'à 2Mo et une mémoire ROM (Read Only Memory) de 1 Mo (jusqu'à 864 Kode SRAM (Static Random Access

Memory) à utiliser et 4 Ko pour la sauvegarde), comprend aussi un contrôleur de mémoire externe avec un bus de données de 32 bits et une unité de calcul de CRC,

- Dispose 168 pins d'entrées/sorties avec possibilités d'interruption,
- Une alimentation USB dédiée intégrant un régulateur interne de 3,3,
- Equipés d'un ensemble de 35 périphériques de communication, notamment une interface Ethernet avec contrôleur DMA, des ports série (USART, UART, I2C, SPI), Port USB, Port pour contrôleur CAN, deux avec CANFD et un avec TT-CAN (Time Triggered-CAN),
- Un ensemble de 11 périphériques analogiques.
- Dispose des interfaces JTAG (Joint Test Action Group)/SWD(Serial Wire Debug) pour le débogage et la programmation.

On se basant sur les caractéristiques et différents critères déjà cités en haut, nous avons décidé de choisir la carte STM32H743vit6, notamment pour les principales raisons suivantes qui ont renforcé notre choix de cette carte :

- La carte possède des périphériques pour le CAN FD qui sont compatibles avec le bus CAN, ceci n'est pas le cas pour toutes les autres cartes,
- Les cartes STM sont des produits sans licences et donc pas besoin d'acheter pour pouvoir les utiliser pour notre projet,
- Une large disponibilité de projet open source avec cet environnement,
- Les produits STM sont considérés comme des produits professionnels pour des personnes affranchis et non pas pour des débutants,
- Une diversité de communautés d'ingénieurs et programmeurs travaillant sur les produits STM ce qui les rend grand public,
- La disponibilité de la carte sur le marché et son coût abordable par rapport à d'autres,
- Les différents logiciels mis à disponibilité par STM pour simplifier la programmation de notre projet comme : STM32CubeIde, STM32CubeMx et STM32Studio...etc,
- La puissance de la carte H7 qui est l'une des cartes les plus puissantes de la dernière gamme de STM,
- Et encore, le dernier critère qui nous a poussé à faire ce choix et le fait que nous avons déjà une certaine expérience avec l'environnement STM après avoir réalisé quelques projets dessus durant notre formation,

II.2 Contrôleurs CAN

Le CAN Controller MCP2551 est un module émetteur-récepteur beaucoup utilisé dans le réseau CAN et qui sert à assurer l'interface entre le contrôleur de protocole CAN et le bus physique et sert également à interpréter et convertir les signaux logiques en niveau de tension, disposant d'une capacité d'émettre et de recevoir à grande vitesse qui peut aller jusqu'à 1 Mbps [17].

Le MCP2551 comprend une capacité d'émission et de réception par un bus différentiel ce qui permet de minimiser le bruit ainsi que d'autres caractéristique visant à garantir un fonctionnement fiable telles que [17] :

- La détection et la correction des erreurs,
- La retransmission des messages perdus,
- Corresponds au systèmes 12v et 24v,
- Conçu pour fonctionner à des températures comprises entre, de -40°C à 125°,
- Capacité de fonctionnement en mode veille à faible courant et comprend une fonction de réveil qui permet l'appareil (d'être mis sous tension),
- Comprend plusieurs modes de fonctionnement, high-speed, Slope-Control, Standby ainsi que d'autres,
- Ce dispositif fournit également des fonctions de protection contre les décharges électrostatiques, les transitoires de haute tension ainsi que les surcharge et les courts-circuits, ce qui protège l'appareil en cas de défaillance,

Le boîtier de MCP2551 possède 8 broches, comme le montre la Figure II.3, dont les broches d'alimentation (doit être connecte à une alimentation de 5V) et de masse. Les différentes broches constituant ce circuit sont décrites ci-après.

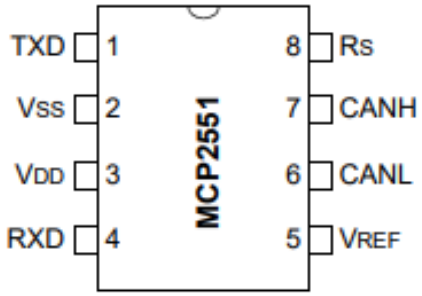


Figure II.3 : Brochage du contrôleur CAN, le MCP2551 [17].

Une broche pour la sortie de tension de référence $V_{Réf}$ qui est égale à $\frac{V_{DD}}{2}$. Deux broches CANH pilote le coté haut du bus différentiel et CANL pilot le coté bas du bus,

ces deux broches sont liées en interne au comparateur d'entrée de réception pour avoir la possibilité de mesurer la tension différentielle entre ces deux broches (voir Figure B.2), tandis que les broches $T_X$et R_X sont spécifiques à la communication CAN, et pour finir une broche R_S utilisée pour sélectionner le mode de fonctionnement (High-Speed, Slope-Control , Standby) via une résistance de polarisation externe. La Figure II.4 représente le branchement du contrôleur MCP2551. Voir Annexe B

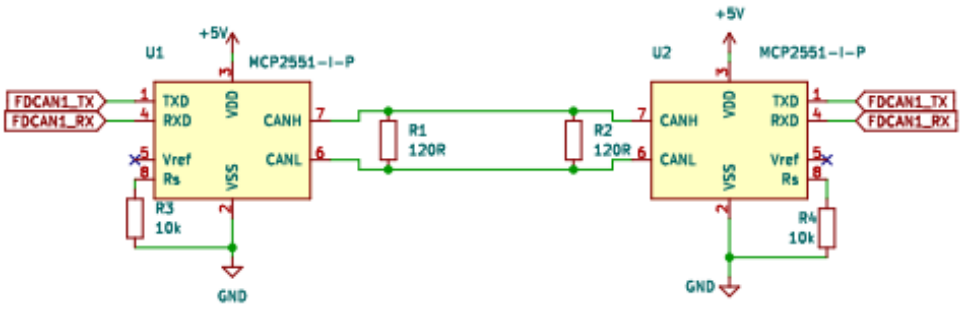


FIGURE II.4 : Branchement du contrôleur MCP2551.

Le tableau II.2 ci-dessous résume les fonctions de chaque Pin.

TABLEAU II.1 : Description du brochage du contrôleur CAN, le MCP2551 [17].

Numéro de la Pin	Nom de la Pin	Fonction de Pin
1	T_X	Entrée de transmission de données
2	V_{SS}	La masse
3	V_{DD}	Tension d'alimentation
4	R_X	Sortie de réception de données
5	V_{REF}	Tension de référence
6	CANL	CAN Low
7	CANH	CAN High
8	R_S	Entrée de contrôle de la pente

Le MCP2551 est utilisé largement dans les applications automobiles et industrielles très en vogue dans le domaine de l'embarqué, bien évidemment, il existe d'autre modèles ou références d'émetteur récepteur CAN, comme le MCP25625, MCP2515, MCP2518fd, MCP2510-I/P et d'autres dans le marché.

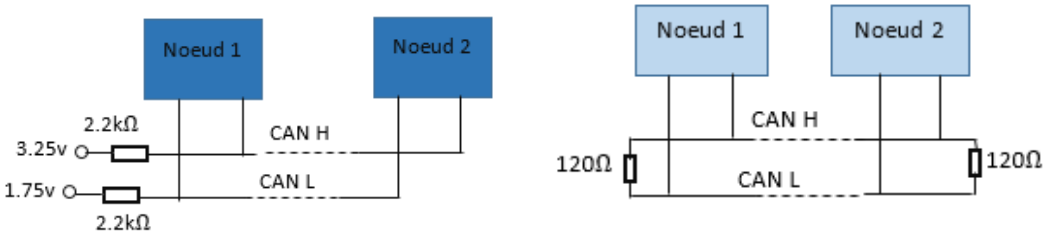
Notre choix s'est focalisé sur le MCP2551 en raison de sa facilité d'utilisation, ses caractéristiques que nous avons déjà vues et déjà travaillé avec précédemment, sa fiabilité, ses performances et sa compatibilité avec les autres composants de notre système, ce qui simplifie la mise en œuvre de la communication CAN et permet de réduire les délais et les coûts de développement, sans oublier la disponibilité de sa documentation.

Résistance de polarisation : Les résistances de polarisation permettent de contrôler la pente de transition entre les signaux CANH et CANL, ce qui réduit les émissions

des interférences électromagnétiques qui affectent le fonctionnement par des signaux indésirables. La valeur de ces deux résistances est de 10KΩ [17].

II.3 Résistances de terminaison

Les résistances de terminaison de bus sont indispensables au bon fonctionnement de la communication, elles permettent de réduire le bruit électrique dû au fléchissement de signaux sur les extrémités. La valeur de chaque résistance de terminaison doit être égale à l'impédance du câble, donc 120Ω. Ces résistances sont placées à l'extrémité physique du réseau CAN en tenant compte la topologie et l'architecture du réseau CAN Low Speed/High Speed, voir la Figure II.5 [18].



(a) CAN Low-speed (b) CAN High-speed

FIGURE II.5 : Branchement de résistances de terminaison.

II.4 Ecran LCD

Le module LCD TFT ILI9488 (Thin-Film Transistor Liquid Crystal Display) est l'un des afficheurs qui utilisent une communication SPI à l'aide de quatre fils pourvu d'un contrôleur intégré et d'une résolution de 320x480 avec la possibilité d'utiliser l'écran tactile. On retrouve aussi un circuit de contrôle pour le rétroéclairage de l'écran. Parmi ces caractéristiques notables [19], on peut citer :

- Résolution de 480x320,
- Écran riche en couleurs de 3,5 pouces, supportant l'affichage des couleurs 18 bits 65k,
- Capacité d'éclairer l'écran avec seulement quelques entrées-sorties, utilisant le bus série SPI,
- Possibilité d'ajouter une carte SD,
- Procure une multitude d'assistances dans plusieurs domaines différents,

- Comprend 9 pins, le Tableau II.2 résume les noms de chaque pin avec une petite description.

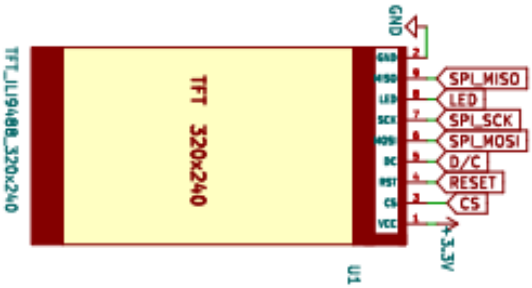


FIGURE II.6 : Schéma électrique de l'afficheur LCD.

TABLEAU II.2 : Description du brochage du LCD [?]

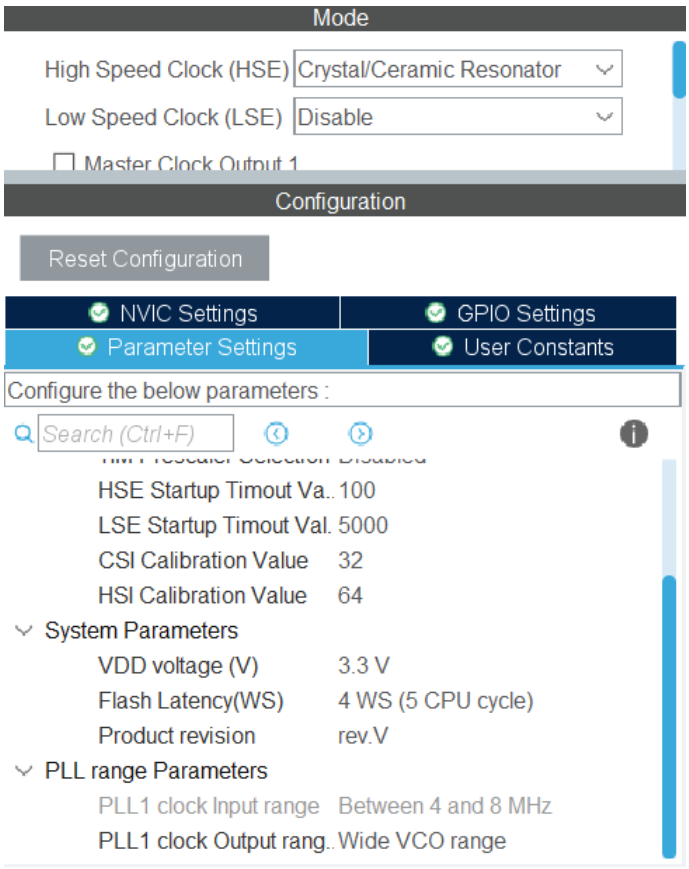
Numéro	Pin	Description
1	V_cc	Alimentation de LCD (3.5 5v)
2	GND	La masse
3	CS	Chip Select
4	RESET	Remise à zéro
5	DC/RS	Pin pour la sélection : état haut : mode registre, état bas : donnée
6	SDI (MOSI)	Pin pour la communication SPI
7	SCK	Pin pour l'horloge de la communication SPI
8	LED	Pin pour le contrôle de la luminosité et de l'électro-éclairage, s'il n'est pas utilisé, elle doit être banchée sur 3.3V
9	SDO (MISO)	Pin pour la communication SPI

II.5 Principe de fonctionnement du circuit et principe de la communication

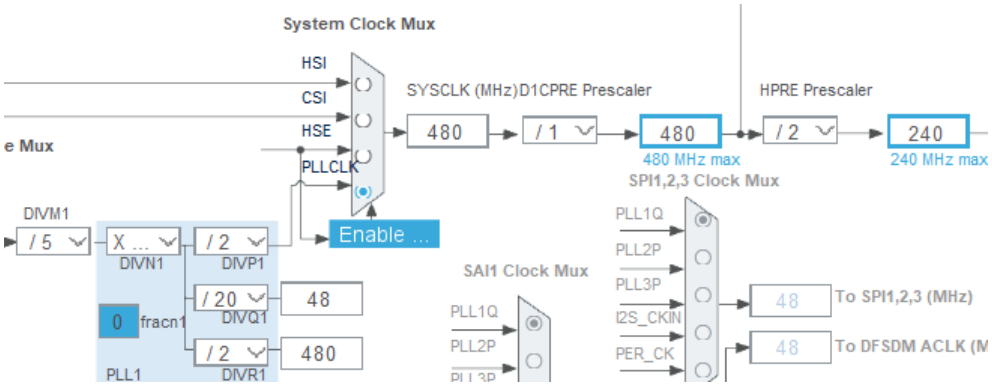
Pour la réalisation de la passerelle/analyseur le travail est divisé en deux parties distincts : hardware et software. Pour la partie hardware en commence par relié les deux cartesSTM entre elles, créant ainsi un bus CAN de 2 nœuds, ensuite on rajoute deux contrôleurs CAN MCP2551 et deux résistances de lignes de 120Ω pour améliorer et nous assurer d'une bonne transmission des données en diminuant les risques d'erreurs et de parasites sur le bus, puis relier la carte qui sert à la réception avec un afficheur LCD TFT afin d'illustrer ou visualiser les données transmisses avec un chronogramme ainsi que leur significations comme indiqué sur le schéma synoptique Figure II.1.

Une fois toutes ces étapes terminées, nous pouvons commencer le travail sur la partie software à l'aide des différents éditeurs mis à disposition par STM comme le STM32CUBEIDE et STM32CUBEMX.

Nous commençons par configurer l'horloge de notre carte STM32H743vit6 qui peut aller jusqu'à 480MHz et nous comptons de bien profiter pleinement de toutes ses capacités, en modifiant ainsi quelques paramètres comme indiqué sur la figure ci-dessous (Figure II.7) notre carte fonctionnera ainsi avec une horloge de 480MHz.



(a)



(b)

FIGURE II.7 : Configuration d'horloge.

Nous changeons la version du produit qui se trouve sous la section « System Parameters » qui par défaut est réglé en « REV. Y» ce qui diminue les capacités de la carte et empêche l'utilisation de l'horloge au-dessus de 240MHz, alors on la règle en «rev. V ».

Une fois l'horloge calibrée, la deuxième étape est la configurations des pins essentielles pour la transmissio, nous commençons par les pins du périphériques FDCAN de la carte qui consistent en 2 pins «FDCAN1_TX» et «FDCAN1_RX» avec bien entendu toutes les configurations recuisent tel que le débit de transmission, la taille de la donnée, le nombre de filtre à appliquer, l'activation de l'auto retransmission pour ré-envoyer les données s'il y a eu une erreur et l'utilisation ou non des interruptions, dans notre cas il est beaucoup plus intéressant d'utiliser les interruptions pour la transmission en raison de :

- Leurs efficacités dans cette dernière car ils permettent une communication asynchrone, ce qui signifie que les nœuds peuvent transmettre et recevoir des données de manière indépendante sans nécessiter une synchronisation stricte.
- Un temps de réponse faible ce qui permet une réactivité rapide.
- Priorisation des événements, les interruptions permettent également de hiérarchiser les événements et les transmissions de données en leur attribuant des niveaux de priorité. Cela est particulièrement utile dans les systèmes où des données critiques ou urgentes doivent être transmises rapidement, car les interruptions de priorité plus élevée peuvent interrompre les interruptions de priorité inférieure pour transmettre leurs données, ce qui est notre cas pour les instructions entre le LCD qui utilise le DMA et la réception des données sur le bus CAN.
- Réduction de la charge du processeur : En utilisant les interruptions, le processeur peut être mis en veille lorsque le bus est inactif, réduisant ainsi la consommation d'énergie globale du système.

Une fois tous ces critères pris en compte, la configuration du périphérique FDCAN de notre carte est illustrée dans les figures ci-dessous, FigureII.8 :

Basic Parameters	
Frame Format	FD mode with BitRate Switching
Mode	Normal mode
Auto Retransmission	Enable
Transmit Pause	Disable
Protocol Exception	Disable
Nominal Sync Jump Width	1
Data Prescaler	1
Data Sync Jump Width	1
Data Time Seg1	1
Data Time Seg2	1
Message Ram Offset	0
Std Filters Nbr	1
Ext Filters Nbr	0
Rx Fifo0 Elmts Nbr	1
Rx Fifo0 Elmt Size	8 bytes data field

(a)

Rx Fifo1 Elmt Size	8 bytes data field
Rx Buffers Nbr	0
Rx Buffer Size	8 bytes data field
Tx Events Nbr	0
Tx Buffers Nbr	0
Tx Fifo Queue Elmts Nbr	1
Tx Fifo Queue Mode	FIFO mode
Tx Elmt Size	8 bytes data field
Clock Calibration Unit	
Bit Timings Parameters	
Nominal Prescaler	1
Nominal Time Quantum	20.833333333333332 ns
Nominal Time Seg1	93
Nominal Time Seg2	2
Nominal Time for one Bit	2000 ns
Nominal Baud Rate	500000 bit/s

(b)

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
FDCAN1 interrupt 0	☑	0	0
FDCAN1 interrupt 1	☐	0	0
FDCAN calibration unit interrupt	☐	0	0

Pin Na...	Signal on ...	Pin Conte...	GPIO outp...	GPIO mode	GPIO Pull-...	Maximum ...	Fast Mode	User Label
PB8	FDCAN1...	n/a	n/a	Alternate ...	No pull-up ...	Low	Disable	
PB9	FDCAN1...	n/a	n/a	Alternate ...	No pull-up ...	Low	Disable	

(c)

FIGURE II.8 : Configuration de FDCAN.

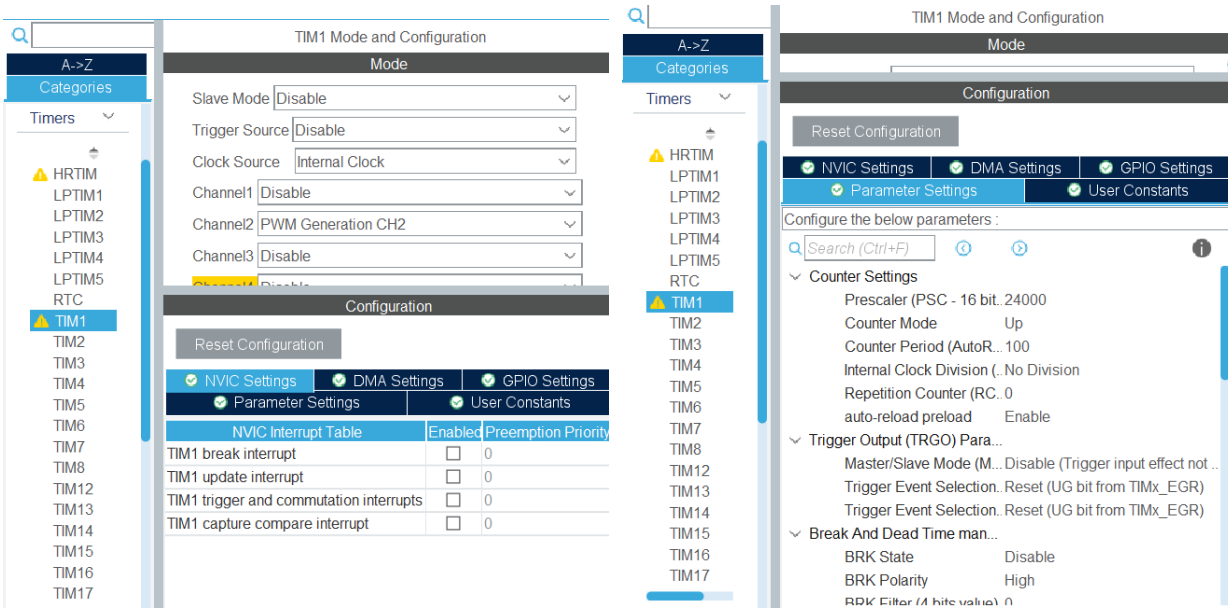
Après cela nous configurant le périphérique USB de la carte pour l'utiliser dans notre passerelle CAN/USB comme ceci (Figure II.9) :

Speed	Full Speed 12MBit/s
Enable internal IP DMA	Disabled
Low power	Disabled
Battery charging	Disabled
Link Power Management	Disabled
Use dedicated end point 1 interrupt	Disabled
VBUS sensing	Disabled
Signal start of frame	Disabled

Pin Na...	Signal on ...	Pin Conte...	GPIO outp...	GPIO mode	GPIO Pull-...	Maximum ...	Fast Mode	User Label
PA11	USB_OT...	n/a	n/a	Alternate ...	No pull-up ...	Low	n/a	
PA12	USB_OT...	n/a	n/a	Alternate ...	No pull-up ...	Low	n/a	

FIGURE II.9 : Configuration de l'USB.

Pour finir, nous devons configurer les pins du LCD TFT ILI9488, nous commençons par l'activation d'un port SPI et d'un timer pour la génération de pulsation PWM qui servira à contrôler la puissance fournie au LCD, aussi la configuration d'autre pins en mode GPIO_OUTPUT pour diverses contrôle propre au LCD comme CS, D/C et RESET en retrouve toutes ces configurations représenter dans les figures suivantes, Figure II.10a et Figure II.10b :



(a) (b)

FIGURE II.10 : Configuration du Timer.

Nous configurons le Mode SPI en FULL-Duplex Master avec une taille de 8 Bits et en mode « MSB First » comme le montre la Figure II.11.

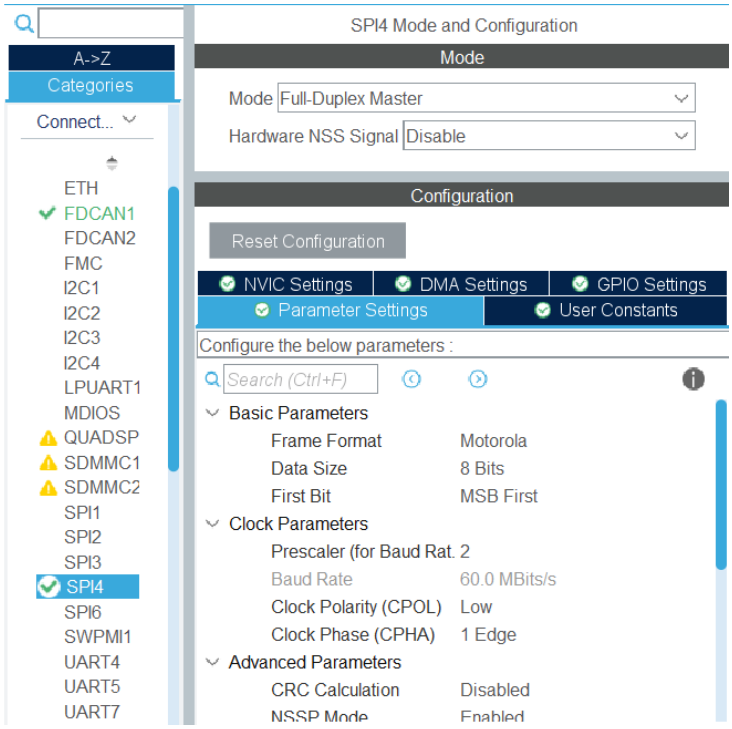


FIGURE II.11 : Configuration du SPI.

Les pins PE9, PE10 et PE15 (Figure II.12) sont configurées comme des GPIO_OUTPUT, ces pines seront utilisés respectivement pour représenter les pins RESET, DC et CS du LCD ILI9488 qu'on a vu précédemment.

Pin...	Signal ...	Pin Co...	GPIO ...	GPIO ...	GPIO ...	Maxim...	Fast M...	User L...	Modified
PE9	n/a	n/a	Low	Output ...	No pull...	Low	n/a	RESET	☑
PE10	n/a	n/a	Low	Output ...	No pull...	Low	n/a	DC	☑
PE15	n/a	n/a	Low	Output ...	No pull...	Low	n/a	CS	☑

FIGURE II.12 : Configuration des pins PE9, PE10, PE15.

Pour pouvoir basculer entre nos deux protocoles CAN et CAN FD, ainsi que de changer le débit de la transmission, nous utiliserons les deux boutons de la carte STM32H743vit6 en mode interruption pour la carte d'émission et nous utiliserons l'interface pour la carte de réception en envoyant des commandes via USB.

A chaque fois qu'on appuie sur le bouton K1 représenté par la pin PE3, le protocole change, tandis qu'un appuie sur le bouton K2 représenté par la pin PC5 augmente le débit de transmission avec comme limite de 2 Mbps pour les transmission CAN FD, et bien sur 1 Mbps pour le protocole CAN et si on appuie trop sur le bouton et qu'on dépasse le débit Maximum alors le débit est reprogrammer au minimum pour les deux protocoles.

Pin...	Signal ...	Pin Co...	GPIO ...	GPIO ...	GPIO ...	Maxim...	Fast M...	User L...	Modified
PC5	n/a	n/a	n/a	Extern...	Pull-up	n/a	n/a		☑
PE3	n/a	n/a	n/a	Extern...	Pull-up	n/a	n/a		☑

FIGURE II.13 : Configuration des pins pour les boutons.

La figure suivante (Figure II.14) représente la configuration finale des pins de notre processeur

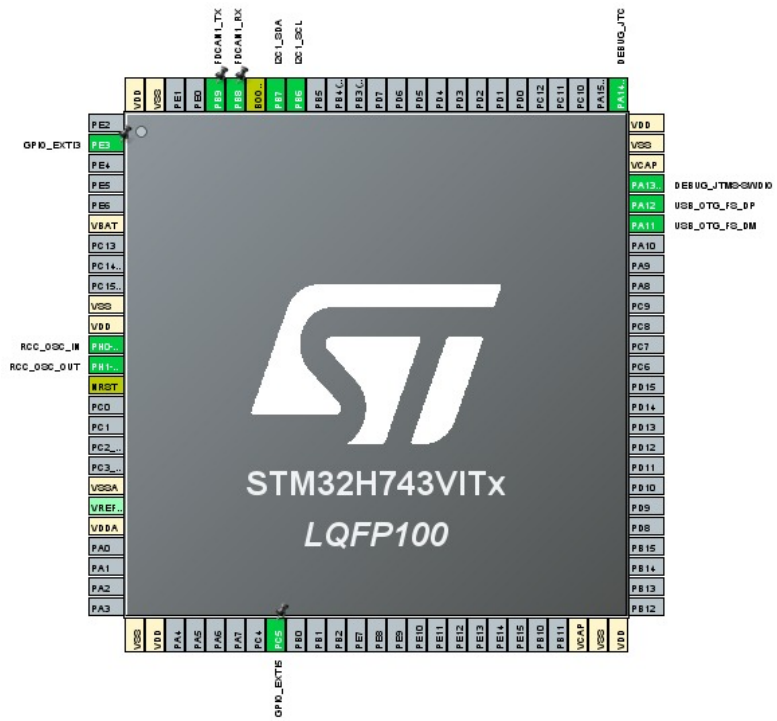


FIGURE II.14 : Configuration finale des pins (Transmission).

Une fois les configurations des différents pins sont effectuées comme indiqué dans la Figure II.14, le code de base est généré et ainsi nous pouvons commencer à écrire un code en langage C qui servira à réceptionner les données et à effectuer un diagnostic de ceux-là, selon les normes internationales et à les afficher sur l'écran LCD, toute en les envoyant via l'USB vers une interface que nous avons créée et qui fera le même diagnostic que la carteSTM et affichera les résultats

Nous aurons aussi besoin d'un autre code pour les envoyer d'une façon continue. La configuration sera la même que celle du premier code sauf que nous n'aurons besoin que des pins « FDCAN1_TX » et « FDCAN1_RX », l'horloge de la carte ainsi que le débit et le nombre de filtre et le type de transmission doivent être les mêmes que celle utilise dans le premier code.

Au final, nous aurons une configuration des pins comme illustré dans la FigureII.15 :

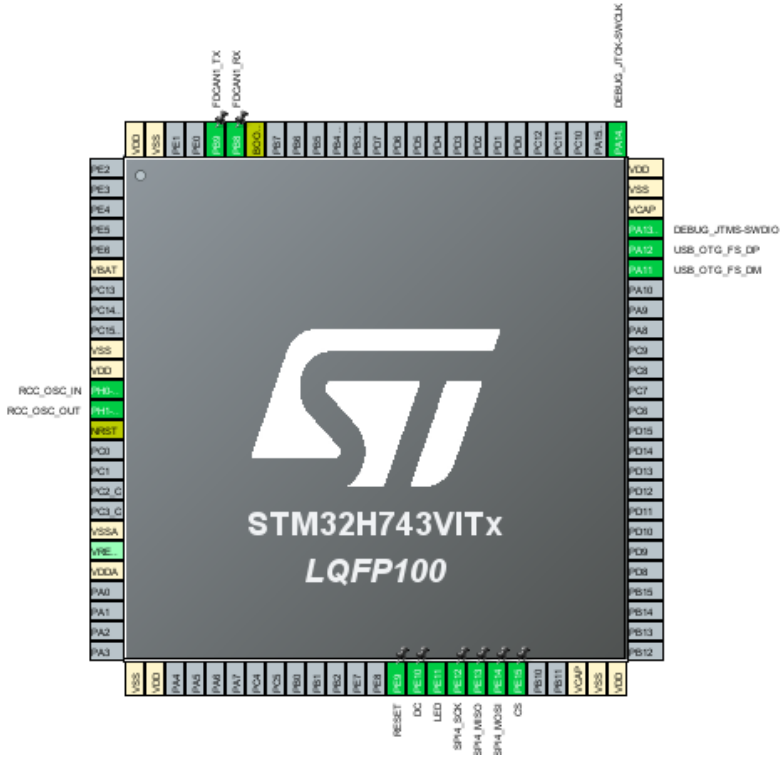


FIGURE II.15 : Configuration finale des pins (Réception).

Les cartes STM peuvent être programmées soit en C ou bien C++, l'éditeur STM32-CUBEIDE que nous avons choisi d'utiliser permet l'utilisation de ces deux langages, mais nous avons opté pour le langage C car c'est un langage pour sa simplicité ; nous commençons donc la programmation de la partie software avec le code qui servira à envoyer en continue les données sur le bus CAN. Ces données ne sont pas choisies arbitrairement, mais se sont celles qui existent vraiment dans le monde de l'automobile (le type de la donnée et son ID sont tirés d'une base de données authentique de véhicule, donc, si on branche une voiture dans un model compatible avec celui de la base de données, notre analyseur pourra récupérer les récupérer naturellement.

Pour commencer on déclare les bibliothèques à utiliser comme indiquée sous le Figure II.16 :

```
/* Includes
#include "main.h"
#include "usb_device.h"
/* Private includes
/* USER CODE BEGIN Includes */
#include "usbd_cdc_if.h"
#include "string.h"
/* USER CODE END Includes */
```

FIGURE II.16 : Déclaration des bibliothèques.

Ensuite nous déclarons les variables qui serviront à l'envoie des données et qui n'ont pas été déclaré lors de la génération du code par CUBEIDE (Figure II.17).

```
/* Private variables
FDCAN_HandleTypeDef hfdcan1;
/* USER CODE BEGIN PV */
uint8_t buffer[1],R[16],s,ss,b[90]
FDCAN_FilterTypeDef sFilterConfig;
FDCAN_TxHeaderTypeDef
VEHICLE_SPEED,AMB_AIR_TEMP,BAROMETRIC_PRESSURE,FUEL_SYS_STATUS,FUEL_PRESSURE,ENGINE_RUNTIME;
FDCAN_TxHeaderTypeDef DIST_TRAVELED_MIL,TIME_RUN_WITH_MIL,FUEL_TYPE;
FDCAN_TxHeaderTypeDef Temperaturefd,Pressionfd,lumierefd,ABSfd;
uint8_t vit[] ={0x0, 0x0,0x0,0x14, 0x0, 0x0, 0x00, 0x0};
uint8_t Temp[] ={0x0, 0x0, 0x0,0x32, 0x0, 0x0, 0x00, 0x0};
uint8_t Pressure[] ={0x0, 0x0,0x0,0x65, 0x0, 0x0, 0x00, 0x0};
uint8_t fuel_sys_status[] ={0x0, 0x0,0x0,0x01, 0x0, 0x0, 0x00, 0x0};
uint8_t Fuel_Pressure[] ={0x0, 0x0,0x0,0x1c, 0x0, 0x0, 0x00, 0x0};
uint8_t engine_runtime[] ={0x0, 0x0,0x0,0x0, 0x0, 0x0, 0x00, 0x0};
uint8_t distance[] ={0x0, 0x0,0x0,0x0, 0x0, 0x0, 0x00, 0x0};
int i=0,j=0;
FDCAN_RxHeaderTypeDef RxHeader;
```

FIGURE II.17 : Déclaration des variables.

La variable « hfdcan1 » (Figure II.18) fut générée par l'éditeur CUBEIDE, représente une structure regroupant les configurations essentielles du périphérique utilisé, parmi lesquelles on a :

- Le périphérique FDCAN utilisé,
- Le type de protocole CAN utilisé,

- La valeur des « Prescaleurs » et des « Time quanta » utilisé pour avoir un débit de 125Kbps,
- Taille de la donnée transmisse,
- Nombre d'éléments utilisé pour l'envoie de la donnée à l'aide de l'interruption.

```
hfdcan1.Instance = FDCAN1;                hfdcan1.Init.FrameFormat = FDCAN_FRAME_CLASSIC;
hfdcan1.Init.Mode = FDCAN_MODE_NORMAL;hfdcan1.Init.AutoRetransmission = ENABLE;
hfdcan1.Init.TransmitPause = DISABLE; hfdcan1.Init.ProtocolException = DISABLE;
hfdcan1.Init.NominalPrescaler = 4;    hfdcan1.Init.NominalSyncJumpWidth = 2;
hfdcan1.Init.NominalTimeSeg1 = 93;    hfdcan1.Init.NominalTimeSeg2 = 2;
hfdcan1.Init.DataPrescaler = 1;       hfdcan1.Init.DataSyncJumpWidth = 1;
hfdcan1.Init.DataTimeSeg1 = 1;        hfdcan1.Init.DataTimeSeg2 = 1;
hfdcan1.Init.MessageRAMOffset = 0;    hfdcan1.Init.StdFiltersNbr = 1;
hfdcan1.Init.RxFifo0ElmtsNbr = 1;     hfdcan1.Init.RxFifo0ElmtSize = FDCAN_DATA_BYTES_8;
hfdcan1.Init.RxBuffersNbr = 0;        hfdcan1.Init.RxBufferSize = FDCAN_DATA_BYTES_8;
hfdcan1.Init.TxFifoQueueElmtsNbr = 1; hfdcan1.Init.TxFifoQueueMode = FDCAN_TX_FIFO_OPERATION;
hfdcan1.Init.TxElmtSize = FDCAN_DATA_BYTES_8;
if (HAL_FDCAN_Init(&hfdcan1) != HAL_OK){ Error_Handler();}
```

FIGURE II.18 : La variable hfdcan1.

Nous avons déclaré toutes les autres variables comme : la variable « sFilterConfig » de type « FDCAN_FilterTypeDef » qui est une structure propre au périphérique FDCAN, elle sert à configure le filtre à appliquer comme les IDs à refuser, le type de masque a appliqué la façon dont la donnée sera reçue.

```
  sFilterConfig.IdType = FDCAN_STANDARD_ID;
  sFilterConfig.FilterIndex = 0;
  sFilterConfig.FilterType = FDCAN_FILTER_MASK; // Ignore because
FDCAN_FILTER_TO_RXBUFFER
  sFilterConfig.FilterConfig = FDCAN_FILTER_TO_RXFIFO1_HP;
  sFilterConfig.FilterID1 = 0x0; // ID Reception
  sFilterConfig.FilterID2 = 0x0; // Ignore because FDCAN_FILTER_TO_RXBUFFER
  sFilterConfig.RxBufferIndex = 0;
  if(HAL_FDCAN_ConfigFilter(&hfdcan1, &sFilterConfig) != HAL_OK) {Error_Handler();}
```

FIGURE II.19 : Configuration de la variable sFilterConfig.

Nous avons déclaré le « FilterID1 » à 0x0 et cela signifiera que n'importe quelle ID sera accepté par la carte. Nous avons déclaré aussi 9 vecteurs de type « Bytes » ou « uint8_t » de taille 8 pour représenter les 8 octets à envoyer, avec eux nous déclarons aussi 9 variables de type « FDCAN_TXHeaderTypeDef » qui est aussi une structure interne au périphérique FDCAN de la carte STM32, dans ces structures nous allons déclarer l'ID de cette variable que nous avons récupéré de la base de donnée, le type de protocole CAN utilisé, la taille de la donnée à transmettre, le type d'ID.... etc. Toutes ces configurations sont représentées dans la figure ci-dessous :

```
VEHICLE_SPEED.Identifier = 0x0D;
VEHICLE_SPEED.IdType = FDCAN_STANDARD_ID;
VEHICLE_SPEED.TxFrameType = FDCAN_DATA_FRAME;
VEHICLE_SPEED.DataLength = FDCAN_DLC_BYTES_8;
VEHICLE_SPEED.ErrorStateIndicator = FDCAN_ESI_ACTIVE;
VEHICLE_SPEED.BitRateSwitch = FDCAN_BRS_OFF;
VEHICLE_SPEED.FDFormat = FDCAN_CLASSIC_CAN;
VEHICLE_SPEED.TxEventFifoControl = FDCAN_NO_TX_EVENTS;
VEHICLE_SPEED.MessageMarker = 0x0; // Ignore because FDCAN_NO_TX_EVENTS
```

FIGURE II.20 : Configuration des structures TxHeader.

Nous avons aussi créé une fonction qui se nomme « change () » qui sert à changer le contenu des 9 vecteurs à chaque transmission pour simuler au mieux un véhicule, et dans la boucle «while» infinie du programme, nous allons utiliser les fonctions spécifiques du périphérique FDCAN pour envoyer ces données et appeler la fonction « change () » avec un intervalle d'une seconde entre chaque transmission et de deux secondes entre chaque cycle comme illustre la Figure II.21 :

```
switch(j){
case 0:
        HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &VEHICLE_SPEED, vit);
        HAL_Delay(1000);
        HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &AMB_AIR_TEMP, Temp);
        HAL_Delay(1000);
        HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &BAROMETRIC_PRESSURE, Pressure);
        HAL_Delay(1000); ...
        change();
        HAL_Delay(2000);
 break;
```

FIGURE II.21 : Code pour la transmission.

Lorsqu'on appuiera sur un des deux boutons comme expliquer précédemment, nous allons changer de protocole ou de débit, pour réaliser cela nous utiliserons l'état de la variable «j» dit un «Flag», avec les interruptions des boutons, chaque appuie changera la valeur de «j» et selon sa valeur nous changerons le type de protocole ou de débit, comme illustre dans la Figure II.22, la même procédure sera appliqueé sur la carte de réception avec la seule différence le changement de valeur de la variable «Flag» se fera via USB.

```
void EXTI3_IRQHandler(void)
{/* USER CODE BEGIN EXTI3_IRQn 0 */
push++;
        switch(push){
        case 1:
              j=0;       speed=1;     test_bascule=1;
              break;
        case 2:
              j=10;      speed=4;     test_bascule=2;     push=0;
              break;
        default:
              break;}
  /* USER CODE END EXTI3_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_3);
```

FIGURE II.22 : Code de l'interruption du bouton K1.

```
case 10:
{if(test_bascule==2){
HAL_FDCAN_Stop(&hfdcan1); hfdcan1.Instance = FDCAN1;
hfdcan1.Init.FrameFormat = FDCAN_FRAME_FD_BRS; hfdcan1.Init.Mode = FDCAN_MODE_NORMAL;
hfdcan1.Init.AutoRetransmission = ENABLE;  hfdcan1.Init.TransmitPause = DISABLE;
hfdcan1.Init.ProtocolException = DISABLE;  hfdcan1.Init.NominalPrescaler = 1;
hfdcan1.Init.NominalSyncJumpWidth = 2;     hfdcan1.Init.NominalTimeSeg1 = 93;
hfdcan1.Init.NominalTimeSeg2 = 2;          hfdcan1.Init.DataPrescaler = 8;
hfdcan1.Init.DataSyncJumpWidth = 13;       hfdcan1.Init.DataTimeSeg1 = 21;
hfdcan1.Init.DataTimeSeg2 = 13;            hfdcan1.Init.MessageRAMOffset = 0;
hfdcan1.Init.StdFiltersNbr = 1;     hfdcan1.Init.ExtFiltersNbr = 0;
hfdcan1.Init.RxFifo0ElmtsNbr = 1;
hfdcan1.Init.RxFifo0ElmtSize = FDCAN_DATA_BYTES_16; hfdcan1.Init.RxFifo1ElmtsNbr = 1;
hfdcan1.Init.RxFifo1ElmtSize = FDCAN_DATA_BYTES_16;hfdcan1.Init.RxBuffersNbr = 0;
hfdcan1.Init.RxBufferSize = FDCAN_DATA_BYTES_16;hfdcan1.Init.TxEventsNbr = 0;
hfdcan1.Init.TxBuffersNbr = 0;      hfdcan1.Init.TxFifoQueueElmtsNbr = 1;
hfdcan1.Init.TxFifoQueueMode = FDCAN_TX_FIFO_OPERATION; hfdcan1.Init.TxElmtSize =
FDCAN_DATA_BYTES_16;         HAL_FDCAN_Init(&hfdcan1);
HAL_FDCAN_Start(&hfdcan1);
if (HAL_FDCAN_ActivateNotification(&hfdcan1, FDCAN_IT_RX_FIFO1_NEW_MESSAGE, 0) !=
HAL_OK){Error_Handler();}
HAL_Delay(3000);      test_bascule=0;
ss= snprintf(b,30,"fdcan 125Kbps\nEND\n"); CDC_Transmit_FS(b, ss);}
                ss= snprintf(b,30,"Start FDCAN\n"); CDC_Transmit_FS(b, ss);
                HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &Temperaturefd, Temperature2);
                HAL_Delay(1000);
                HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &ABSfd, ABS2);
                HAL_Delay(1000);
                change();
                ss= snprintf(b,30,"END\n"); CDC_Transmit_FS(b, ss);
                        break;
```

FIGURE II.23 : Code pour le changement de protocole.

Maintenant que nous avons terminé avec le code de la première carte qui sert à envoyer les données, nous allons alors nous concentrer sur le code de la 2éme carte qui sert à récupérer les données et les diagnostiquer et les ré-envoyer via USB.

Nous commençons par déclare les bibliothèques à utiliser comme indiquer sous la Figure II.24 ci-dessous :

```
/* Includes
#include "main.h"
#include "usb_device.h"
/* Private includes
/* USER CODE BEGIN Includes */
#include "string.h"
#include "usbd_cdc_if.h"
#include "stdio.h"
#include "ILI9488.h"
#include "besma.h"
#include "obd.h"
#include "ND.h"
/* USER CODE END Includes */
```

FIGURE II.24 : Déclaration des bibliothèques.

Et ensuite, nous déclarons les variables qui serviront à l'envoie des données et qui n'ont pas été déclarées lors de la génération du code par CUBEIDE (Figure II.25) :

```
/* Private variables
/* USER CODE BEGIN PV */
FDCAN_HandleTypeDef hfdcan1;
extern SPI_HandleTypeDef hspi4;
extern DMA_HandleTypeDef hdma_spi4_tx;
extern TIM_HandleTypeDef htim1;
FDCAN_FilterTypeDef sFilterConfig;
uint8_tR[16],a[8],d[64],buffer[64],s,ss,bs,b[90],buffer2[16],buffer3[16],buffer4[16],buf[64]
 ,text[40],text2[40],bufferr_receive[8];
uint16_t col,x=380,y=60;
FDCAN_RxHeaderTypeDef RxHeader;
FDCAN_TxHeaderTypeDef filter,Ms;
uint16_t _width = DISPL_WIDTH,_height = DISPL_HEIGHT;
extern float engine_data;
/* USER CODE BEGIN PV */
```

FIGURE II.25 : Déclaration des variables.

Pour la variable « hfdcan1 » comme expliquer plus haut, elle doit être similaire à celle déclarée dans le premier code vu que nous avons configuré le périphérique de la même façon donc pas besoin de changer cette structure, même chose pour la déclaration de la variable « sFilterConfig » qui restera la même car nous voulons récupérer tous les ID, sans avoir besoin de faire le tri.

Les données récupérées seront stockées à l'intérieur de la variable « R » et les autres

informations que nous avons préalablement déclaré dans la structure de type « FDCAN_TXHeaderTypeDef » seront, quant à eux, stockées dans une structure similaire dans la variable nommée « Rx_Header ».

Les variables « buffer » et « texte » sont, quant à eux, utilisées pour envoyer respectivement les données sous format string via USB vers un port connecte qui sera notre interface et vers l'afficheur LCD.

Dans la figure ci-dessus nous pouvons observer toutes les fonctions que nous avons créé pour faire certaines animations sur l'afficheur LCD et pour dessiner le chronogramme.

```
void haut(uint16_t col);
void bas(uint16_t col);
void lath(uint16_t col);
void lat(uint16_t col);
uint16_t rand_color();
void dinam(uint16_t col);
void horizanimD();
void horizanim();
void chronogramme();
```

FIGURE II.26 : Déclaration des Fonctions.

La fonction la plus importante de toutes est « obd () », qui sert à diagnostiquer les données récupérées et à les renvoyer sous une grandeur plus compréhensible pour nous, êtres humains, tel que : « Température, vitesse, Pression du carburant... etc ».

Comme expliqué plus haut nous avons décidé d'utiliser la méthode des interruptions pour la transmission, ce qui signifie que lorsqu'une trame est reçue il y'a une fonction d'interruption bien spéciale appelée « HAL_FDACN_RXFifo0Callback » Figure II.27. « Fifo0 » est dû au fait que lorsque nous avons configuré le périphérique FDCAN, nous avons configuré la mémoire « Fifo0 » comme zone mémoire pour la réception des messages.

Nous pouvons voir que dès que cette fonction est appelée, nous commençons par vérifier s'il y a eu un nouveau message enregistré sur la mémoire « Fifo0 », si tel est le cas, alors nous récupérons ce message dans la structure design « RxHeader » et nous stockons la donnée de 8 octets dans la variable « R » à l'aide de la fonction « HAL_FDCAN_GetRxMessage() ».

Une fois le message récupéré, nous l'envoyons, ainsi que l'ID correspondent via USB vers l'interface et nous commençons la partie dite diagnostic de notre analyseur avec, comme expliqué plus haut, l'utilisation de la fonction « obd () ».

```
void HAL_FDCAN_RxFifo0Callback(FDCAN_HandleTypeDef *hfdcan, uint32_t RxFifo0ITs)
{  if((RxFifo0ITs & FDCAN_IT_RX_FIFO0_NEW_MESSAGE) != RESET)
  { HAL_FDCAN_GetRxMessage(hfdcan, FDCAN_RX_FIFO0, &RxHeader, R);
   if(RxHeader.FDFormat == FDCAN_CLASSIC_CAN )
   {ss=snprintf(b,30,"%02x %02x %02x %02x %02x %02x %02x %02x %02x"
                            ,RxHeader.Identifier,R[0],R[1],R[2],R[3],R[4],R[5],R[6],R[7]);
      CDC_Transmit_FS(b, ss);
      obd();
    }     if(j==0){
              switch (RxHeader.Identifier){
              case 0x0d:
                        m++;   col=ILI9488_OLIVE;  x=380; y=60;
                        sprintf(text,"Data :");
                        Displ_WString(20, 30, text, Font16, 1, ILI9488_WHITE, ILI9488_BLACK);
             sprintf(text,"%02x %02x %02x %02x %02x %02x %02x %02x
             %02x",R[0],R[1],R[2],R[3],R[4],R[5],R[6],R[7]);
                        Displ_WString(90, 30, text, Font16, 1, ILI9488_BLACK, ILI9488_OLIVE);
                        e_data=engine_data;
            if(m>1){
                    x=380;     y=60;  c=1;   col=ILI9488_OLIVE;  e_data=engine_data;
                    }
```

FIGURE II.27 : Code de la fonction « HAL_FDACN_RXFifo0Callback ».

Dans cette partie nous commençons d'abord par déclarer toutes les variables du véhicule avec leurs ID propres que nous avons récupéré de la base de données comme illustre la Figure II.28 :

```
#define VEHICLE_SPEED       0x0D
#define VEHICLE_SPEED_DESC          "Vehicle Speed"
#define TIMING_ADVANCE      0x0E    //
#define TIMING_ADVANCE_DESC         "Timing Advance"
#define INTAKE_TEMP         0x0F    //
#define INTAKE_TEMP_DESC            "Intake Air Temperature"
#define MAF_SENSOR          0x10
#define MAF_SENSOR_DESC             "MAF Sensor Air Flow Rate"
#define THROTTLE            0x11
#define THROTTLE_DESC               "Throttle Position"
#define COMMANDED_SEC_AIR   0x12    ///
#define COMMANDED_SEC_AIR_DESC      "Commanded Secondary Air Status"
```

FIGURE II.28 : Déclaration des IDs.

Une fois cette étape terminée, nous commençons le traitement de la donnée, chaque donnée et traitée de façon unique selon sa nature propre, les données qui représentent : la température, la vitesse ou la distance parcourus ne seront évidemment pas traitées de la même manière.

Nous créons alors une grande boucle Switch qui, selon l'ID de la donnée reçue, fera un traitement différent comme nous pouvons le constater dans les deux figures suivantes, Figure II.29 :

```
void obd (){
/* Details from http://en.wikipedia.org/wiki/OBD-II_PIDs */
     switch (RxHeader.Identifier) {
              case PID_0_20: // PID 0-20 Supported
              PID020 = (R[3] << 24) | (R[4] << 16) | (R[5] << 8) | (R[6]);
                       break;
             case VEHICLE_SPEED:// A[km]
           engine_data = R[3];
            s=snprintf(buffer,30,"The speed car: %dkm\n",(int) engine_data);
            CDC_Transmit_FS(buffer,s);
                       break;
```

FIGURE II.29 : Code de traitement de données.

On peut constater que selon l'ID du message, les données sont traitées différemment.

Une fois la partie du diagnostic terminée, nous passons à l'affichage sur l'écran du LCD, nous commençons avec une petite animation du nom de notre projet « CANalyzer » et par afficher les différents ID que nous allons recevoir, une fois chaque trame reçue, le chronogramme qui correspond aux données reçus est affiché sur l'écran à l'aide des fonctions que nous avons créé au préalable (Figure II.30).

```
ILI9488_Init();
setRotation(Displ_Orientat_270);
fillScreen(ILI9488_BLACK);
for(w=0;w<100;w++){
                  x = rand() % (_width - 11);y = rand() % (_height - 18);
                  uint16_t taille=rand()%60; DrawCircle(x, y,taille,rand_color());
       }
       HAL_Delay(500);
       dinam(ILI9488_BLACK);
       HAL_Delay(500);
       Displ_CString(120,120,180,180,"CANalyzer",Font24,3,ILI9488_WHITE,ILI9488_BLACK);
       HAL_Delay(1000);
       horizanim();
       Displ_CString(0,60,20,10,"0x0D",Font16,1,ILI9488_WHITE,ILI9488_OLIVE);
       y=60;x=380;
       for(int p=0;p<64;p++) {bas(ILI9488_OLIVE);}
       Displ_CString(0,100,20,10,"0x46",Font16,1,ILI9488_WHITE,ILI9488_BLUE);
       y=100;x=380;
       for(int p=0;p<64;p++)
       {bas(ILI9488_BLUE);}
```

FIGURE II.30 : Initialisation du LCD

Nous vérifions chaque bit reçu, s'il est égal à « 1 » ou un « 0 » pour pouvoir dessiner le chronogramme correspondant comme illustré par la Figure II.31.

```
void haut(uint16_t col){
        drawLine(x, y, x-5,y, col);//line hori
        drawLine(x-5, y, x-5,y+2, col);//line hori
        x=x-5;}
void bas(uint16_t col){
       drawLine(x, y+10, x-5,y+10, col);
       drawLine(x-5, y+10, x-5,y+8, col);
       x=x-5;}
```

(a)

```
void lath(uint16_t col)
{drawLine(x+5, y, x+5,y+10, col);//line vert + de
px vers le bas
}
void lat(uint16_t col)
{ drawLine(x+5, y, x+5,y+10, col);
}
```

(b)

```
void chronogramme()
{for(int z=0;z<8;z++){
                a[0] = (R[z] & (0x01));  a[1] = (R[z] & (0x02));
                a[2] = (R[z] & (0x04));  a[3] = (R[z] & (0x08));
                a[4] = (R[z] & (0x10));  a[5] = (R[z] & (0x20));
                a[6] = (R[z] & (0x40));  a[7] = (R[z] & (0x80));
                for(int i=0;i<8;i++){  if(a[i]==0x00)  { bas(col);
                                       if(l==2){lath(col);}
                                       if((i==3) | (i==7))
                                       {drawLine(x, y+10, x,y+8, ILI9488_RED);}
                                       l=1;}
                                       else{haut(col);x=x+5;
                                            bas(ILI9488_BLACK);
                                            if(l==1){lat(col);}
                                            l=2;
                                            if((i==3) | (i==7)){drawLine(x, y, x,y+2, ILI9488_RED);}
                  }}}}
```

(c)

FIGURE II.31 : Les fonctions utilisées pour afficher le chronogramme.

On peut résumer toutes les étapes de ce fonctionnement dans l'organigramme suivant :

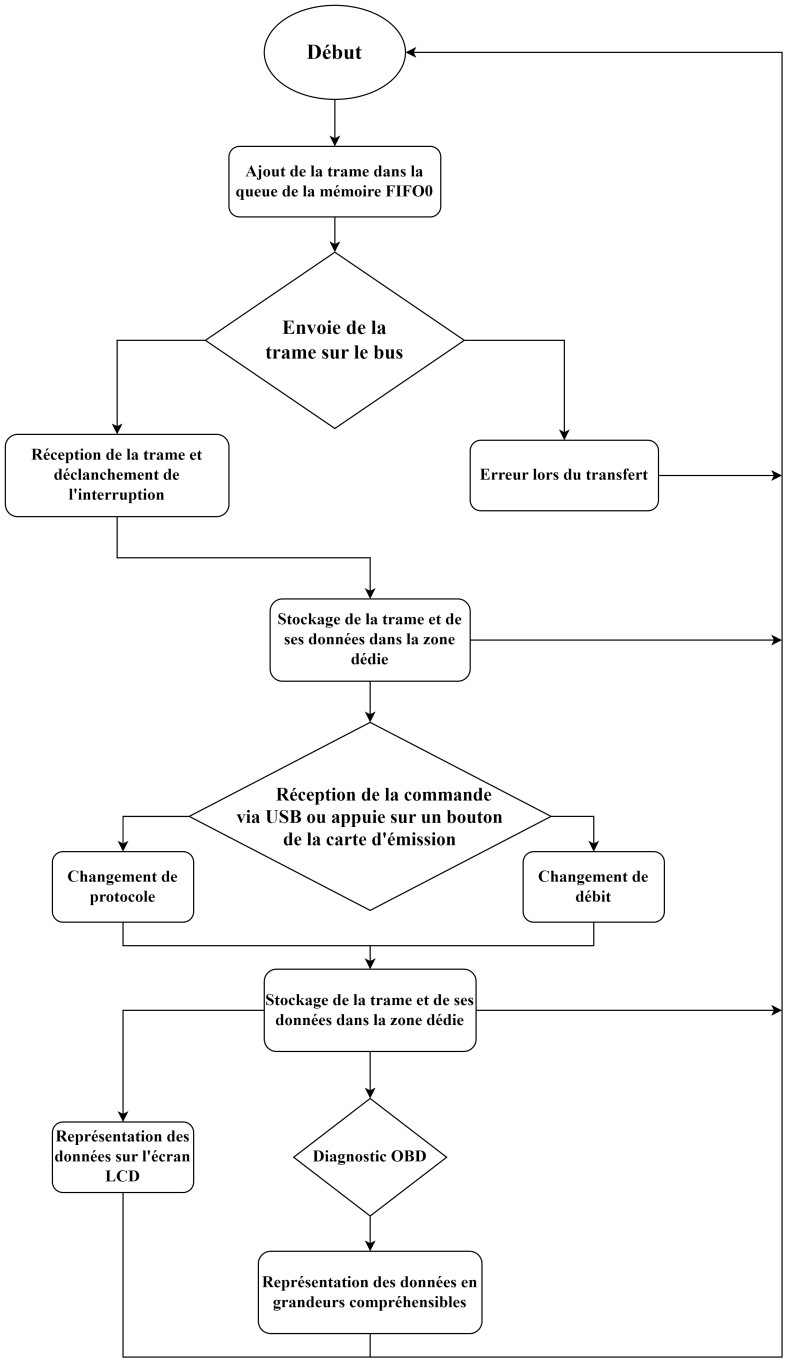


FIGURE II.32 : organigramme de fonctionnement de l'analyseur.

Conclusion

Nous avons résumé dans ce chapitre, la conception globale de notre système, ainsi que les différentes étapes de notre travail sur ce projet, néanmoins, pour avoir pu parvenir à ce point-là, nous sommes passés par une multitude de tests et d'essaies différents pour répondre aux exigences de notre cahier de charge, tout en essayant toujours d'optimiser et de rajouter de nouvelles fonctionnalités, ces tests seront détaillés avec plus de précision dans le prochain chapitre.

Chapitre III

Mise en Œuvre, Résultats Expérimentaux et Interprétations

Introduction

Dans ce chapitre, nous allons passer à l'étape d'implémentation des différents programmes sur notre système et de sa mise en œuvre, ainsi qu'à l'analyse et la présentation des résultats obtenus.

Dans ce chapitre, nous abordons les différents aspects de réalisation et aboutissement de notre projet, tels que les performances techniques, les résultats des tests et les éventuelles limitations rencontrées. Nous présentons également les perspectives d'amélioration pour l'avenir.

III.1 Description du matériel et de l'environnement logiciel du travail

Au cours de la réalisation des différents tests, nous avons eu recours à l'utilisation de divers composants que nous avons déjà décrit dans le chapitre précédent, tous ce travail fut réalisé sur un PC ayant les caractéristiques suivantes :

- Processeur : Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz.
- Mémoire RAM installée : 8,00 Go.
- Type du système : Système d'exploitation 64 bits, processeur x64.

Pour les premières phases de test, nous avons utilisé une carte STM différentes, la STM32-F466RE illustrée par la Figure III.1. Celle ci possède seulement un périphérique supportant le protocole CAN.

La raison de l'utilisation de cette carte durant les premières phases du projet revient seulement à un gain du temps, car la carte STM32H743VIt6 n'était pas encore en notre possession. Et vu que nous devions aussi faire des transmissions avec le protocole CAN et non seulement en CAN FD. Nous avons donc utilisé cette carte.

Aussi, une autre raison d'utilisation de cette carte est la disponibilité abondante de documentation pour la gamme F4 des cartes STM. Grâce aussi aux performances techniques qu'elle possède et le nombre si important de périphériques qui lui sont reliés (Figure III.2).

Nous allons programmer cette cartes STM à l'aide des logiciels STM32CUBEIDE et STM32CUBEMX en langage C pour réaliser les différentes tâches nécessaires de l'analyseur, allant de la réception, du diagnostic des données et leurs interprétations et de leurs transferts via USB jusqu'à l'affichage sur l'écran LCD.

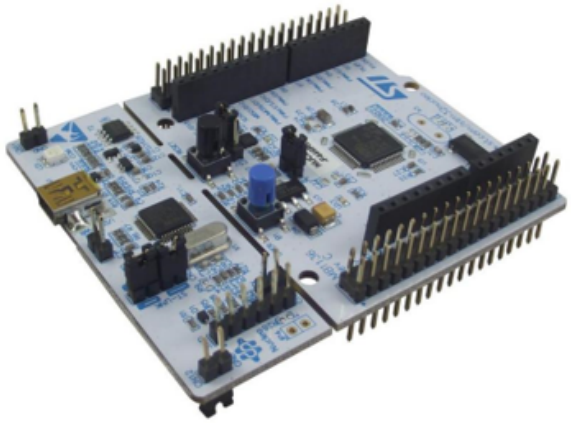


FIGURE III.1 : Carte STM32F446RE.

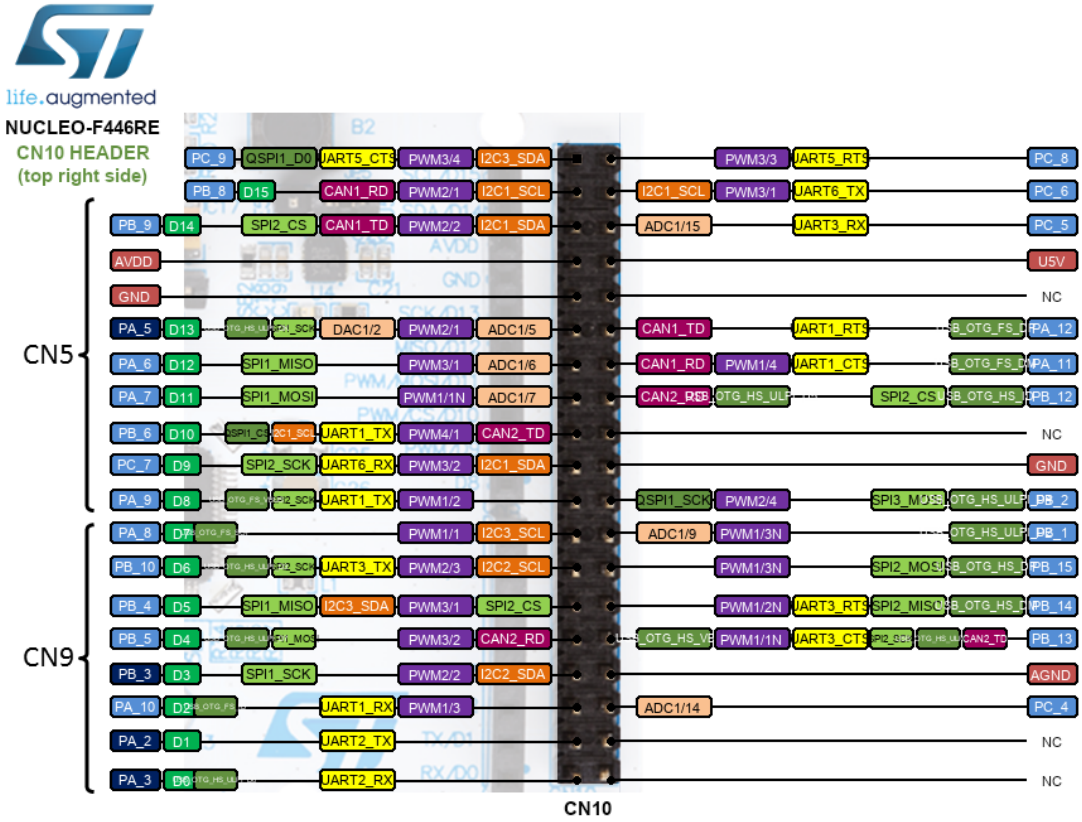


FIGURE III.2 : Périphériques de la STM32F446RE.

III.2 Photographie du système.

Une fois le matériel requis pour la réalisation de chaque bloc du système obtenue on passe à leurs associations pour effectuer les différents tests et réaliser notre passrelle can,canfd/usb.

La Figure III.3 ci-dessous représente une photo reel du bloc de transmission et de

réception, chaque bloc est représente par une carte stm32h743viT6 qui servira pour simuler un véhicule ou une machine qui utilise le protocole can ou can fd et ainsi envoyer des donnees sur le bus et de les recevoir dans le bloc de réception l'aide de l'autre carte.

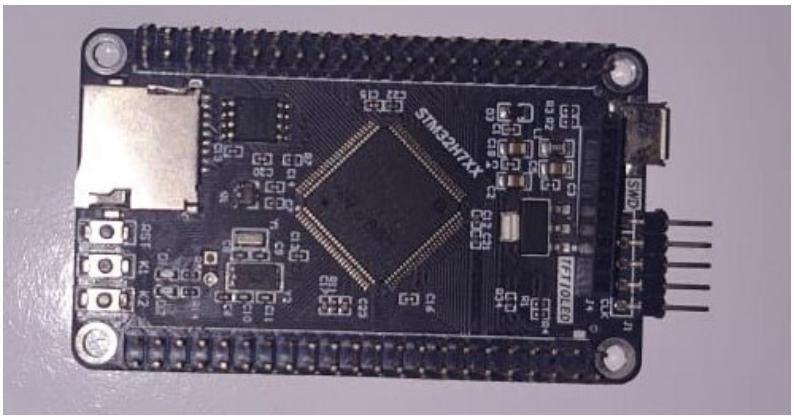


FIGURE III.3 : Bloc de transmission et de réception.

La Figure III.4 quant à elle est une photographie du bloc de transmission caractérise par la présence de deux contrôleurs can mcp2551.

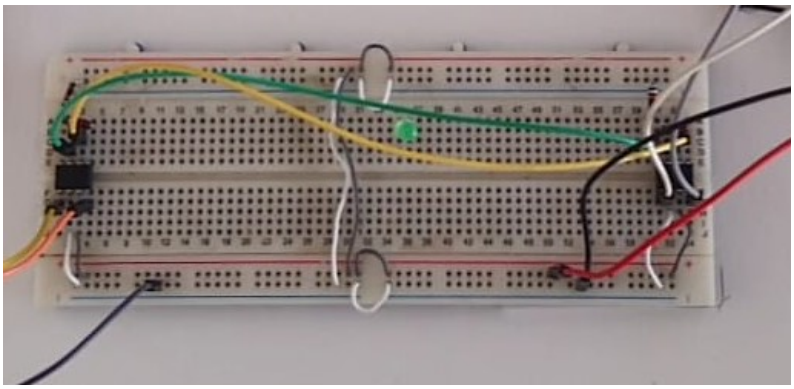


FIGURE III.4 : Bloc de transmission.

La Figure III.5 montre le bloc d'affichage qui est représenté par un écran LCD qui sert à affiche les données reçu par la carte de réception.

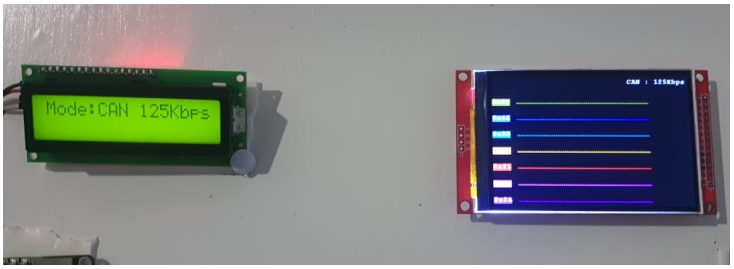


FIGURE III.5 : Bloc d'affichage.

Le deuxième LCD qui est représenté par la Figure III.6, quant à lui sert uniquement pour signaler avec quel type de protocole les données sont envoyés et avec quels débits.



FIGURE III.6 : LCD pour l'affichage de débit et de protocole de transmission.

La connexion entre ces différents blocs permet de réaliser notre système final, une passerelle can,canfd/usb représente dans la Figure III.7

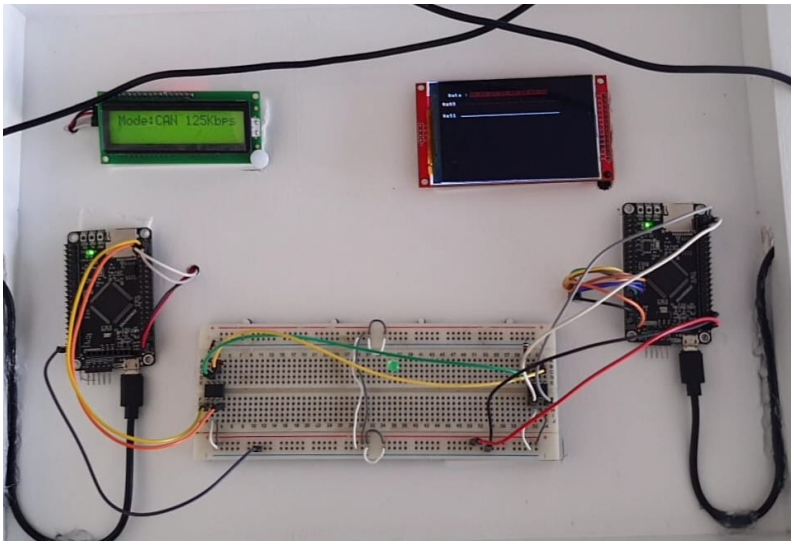


FIGURE III.7 : Sysème final.

III.3 Déscription des étapes de tests

Pour pouvoir réaliser notre version de l'analyseur CAN, nous sommes passés par plusieurs étapes de tests pour diverses raisons techniques, ainsi que l'obligation de se familiariser au mieux avec l'environnement STM et celui du bus CAN et CAN FD.

III.3.1 Test de transmission avec protocole CAN

Pour ce test nous avions besoins de deux cartes STM32F446RE, deux contrôleurs CAN et de deux résistances de 120Ω comme nous pouvons le voir dans la Figure III.8.

Pour transmettre les données, nous utilisons les boutons comme déclencheurs et la

LED de la carte comme témoin de la réception uniquement si les données transmissent ne sont pas erronées, tel que lorsqu'on appui sur le bouton de la carte N°1 elle envoie le message vers la deuxième carte et on vérifie si les données concordent ; si c'est le cas, alors cela allumera la LED sur la carte et nous pouvons faire l'opération en sens inverse, cette transmissions est réalisée grâce aux interruptions comme expliqué dans le chapitre précédent.

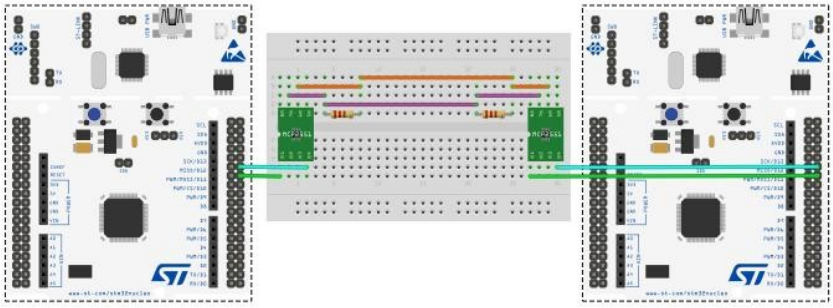


FIGURE III.8 : Circuit du premier Test.

Étape 1 : Configurations des pins du processeur de la carte STM32F446RE,tel que ces configurations sont les mêmes sur les deux cartes, comme illustre dans la Figure III.9).

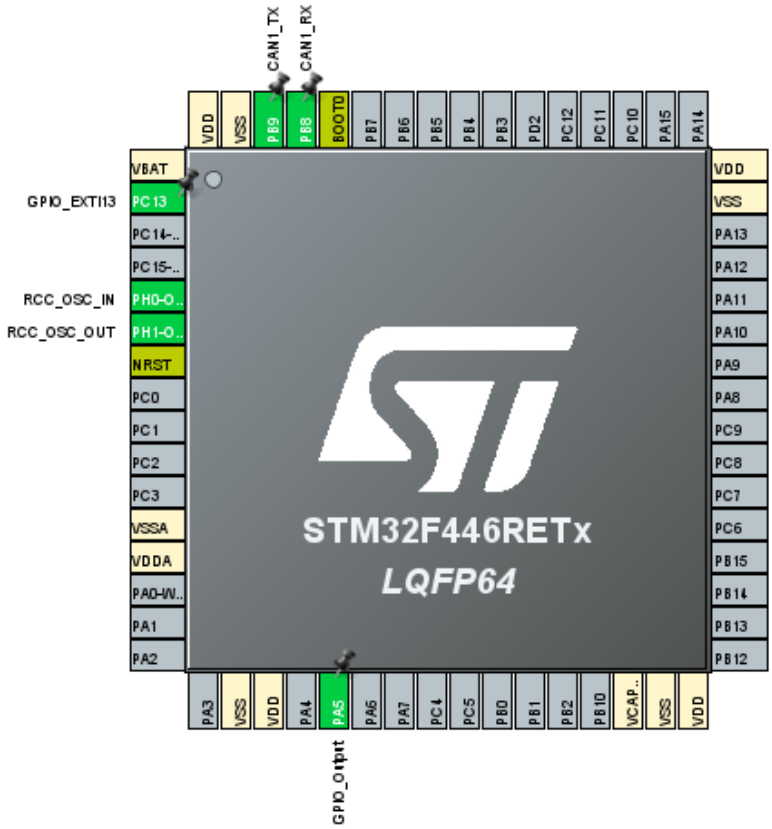


FIGURE III.9 : Configuration de la STM32F446RE.

Étape 2 : Configuration de l'horloge de la carte au maximum de ces capacités

180MHz comme illustre la Figure III.10

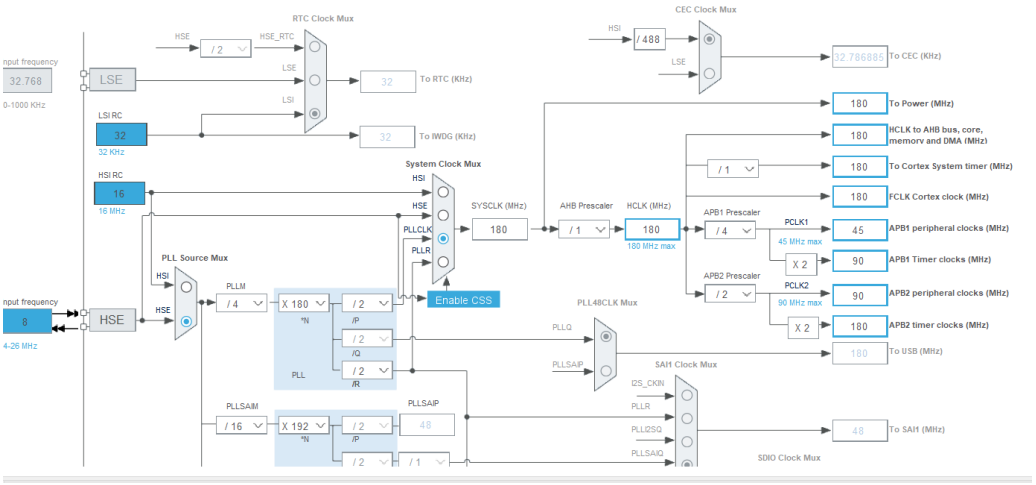


FIGURE III.10 : Configuration de l'horloge.

Étape 3 : Déclaration des variables qui contiendrons lesdonnées sous format de vecteurs de huit octets, et nous déclarons aussi les variables qui serviront à préciser les identifiants et leurs caractéristiques : taille, méthode de transmissions,..etc, ce genre de variable est sous la forme d'une structure de type «CAN_TxHeaderTypeDef» pour la transmission et «CAN_RxHeaderTypeDef» pour la réception, comme on peut le voir dans la Figure III.11.

```
/* USER CODE END PM */
/* Private variables
CAN_HandleTypeDef hcan1;
/* USER CODE BEGIN PV */
CAN_FilterTypeDef Scanfilter;
CAN_RxHeaderTypeDef RxHeader;
CAN_TxHeaderTypeDef TxHeader;
uint32_t TxMailbox;
uint8 t TxData[8];
```

```
/* USER CODE BEGIN 1 */
TxData[0]=0x01;
TxData[1]=0xaa;
TxData[2]=0xbb;
TxData[3]=0xcd;
TxData[4]=0xff;
TxData[5]=0xdd;
TxData[6]=0x14;
```

```
TxHeader.IDE=CAN_ID_STD;
TxHeader.StdId=0x200;
TxHeader.RTR=CAN_RTR_DATA;
TxHeader.DLC=8;
```

FIGURE III.11 : Déclaration des variables.

Étape 4 : Quand le bouton d'une des cartes appuyé, une interruption sera alors déclenchée ce qui fera appel à la fonction dans la Figure III.12, où les données du vecteur « TxData » et de la structure « TxHeader » seront transmises sur le bus en direction de l'autre carte.

```
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */
  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */
      if(HAL_GPIO_ReadPin(GPIOC,GPIO_PIN_13)== 0)
      {
            HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox);
```

FIGURE III.12 : Fonction d'interruption pour la transmission.

Étape 5 : Une fois la trame reçu une fonction reliée à l'interruption est déclenchée, les données liées à la structure TxHeader seront stockées dans la structure RxHeader, tandis que les valeurs des données seront quant à elles, stockées dans le vecteur RxData comme on peut le voir dans la Figure III.13. Une fois ces données stockées, la phase de vérification débute, si les données reçues sont bien ceux que nous avons envoyé alors la LED de la carte de réception s'allumera sinon rien ne se passe comme illustre la Figure III.14.

```
/* USER CODE BEGIN 4 */
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
      HAL_CAN_GetRxMessage(&hcan1, CAN_RX_FIFO0, &RxHeader, RxData);
      __NOP();
```

FIGURE III.13 : Fonction d'interruption pour la réception.

```
if(RxData[0]==0x01)
{
      HAL_GPIO_WritePin(GPIOA,GPIO_PIN_5,GPIO_PIN_SET);
```

FIGURE III.14 : Vérification des données reçues.

Une fois cette phase de tests et de familiarisation avec l'environnement du bus et du périphérique CAN terminée et réussie, nous avons essayé de reproduire la même chose, mais cette fois-ci avec le protocole CAN FD

Toutes les étapes de ce code peuvent être résume avec l'organigramme suivant :

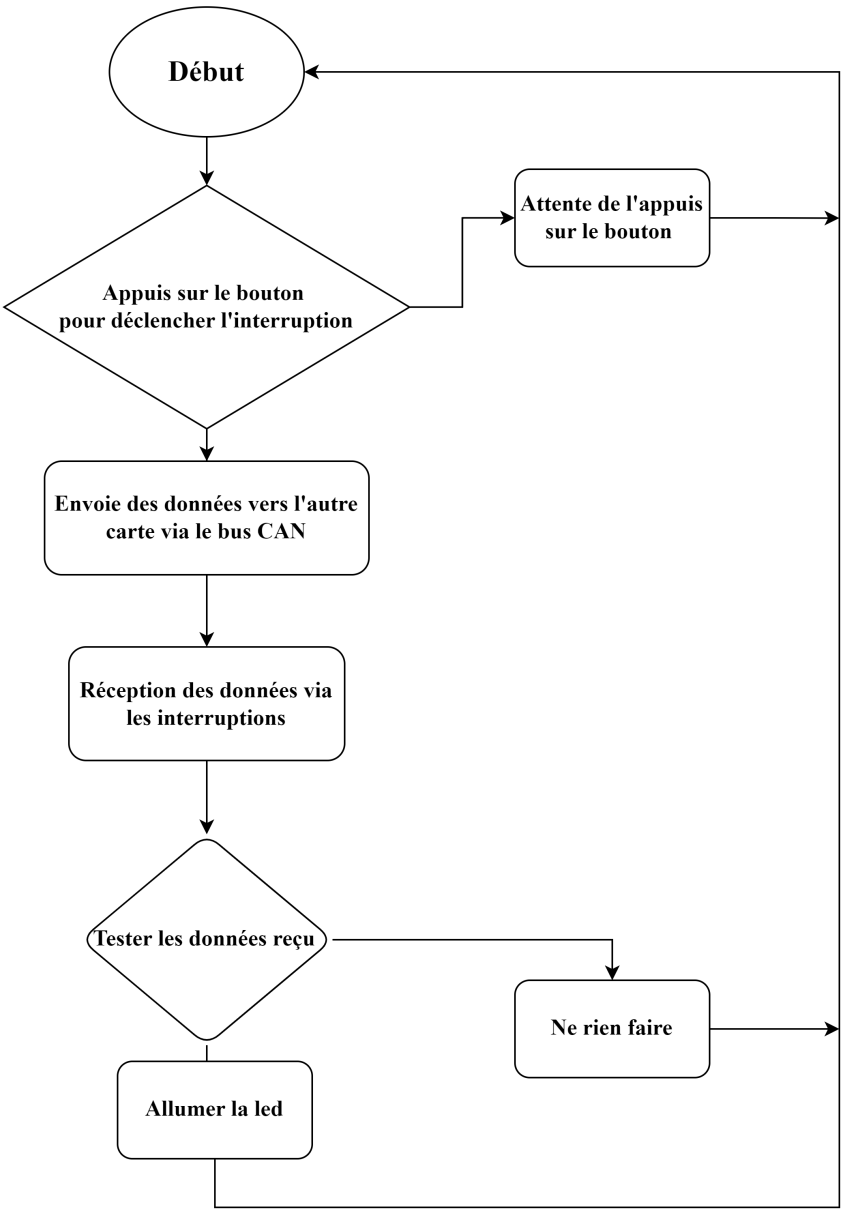


FIGURE III.15 : Organigramme de la transmission CAN.

III.3.2 Test de transmissions avec protocole CAN-FD

III.3.2.1 Transmission avec les Buffers

Pour ce test et le reste nous utiliserons la carte STM32H743vit6 qui dispose bien d'un périphérique FDCAN, pour bien démontrer qu'il s'agit bien d'une transmission en utilisant le bus CANFD, nous enverrons un vecteur de plus de seize octets de la première vers la deuxième carte, en utilisant cette fois-ci non pas les interruptions comme le premier test, mais une transmission à l'aide d'un des 32 différents buffers dédiés des périphériques mises à disposition dans la carte STM.

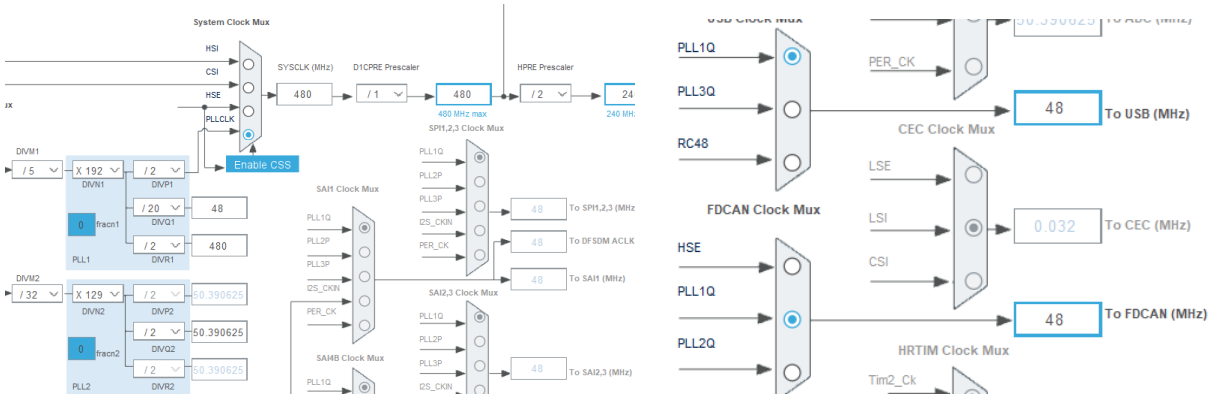
Cette méthode d'utilisation des buffers est fiable, mais elle reste néanmoins moins

performante par rapport aux interruptions pour plusieurs raisons :

- Un temps de réponse nettement plus court, lors de l'utilisation des interruptions pour la réception de données lorsqu'une se déclenche, cela indique que des données ont été reçues et donc cela permet de les traiter immédiatement sans attendre la vérification de l'état des buffers, ce qui réduit le temps de latence entre la réception et le traitement,
- Une plus grande économie des ressources, en effet l'utilisation des interruptions permet une gestion efficace des ressources du système, au lieu de devoir vérifier en permanence les buffers pour voir si de nouvelles données sont disponibles, les interruptions se déclenchent uniquement lorsque des données sont reçues. Cela permet d'économiser les ressources du processeur qui seraient autrement utilisées pour effectuer cette vérification en continu,
- Priorité et gestion des événements : Les interruptions permettent de hiérarchiser les événements et de gérer leur traitement en fonction de leur priorité,
- Fiabilité de la réception : Les interruptions facilitent une réception plus fiable des données,

Malgré cela, nous avons décidé de les utiliser pour ce test afin de s'assurer, d'une part, que c'est bien possible d'envoyer et de recevoir des trames sur le bus à l'aide des buffers, mais aussi pour pouvoir comparer les résultats obtenus avec les interruptions et ainsi, confirmer que ce sont ces raisons qui, théoriquement, mettent la méthode des interruptions comme la meilleure.

Étape 1 : Configuration de l'horloge de la carte au maximum de ces capacités 480MHz et de l'horloge des périphériques FD CAN et USB. Pour obtenir un débit de transmission de 500Kbps avec l'utilisation de 4 buffers différents. Comme illustre dans la Figure III.16 etla Figure III.17.



(a) (b)

FIGURE III.16 : Configuration de l'horloge.

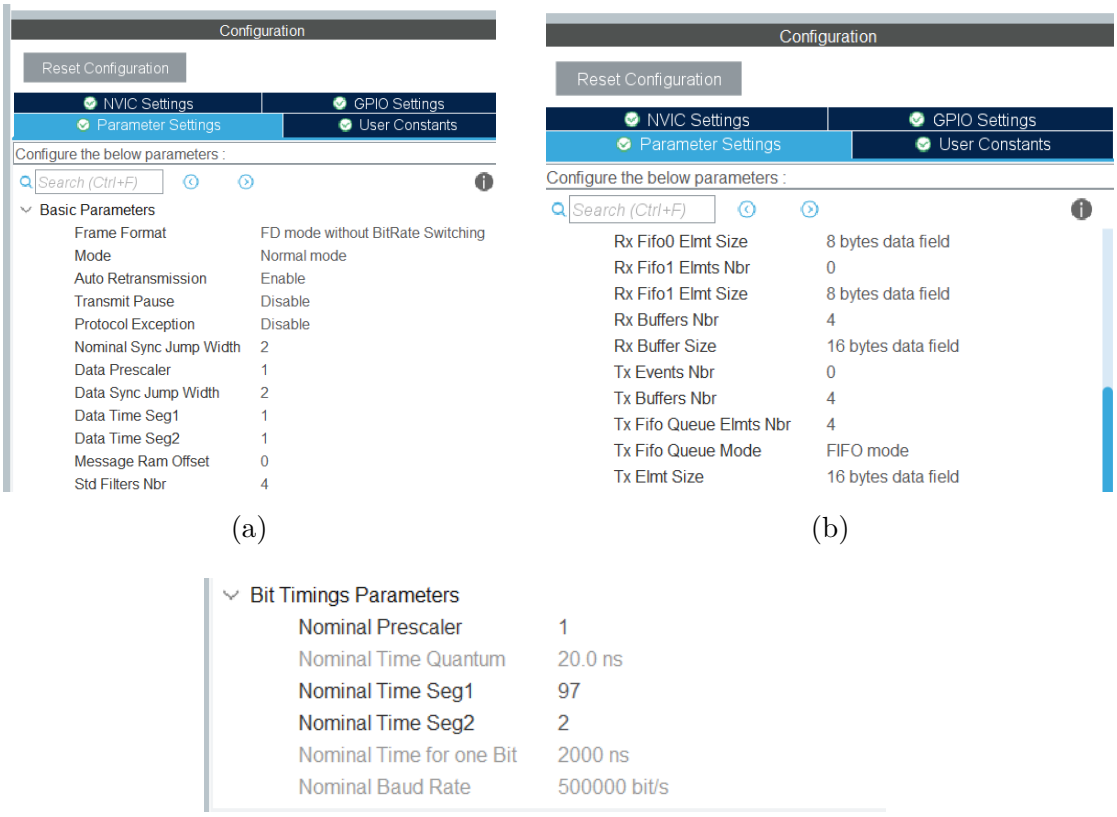


FIGURE III.17 : Configuration du périphérique FD CAN.

Étape 2 : Déclaration des variables de transmission de type « FDCAN_TxHeaderTypeDef » et «FDCAN_FilterTypeDef », ainsi que les vecteurs qui contiendrons les données déclarées que nous avons remplis d'une manière arbitraire juste pour le test comme démontre dans la Figure III.18.

```
/* Private variables -

FDCAN_HandleTypeDef hfdcan1;

/* USER CODE BEGIN PV */

FDCAN_FilterTypeDef sFilterConfig;

FDCAN_TxHeaderTypeDef Temperature,Pression,lumiere,vitesse;

uint8_t Temperature1[] = {0x10, 0x32, 0x54, 0x76, 0x98, 0x00, 0x11, 0x22, 0x33, 0x44, 0x55,
0x66,0x00,0xdd,0xdc,0x1c};

uint8_t Pression1[] =
{0x40,0x50,0x60,0x70,0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0,0xf1,0xdd,0x40,0x10};

uint8_t lumiere1[] =
{0xa1,0xa2,0xa3,0xa4,0xa5,0xa6,0xa7,0xa8,0xa9,0xaa,0xab,0xac,0xad,0xae,0xaf,0xa1};

uint8_t vitesse1[] =
{0x1e,0x2e,0x3e,0x4e,0x5e,0x6e,0x7e,0x8e,0x9e,0xae,0xbe,0xce,0xde,0xe0,0xe0,0xe1};

int i=0;

int j=0;

uint8_t data3[16];

uint8_t buffer[64],s;
```

FIGURE III.18 : Les variables de transmission.

Étape 3 : Déclaration des variables du filtre de la carte d'une manière de pouvoir

recevoir à partir de n'importe quel ID, donc pas de filtrage et de recevoir les trames uniquement par la méthode des buffers. Comme illustre dans Figure III.19.

```
sFilterConfig.IdType = FDCAN_STANDARD_ID;
sFilterConfig.FilterIndex = 0;
sFilterConfig.FilterConfig = FDCAN_FILTER_TO_RXBUFFER;
sFilterConfig.FilterID1 = 0x8; // ID Reception
sFilterConfig.RxBufferIndex = 0;
if(HAL_FDCAN_ConfigFilter(&hfdcan1, &sFilterConfig) != HAL_OK)
{
      Error_Handler();
}
```

FIGURE III.19 : Configuration du filtre.

Étape 4 : Déclaration de quatre variables de type structure de la forme FDCAN_ TxHeaderTypeDef : « Température, Vitesse, Pression, Lumière», chaque variable sera associée à un et à un seul buffer unique, d'un ID unique, d'un type de trame, un vecteur de 16 octets contenant les données à transmettre, le type d'identifiant «Standard» ou «Extend» et le protocole à utiliser CAN ou CAN FD. toutes ces configurations peuvent être illustrées par la Figure III.20.

```
Temperature.Identifier = 0x1;
Temperature.IdType = FDCAN_STANDARD_ID;
Temperature.TxFrameType = FDCAN_DATA_FRAME;
Temperature.DataLength = FDCAN_DLC_BYTES_16;
Temperature.ErrorStateIndicator = FDCAN_ESI_ACTIVE;
Temperature.BitRateSwitch = FDCAN_BRS_OFF;
Temperature.FDFormat = FDCAN_FD_CAN;
Temperature.TxEventFifoControl = FDCAN_NO_TX_EVENTS;
Temperature.MessageMarker = 0x0; // Ignore because FDCAN_NO_TX_EVENTS
```

FIGURE III.20 : FDCAN_TxHeaderTypeDef.

Étape 5 : A l'interrieur de la boucle infini While, l'association de chaque variable de Type FDCAN_TxHeaderTypeDef avec un buffer se fait à l'aide de la fonction HAL_ FDCAN_AddMessageToTxBuffer. Une fois cela terminé deux autres fonctions HAL_ FDCAN_EnableTxBufferRequest et HAL_FDCAN_IsTxBufferMessagePending sont utilisées pour s'assure que le buffer en question soit libre. une fois que nous recevons la confirmation que le buffer est disponible, le message peut alors être envoyé comme indiqué dans la Figure III.21.

```
while (1)
{/* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
if(j==1)
 {
  HAL_FDCAN_AddMessageToTxBuffer(&hfdcan1, &Temperature, Temperature1, FDCAN_TX_BUFFER0);
  HAL_FDCAN_EnableTxBufferRequest(&hfdcan1, FDCAN_TX_BUFFER0);
  HAL_FDCAN_IsTxBufferMessagePending(&hfdcan1, FDCAN_TX_BUFFER0);
  HAL_Delay(100);
      for (i=0;i<16;i++){Temperature1[i]++;}
   }
 if(j==2)
   {
   HAL_FDCAN_AddMessageToTxBuffer(&hfdcan1, &Pression, Pression1, FDCAN_TX_BUFFER1);
   HAL_FDCAN_EnableTxBufferRequest(&hfdcan1, FDCAN_TX_BUFFER1);
   HAL_FDCAN_IsTxBufferMessagePending(&hfdcan1, FDCAN_TX_BUFFER1);
   HAL_Delay(100);
       for (i=0;i<16;i++){Pression1[i]++;}
   }
```

FIGURE III.21 : Code de transmission.

Étape 6 : Avec ceci la programmation de la carte d'émission est donc terminée et nous passons donc vers la programmation de la carte de réception, pour cette partie nous effectuons les mêmes étapes de configurations d'horloge, périphériques et de variables pour s'assure de la transmission comme illustre dans la Figure III.22.

```
/* Private variables
FDCAN_HandleTypeDef hfdcan1;
/* USER CODE BEGIN PV */
FDCAN_FilterTypeDef sFilterConfig;
FDCAN_RxHeaderTypeDef Temperature,Pression,lumiere,vitesse;
uint8_t T1[16],P1[16],l1[16],v1[16];
int i=0;
uint8_t affichT1[16],affichP1[16],affichl1[16],affichv1[16];
```

FIGURE III.22 : Code de transmission.

Étape 7 : La figure III.23 représente l'étape de réception ,où nous récupérons le contenu de chaque buffer et nous stockons la partie des identifiants dans des variables dédiées et les données dans les vecteurs spécialement déclarés pour cela.

```
while (1)
{  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
       s= snprintf(buffer,50,"-start-\r\n");
       CDC_Transmit_FS(buffer, s);
       HAL_FDCAN_GetRxMessage(&hfdcan1, FDCAN_RX_BUFFER0, &Temperature, T1);
       s= snprintf(buffer,50,"-Temperature-\r\n");
       CDC_Transmit_FS(buffer, s);
       for(i=0;i<16;i++)
            {if(affichT1[i]!=T1[i])
                {affichT1[i]=T1[i];
                 s= snprintf(buffer,50,"-> %02x\n",affichT1[i]);
                 CDC_Transmit_FS(buffer, s);
                  if(i==15)
                        { s= snprintf(buffer,50,"-----end Temperature-----\r\n");
                           CDC_Transmit_FS(buffer, s);
                        }}}
```

FIGURE III.23 : Code de réceptions.

La Figure III.24 représente les résultats obtenus lors de la réception des données sur la deuxième carte.

-start-	-Pression-	-Lumiere-
-Temperature-	-> 40	-> a1
-> 10	-> 50	-> a2
-> 32	-> 60	-> a3
-> 54	-> 70	-> a4
-> 76	-> 80	-> a5
-> 98	-> 90	-> a6
-> 11	-> a0	-> a7
-> 22	-> b0	-> a8
-> 33	-> c0	-> a9
-> 44	-> d0	-> aa
-> 55	-> e0	-> ab
-> 66	-> f0	-> ac
-> dd	-> f1	-> ad
-> dc	-> dd	-> ae
-> 1c	-> 40	-> af
-----end Temperature-----	-> 10	-> a1
	-----end Pression-----	-----end lumiere -----

FIGURE III.24 : Résultats de réception des différentes données de la carte STM.

Comme on peut le constater, il y a certaines petites erreurs : comme le fait que parfois la trame n'est pas affichée complètement ou bien qu'on ne reçoive pas de nouvelles trames et on se retrouve alors à avoir la même trame que précédemment, cela revient en grande partie à la latence de l'USB pour le transfert vers le terminal pour affichage, mais aussi du fait de devoir attendre que le buffer soit prêt à recevoir les nouvelles trames.

Ainsi, parfois dans la boucle infinie on doit envoyer, par exemple, la trame correspondante à la température, mais le buffer associé n'est pas encore prêt, alors la boucle passe cette étape et va essayer d'envoyer une autre trame, mais dans la partie de la réception, ce n'est pas le cas puisqu'on va envoyer via USB, un vecteur correspondent au buffer de la température qui est censé avoir les nouvelles données, mais vu que ce buffer n'était pas

prêt, résultats on se retrouve avec les mêmes données que la trame précédente, donc une répétition qui n'est clairement pas voulu.

La figure III.25 résume ce test sous format d'organigramme :

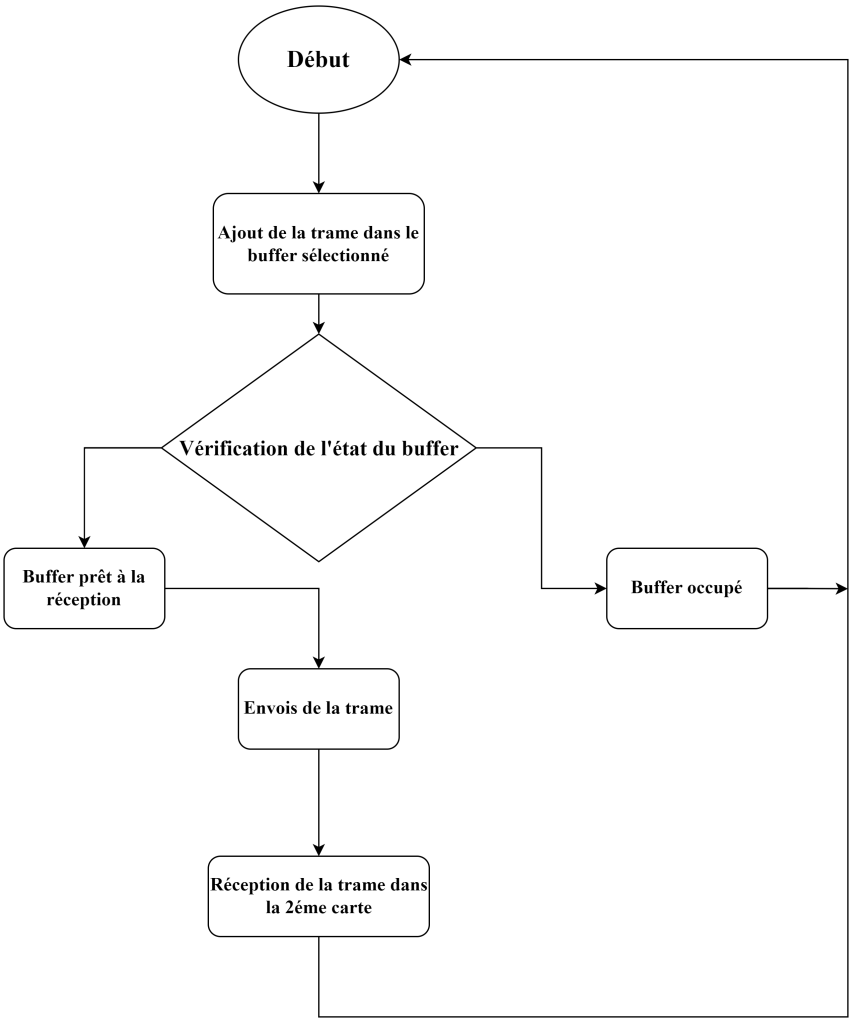
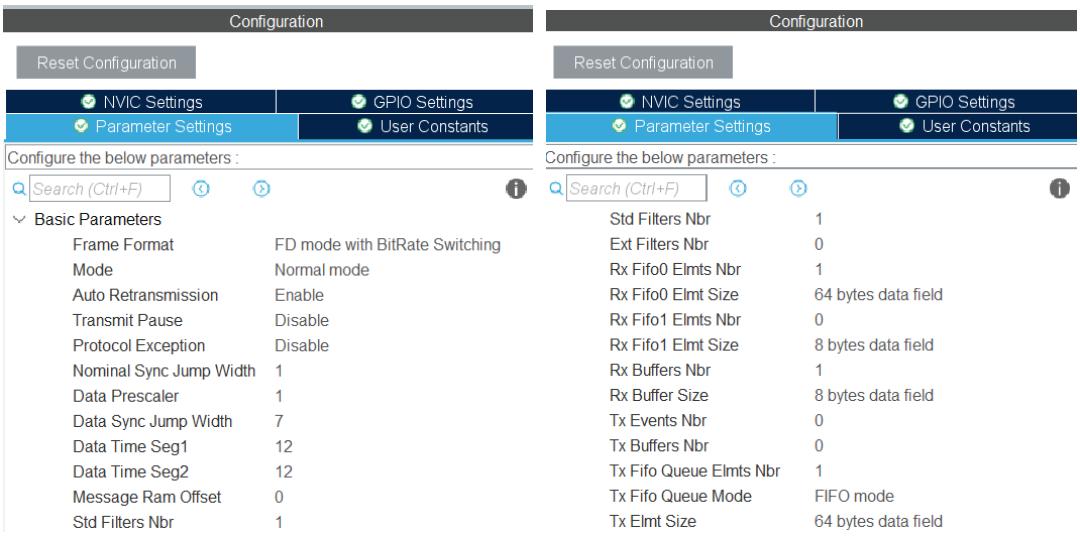


FIGURE III.25 : Organigramme de la transmission avec buffer.

III.3.2.2 Transmission avec les Interruptions

Une fois le test avec la méthode des buffers terminé, et que nous avons noté les erreurs et complications liées à cette méthode, nous avons décidé de réutiliser les interruptions pour des résultats plus optimaux.

Étape 1 : Modifié les configurations du périphérique FD CAN pour transmettre et recevoir les trames à l'aide des interruptions avec un débit de 500Kbps, ainsi que la taille des trames de 16 à 64 octets. La configuration de la partie «Frame Format» du périphérique est mise à «FD mode with BitRate Switching», pour avoir une transmission avec changement de débit pour la partie des données fixé pour ce test à 1Mbps, comme montre la Figure III.26.



(a) (b)

FIGURE III.26 : Configuration du FDCAN

Étape 2 : Configuration des variables et structures de transmission qui contiennent les données et les identifiants voir la Figure III.27.

```
Temperature.Identifier = 0x1;
Temperature.IdType = FDCAN_STANDARD_ID;
Temperature.TxFrameType = FDCAN_DATA_FRAME;
Temperature.DataLength = FDCAN_DLC_BYTES_64;
Temperature.ErrorStateIndicator = FDCAN_ESI_ACTIVE;
Temperature.BitRateSwitch = FDCAN_BRS_ON;
Temperature.FDFormat = FDCAN_FD_CAN;
Temperature.TxEventFifoControl = FDCAN_NO_TX_EVENTS;
```

FIGURE III.27 : Configuration des variables de transmissions.

Cette figure représente un exemple sur les variables qui contiendront les données intrinsèques des trames : ID et son format, type de protocole, taille et méthode de transmission...etc. On peut ainsi voir les différences qu'il y a avec les configurations du premier test.

Étape 3 : Envoie des trames sur le bus a l'aide des fonctions spéciales aux interruptions où on précise le périphérique utilisé, la structure à envoyer et le vecteur qui contient les données à envoyer, comme l'illustre la Figure III.28.

```
HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &Temperature, Temperature1);
for (i=0;i<64;i++){Temperature1[i]++;}
HAL_Delay(1500);
```

FIGURE III.28 : Code pour la transmission.

Étape 4 : Déclenchement d'une interruption qui fera appel à la fonction représenté dans la Figure III.29. Nous précisons le périphérique utilisé, la structure où stocker les données intrinsèques, et la zone mémoire responsable de cette interruption (FIFO0 ou FIFO1) et un vecteur qui contiendra les données.

```
void HAL_FDCAN_RxFifo0Callback(FDCAN_HandleTypeDef *hfdcan, uint32_t RxFifo0ITs)
{
      if((RxFifo0ITs & FDCAN_IT_RX_FIFO0_NEW_MESSAGE) != RESET)
   {
   HAL_FDCAN_GetRxMessage(hfdcan, FDCAN_RX_FIFO0, &RxHeader, R);
      if (RxHeader.Identifier == 0x1)
          {  for(i=0;i<64;i++){T1[i]=R[i];}
               s= snprintf(buffer,300,"--Temperature--\nID : %02x - %02x %02x %02x %02x %02x
                                       %02x %02x %02x\n%02x %02x %02x %02x %02x %02x %02x
                                       %02x\n%02x %02x %02x %02x %02x %02x %02x %02x\n%02x %02x
                                       %02x %02x %02x %02x %02x %02x\n%02x %02x %02x %02x %02x
                                       %02x %02x %02x\n%02x %02x %02x %02x %02x %02x %02x
                                       %02x\n%02x %02x %02x %02x %02x %02x %02x %02x\n%02x %02x
                                       %02x %02x %02x %02x %02x %02x\n--end Temperature--\n",
                           RxHeader.Identifier,R[0],R[1],R[2],R[3],R[4],R[5],R[6],R[7],
                           R[8],R[9],R[10],R[11],R[12],R[13],R[14],R[15],
                           R[16],R[17],R[18],R[19],R[20],R[21],R[22],R[23],
                           R[24],R[25],R[26],R[27],R[28],R[29],R[30],R[31],
                           R[32],R[33],R[34],R[35],R[36],R[37],R[38],R[39],
                           R[40],R[41],R[42],R[43],R[44],R[45],R[46],R[47],
                           R[48],R[49],R[50],R[51],R[52],R[53],R[54],R[55],
                           R[56],R[57],R[58],R[59],R[60],R[61],R[62],R[63]);
              CDC_Transmit_FS(buffer, s);
              }
```

FIGURE III.29 : Code pour la réception.

Une fois les données reçues on peut remarquer que les problèmes rencontres avec la transmission avec les buffers ont été supprimé. ainsi avec les interruptions il y a une manipulation des trames que ce soit pour l'envoie ou pour la réception bien plus intuitives. Et une plus grande fiabilité, vu que les erreurs rencontrées avec l'ancienne méthode ont été évité.

La figure III.30 illustre les résultats obtenus avec la méthode des interruptions.

```
--Temperature--
ID : 01 - 10 32 54 76 98 00 11 22
33 44 55 66 00 dd dc 1c
40 50 60 70 80 90 a0 b0
c0 d0 e0 f0 f1 dd 40 10
41 50 60 70 80 90 a0 b0
c0 d0 e0 f0 f1 dd 40 10
40 50 60 70 80 90 a0 b0
c0 d0 e0 f0 f1 dd 40 10
--end Temperature--

--Pression--
ID : 02 - 40 50 60 70 80 90 a0 b0
c0 d0 e0 f0 f1 dd 40 10
1e 2e 3e 4e 5e 6e 7e 8e
9e ae be ce de e0 e0 e1
1e 2e 3e 4e 5e 6e 7e 8e
9e ae be ce de e0 e0 e1
1e 2e 3e 4e 5e 6e 7e 8e
9e ae be ce de e0 e0 e1
--end Pression--
```

```
--Vitesse--
ID : 03 - a1 a2 a3 a4 a5 a6 a7 a8
a9 aa ab ac ad ae af a1
 a1 a2 a3 a4 a5 a6 a7 a8
a9 aa ab ac ad ae af a1
 a1 a2 a3 a4 a5 a6 a7 a8
a9 aa ab ac ad ae af a1
 a1 a2 a3 a4 a5 a6 a7 a8
a9 aa ab ac ad ae af a1
--end Vitesse--

--ABS--
ID : 04 -1e 2e 3e 4e 5e 6e 7e 8e
9e ae be ce de e0 e0 e1
1e 2e 3e 4e 5e 6e 7e 8e
9e ae be ce de e0 e0 e1
1e 2e 3e 4e 5e 6e 7e 8e
9e ae be ce de e0 e0 e1
1e 2e 3e 4e 5e 6e 7e 8e
9e ae be ce de e0 e0 e1
--end ABS--
```

FIGURE III.30 : Résultats de la réception.

Les étapes de ce test peuvent être résume dans l'organigramme suivant :

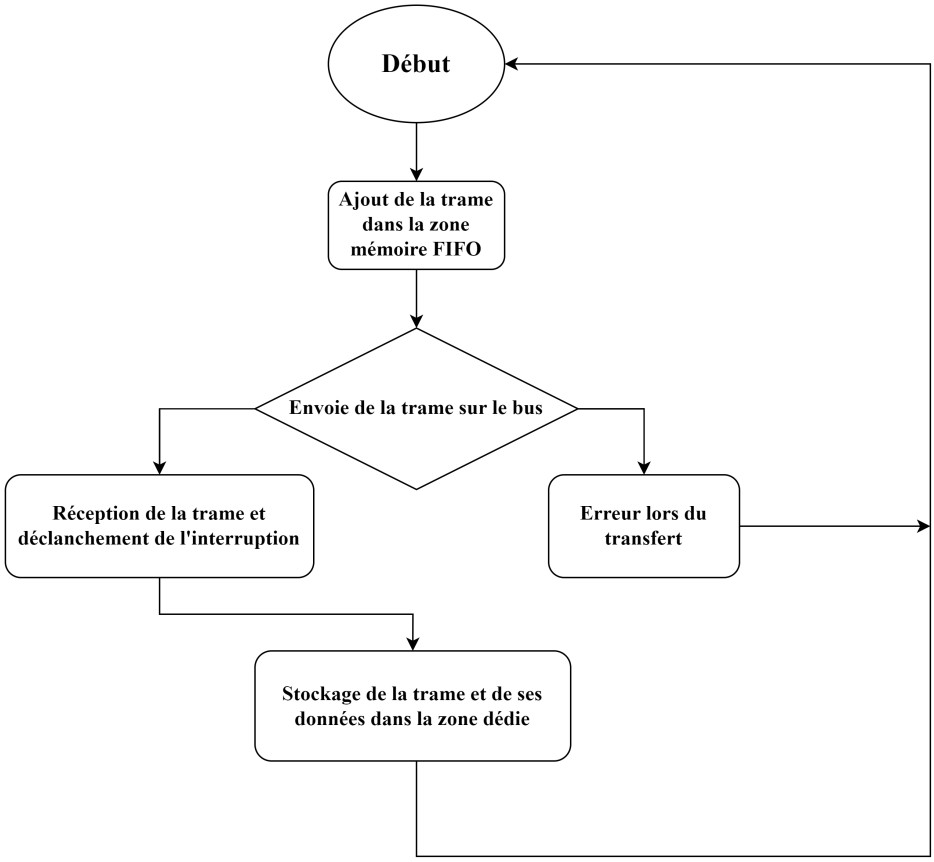


FIGURE III.31 : Organigramme de la transmission avec interruption.

A la fin de ces deux tests nous avons pu conclure et constater pas mal de points et corriger et affirmer nos choix concernant les méthodes à utiliser pour notre réalisation, en effet, l'utilisation des interruptions pour la transmission semble clairement être la meilleure solution en termes d'efficacité et de traitement.

Maintenant que l'on a réussi à réaliser des transmissions avec les deux protocoles séparément, et avec des méthodes différentes et un changement de débit entre la partie des données et de l'identifiant, nous avons eu l'idée d'ajouter une fonctionnalité pour pouvoir basculer de protocole et de débit à partir de notre interface, sans avoir à débrancher ni à reprogrammer entièrement l'analyseur, seulement avec un simple clic et on peut ainsi basculer du protocole CAN vers le CAN FD ou inversement.

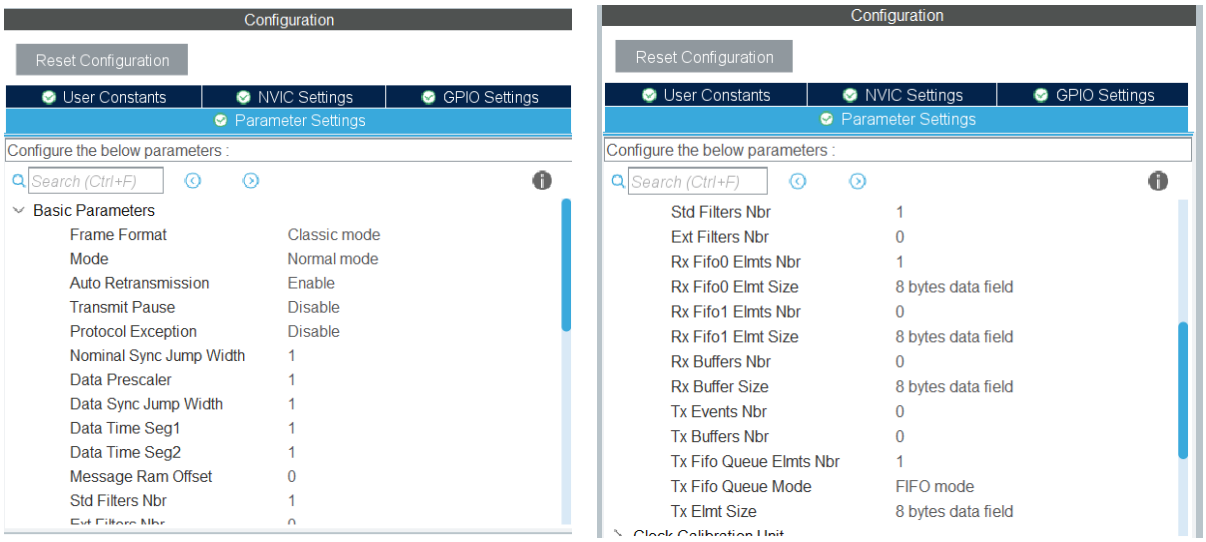
III.3.3 Transmission CAN et CAN FD

Pour pouvoir réaliser ce test important et simuler le fameux bouton de l'interface, nous utilisons pour cela un terminal pour envoyer des commandes via USB et les deux boutons de la carte STM32H743vit6 en mode interruption.

Le but de ce test est de créer un analyseur capable de recevoir les données en format CAN et CAN FD, avec la possibilité de pouvoir changer de protocole de communication avec un simple clic, pour réussir à réaliser cela, nous allons tirer parti des précédents tests et de notre compréhension du fonctionnement du périphérique FD CAN de notre carte STM.

Étape 1 : On commence alors le test par définir une configuration par défaut, qui consiste en une communication avec le protocole CAN de 8 octets avec un débit de 125kbps. Pour cela nous configurons l'horloge à sa pleine puissance à 480 MHz.

Étape 2 : La sélectionne du type de protocole, la taille de la trame, le type d'identifiant, la méthode de transmission, et la configuration du débit qu'on fixera en premier lieu à 125Kbps comme démontre dans la figure suivante :



(a) (b)

FIGURE III.32 : Configuration du FD CAN.

Étape 3 : Après l'étape des configurations, nous déclarons les variables qui contiendrons les données et les structures à envoyé.

Étape 4 : Dans la boucle infinie où se trouve la partie du code qui servira à envoyer de façon continue les différentes trames sur le bus.

On a créé une grande boucle «Switch» pour pouvoir changer de type de protocole et le débit de transmission, sans avoir à tout débrancher et à reprogramme de nouveau, qui fonctionnera selon ce que la carte reçoit via USB du terminal pour la partie réception et avec l'appui sur les boutons de la carte pour la partie transmissions, ceci permettra de faire les changements nécessaires, toute fois, il faut bien noter qu'une fois que le périphérique FD CAN est configuré et initialisé, il est strictement impossible de changer ses caractéristiques, néanmoins, il y a une petite astuce pour pouvoir y remédier, on arrêt le périphérique avec une fonction spécialement faite pour cela «HAL_FDCAN_Stop» et on le reconfigure avec les nouvelles caractéristiques souhaitées, puis on refait son initialisation ainsi que

son démarrage comme illustrent les Figures III.33 et III.34.

Ainsi, on peut tout changer que ça soit le type de protocole, la taille des données, le débit, toute fois il faut noter que pour éviter toutes sortes d'erreurs lors de ce basculement, on a instauré un petit délai de deux secondes entre ces changements et l'envoie des trames sur le bus qui corresponde à ces nouvelles configurations.

```
case 0xc2://Le cas où la donnée reçu par USB qui est stocker dans la variable
bufferr_receive est égal à «0xc2» pour une transmission CAN avec un débit de
125Kbps
//stopper les interruptions sur le périphérique FD CAN
HAL_FDCAN_TT_DeactivateNotification(&hfdcan1, FDCAN_IT_RX_FIFO0_NEW_MESSAGE);

//stopper le périphérique FD CAN
 HAL_FDCAN_Stop(&hfdcan1);
//Nouvelle configuration du périphérique FD CAN
 hfdcan1.Instance = FDCAN1;
 hfdcan1.Init.FrameFormat = FDCAN_FRAME_CLASSIC;
 hfdcan1.Init.NominalPrescaler = 2;
//Initialisation du périphérique FD CAN
 HAL_FDCAN_Init(&hfdcan1);
//Démarrage du périphérique FD CAN
 HAL_FDCAN_Start(&hfdcan1);
 HAL_FDCAN_ActivateNotification(&hfdcan1, FDCAN_IT_RX_FIFO0_NEW_MESSAGE,0);
 HAL_Delay(2000);//Délai de 2 secondes
 bufferr_receive[0]=0xff;

 speed=0;}
break;
```

FIGURE III.33 : Code pour le changement de débit.

Dans la Figure III.34 suivante, nous allons voir cette fois-ci les changements opérés pour basculer d'un protocole à un autre, et nous avons mêmes changer la taille des données transmises, en effet dans le protocole CAN nous envoyons 8 octets or lorsqu'on utilise le protocole CAN FD on envoie des trames de 16 octets pour bien montrer que nous sommes en CAN FD.

```
case 0xf1://Le cas où la donnée reçu par USB est égal à «0xf1» pour une
transmission CAN FD avec un débit de 125Kbps pour la partie nominale et 1Mbps pour
la partie des données

//stopper les interruptions sur le périphérique FD CAN
HAL_FDCAN_TT_DeactivateNotification(&hfdcan1, FDCAN_IT_RX_FIFO0_NEW_MESSAGE);

//stopper le périphérique FD CAN

 HAL_FDCAN_Stop(&hfdcan1);

//Nouvelle configuration du périphérique FD CAN

 hfdcan1.Instance = FDCAN1;

 hfdcan1.Init.FrameFormat = FDCAN_FRAME_FD_BRS;

 hfdcan1.Init.NominalPrescaler = 1; hfdcan1.Init.NominalSyncJumpWidth = 2;

 hfdcan1.Init.NominalTimeSeg1 = 93; hfdcan1.Init.NominalTimeSeg2 = 2;

 hfdcan1.Init.DataPrescaler = 8;    hfdcan1.Init.DataSyncJumpWidth = 13;

 hfdcan1.Init.DataTimeSeg1 = 21;    hfdcan1.Init.DataTimeSeg2 = 13;

 hfdcan1.Init.RxFifo0ElmtSize = FDCAN_DATA_BYTES_16;

 hfdcan1.Init.TxElmtSize = FDCAN_DATA_BYTES_16;

 //Initialisation du périphérique FD CAN

 HAL_FDCAN_Init(&hfdcan1);

//Démarrage du périphérique FD CAN

 HAL_FDCAN_Start(&hfdcan1);

 HAL_FDCAN_ActivateNotification(&hfdcan1, FDCAN_IT_RX_FIFO0_NEW_MESSAGE,0);

 HAL_Delay(2000);//Délai de 2 secondes

 bufferr_receive[0]=0xff;    tftfd=0;      fd_speed=0;    speed=0;

}

break;
```

FIGURE III.34 : Code pour le changement de protocole.

Ce test nous a offert les résultats escomptés avec aucune erreur ou de résultats inattendus, la Figure III.35 ci-dessous montre ces fameux résultats :

ID :01--temperature--	ID :03--lumiere--
60 82 a4 c6 e8 50 61 72	cb cc cd ce cf d0 d1 d2
--end-temperature	--end-lumiere
ID :02--Pression--	ID :04--ABS--
90 a0 b0 c0 d0 e0 f0 00	48 58 68 78 88 98 a8 b8
--end-pression	--end-ABS
i am fdcan v1 125kbps	i am can v2
ID :03---lumiere--	ID :01--temperature--
20 20 20 20 20 20 50 20 01 01 01 01 01 01 01 01	3b 5d 7f a1 c3 2b 3c 4d
--end-lumiere	--end-temperature
ID :04---ABS--	ID :02--Pression--

FIGURE III.35 : Résultats du test.

Les mêmes opérations sont effectuées sur la carte de transmission pour pouvoir basculer entre les protocoles et le changement de débit, avec seul différence la méthode d'enclenchement du changement dans la carte d'émission on doit appuis sur un des boutons pour les effectués.

Arrive à ce stade, nous sommes capables de recevoir les données avec les deux protocoles et de pouvoir basculer entre eux aisément et même de changer le débit de la transmission, et ainsi on a pu commencer l'étape de diagnostic des données.

L'organigramme ci-dessus résume la totalité du test

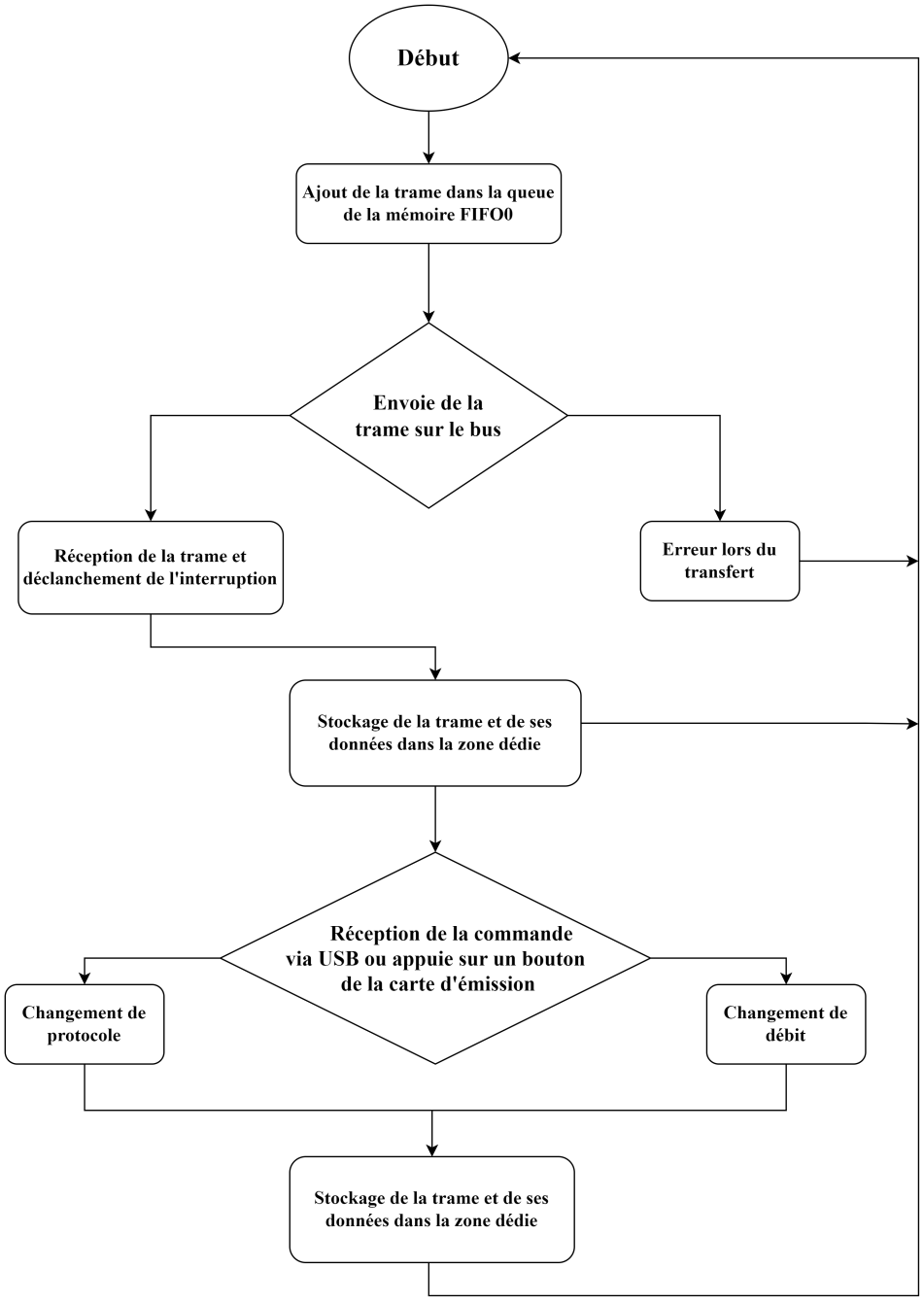


FIGURE III.36 : Organigramme du basculement.

III.3.4 Diagnostic OBD

Dans cette étape, une fois que nous avons fini le travail sur les données et les ID aléatoires, nous allons travailler avec des valeurs qu'on retrouve dans le monde réel.

On se basant sur une base de données, nous avons tiré les véritables valeurs des ID de certaines grandeurs, nous avons pu construire une fonction nommée « obd », qui dans ce test, nous permettra de passer d'une simple émission et réception de données à un traitement et une interprétation précise de ces dernières.

Car, selon l'ID reçu, les données de la trame recevront des conversions et des traitements différents, afin de les présenter sous un format plus clair, plus compréhensible à l'échelle humaine et ainsi changer les « 0 » et « 1 » en grandeurs physiques concret tel que : Température en C°, Pression en KPa, Vitesse en KM/H... etc.

Étape 1 : On refait les mêmes configurations d'horologes, de variables et de périphériques que le test précédent.

Étape 2 : La Figure III.37 ci-dessus, montre une partie de la fonction « obd », où on peut remarquer que selon la valeur de « RxHeader.Identifier» qui est l'ID de la trame reçue, alors on aura un traitement, une conversion et une interprétation différentes.

```
extern float engine_data;
void obd (){
/* Details from http://en.wikipedia.org/wiki/OBD-II_PIDs */
switch (RxHeader.Identifier) {
case FUEL_PRESSURE://A*3
engine_data = (R[3]*3);
s=snprintf(buffer,40,"The fuel pressure :%d KPa\n",(int)engine_data);
CDC_Transmit_FS(buffer, s);
break;
case VEHICLE_SPEED://A[Km/h]
engine_data = R[3];
```

FIGURE III.37 : Fonction OBD pour le diagnostic.

III.3.5 Affichage sur le LCD

Nous afficherons les données reçues sous forme de chronogramme sur l'écran de notre LCD « ILI9488 ».

Étape 1 : On refait les mêmes configurations d'horologes, de variables et de périphériques que le test précédent.

Étape 2 : Ajouter les pins et les configurations nécessaires au fonctionnement du LCD sur notre STM32CUBEMX projet.

Étape 3 : Configuration du port SPI ainsi qu'un timer pour générer des pulsation « PWM », pour pouvoir manipuler les pixels sur l'écran d'affichage, nous nous sommes aidés d'une bibliothèque déjà disponible contenant les fonctions de base à utiliser et nous avons dû nous inspire et nous baser dessus pour créer plusieurs fonctions différentes afin de manipuler au mieux les données et les affiches sur notre écran.

Nous pouvons voir de la Figure III.38 certaines de ces fonctions, notamment les fonctions pour dessiner les fronts montants et descendants du chronogramme correspondant aux vecteurs de données reçues.

```
void haut(uint16_t col)
{       drawLine(x, y, x-5,y, col);
        drawLine(x-5, y, x-5,y+2, col);
        x=x-5;
}
void bas(uint16_t col)
{       drawLine(x, y+10, x-5,y+10, col);
        drawLine(x-5, y+10, x-5,y+8, col);
        x=x-5;
}
void lath(uint16_t col)
{
   drawLine(x+5, y, x+5,y+10, col);//line vert + de px vers le bas
}
void lat(uint16_t col)
{  drawLine(x+5, y, x+5,y+10, col);
}
```

FIGURE III.38 : Fonction pour l'affichage LCD

Ce dernier test est alors la conclusion et l'aboutissement de notre travail pour la réalisation d'un analyseur CAN capable de recevoir des données avec les 2 formats CAN et CAN FD, possédant aussi la possibilité de basculer entre eux et de réaliser un diagnostic complet sur les données reçues. mais aussi de les interpréter et de les afficher d'une manière simple et efficace sur un écran LCD.

La Figure III.39 illustre certains résultats.

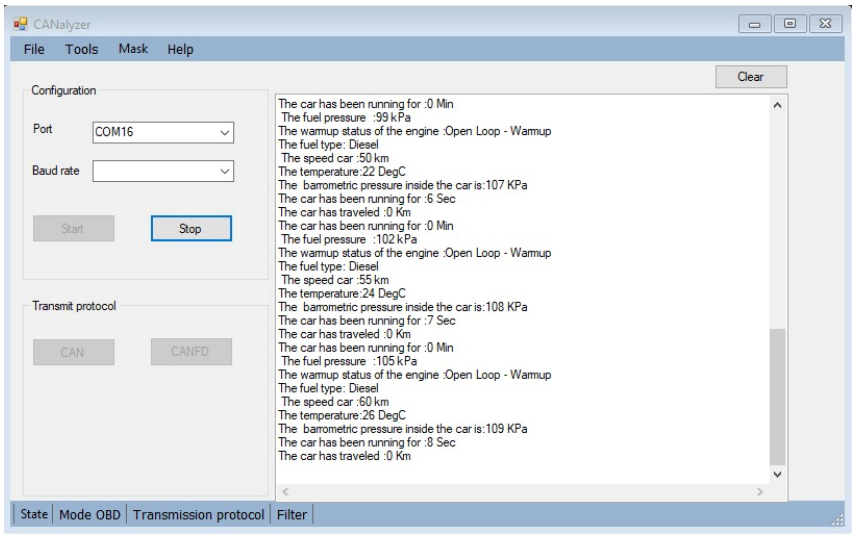


FIGURE III.39 : Résultats du diagnostic

Conclusion

En conclusion de ce chapitre qui est dédié principalement aux tests et résultats de la réalisation de l'analyseur CAN, nous avons pu constater l'importance cruciale de suivre ces étapes dans le processus de développement, ainsi, ces tests ont joué un rôle essentiel pour garantir la fiabilité, la précision et la performance de l'analyseur CAN.

Au cours de ce chapitre, nous avons vu différentes méthodes de test, qui permettent de vérifier le bon fonctionnement de l'analyseur dans divers scénarios. Ces tests nous permettent de valider sa capacité à interpréter les données CAN ou CAN FD et à fournir des résultats cohérents.

Les résultats obtenus grâce à ces tests sont d'une grande valeur, car ils nous permettent de comprendre en profondeur le comportement de l'analyseur et d'identifier d'éventuels problèmes ou limitations, ainsi que la possibilité d'ajuster et d'optimiser les algorithmes et les paramètres afin d'améliorer les performances globales de l'analyseur.

En résumé, les tests et les résultats obtenus sont ceux que nous escomptions et ainsi nous permettent de fournir un outil fiable, précis et adapté aux besoins de l'industrie automobile.

Conclusion générale

Arrivant à l'aboutissement de notre projet qui porte sur la réalisation d'une passerelle CAN/USB combinée à un analyseur CAN, compatible à la fois avec les données des deux protocoles CAN et CAN FD, nous sommes extrêmement satisfaits des résultats obtenus, les tests approfondis que nous avons effectués ont confirmé l'efficacité et la polyvalence de notre solution, répondant ainsi aux besoins croissants de l'industrie automobile en matière d'outils d'analyse et de diagnostic pour les réseaux CAN modernes.

La conception de notre passerelle a été minutieusement étudiée pour permettre la réception et le traitement précis des données provenant du bus, ainsi au cours de nos tests nous avons démontré sa compatibilité avec divers types de messages et des vitesses de transmission variables, tout en garantissant l'intégrité des données.

L'intégration d'un analyseur CAN performant au sein de notre dispositif a grandement contribué à l'efficacité et à la précision des diagnostics effectués sur les données reçues. Grâce à des algorithmes sophistiqués, nous avons été en mesure d'interpréter les informations contenues dans les messages CAN, de les convertir en grandeurs compréhensibles pour les utilisateurs, et de les afficher de manière claire et concise sur notre interface logicielle.

La visualisation en temps réel des données analysées revêt une importance primordiale. C'est pourquoi nous avons intégré un écran LCD à notre dispositif, permettant aux utilisateurs d'observer instantanément les grandeurs et d'obtenir une vue d'ensemble des performances du réseau CAN. Cette fonctionnalité facilite l'identification rapide de tout dysfonctionnement ou problème de communication.

Grâce à notre projet, nous avons répondu aux besoins de l'industrie automobile en proposant un outil d'analyse et de diagnostic complet pour les réseaux CAN et CAN FD. Les professionnels du secteur disposent désormais d'une solution puissante leur permettant de comprendre en profondeur le fonctionnement des systèmes électroniques dans les véhicules, de diagnostiquer efficacement les problèmes de communication, et de valider les protocoles utilisés.

L'interface utilisateur de notre passerelle/analyseur facilite l'accès aux données analysées, leur interprétation et la prise de décisions éclairées pour améliorer les performances des systèmes embarqués, les utilisateurs peuvent visualiser les informations clés, générer des rapports détaillés, et ainsi optimiser les réglages et les réparations nécessaires.

En conclusion, notre projet a démontré notre expertise dans le domaine des protocoles de communication automobile et notre capacité à développer des solutions innovantes répondant aux besoins spécifiques de l'industrie.

Bibliographie

[1] P. G. A. G. Marco Di Natale, Haibo Zeng, *Understanding and Usingthe Controller Area Network Communication Protocol*, 1st ed. Springer, 2012.

[2] W. Lawrenz, *CAN System Engineering.* Springer-Verlag London 1997, 2013.

[3] P. Kadionik, "Le bus can." ENSEIRB, 2001, consulté en Février 2023.

[4] C. Lery, "Mise en œuvre d'une communication par bus can." Université BlaisePascal, Clermont Ferrand, 2009.

[5] K. J. Matwiejczyk Jajczyk, "Can bus diagnostics." Poznań University ofTechnology.

[6] F. A. COLLET RACHID, "Bus can." Laboratoire des technologies innovantes, université de Picardie Jules-Verne, France.

[7] "Can with flexible data-rate." BOSCH, April 17th, 2012, consulté en Mars 2023.

[8] "Fdcan peripheral on stm32 devices," STMicroelectronics, Tech. Rep., 2019.

[9] B. Westoff, "Can fd – tout sauf automobil," 14 septembre 2022.

[10] "Can bus analyzer." ADFweb, 2022, consulté en Mars 2023.

[11] "USB in a NutShell - Chapter 1 - Introduction — beyondlogic.org," https://www.beyondlogic.org/usbnutshell/usb1.shtml, consulté en Juillet 2023.

[12] "Front Page | USB-IF — usb.org," http://www.usb.org/, consulté en Juillet 2023.

[13] "Les protocoles USB.pdf — fr.scribd.com," https://fr.scribd.com/document/290665066/Les-protocoles-USB-pdf, consulté en Juillet 2023.

[14] M. Falch, "Can fd explained," Danemark 202.

[15] M. S. S. Edition, in *Carmine Noviello.* Leanpub, fevrier 2022, pp. 21–51.

[16] STMicroelectronics, "Stm32h742xi/g stm32h743xi/g," 2019.

[17] M. Technology, "High-speed can transceiver," 2016.

[18] Mr.Belloula, "Cours multiplexage." ENST, 2019, p. 18.

[19] LCDWIKI, "3.5inch spi module msp3520 user manual," 2018.

Annexe A

A.1 Sous-couches de la couche physique

- **La sous-couche PLS** : gère la représentation du bit, Codage/Décodage de bit, Bit timing et la synchronisation.
- **La sous-couche PMA** : définit les niveaux électriques des signaux
- **La sous-couche MDI** : définit le support de transmission (les connecteurs).

A.1 Sous couches de la couche liaison de données

La sous couche LLC : responsable du filtrage des messages, la notification des surcharges (overload) et la procédure de recouvrement des erreurs.

La sous-couche MAC : considérée comme le cœur du protocole CAN. Elle est responsable de la détection et la signalisation des erreurs, la mise en trame du message, l'arbitrage et l'acquittement des erreurs, Encapsulation/Décapsulation des données ainsi que le Codage de trame (Stuffing/Destuffing). Les opérations MAC sont supervisées par une entité de contrôle qui surveille l'état des erreurs et limite les opérations d'un nœud en cas de détection d'une défaillance permanente afin de contenir les erreurs et d'assurer une plus grande fiabilité.

A.3 Champ de CRC

Utilisé pour la détection des erreurs, il est possible de détecter jusqu'à cinq erreurs d'un seul bit dans le message. Seuls les champs de SOF,d'arbitrage, de contrôle et de données sont utilisés pour le calcul de ce champ.

La séquence de CRC est obtenue en sélectionnant le polynôme d'entrée (à diviser) en tant que flux de bits à partir du bit de début de trame (SOF) jusqu'au champ de données suivi de 15 zéros. Ce polynôme est divisé par un polynôme générateur, $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$, et le reste de la division constitue la séquence CRC de la trame.

A.4 Arbitrage

Pour déterminer la priorité de messages, Le bus CAN utilise la méthode d'accès CSMA/CR, deux nœuds commencent une transmission en même temps sur un bus libre. Tout au long de la trame, chaque nœud relié via sans transmetteur la valeur du bus pour chaque bit transmis, au début les deux nœuds envoient leur bit dominant de début du trame SOF, ensuite le MSB de l'identificateur est envoyé, Si un nœud transmet un bit récessif lit un bit dominant, il arrête la transmission.

A.5 Types d'erreurs

Le protocole CAN peut détecter et traiter cinq types d'erreurs :

- Erreur de bit (Bit error) : Une erreur de bit est détectée lorsque la valeur du bit reçu, lue sur le bus par l'émetteur, est différente que celle qu'il a envoyée.
- L'erreur de Stuffing (stuff error) : Une erreur de Stuffing est détectée quand il y a plus de cinq bits consécutifs de même polarité sur le bus.
- L'erreur du CRC (CRC error) : Si le CRC calculé par le récepteur est différent de celui qui est stocké dans la trame envoyée par l'émetteur.
- L'erreur d'Acknowledge Delimiter : Le récepteur détecte une erreur d'acquittement lorsqu'il n'observe pas un bit récessif lors du champ de Acknowledge Delimiter.
- L'erreur de Slot Acknowledge Ack (Acknowledgment Error) : Une erreur de Slot Ack est signalée par l'émetteur s'il ne lit pas un bit dominant lors du champ de slot Ack.

A.6 Traitement des erreurs

Lorsqu'une des erreurs décrites précédemment est détectée, chaque nœud est informé par une trame d'erreur. A cause de l'envoie d'un drapeau d'erreur actif, les autres nœuds voient une erreur de bit stuffing, donc ils commencent également à envoyer une trame d'erreur. Cet effet se produit lorsque l'erreur est locale et sa gestion se fait ainsi :

- Une erreur est détectée.
- Une trame d'erreur est envoyée par tous les nœuds qui on détecter celle-ci.
- Le message en cours est rejeté par tous les nœuds CAN.
- Les compteurs d'erreurs (décrits ci-dessous) de chaque nœud CAN sont affectés conformément aux règles de confinement des erreurs.

Confinement des erreurs : Le CAN comprend un mécanisme spécifique ayant pour but de détecter la défaillance des nœuds et les place dans des états passifs ou éteints, de sorte qu'elles ne peuvent pas affecter l'état du bus. Un nœud peut être dans l'un des trois états :

- Erreur active : Les unités dans cet état sont supposées fonctionner correctement, des messages peuvent être envoyés et reçus et les erreurs sont signalées à l'aide d'un drapeau d'erreur active.
- Erreur passive : Les unités dans cet état sont atteintes après détection plusieurs erreurs sur le bus. Elles peuvent toujours transmettre sur le bus, mais en cas d'erreur, seul un drapeau d'erreur passive est envoyé. D'où, un nœud CAN passif en cas d'erreur ne peut gêner le reste du trafic CAN. Il ne marquera que les erreurs dans ses propres messages transmis.
- Bus off : Les unités dans cet état sont très probablement déconnectées du bus CAN et ne peuvent avoir aucune influence sur le bus.

Chaque nœud possède deux compteurs d'erreurs, un compteur d'erreur de transmission (Transmit Error Count) et un compteur d'errer de réception (Receive Error Count) ayant pour rôle de compter les erreurs détectées lors de la transmission et l'émission des trames.

Les unités changent d'état en fonction de la valeur de ces deux compteurs. Lorsque le nombre d'erreurs devient trop important et que le gestionnaire est déjà en erreur passive, le nœud se met en Bus Off et se déconnecte du bus. L'état Bus Off ne peut être quitté que par des actions appropriées du contrôleur hôte (réinitialisation logicielle ou matérielle). Les transitions dans les différents modes s'effectuent suivant la valeur des compteurs, Figure fig :erreur.

- Erreur active : le compteur de réception et le compteur d'émission ont une valeur inférieure à 127
- Erreur passive : Au moins un des deux compteurs d'erreurs est supérieur à 127 et le compteur d'erreurs d'émission est inférieur à 256.
- Bus Off : Le compteur d'erreurs d'émission est supérieur à 255.

Les compteurs d'erreurs sont incrémentés en cas d'erreurs détectées ou décrémentés en cas de messages réussis. L'incrémentation et décrémentation se fait suivant ces règles :

- Le compteur de réception est incrémenté de 1 si : le récepteur détecte une erreur, sauf si l'erreur est un Bit Error durant un flag d'erreur active ou un flag de surcharge.
- Le compteur de réception est incrémenté de 8 si : le récepteur détecte un bit dominant après avoir envoyé un flag d'erreur, ou lorsqu'il voit un Bit Error tandis qu'il reçoit un flag d'erreur active ou un flag de surcharge.

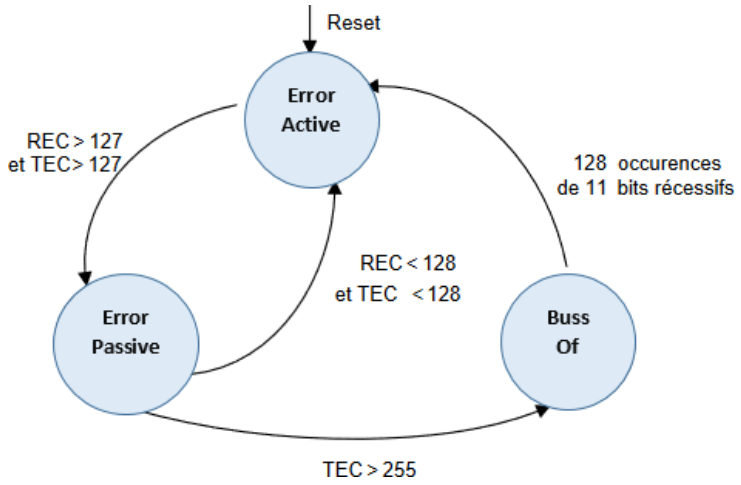


FIGURE A.1 : Diagramme d'Etat des erreurs du bus CAN

- Le compteur de transmission incrémenté de 8 si un émetteur envoie un flag d'erreur, sauf :
  - Lorsqu'un émetteur est en état d'erreur passive et qu'il détecte une erreur d'acquittement.
  - Si l'émetteur envoie un flag d'erreur lors d'une erreur de bit-stuffing durant la période d'arbitrage.
  - Lorsque l'émetteur détecte un bit error pendent qu'il émet un flag erreur active ou un flag de surcharge.
- Tout nœud tolère 7 bits dominants après la réception d'un drapeau d'erreur active ou passive, ou d'un drapeau de surcharge. Les compteurs d'émission et de réception s'incrémentent de 8 s'ils détectent un 8eme bit dominant consécutif suivant le drapeau d'erreur passive, ou un 14eme bit dominant consécutif après un drapeau d'erreur active ou un drapeau de surcharge, et après toutes les suites de huit bits dominants consécutifs.
- Le compteur de réception quand le REC est compris entre 1 et 127, se décrémente à 1 après chaque réception d'une trame sans erreur, si le REC est à 0 sa valeur ne change pas.
- Le compteur d'émission se décrémente à 1 après une transmission réussie d'une trame. Si la valeur du compteur est à 0, le compteur ne s'incrémente pas.

Recouvrement des erreurs : Lorsqu'une erreur est détectée, le recouvrement de cette dernière se fait par la retransmission automatique de la trame jusqu'à qu'elle soit transmise sans erreur. (Son compteur d'erreur passe en mode passive).

Le temps entre le moment de détection d'erreur et le moment de retransmission d'une nouvelle trame est au maximum égale au temps de transmission de 29 bits pour CAN 2.0A et 31 bits pour CAN 2.0B.

A.7 Protocole USB

Les paquets de début de trame SOF

Ce paquets est composé de 11 bits envoyé par l'hôte toutes les 1ms sur un bus en pleine vitesse. La figure suivante illustre le paquet de début de trame : **Les paquets jetons**

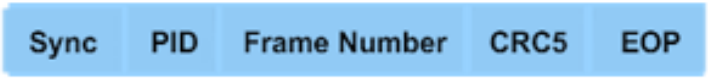


FIGURE A.2 : Paquet de début de trame [13].

Il existe 3 sortes différentes de ces paquets :

- In : ce qui signifie que l'hôte veut lire des informations,
- Out : pour signifie que l'hôte désire envoyer/écrire des informations,
- Setup : ce mode est utilisé pour envoyer des commandes.

La figure suivante nous permet d'avoir une meilleure visuelle du format de ce paquet : **Les paquets de données**



FIGURE A.3 : Les paquets jetons [13].

Il existe 2 type de paquets de données :

- Data0,
- Data1.

En mode haute vitesse il existe 2 autres type de paquets :

- DATA2,
- MDATA.

La figure suivante illustre la forme de ces paquets : Les paquets de données peuvent avoir différents débit de transmission tel que :

- La taille maximale pour une transmission basse vitesse est de 8 octets,
- La taille maximale pour une transmission pleine vitesse est de 1023 octets,



FIGURE A.4 : Les paquets de données [13].

- La taille maximale pour une transmission haute vitesse est de 1024 octets.

Les paquets d'Etat

Il en existe 3 types qui font partis du PID :

- ACK pour valide la réception,
- NAK pour signifie que le périphérique ne peut ni recevoir ni envoyé de données, utilisé aussi dans le mode d'interruptions pour signale de ne pas envoyer de données à ce périphérique,
- STALL pour signifie que le périphérique est bloqué.

La figure suivante détaille le format de ce paquet :



FIGURE A.5 : Les paquets d'Etat [13].

La figure suivante illustre une communication USB simple avec les différents paquets utilisés :

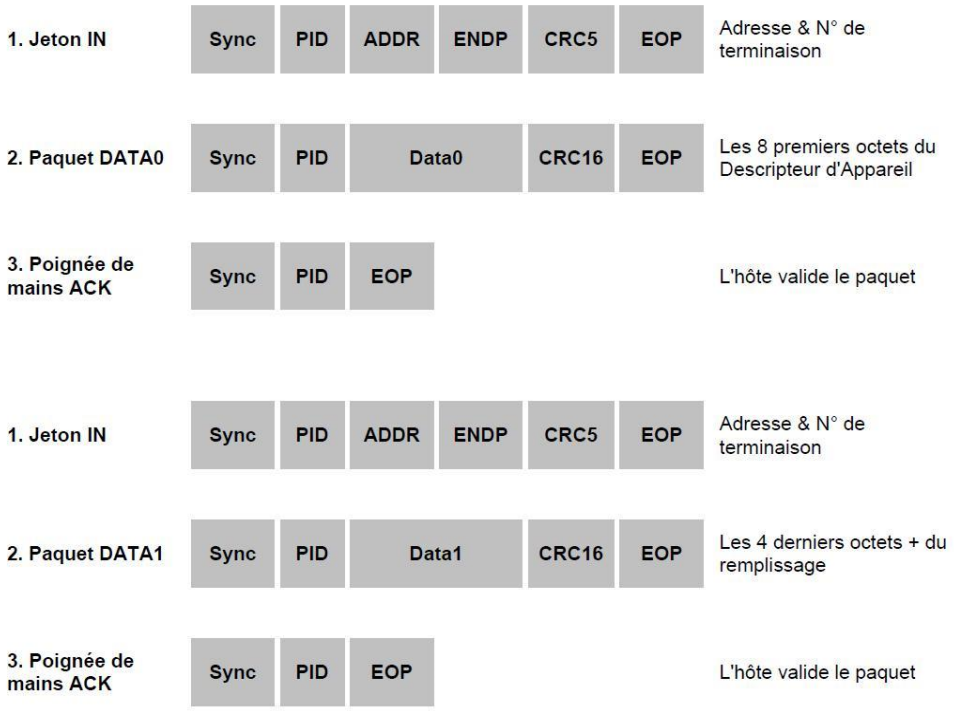


FIGURE A.6 : Communication USB [13].

A.8 Type de trames USB

Voici les principaux types de trames :

- Trame de synchronisation (Sync Frame) : Cette trame marque le début d'un paquet de données. Elle est composée d'un motif de bits spécifique qui permet à l'hôte et au périphérique de synchroniser leurs horloges de communication.
- Trame de donnée (Data Frame) : Cette trame contient les données à transférer entre l'hôte et le périphérique ou vice versa. Les trames de données sont utilisées pour le transfert de données dans les trois différents modes. Elles contiennent des informations sur le type de transfert, la longueur des données et les données elles-mêmes.
- Trame de gestion de bus (Bus Management Frame) : Cette trame est utilisée pour la gestion du bus USB. Elle peut être utilisée pour indiquer la fin d'un paquet de données, signaler une erreur de transmission ou effectuer d'autres opérations de contrôle du bus.
- Trame de token (Token Frame) : Les trames de token sont utilisées pour initier une transaction USB. Elles contiennent des informations telles que l'adresse du périphérique, le type de transaction lecture, écriture ou configuration ainsi que le type de donnée attendu : données, statut ou handshake.

- Trame de réponse (Handshake Frame) : Ces trames sont utilisées pour confirmer la réception ou la transmission de données. Elles sont envoyées en réponse à une trame de token et peuvent indiquer l'acceptation ou le rejet de la transaction.
- Trame de prévention des erreurs (Error Frame) : Cette trame est utilisée pour signaler une erreur de transmission sur le bus USB.

A.9 OBD

La norme OBD qui est pour « On Board Diagnostic » a été mise en place au départ par la CARB « Californian Air Resources Board» pour contrôler les émissions polluantes des véhicules, l'arrivée des calculateurs électroniques de gestion du moteur et des capteurs associés a permis aux véhicules de diminuer leurs rejets polluants.

L'OBD stipule que le véhicule doit sur toute sa durée de vie, surveiller en permanence le bon fonctionnement de son moteur.
Il existe plusieurs normes de l'OBD [18] :

- L'OBD ou OBDI qui standardise le connecteur afin qu'il soit identique pour tous les véhicules. Le protocole de communication lui reste plus ou moins spécifique suivant les marques,
- L'OBDII est venu en 1996 aux États Unis pour spécifier des protocoles communs,
- L'EOBD pour Européen OBD reprenant l'OBDII est spécifique pour les véhicules européens.

Le connecteur de diagnostic OBD

Le connecteur doit obligatoirement se situer dans l'habitacle, on le trouve généralement sous le volant dans le compartiment à fusibles ou sous le cendrier près du frein à main, une valise de diagnostic OBD y est brancher pour avoir accès aux informations.

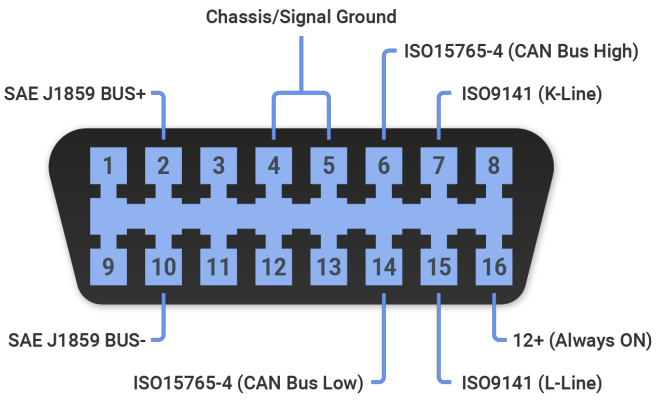


FIGURE A.7 : Le connecteur de diagnostic OBD [18].

Annexe B

B.1 Les microcontrôleurs STM32

Les microcontrôleurs sont divisés en dix-sept sous famille, le tableau B.1 et la Figure B.1 donnent une brève description et les principales caractéristiques de chaque famille STM32.

Tableau B.1 : Caractéristiques des familles STM [18]

Famille	Caractéristiques
STM32H7	Haute performances, gamme de mémoire facilement extensible, périphériques avancés (la vitesse de périphériques est indépendante de la vitesse du processeur), une mémoire plus grande, dote d'une interface USB-C.
STM32F4	Haute performance, vitesse d'horloge enlevée, destinée aux applications multimédias.
STM32F7	Performances théoriques maximales du Cortex-M, haut puissance et mémoire élevée.
STM32L0	Destinées aux applications à faible consommation d'énergie.
STM32F0	Faible consommation d'énergie, coût abordable.
STM32G0	Faible consommation d'énergie, boîtiers de petite taille et un coût plus bas.
STM32F1	Faible coût et faible consommation d'énergie, meilleur choix pour débuter avec les STM32.
STM32G4	Haute efficacité énergétique, traitement en temps réel, dote d'un contrôleur USB-C Power Delivery.
STM32WB	Dédiés à la connectivité sans fil, facile à utiliser, fiable et parfaitement adaptés aux applications industrielles.

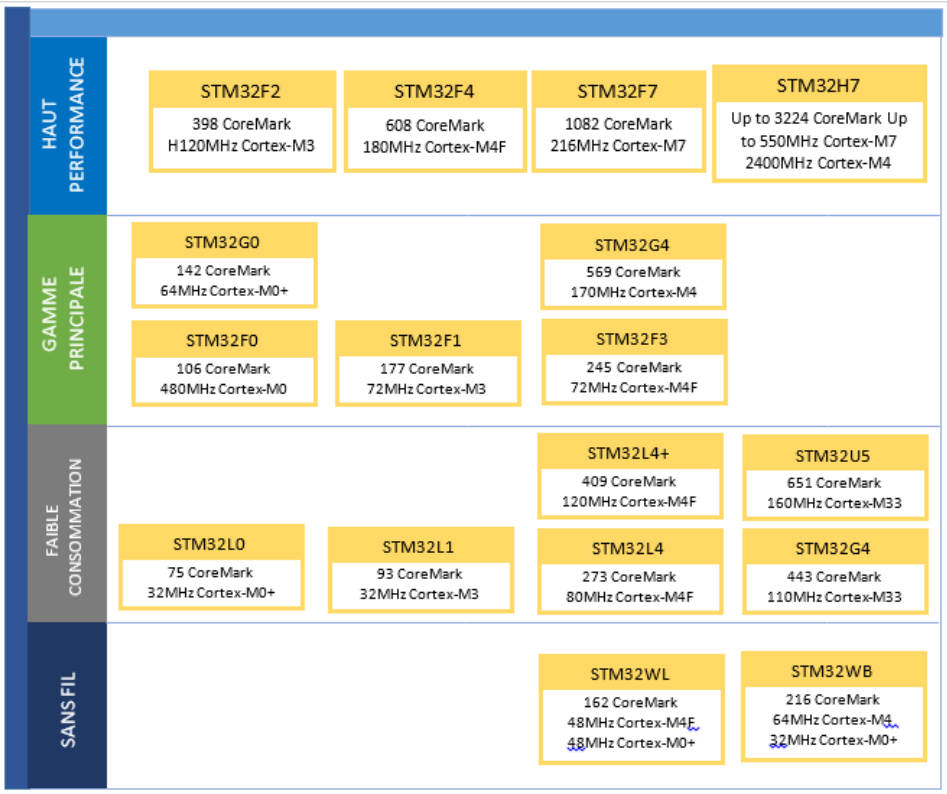


FIGURE B.1 : Caractéristiques des familles STM [18]

B.2 Caractéristiques de la carte STM32H743vit6

- Équipé de plusieurs fonctionnalités de sécurité telles que le ROP, PC-ROP pour protéger les données des attaques extérieures.
- Réinitialisation et gestion de l'énergie : comprend trois domaines séparés d'alimentation (High-performance, périphériques de communication et temporisateurs, réinitialisation/commande d'horloge/gestion de l'énergie).
  - Une alimentation USB dédiée intégrant un régulateur interne de 3,3V.
  - Un régulateur intégré (LDO) avec sortie configurable et évolutive pour alimenter les circuits numériques.
  - Référence de tension pour périphérique analogique/ V_REF+ .
  - Plusieurs modes de fonctionnement à faible consommation tel que le mode veille, Arrêt, Standby et V_BAT pour la charge de batterie.
- Faible consommation d'énergie : dispose des broches de surveillance de l'état d'alimentation de l'unité centrale et du domaine.
- Il prend en charge l'accélérateur graphique ART pour réduire la charge sur le CPU ce qui permet une exécution rapide du code.

Les mode de fonctionnement du contrôleur MCP255

- Le mode High-Speed : dans ce mode les pilotes de sortie de l'émetteur ont des temps de montée et de descente rapide ce qui résulte une transmission rapide de données en supportant les vitesses élevées de bus CAN, en reliant la broche R_S avec V_{SS}, ce mode est sélectionné.
- Le mode Slope-Control : Ce mode est sélectionné en connectant une résistance externe entre RS et la masse, ceci est pour limiter les temps de montée et de descente de CANH et CANL dans le but de réduire les interférences électromagnétiques.
- Le mode Standby : En mode Standby, l'émetteur est éteint et le récepteur fonctionne avec un courant très faible, la broche RXD fonctionne aussi mais avec une vitesse très faible également [19]. Les Figures B.2 représente le schéma block du contrôleur MCP2551.

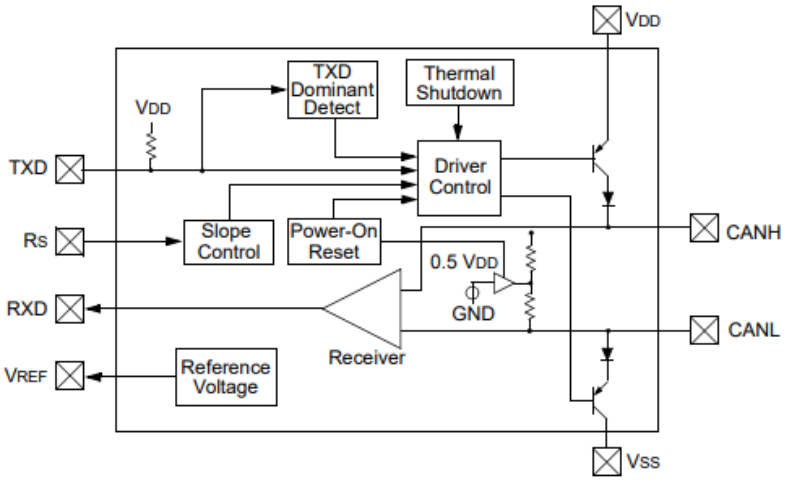


FIGURE B.2 : Schéma bloc du contrôleur MCP2551 [19]

B.3 Caractéristiques de l'afficheur LCD

Le tableau B.2 ci-dessous présente plus de détails sur ce module :

Tableau B.2 : Caractéristiques de l'afficheur LCD [19]

Couleur d'affichage	Couler RGB 65K
SKU	MSP3521
Taille d'écran	3.5 inch
Type	TFT
Circuit de commande	ILI9488
Résolution	480*320 Pixel
Interface du module	SPI avec 4 fils
Zone active	48.96x73.44 mm
Taille de PCB	56.34x98 mm
Angle de vue	>60 °
Température de fonctionnement	-20 °C 70 °C
Température de stockage	-30 °C 80 °C
Voltage de fonctionnement	90mA
Poids du produit	45g

Pour pouvoir ajouter une meilleur illustration du basculement de type de protocole dans la carte d'émission, nous avons décidé d'ajouter un écran LCD pour afficher l'état et le débit de la transmission en temps réel, pour faire cela cette écran sera relie à la carte a travers un module I2C standard qui est le «PCF8574».

On commence par le LCD, qui est l'abréviation du terme anglais "Liquid Crystal Display" qui signifie en français « Écran à cristaux liquides » d'où afficheur LCD.

L'afficheur LCD est en particulier une interface visuelle entre un système et l'utilisateur, son rôle est de transmettre les informations utiles, il affichera donc des données susceptibles d'être exploiter par l'utilisateur d'un système, comme dans notre cas le débite et le type de protocole en émission.

Le nombre de caractère affichable par ligne pour ce modèle est de 16 caractères sur deux lignes soit au total 32 caractères, on retrouve cette information dans la datasheet sous la forme 16 X 02 et aussi dans la référence 1602 A.

Le tableau ci-dessous représente bien le rôle de chaque broche et définit le brochage de l'afficheur LCD.

TABLEAU B.3 : Le rôle des broche de l'afficheur LCD..

Non	Symbole	Level	Fonction
1	V_{ss}	–	0v
2	V_{DD}	–	+5V
3	V0	–	for LCD
4	R_S	H/L	Selection de registre
5	R/W	H/L	H-Read L-Write
6	E	H.H-L	Activer le signale
7	DB0	H/L	Bus de données utiliser pour le transfert de 8 bits
8	DB1	H/L	
9	DB2	H/L	
10	DB3	H/L	
11	DB4	H/L	Bus de données utiliser pour le transfert de 4 et 8 bits
12	DB5	H/L	
13	DB6	H/L	
14	DB7	H/L	
15	BLA	–	BLCKLIGHT +5V
16	BLK	–	BLACKLIGHT 0V

Pour relier cette afficheur à la carte et diminuer le nombre de de fils à utiliser, nous allons le ratacher à un module, et ce module est le PCF8574. Le module PCF8574 est un circuit intégré utilisé pour étendre le nombre de broches d'E/S (Entrée/Sortie) d'un microcontrôleur ou d'un système embarqué. Il est couramment utilisé dans les projets électroniques pour ajouter des broches supplémentaires à un microcontrôleur limité en nombre de ports d'E/S.

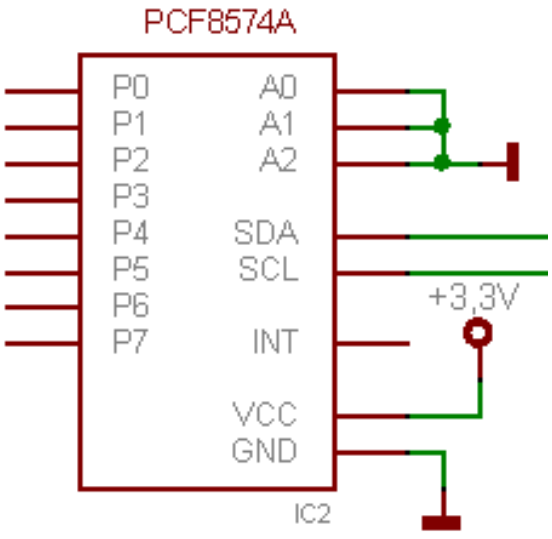


FIGURE B.3 : Le module PCF8574.

Ce module est un expander de bus bidirectionnel qui peut contrôler jusqu'à 8 broches d'E/S comme le montre la Figure B.3. Il communique avec le microcontrôleur principal via un bus I2C (Inter-Integrated Circuit), ce qui permet de transférer des données en série en utilisant seulement deux fils de connexion.

Son utilisation simplifie la conception matérielle, car il permet de réduire le nombre de fils nécessaires pour connecter les composants externes au microcontrôleur. De plus, il est compatible avec de nombreux microcontrôleurs populaires et dispose de bibliothèques logicielles qui facilitent son intégration dans les projets.

Les configurations du bus I2C utilisé sur la carte sont comme démontre dans la Figure suivante :

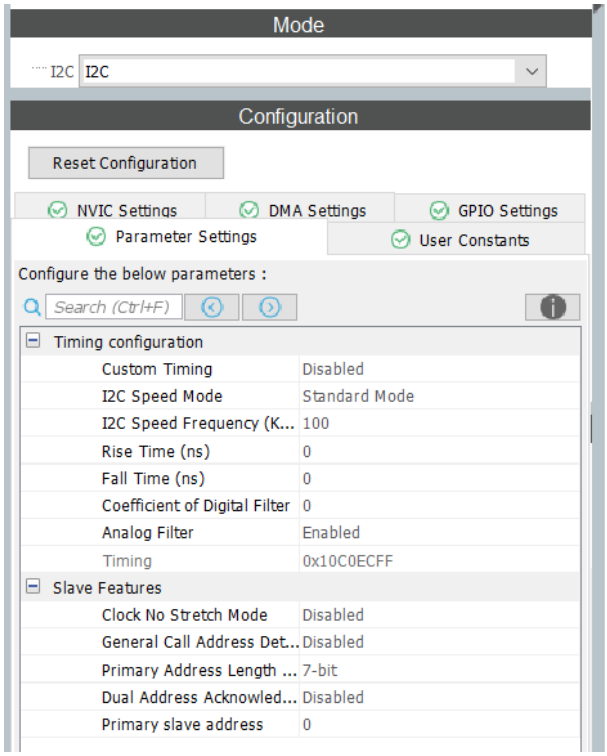


FIGURE B.4 : Configurations du bus I2C .

ملخص

تطورات التكنولوجيا في صناعة السيارات أدت إلى تطوير بروتوكول الذي يوفر سرعات نقل أعلى وسعة متزايدة. و لكن لم تكن المحللات متوافقة مع هذا البروتوكول الجديد، مما استدعى حلاً منا: بوابة مدمجة لتلبية هذه الاحتياجات. توفر هذه البوابة\المحلل واجهة سهلة الاستخدام للمهندسين لإجراء تحليل شامل للبيانات، وتقدم ميزات متقدمة لتفسير رسائل وتحويلها إلى قيم مفهومة، مما يمكّن المستخدمين من الحصول على نظرة شاملة عن أداء الشبكة. بالإضافة إلى ذلك، يتيح عرض البيانات في الوقت الحقيقي على شاشة . تلبي حلولنا أيضاً التطورات حيث نأخذ بروتوكول الجديد بعين الاعتبار للاستفادة من مزاياه والحفاظ في الوقت نفسه على التوافق مع الموجود. باختصار، نجح مشروعنا في تلبية الاحتياجات الجديدة والمحددة لصناعة السيارات. كلمات المفتاحية

الكلمات المفتاحية: بروتوكول، البوابة\المحلل، السيارات

Résumé

Le protocole CAN dans l'industrie automobile a été amélioré pour créer le protocole CAN FD, offrant une transmission plus rapide et une capacité accrue. Cependant, les analyseurs existants n'étaient pas compatibles. Nous avons donc conçu une passerelle CAN/USB combinée à un analyseur CAN. Cette solution conviviale permet aux ingénieurs d'analyser les données, d'interpréter les messages CAN et de visualiser en temps réel les performances du réseau. Elle répond également aux évolutions technologiques en offrant les avantages du protocole CAN FD tout en maintenant la compatibilité avec le protocole CAN existant. En conclusion, notre projet répond aux besoins spécifiques de l'industrie automobile.

Mots Clés : CAN/CAN FD protocole, Analyseur, IndustrieAutomobile,Diagnostic,CAN/USB.

Abstract

Technological advancements in the automotive industry have led to the development of the CAN FD protocol, offering higher transmission rates and increased capacity. However, the lack of compatibility between existing analyzers and this protocol posed a challenge. To tackle this issue, we developed a combined CAN/USB gateway with a CAN analyzer. This innovative enables engineers and technicians to conduct comprehensive data analysis, interpret CAN messages, and convert them into easily understandable values. Equipped with a real-time LCD screen display and it considering the CAN FD protocol, our project ensures compatibility and improved performance. Through successful implementation, our solution effectively meets the industry's specific needs, providing engineers and technicians with a powerful tool for analyzing and troubleshooting CAN FD networks.

Keywords: CAN/CAN FD protocol, Analyzer, Automotive Industry, Diagnostic, CAN/USB.