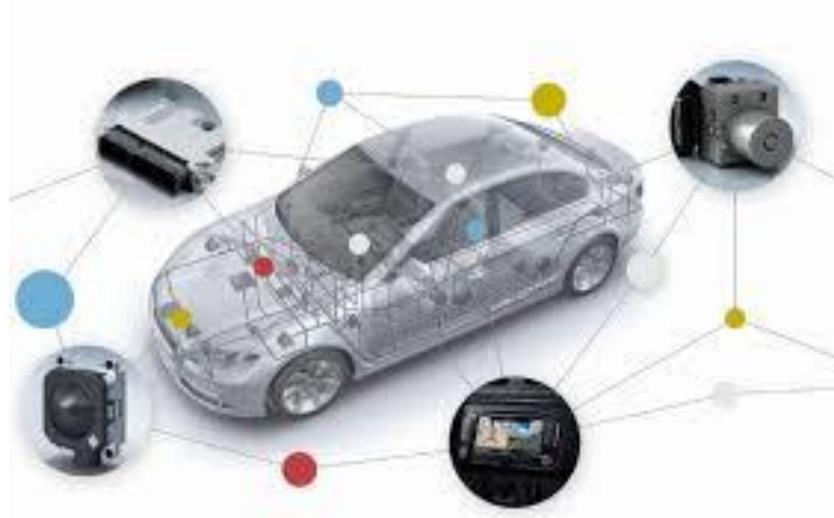


Université Toulouse III - Paul Sabatier
Faculté des Sciences et de l'Ingénierie
Département Electronique, Energie électrique, Automatique

Rapport de projet :

Implémentation pratique et programmation d'un contrôleur BUS CAN pour les systèmes embarqués dans l'automobile



Réalisé par :

- EL-HIHI Hamza
- EL BOUNI Ilyas

Encadré par :

- Mr. PERISSE Thierry

REMERCIEMENTS

On tient à exprimer notre profonde gratitude à toutes les personnes qui ont contribué à la réalisation de ce projet.

En premier lieu, on remercie vivement Monsieur **Thierry Perisse**, professeur à l'Université Toulouse III Paul Sabatier, pour son encadrement, sa disponibilité et ses conseils tout au long de cette expérience.

Un remerciement particulier va à Franque, technicien à l'Université Toulouse III Paul Sabatier, pour son soutien et pour avoir contribué à créer une atmosphère agréable et motivante tout au long de ce projet

Enfin, nous sommes également reconnaissants envers nos familles, nos proches et nos amis, pour leur amour, leur soutien inconditionnel et leurs encouragements constants. Leur présence et leur soutien moral ont été d'une importance capitale tout au long de cette période, et nous leurs sommes infiniment reconnaissantes.

INTRODUCTION

Ce rapport présente les travaux réalisés dans le cadre de notre projet sur le Bus CAN, un élément essentiel dans les systèmes de communication embarqués. Ce projet s'inscrit dans notre formation en Licence Professionnelle en Conception et Commande Numérique des Systèmes Électriques Embarqués – Gestion de l’Énergie Informatique Industrielle à l’Université Toulouse III Paul Sabatier.

L’objectif de ce travail est d’explorer en profondeur les principes du Bus CAN, ses différentes applications et son implémentation dans un système embarqué. Nous avons ainsi étudié les fondamentaux du protocole CAN, son architecture, ses types de trames et ses mécanismes de communication. Par la suite, nous avons mis en pratique ces connaissances en développant un code pour contrôler un système de communication basé sur le CAN, intégrant la carte EID210 et le Véhicule Multiplexé Didactique (VMD), ainsi qu’un système embarqué basé sur Arduino.

Ce projet nous a permis d’approfondir notre compréhension des réseaux embarqués et de renforcer nos compétences en programmation et en configuration de systèmes de communication industrielle.

Ce rapport est structuré comme suit : dans un premier temps, nous présentons les bases théoriques du Bus CAN, son historique et ses applications. Ensuite, nous détaillons l’architecture matérielle et logicielle du système mis en place, en abordant à la fois la communication via la carte EID210 et le Véhicule Multiplexé Didactique (VMD), ainsi que le développement d’un système embarqué basé sur Arduino. Nous exposons ensuite les résultats obtenus et les tests effectués. Enfin, nous concluons en revenant sur les enseignements tirés de cette expérience et les perspectives d’amélioration possibles.

Table des matières

REMERCIEMENTS	2
INTRODUCTION	3
<i>Chapitre I : Fondements du Bus CAN.....</i>	6
1. Définition du BUS CAN :	7
2. Historique et Évolution du Bus CAN :	7
3. Applications et Domaines d'Utilisation du Bus CAN :.....	8
4. Analyse des Trames CAN :	9
4.1. Types de messages CAN :	9
4.2. Formats d'une trame :.....	10
4.3. Fonctionnement des Trames CAN :	11
5. Les Différents Types de Bus CAN :.....	12
6. Structure et Architecture du Bus CAN :	13
<i>Chapitre II : Intégration du Bus CAN dans le Véhicule Multiplexé Didactique (VMD)</i>	15
1. Introduction :	16
2. Le Véhicule Multiplexé Didactique (VMD) :	17
3. Module CAN01A :	17
3.1. Présentation :.....	17
3.2 Les cartes d'interface CAN01A :.....	18
3.3 Cartes 8 entrées logiques :	19
3.4 Cartes 4 sorties de puissance :.....	20
4. DESCRIPTION DU MCP25050 :	20
5. Gestion des échanges entre la carte EID210 000 et le modèle CAN01A ou le VMD :.....	23
5.1. Carte EID210 000 :	23
5.2. Carte ATON CAN :.....	25
6. Analyse et implémentation du protocole CAN pour le contrôle des modules via MCP25050 :.....	27
6.1. Introduction :	27
6.2. Remplissage des principaux champs des trames CAN :	27
6.3. Exemple de trame du TP1 de VMD :	32
7. Installation du Logiciel EID210 :	33
<i>Chapitre III : Conception et Implémentation du Système Embarqué.....</i>	47

1. Introduction :	48
2. Matériel :	48
3. Explication de Matériel :.....	49
4. Conception et Fonctionnement :	58
5. Communication Arduino avec le Shield CAN :.....	59
6. Fonctionnement Réel du Système :.....	59
7. Architecture du Système :.....	60
8. Partie réalisation :.....	61
8.1. Commande du Moteur à Courant Continu :	61
8.2. Réalisation :.....	62
9. Explication des programmes :.....	66
9.1. Explication de l'utilisation de la fonction millis() dans nos codes :	66
9.2. Programme de première Nœud :.....	68
9.3. Explication de programme de première Nœud :	69
9.4. Programme de deuxième Nœud :	71
9.5. Explication de programme de deuxième Nœud :	72
9.6. Programme de troisième Nœud :	75
9.7. Explication de programme de troisième Nœud :	77
10. Visualisation des Trames sur le Réseau CAN :	79
11. Conclusion :.....	82
CONCLUSION GENERALE	83
Sources d'informations et Références techniques :	84

Chapitre I :
Fondements du Bus CAN

1. Définition du BUS CAN :

Le **Bus CAN** (**C**ontroller **A**rea **N**etwork) est un protocole de communication série conçu pour les systèmes embarqués, permettant l'échange de données entre plusieurs microcontrôleurs et périphériques sans nécessiter un contrôleur central. Il repose sur une architecture **multi-maître**, où chaque nœud peut transmettre et recevoir des messages de manière autonome. Les communications s'effectuent via des **trames** hiérarchisées selon un mécanisme de priorité, garantissant une transmission fiable et optimisée des informations.



2. Historique et Évolution du Bus CAN :

Le Controller Area Network (CAN) est un protocole de communication série développé initialement pour l'industrie automobile afin de réduire la complexité et le coût du câblage. Depuis sa création, le bus CAN a évolué pour répondre aux besoins croissants de communication rapide et fiable entre les dispositifs électroniques, s'imposant comme un standard incontournable dans de nombreuses industries.

Historique et Évolution du Bus CAN :

- **1983** : Début du développement du bus CAN par la société allemande Robert Bosch GmbH.
- **Février 1986** : Présentation officielle du protocole CAN lors du congrès de la Society of Automotive Engineers (SAE) à Détroit, Michigan, États-Unis.
- **1987** : Production des premiers circuits intégrés contrôleurs CAN par Intel (modèle 82526) et Philips (modèle 82C200).
- **1991** : Introduction du bus CAN dans le secteur automobile avec la Mercedes-Benz Classe S (W140), premier véhicule de série équipé d'un système de câblage multiplexé basé sur le CAN.
- **1993** : Publication de la norme internationale ISO 11898, standardisant le protocole CAN.
- **2012** : Bosch introduit le CAN FD (Flexible Data-Rate), une évolution du protocole permettant des taux de transfert de données plus élevés et une taille de trame accrue.

Aujourd'hui, le bus CAN est largement utilisé dans divers secteurs tels que l'automobile, l'aéronautique, l'automatisation industrielle et les équipements médicaux, témoignant de son adaptabilité et de son efficacité en tant que protocole de communication.

3. Applications et Domaines d'Utilisation du Bus CAN :

Le **bus CAN (Controller Area Network)** est un protocole de communication robuste et efficace, largement adopté dans de nombreux domaines industriels. Parmi ces domaines :



1. Secteur Automobile :

Le bus CAN est principalement utilisé dans les véhicules modernes pour la communication entre différents composants électroniques sans recourir à un câblage complexe. Il permet la transmission rapide et fiable des informations entre les capteurs, les actionneurs et les unités de commande électronique (ECU).

2. Aéronautique et Aérospatiale :

Dans le domaine aéronautique, le CAN est utilisé pour la gestion des commandes de vol et la surveillance des systèmes. Son utilisation dans les avions et les drones permet une réduction du poids du câblage tout en garantissant une communication robuste et sécurisée, essentielle dans des environnements critiques.

3. Automatisation Industrielle :

Les systèmes d'automatisation et de contrôle des machines industrielles exploitent le bus CAN pour assurer une communication efficace entre les variateurs de vitesse, les automates programmables, les capteurs et les actionneurs. Il est couramment utilisé dans les robots industriels, les lignes de production automatisées et les machines-outils pour améliorer la précision et la coordination des processus de fabrication.

4. Équipements Médicaux :

Dans le secteur médical, le CAN est intégré aux équipements de diagnostic et de surveillance, tels que les appareils d'imagerie médicale, les pompes à perfusion et les respirateurs. Sa fiabilité et sa faible latence en font une solution idéale pour garantir la transmission rapide et sécurisée des données vitales des patients.

5. Domotique et Bâtiments Intelligents :

Le CAN est également présent dans les systèmes de gestion des bâtiments intelligents, permettant le contrôle et la surveillance de l'éclairage, du chauffage, de la climatisation et des dispositifs de sécurité. Son utilisation favorise l'optimisation énergétique et l'amélioration du confort des utilisateurs.

6. Robotique et Systèmes Embarqués :

Les robots mobiles et les systèmes embarqués autonomes bénéficient du bus CAN pour échanger rapidement des données entre les modules de contrôle. Il est particulièrement utile dans les

applications nécessitant une coordination rapide des mouvements et une gestion optimisée des ressources énergétiques.

4. Analyse des Trames CAN :

Les trames CAN (Controller Area Network) sont des unités de transmission utilisées pour échanger des données entre les différents nœuds d'un réseau CAN. Chaque trame contient un certain nombre de champs qui sont utilisés pour assurer la communication, la vérification des erreurs et la gestion de la priorité des messages.

Il existe deux types de trames : **trame standard** et **trame étendue**.

- **Trame standard** : Elle utilise un identifiant de **11 bits**, ce qui permet de gérer jusqu'à 2048 identifiants différents.
- **Trame étendue** : Elle utilise un identifiant de **29 bits**, permettant ainsi une plus grande capacité d'adressage, avec jusqu'à 536 870 912 identifiants possibles.

4.1. Types de messages CAN :

Il existe plusieurs types de messages CAN sous différentes catégories de trames dans le protocole CAN :

1. **Trame de données** (Data Frame) :

- Utilisée pour transmettre des données.

2. **Trame de demande** (Remote Frame) :

- Permet à un nœud de demander des données d'un autre nœud.

3. **Trame d'erreur** (Error Frame) :

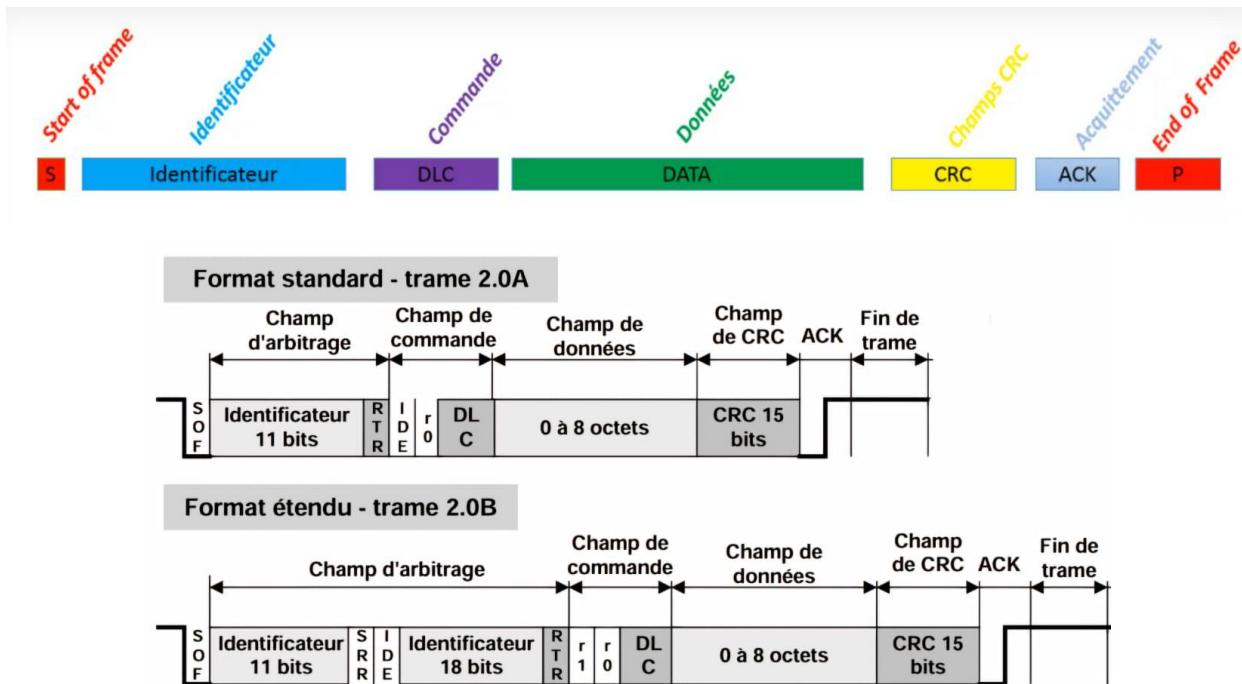
- Envoyée lorsqu'une erreur est détectée par un nœud.

4. **Trame de surcharge** (Overload Frame) :

- Utilisée pour gérer les cas où un nœud ne peut pas recevoir de nouvelles données immédiatement.

4.2. Formats d'une trame :

Une trame CAN se compose de plusieurs segments ou champs, chacun ayant une fonction spécifique :



1. Start of Frame (SOF) :

- Un seul bit qui marque le début d'une trame. Il indique que la trame commence, et il est toujours à **0**.

2. Champ d'arbitrage :

- L'identifiant est un champ de **11 bits** pour les trames standard ou **29 bits** pour les trames étendues (ce qui permet de gérer une plus grande variété de messages). Le **bit RTR** (Remote Transmission Request) détermine s'il s'agit d'une trame de données (**0**) ou d'une trame de demande de message (**1**).
- Le **bit IDE** est établi la distinction entre format standard et format étendu il se trouve dans le champ **d'arbitrage** si la trame est **étendu (1)** et dans le champ de **commande** si la trame est **standard (0)**.
- L'identifiant est important pour déterminer la priorité de la trame. Plus l'identifiant est bas, plus la priorité est élevée (en cas de conflit de bus, la trame avec l'identifiant le plus bas sera transmise).

3. Champ de Commande :

- Ce champ comprend des informations sur la longueur des données.

- les **bits r** sont des bits réservés pour une utilisation future, et ne sont pas utilisés.
- **DLC** est codé sur quatre bits ce permette d'envoyer jusqu'à 8 octet de données.**Champ de Données :**
- C'est la zone où les données sont transmises. Elle peut contenir entre **0 et 8 octets** de données.
- Ce champ peut être vide si la trame est une demande de données.

4. CRC (Cyclic Redundancy Check) :

- Ce champ contient un code de redondance cyclique de **15 bits** qui permet de vérifier l'intégrité des données reçues.
- Lors de la réception d'une trame, le récepteur effectue un calcul CRC et compare le résultat avec celui contenu dans la trame. Si les deux résultats correspondent, cela signifie que la trame est correcte.

5. ACK (Acknowledge) :

- Ce champ contient un seul bit qui est utilisé pour confirmer la réception correcte de la trame.
- Le récepteur envoie un signal ACK pour indiquer qu'il a bien reçu la trame. Si aucune confirmation n'est reçue, la trame est renvoyée.

6. End of Frame (EOF) :

- La trame se termine par un champ de 7 bits à **1**. Cela marque la fin de la trame et permet au récepteur de savoir qu'il n'y a plus de données à venir.

4.3. Fonctionnement des Trames CAN :

Le fonctionnement du Bus CAN repose sur le principe de communication de type différentiel et asynchrone :

- **Transmission différentielle** : CAN utilise deux fils pour transmettre les signaux, généralement appelés **CAN High** (CAN_H) et **CAN Low** (CAN_L). La différence de potentiel entre ces deux lignes permet d'éviter les interférences et d'assurer une communication stable.
- **Arbitrage basé sur la priorité** : Le Bus CAN est un bus **multi-maître**, ce qui signifie que n'importe quel nœud peut initier une transmission. Cependant, si plusieurs nœuds tentent d'envoyer un message en même temps, l'arbitrage entre les messages est basé sur la priorité de l'identifiant. Si un nœud transmet une trame avec un identifiant plus faible, il a la priorité sur les autres.

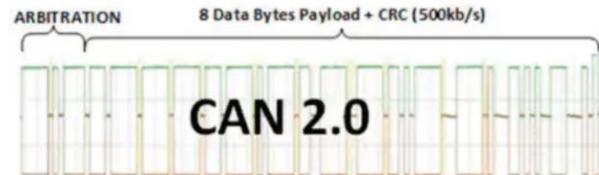
- **Contrôle des erreurs** : Chaque trame CAN comprend plusieurs mécanismes de détection d'erreurs (comme le CRC et l'ACK). En cas d'erreur détectée, la trame est immédiatement renvoyée.
- **Communication asynchrone** : Le Bus CAN ne nécessite pas de synchronisation temporelle entre les nœuds. Chaque périphérique émet sa trame quand il le souhaite, et les nœuds récepteurs se synchronisent grâce aux informations contenues dans la trame.

5. Les Différents Types de Bus CAN :

Le bus CAN a évolué pour répondre aux exigences de diverses applications industrielles, ce qui a conduit au développement de plusieurs variantes adaptées à des besoins spécifiques. Chaque type de bus CAN se distingue par ses caractéristiques en termes de vitesse de transmission, de capacité de gestion des erreurs et d'adaptabilité aux environnements contraignants. Ces différentes versions du protocole permettent d'assurer une communication efficace et fiable, que ce soit dans l'automobile, l'aéronautique, l'industrie ou encore les systèmes embarqués. Les principales versions du bus CAN sont :

1. CAN Standard (ou CAN 2.0) :

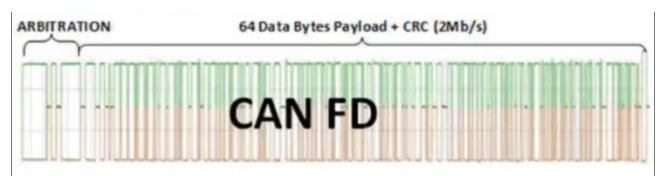
Le CAN standard, aussi appelé **CAN 2.0**, est la première version du protocole, définie par la norme **ISO 11898**. Il est divisé en deux sous-catégories :



- **CAN 2.0A** : Utilise une trame d'identification sur 11 bits (format standard).
- **CAN 2.0B** : Introduit une trame d'identification étendue sur 29 bits (format étendu), permettant plus de possibilités d'adressage.

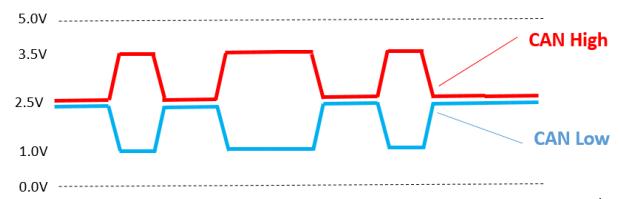
2. CAN FD (Flexible Data-Rate) :

Le **CAN FD**, introduit en **2012 par Bosch**, améliore la version standard en augmentant la vitesse de transmission et la taille des trames de données. Il permet un débit plus rapide allant jusqu'à **8 Mbit/s** (contre 1 Mbit/s pour le CAN 2.0) et peut transporter jusqu'à **64 octets de données** par trame (contre 8 octets pour le CAN classique).



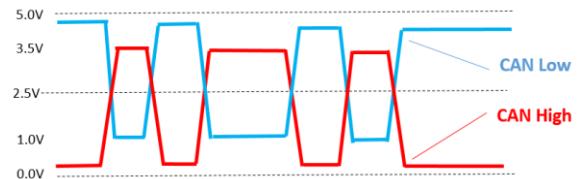
3. CAN High-Speed (ISO 11898-2) :

Le **CAN High-Speed** est utilisé pour les systèmes nécessitant une communication rapide et robuste. Il fonctionne à des débits pouvant atteindre **1 Mbit/s** et est couramment employé dans les réseaux automobiles et industriels.



4. CAN Low-Speed / Fault-Tolerant (ISO 11898-3) :

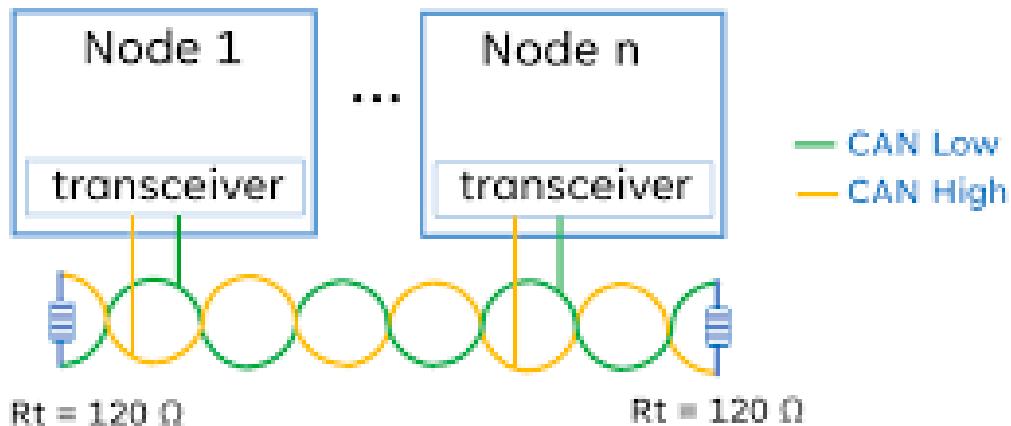
Le CAN Low-Speed est conçu pour des applications où la tolérance aux pannes est essentielle. Il fonctionne à des vitesses plus faibles (125 kbit/s max) et permet au système de continuer à fonctionner même en cas de défaillance d'un nœud ou d'une ligne de transmission.



Comparaison des Différents Types de Bus CAN :

	CAN Low Speed	CAN Hhigh Speed	CAN FD
Débit	125 kb/s	De 125 kb/s à 1 Mb/s	jusqu'à 8Mb/s
Tension d'alimentation	5 Volt	5 Volt	5 Volt
Niveau dominant	CANH = 4V CANL = 1V	CANH = 3.5V CANL = 1.5V	CANH = 3.5V CANL = 1.5V
Niveau récessif	CAN H = 1.75V CAN L = 3.25V	CAN H = CAN L = 2.5V	CAN H= 2.5V CAN L = 1.5V
Caractéristique de cable	30 pF entre CAN H et CAN L	Résistance de 1200 ohms aux deux extrémités du bus entre CAN H et CAN L (=60Q)	Résistance de 1200 ohms aux deux extrémités du bus entre CAN H et CAN L (=60Q)
nombre de noeuds	2 à 20	2 à 30	2 à 30

6. Structure et Architecture du Bus CAN :



Le bus CAN est une **architecture en ligne** où plusieurs nœuds sont connectés à une seule paire de fils torsadés, chaque nœud contient un **transceiver**, qui convertit les signaux du contrôleur CAN en niveaux de tension compatibles avec le bus.

Le bus utilise deux lignes de signalisation :

- CAN High (Jaune sur le schéma).

- CAN Low (Vert sur le schéma).

Ces signaux sont différentiés pour améliorer l'immunité aux interférences électromagnétiques.

À chaque extrémité du bus, une **résistance de 120Ω** est placée pour empêcher les réflexions de signal et assurer une bonne intégrité des données.

Tous les nœuds partagent le même bus et communiquent en **broadcast** (diffusion).

Chapitre II :

Intégration du Bus CAN dans le Véhicule Multiplexé Didactique (VMD)

1. Introduction :

L'objectif de ce projet est de prendre en main et commander le modèle CAN01A du Véhicule Multiplexé Didactique (VMD) à l'aide de la carte EID210 000, en utilisant la carte ATON CAN comme interface de communication. Cette approche permet d'explorer en profondeur le protocole de communication CAN (Controller Area Network) et ses applications dans les systèmes embarqués.

Pour assurer la communication entre ces modules, des **trames CAN** spécifiques sont envoyées afin de configurer et contrôler les différents registres du CAN01A. Ces trames permettent notamment de :

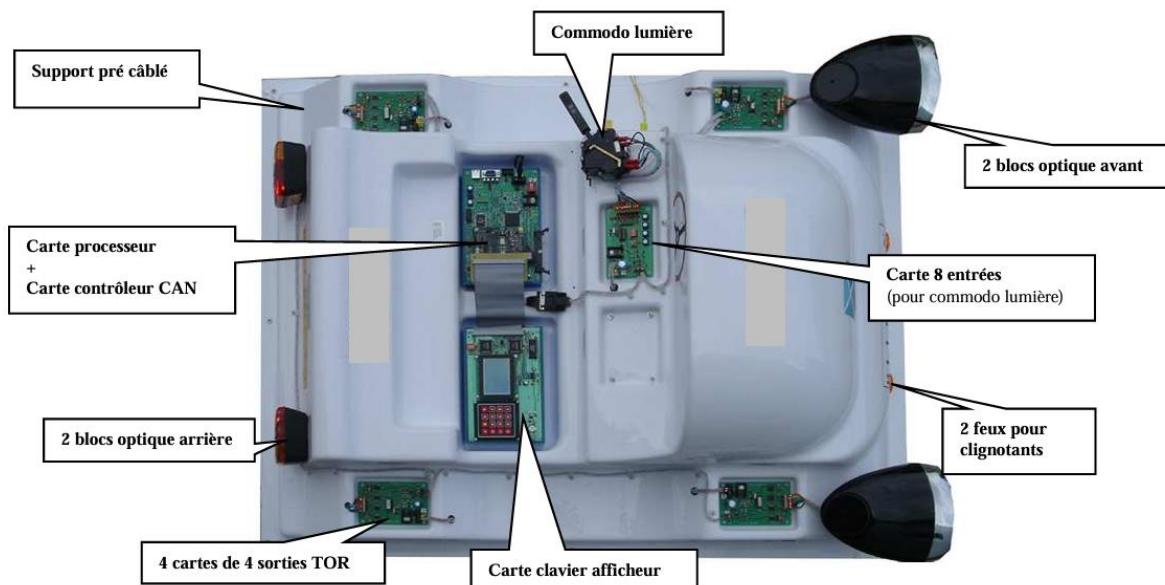
- Gérer les entrées et sorties des cartes.
- Superviser l'état des périphériques connectés (capteurs, actionneurs).
- Garantir un échange de données fiable et en temps réel entre les différentes cartes du système.

Grâce à l'utilisation du bus CAN, ce projet offre une meilleure compréhension des principes du multiplexage, des protocoles de communication embarqués, et des techniques de contrôle des systèmes automatisés. Il constitue ainsi une base solide pour l'étude et l'implémentation des réseaux embarqués dans l'automobile et l'industrie.

2. Le Véhicule Multiplexé Didactique (VMD) :

Le Véhicule Multiplexé Didactique (VMD) est un système didactique conçu pour l'étude des Réseaux Locaux Industriels (RLI), notamment le bus CAN. Il permet d'aborder plusieurs concepts clés liés aux réseaux de communication, comme le modèle OSI, les protocoles de communication, les interfaces électriques et électroniques, ainsi que la structuration des programmes mettant en œuvre ces technologies.

Grâce à l'intégration du **bus CAN**, le VMD permet de simuler un **système automobile multiplexé**, offrant ainsi un environnement idéal pour comprendre et expérimenter les technologies de communication utilisées dans le domaine des véhicules modernes et des systèmes industriels.



Le VMD est composé de plusieurs sous-systèmes, dont les modules **CAN01A** et **CAN01B**, qui servent de plateformes pédagogiques pour comprendre le fonctionnement et l'implémentation du bus CAN dans un environnement industriel ou automobile. Dans ce projet, nous nous sommes basés spécifiquement sur le module **CAN01A**, qui joue un rôle central dans la communication entre les différents modules d'entrées et de sorties du VMD.

3. Module CAN01A :

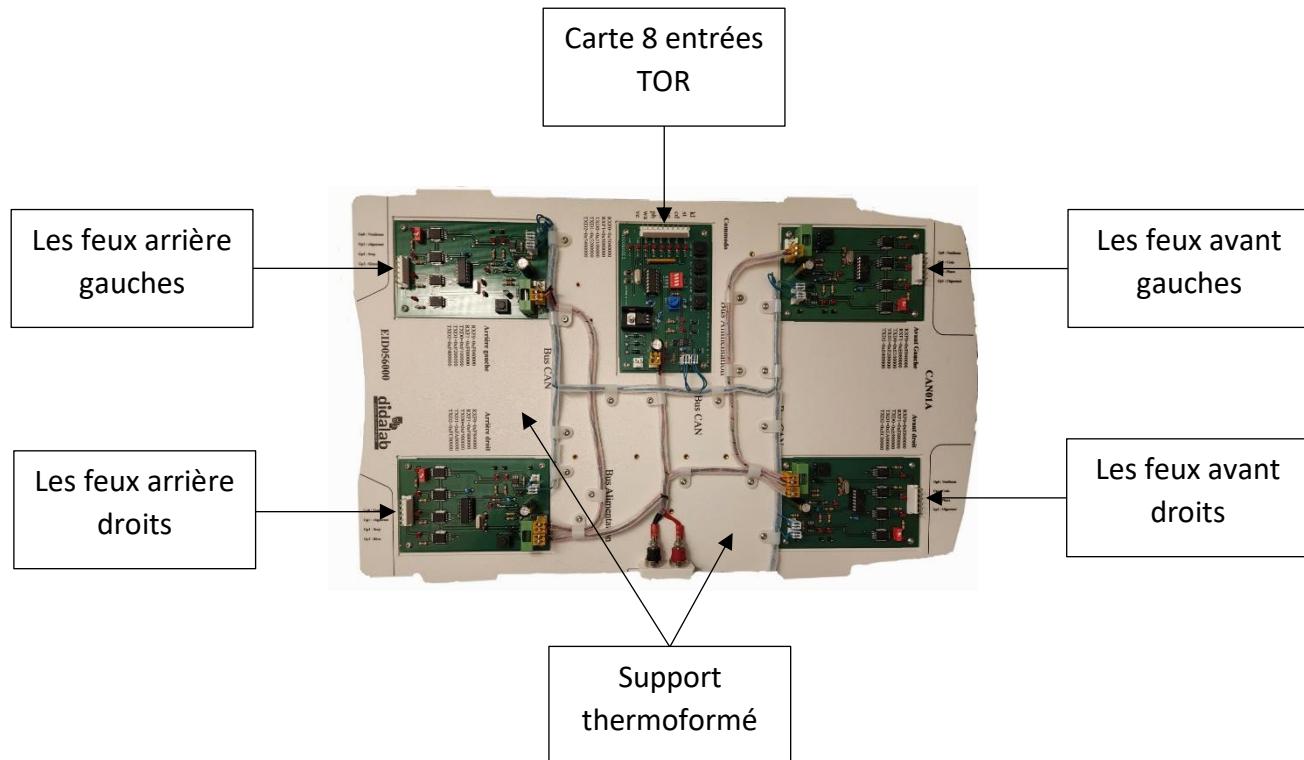
3.1. Présentation :

Le module CAN01A est un sous-système du VMD qui joue un rôle essentiel dans l'étude et l'expérimentation du bus CAN.

Le module CAN01A se compose :

- D'un support thermoformé représentant un véhicule avec ses organes de signalisation.

- D'une carte 8 entrées TOR sur bus CAN gérant le commodo lumière.
- De 4 cartes de sortie TOR, gérant :
 - Les feux avant gauches.
 - Les feux avant droits.
 - Les feux arrière gauches.
 - Les feux arrière droits.



3.2 Les cartes d'interface CAN01A :

Les cartes du module CAN01A permettent l'interfaçage et la communication des différents composants du système via le bus CAN. Ces cartes sont basées sur un circuit intégré **MCP25050**, qui joue un rôle central dans leur fonctionnement. Le MCP25050 est un **expandeur d'entrées/sorties** configurable compatible avec le bus CAN, il agit comme un pont de communication entre les entrées/sorties logiques et le réseau CAN. Grâce à sa programmabilité, il permet d'adapter dynamiquement la configuration des broches :

- Via une trame CAN, il est possible de définir si une liaison est une entrée ou une sortie.
- Il prend en charge jusqu'à **8 lignes d'entrées/sorties** pouvant être reconfigurées selon les besoins du système.

Ce circuit MCP25050 est donc le cœur des cartes et joue un rôle essentiel dans la gestion des échanges de données, permettant une communication fiable et efficace au sein du VMD.

Les cartes du CAN01A possèdent des éléments communs qui assurent leur bon fonctionnement et leur interopérabilité.

Éléments communs des cartes du module CAN01A :

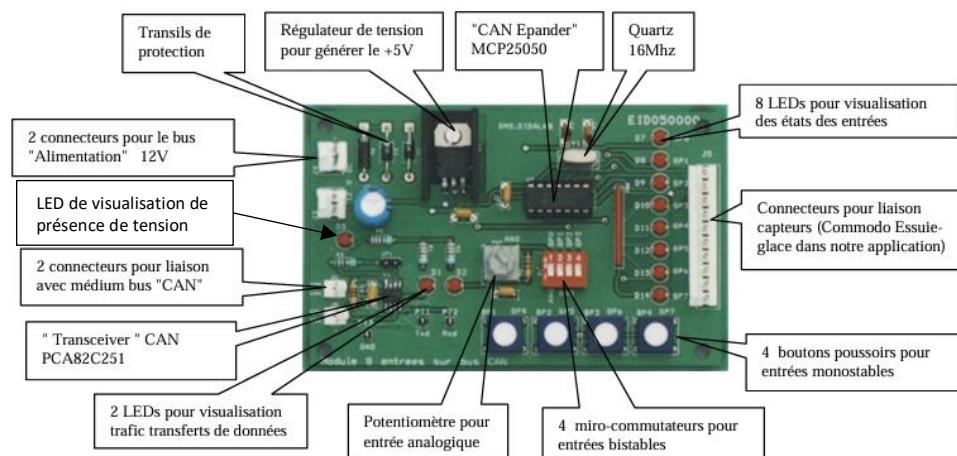
- Deux connecteurs d'alimentation (bus "alimentation énergie").
- Un régulateur de tension générant la tension +5V nécessaire aux circuits intégrés inclus sur le module.
- les protections d'usage.
- Une LED de visualisation de présence tension.
- Une résistance de terminaison de bus connectable ou non par "jumper".
- Un circuit intégré "émetteur récepteur de ligne" (82CA251).
- Un circuit intégré d'extension d'entrées sorties (MPC25050) permettant la communication et l'interprétation des messages.
- un oscillateur à quartz nécessaire au circuit MPC25050.
- Deux LEDs de visualisation de communication en réception et en émission.

3.3 Cartes 8 entrées logiques :

La carte 8 entrées logiques est un module essentiel du CAN01A, permettant l'interfaçage de signaux logiques avec le bus CAN. Elle est utilisée pour la simulation et la gestion des entrées numériques provenant d'interrupteurs, de capteurs ou de boutons poussoirs.

En plus des éléments communs, cette carte comprend :

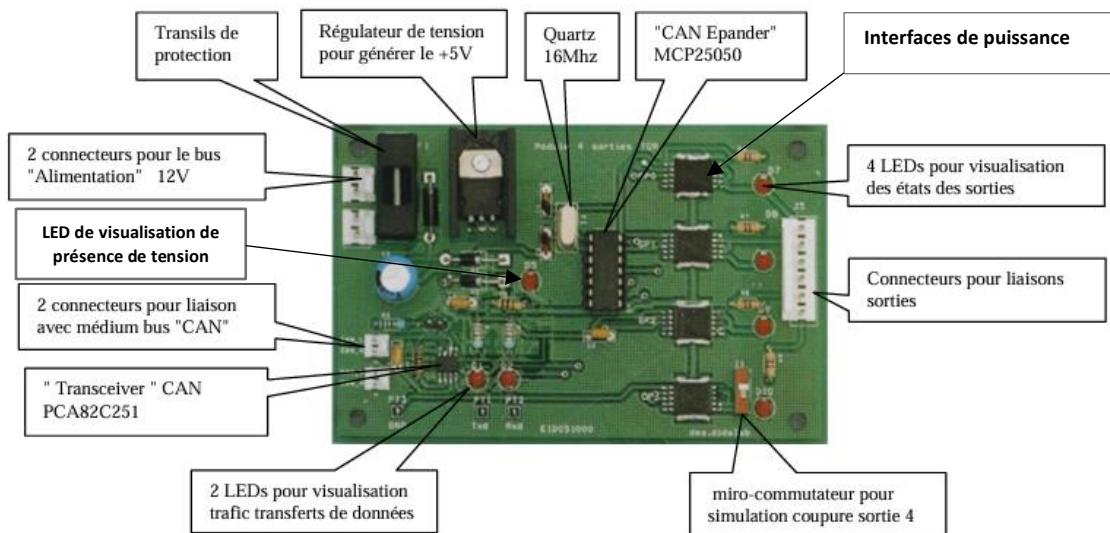
- 4 boutons poussoir.
- 4 commutateurs à 2 positions (fermé ou ouvert).
- 1 connecteur 10 points permettant de relier les capteurs externes de type fin de course.
- 8 LEDs de visualisation des états.
- 1 potentiomètre analogique.



3.4 Cartes 4 sorties de puissance :

Ces cartes sont conçues pour piloter des charges électriques en mode "tout ou rien" sous 12V, ce qui permet d'activer ou de désactiver des dispositifs électriques via le bus CAN. Plusieurs éléments communs comprennent les éléments suivants :

- 4 interfaces de puissance permettant de piloter 4 charges électriques en "tout ou rien", sous 12V (**VN05**).
- 4 LEDs de visualisation des états des sorties puissance.
- 4 entrées de contrôle des charges.
- 1 entrée de simulation de coupure de charge.



Le circuit intégré **VN05**, utilisé pour l'interface de puissance, génère un signal logique permettant de détecter si une charge est bien connectée (en mesurant le courant absorbé).

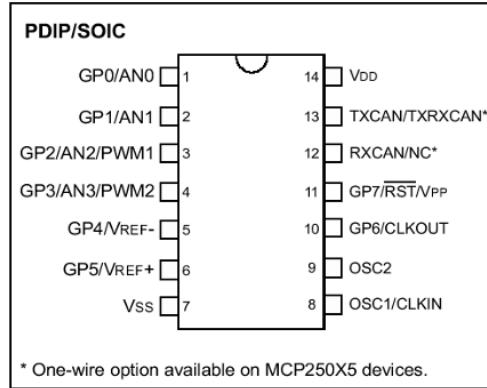
Les circuits de puissance **VN05** peuvent gérer des charges consommant jusqu'à 12 A en continu et sont protégés contre les courts-circuits et les dépassements de température, garantissant ainsi la sécurité et la fiabilité du système.

4. DESCRIPTION DU MCP25050 :

Le MCP25050 est un circuit intégré de 14 broches. Il joue un rôle clé dans l'interface et la communication du module CAN01A avec le bus CAN, permettant l'échange d'informations entre les cartes d'entrée/sortie et les autres systèmes connectés au réseau.

Côté interface CAN : Peut communiquer avec une vitesse de transmission qui peut atteindre 1Mbit/s. Certaines versions du circuit permettent une communication sur 1 fil (One-wire).

Côté application : Il possède 8 lignes d'entrées/sorties (GPO à GP7) configurables individuellement en entrée ou en sortie, seule la ligne GP7 ne peut être utilisée en sortie.



Deux liaisons (GP2 et GP3) peuvent être configurées en sorties modulées (PWM), ces deux sorties peuvent être commandées indépendamment l'une de l'autre. Certaines versions du circuit intègrent un convertisseur analogique numérique (4 voies) sur 10 bits.

Il est capable d'envoyer un message sans qu'il soit interrogé si l'une de ses entrées analogique dépasse des seuils de tension que l'on peut choisir. Il possède un "schéduleur" qui lui permet d'envoyer un message à intervalles de temps réguliers sans qu'on le lui demande (par exemple l'état de ses entrées ou la valeur convertie d'une des entrées analogiques).

C'est un circuit configurable grâce à une banque de registres qui sont gravés dans le circuit lors d'une phase de programmation.

Remarque :

- Ces circuits ne sont pas reprogrammables.
- Le constructeur (MICROCHIP) commercialise des outils logiciels et matériels de configuration et de programmation.
- Un circuit déjà programmé peut être lu par le logiciel de configuration et de programmation.

Le MCP25050 est un émetteur de ports configuré pour gérer des entrées et sorties TOR. Comme illustré dans l'image ci-dessous, il est utilisé pour deux principales cartes : la carte de 8 entrées (Commodo lumière) et la carte de 4 sorties TOR.

➤ **Carte 8 entrées (Commodo lumière) :**

Cette carte est entièrement configurable en mode entrée et permet de récupérer l'état de divers éléments du véhicule, tels que le klaxon, les clignotants, les feux de position (veilleuse), le phare et le warning. Chaque bit du port 8 bits est attribué à une fonctionnalité spécifique afin de transmettre ces informations via le bus CAN.

➤ Carte 4 sorties TOR :

Cette carte combine quatre entrées et quatre sorties. Les sorties contrôlent des éléments tels que les clignotants, le code et le phare, tandis que les entrées permettent de surveiller l'état des feux avant et arrière. Il est important de noter que cette configuration des entrées est fixe et ne peut pas être reprogrammée une fois définie.

6.2.1 Carte 8 entrées (Commodo lumière)

Le port 8 bits du can expander MCP25050 est configurer en entrée TOR.

GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0
E	E	E	E	E	E	E	E

Avec E: entrée TOR.

L'affectation des entrées sur la carte entrée est la suivante :

GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0
klaxon	stop	Clignotant droit	Clignotant gauche	code	phare	warning	Veilleuse

6.2.2 Carte 4 sorties TOR

Le port 8 bit du can expander MCP25050 est configurer :

GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0
E	E	E	E	S	S	S	S

Avec : E: entrée TOR,

S : sortie TOR

6.2.2.1 Feux avant

L'affectation des entrées sur la carte entrée est la suivante :

GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0
Etat clignotant	Etat phare	Etat code	Etat veilleuse	clignotant	phare	code	Veilleuse

6.2.2.2 Feux arrières

L'affectation des entrées sur la carte entrée est la suivante :

GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0
Etat GP3	Etat clignotant	Etat code	Etat veilleuse	(klaxon)	clignotant	code	Veilleuse

5. Gestion des échanges entre la carte EID210 000 et le modèle CAN01A ou le VMD :

Pour commander les cartes d'entrées/sorties du modèle CAN01A ou du VMD, nous utilisons la carte EID210 000 comme carte mère et la carte ATON CAN comme interface de communication. En effet, le microcontrôleur 68332 intégré dans la carte EID210 000 n'est pas capable d'envoyer directement des trames CAN.

La carte ATON CAN joue ainsi un rôle essentiel en assurant la liaison entre la carte EID210 000 et le modèle CAN01A ou le VMD via le bus CAN. Elle est responsable de l'envoi et de la réception des trames, garantissant une communication fluide et efficace entre ces deux éléments.

5.1. Carte EID210 000 :

5.1.1 Fonctions principales :

La carte processeur EID 210 000 est un module d'étude d'un microsystème architecturé autour du microcontrôleur 68332 (de la famille 68000, fabricant Motorola).

Elle dispose d'un certain nombre de périphériques permettant le pilotage, et l'acquisition de données (tout ou rien ou analogiques) à travers un port d'extension.

La carte dispose également d'interfaces de communication série asynchrone et synchrone, d'un bus USB 1.1, et d'un bus d'extension au format "PC104".

5.1.2 Ressources matérielles :

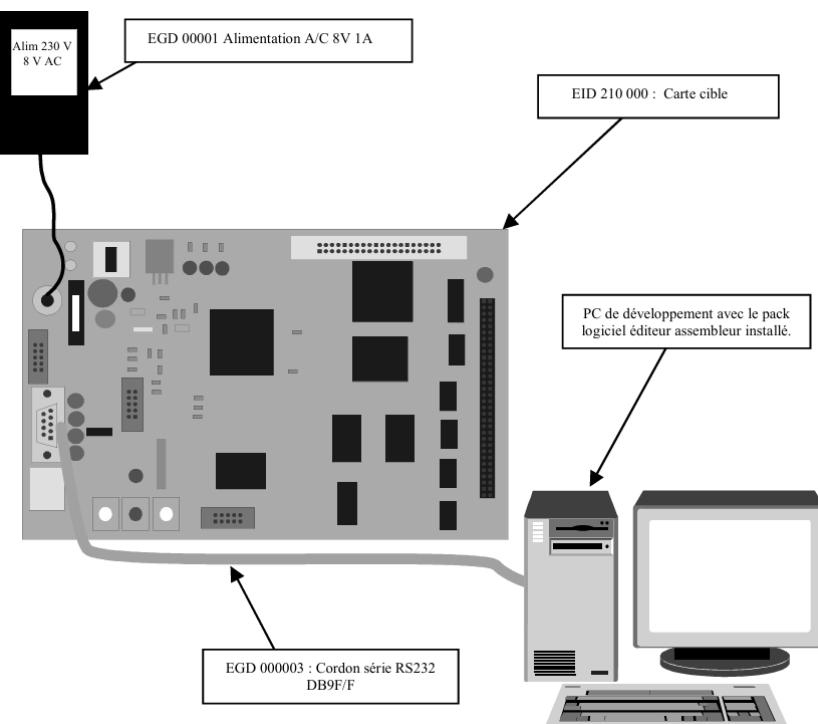
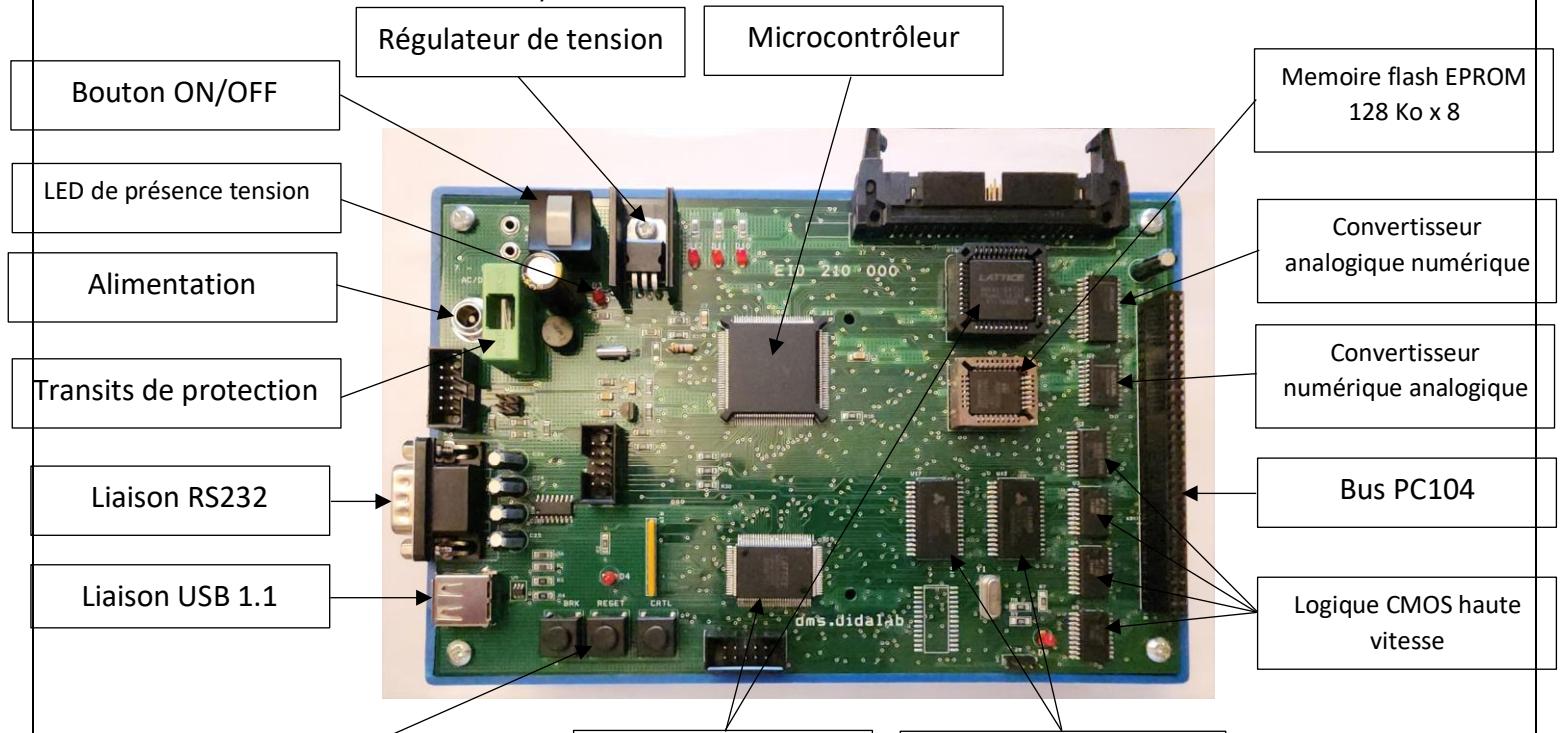
La carte processeur EID210 000 comporte les éléments matériels suivants :

- un microcontrôleur 68332 cadencé à 16,7 MHz.
- 128 Ko x 8 de flash EPROM.
- 128 Ko x 16 de RAM.
- deux réseaux logiques programmables (PLD) permettant:
 - La mise en forme des différents signaux (EPLD de contrôle).
 - D'avoir un port 8 bits bidirectionnel.
- un convertisseur analogique numérique 6 voies, avec 12 bits de résolution.
- un convertisseur numérique analogique 8 bits 4 sorties.
- un bus PC104 8 bits.
- une liaison RS232.
- une liaison USB 1.1.
- une liaison série synchrone de type SPI ou I2C.

5.1.3 INSTALLATION ET MISE EN SERVICE :

Pour installer la carte EID210 000, il faut :

- Relier la liaison RS232 à un port d'un ordinateur de type P.C.
- Relier l'alimenter avec une alimentation 230V/8V AC.
- Appuyer sur le bouton ON/OFF pour mettre le système sous tension (la LED de présence tension doit s'allumer).

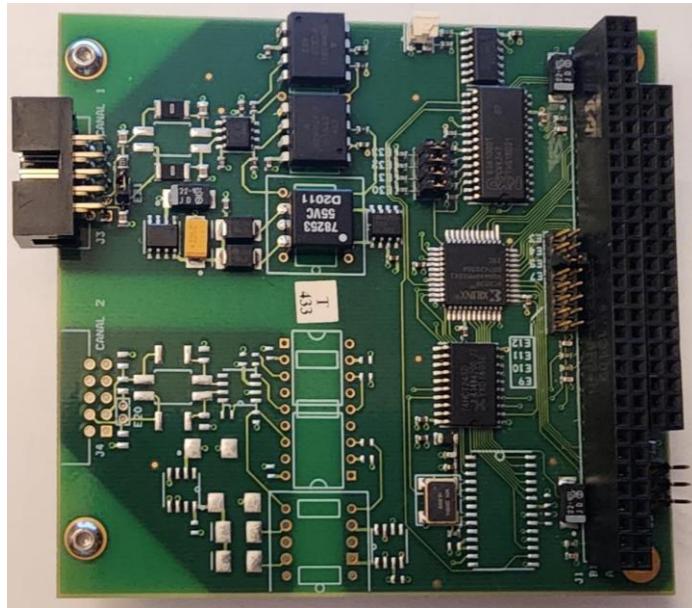


5.2. Carte ATON CAN :

5.2.1. Définition :

La carte **ATON CAN PC/104** est une carte d'interface CAN basée sur le standard PC/104. Elle est conçue pour des applications industrielles ou embarquées nécessitant une communication via le bus CAN.

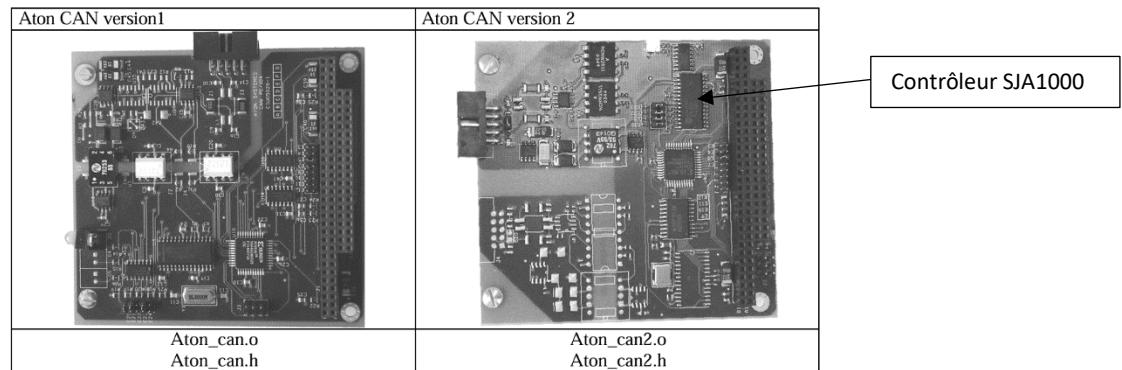
La carte **ATON CAN** assure la communication entre la carte EID210 000 et le modèle CAN01A OU le VMD via le protocole CAN bus. Elle gère l'envoi et la réception des trames entre ces deux composants.



5.2.2. Caractéristiques générales :

1. **Standard PC/104 :**
 - Compact et empilable, idéal pour les systèmes embarqués.
 - Compatible avec les systèmes PC/104.
2. **Protocole CAN :**
 - Supporte généralement le CAN 2.0A (Standard) et CAN 2.0B (Étendu).
 - Vitesses de communication configurables (jusqu'à 1 Mbit/s).
3. **Microcontrôleur ou contrôleur CAN :**
 - Intègre un contrôleur CAN comme le **SJA1000**.
 - Couplée avec un microcontrôleur pour gérer les trames et protocoles.
4. **Connectivité :**
 - Connecteurs pour interfaçer avec les réseaux CAN via des ports DB9.
 - Options pour des communications dual-CAN ou CAN-FD selon le modèle.
 - pour faciliter l'intégration dans des applications spécifiques.
5. **Alimentation :**
 - utilise l'alimentation disponible sur le bus PC/104.

Il existe 2 versions de la carte **ATON CAN**. Il faut sélectionner le fichier de la bibliothèque en fonction de la version de la carte, dans notre cas en utilise la deuxième version alors on ajoutera la bibliothèque «**aton_can2.o**».



Pour configurer la carte **ATON CAN**, il faut lier la bibliothèque «**aton_can.o**» ou «**aton_can2.o**» pour l'initialisation du contrôleur de bus CAN SJA1000 pour la gestion du VMD.

6. Analyse et implémentation du protocole CAN pour le contrôle des modules via MCP25050 :

6.1. Introduction :

Pour contrôler les cartes du **Module CAN01A ou le VMD** via le bus CAN, il faut envoyer des trames CAN à le MCP25050.

Une première trame qui configure les broches en **mode sortie ou en mode entrée** en écrivant dans le registre **GDDR (registre est uniquement responsable de définir l'état des broches en mode entrée ou sortie)**.

Et une deuxième trame qui définit l'état logique des sorties (haut ou bas) en écrivant dans le registre **GPLAT (registre gère l'état des entrées et des sorties)**.

La première étape consiste à configurer les broches en **mode sortie** via GDDR, une fois les broches configurées, en envoi une **deuxième trame** ciblant GPLAT pour définir l'état des sorties (allumer/éteindre les lampes). Pour effectuer cela, il est nécessaire de définir les paramètres de la trame CAN.

6.2. Remplissage des principaux champs des trames CAN :

Pour définir les paramètres de la trame CAN, nous nous basons sur les documentations fournies par **Didalab** concernant le matériel utilisé.

Nous commencerons par **définir l'ID de la trame**, qui permet d'identifier le message et sa priorité sur le bus. Cet **identifiant (ID)** peut être en **format standard (11 bits)** ou **étendu (29 bits)**, selon la configuration du réseau CAN utilisé dans notre projet.

Une fois l'ID déterminé, nous préciserons les autres champs essentiels de la trame, tels que :

- **DLC (Data Length Code)** : indique le nombre d'octets de données dans la trame (de 0 à 8)
- **Données (Data Field)** : contient les informations à transmettre aux modules du système

Les autres paramètres, tels que le **CRC (Cyclic Redundancy Check)**, le **bit ACK** et les champs de contrôle, sont gérés automatiquement par le microcontrôleur.

6.2.1. Remplissage du champ ID :

Chaque carte du modèle **CAN01A** possède un **ID unique** pour chaque type de message transmis ou reçu sur le bus CAN.

Dans notre projet, les messages échangés suivent une structure bien définie, avec des identifiants spécifiques en fonction du type de communication. On distingue plusieurs types de messages, notamment :

- **IM (Input Message)** : Utilisée pour transmettre des données.
- **IRM (Input Request Message)** : Permet à demander des données.
- **OM (Output Message)** : C'est une repense du message IRM.

Chaque ID est propre à un type de message et à une carte spécifique, ce qui permet d'assurer une communication structurée et d'éviter les conflits sur le bus.

Pour trouver les **ID des cartes** du modèle **CAN01A**, nous pouvons nous référer directement sur le modèle CAN01A ou dans le tableau suivant **d'identification des différents modules CAN** (fourni par Didalab) : Ce tableau répertorie les **ID attribués à chaque carte** en fonction du **type de message** échangé.

Registre MCP25025 Et fonction	- SIDH (en bin)	- SIDL (en bin)	SIDH (Hex)	SIDL (Hex)	Identificateur (Hex) (! sur 29 bits)	Labels définis dans fichier CAN_VMD.h
-------------------------------	-----------------	-----------------	------------	------------	--------------------------------------	---------------------------------------

Nœud "Commodo feux"

RXF0 -> IRM et OM	001 0 10 00	001 - 1-xx	28	28	0504 xx xx	T_Ident_IRM_Commodo_Feux
RXF1 -> IM	001 0 10 00	010 - 1-xx	28	48	0508 xx xx	T_Ident_IM_Commodo_Feux
TXD0 -> On Bus	001 0 10 00	100 - 1-xx	28	88	0510 xx xx	
TXD1 -> Acq IM	001 0 10 01	000 - 1-xx	29	08	0520 xx xx	T_Ident_AIM_Commodo_Feux
TXD2 -> Mes. Auto.	001 0 10 10	000 - 1-xx	2A	08	0540 xx xx	

Nœud "Feux avant gauche"

RXF0 -> IRM et OM	011 1 00 00	001 - 1-xx	70	28	0E04 xx xx	T_Ident_IRM_FVG
RXF1 -> IM	011 1 00 00	010 - 1-xx	70	48	0E08 xx xx	T_Ident_IM_FVG
TXD0 -> On Bus	011 1 00 00	100 - 1-xx	70	88	0E10 xx xx	
TXD1 -> Acq IM	011 1 00 01	000 - 1-xx	71	08	0E20 xx xx	T_Ident_AIM_FVG
TXD2 -> Mes. Auto.	011 1 00 10	000 - 1-xx	72	08	0E40 xx xx	

Nœud "Feux avant droit"

RXF0 -> IRM et OM	011 1 01 00	001 - 1-xx	74	28	0E84 xx xx	T_Ident_IRM_FVD
RXF1 -> IM	011 1 01 00	010 - 1-xx	74	48	0E88 xx xx	T_Ident_IM_FVD
TXD0 -> On Bus	011 1 01 00	100 - 1-xx	74	88	0E90 xx xx	
TXD1 -> Acq IM	011 1 01 01	000 - 1-xx	75	08	0EA0 xx xx	T_Ident_AIM_FVD
TXD2 -> Mes. Auto.	011 1 01 10	000 - 1-xx	76	08	0EC0 xx xx	

Nœud "Feux arrière gauche"

RXF0 -> IRM et OM	011 1 10 00	001 - 1-xx	78	28	0F04 xx xx	T_Ident_IRM_FRG
RXF1 -> IM	011 1 10 00	010 - 1-xx	78	48	0F08 xx xx	T_Ident_IM_FRG
TXD0 -> On Bus	011 1 10 00	100 - 1-xx	78	88	0F10 xx xx	
TXD1 -> Acq IM	011 1 10 01	000 - 1-xx	79	08	0F20 xx xx	T_Ident_AIM_FRG
TXD2 -> Mes. Auto.	011 1 10 10	000 - 1-xx	7A	08	0F40 xx xx	

Nœud "Feux arrière droit"

RXF0 -> IRM et OM	011 1 11 00	001 - 1-xx	7C	28	0F84 xx xx	T_Ident_IRM_FRD
RXF1 -> IM	011 1 11 00	010 - 1-xx	7C	48	0F88 xx xx	T_Ident_IM_FRD
TXD0 -> On Bus	011 1 11 00	100 - 1-xx	7C	88	0F90 xx xx	
TXD1 -> Acq IM	011 1 11 01	000 - 1-xx	7B	08	0FA0 xx xx	T_Ident_AIM_FRD
TXD2 -> Mes. Auto.	011 1 11 10	000 - 1-xx	7E	08	0FC0 xx xx	

On remarque que les identifiants (ID) dans le tableau contiennent des valeurs incomplètes représentées par "xx xx". Ces valeurs doivent être complétées en fonction du type de message utilisé (IM, IRM, OM).

Pour déterminer les valeurs exactes à insérer, nous nous basons sur un second tableau **de command messages** (Le tableau est en binaire) qui fournit les informations nécessaires au remplissage des parties manquantes des **identifiants** et de champ **DATA**. Ce tableau associe chaque type de message à une structure spécifique d'identifiant, nous permettant ainsi de finaliser les ID de manière précise et cohérente.

Information Request Messages (to MCP2502X/5X)																								
	Standard ID				Extended ID				Data Bytes															
0	1	9	8	7	6	5	4	3	2	1	R/I	DLC	1	1	RXB1D8 (8 bits)	RXB1D0 (8 bits)								
Read A/D Regs	x	x	x	x	x	x	x	x	x	1	0	0	0	8	x	xxxx xxxx	xxxx *000	n/a						
Read Control Regs	x	x	x	x	x	x	x	x	x	1	0	1	1	7	x	xxxx xxxx	xxxx *001	n/a						
Read Config Regs	x	x	x	x	x	x	x	x	x	1	0	1	0	5	x	xxxx xxxx	xxxx *010	n/a						
Read CAN Error	x	x	x	x	x	x	x	x	x	1	0	0	1	3	x	xxxx xxxx	xxxx *011	n/a						
Read PWM Config	x	x	x	x	x	x	x	x	x	1	0	1	0	6	x	xxxx xxxx	xxxx *100	n/a						
Read User Mem	x	x	x	x	x	x	x	x	x	1	1	0	0	8	x	xxxx xxxx	xxxx *110	n/a						
Read User Mem (bank)	x	x	x	x	x	x	x	x	x	1	1	0	0	8	x	xxxx xxxx	xxxx *110	n/a						
Read Register	x	x	x	x	x	x	x	x	x	1	0	0	0	1	x	addr	xxxx *111	n/a						

Output Messages (from MCP2502X/5X)																									
	Standard ID				Extended ID				Data Bytes																
0	1	9	8	7	6	5	4	3	2	1	R/I	DLC	1	1	RXB1D8 (8 bits)	RXB1D0 (8 bits)									
Read A/D Regs	x	x	x	x	x	x	x	x	x	0	1	0	0	8	x	xxxx xxxx	xxxx *000	I0INTFL	GPIO	AN0H	AN1H	AN10L	AN2H	AN3H	AN23L
Read Control Regs	x	x	x	x	x	x	x	x	x	0	1	0	1	7	x	xxxx xxxx	xxxx *001	ADC0N0	ADC0N1	OPTREG1	OPTREG2	STCON	I0INTEN	I0INTPO	n/a
Read Config Regs	x	x	x	x	x	x	x	x	x	0	1	0	0	5	x	xxxx xxxx	xxxx *010	DDR	GPIO	CNF1	CNF2	CNF3	n/a	n/a	n/a
Read CAN Error	x	x	x	x	x	x	x	x	x	0	1	0	1	3	x	xxxx xxxx	xxxx *011	EFLG	TEC	REC	n/a	n/a	n/a	n/a	
Read PWM Config	x	x	x	x	x	x	x	x	x	0	1	0	1	6	x	xxxx xxxx	xxxx *100	PR1	PR2	T1CON	T2CON	PWM1DCH	PWM2DCH	n/a	n/a
Read User Mem (bank1)	x	x	x	x	x	x	x	x	x	0	1	0	0	8	x	xxxx xxxx	xxxx *101	USERID0	USERID1	USERID2	USERID3	USERID4	USERID5	USERID6	USERID7
Read User Mem (bank)	x	x	x	x	x	x	x	x	x	0	1	0	0	8	x	xxxx xxxx	xxxx *110	USERID8	USERID9	USERID10	USERID11	USERID12	USERID13	USERID14	USERID15
Read Register	x	x	x	x	x	x	x	x	x	0	1	0	0	1	x	addr	xxxx *111	value	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Input Messages (to MCP2502X/5X)																									
	Standard ID				Extended ID				Data Bytes																
0	1	9	8	7	6	5	4	3	2	1	R/I	DLC	1	1	RXB1D8 (8 bits)	RXB1D0 (8 bits)									
Write Register	x	x	x	x	x	x	x	x	x	0	1	0	1	3	x	xxxx xxxx	xxxx x000	addr	mask	value	n/a	n/a	n/a	n/a	
Write TX Message ID 0	x	x	x	x	x	x	x	x	x	0	1	0	0	4	x	xxxx xxxx	xxxx x001	TX0SIDH	TX0SIDL	TX0EID8	TX0EID0	n/a	n/a	n/a	n/a
Write TX Message ID 1	x	x	x	x	x	x	x	x	x	0	1	0	0	4	x	xxxx xxxx	xxxx x010	TX1SIDH	TX1SIDL	TX1EID8	TX1EID0	n/a	n/a	n/a	n/a
Write TX Message ID 2	x	x	x	x	x	x	x	x	x	0	1	0	0	4	x	xxxx xxxx	xxxx x011	TX2SIDH	TX2SIDL	TX2EID8	TX2EID0	n/a	n/a	n/a	n/a
Write I/O Configuration	x	x	x	x	x	x	x	x	x	0	1	0	0	5	x	xxxx xxxx	xxxx x100	I0INTEN	I0INTPO	DDR	OPTREG1	ADC0N1	n/a	n/a	n/a
Write RX Mask	x	x	x	x	x	x	x	x	x	0	1	0	0	4	x	xxxx xxxx	xxxx x101	RXMSIDH	RXMSIDL	RXMEID8	RXMEID0	n/a	n/a	n/a	n/a
Write RX Filter0	x	x	x	x	x	x	x	x	x	0	1	0	0	4	x	xxxx xxxx	xxxx x110	RXF0SIDH	RXF0SIDL	RXF0EID8	RXF0EID0	n/a	n/a	n/a	n/a
Write RX Filter1	x	x	x	x	x	x	x	x	x	0	1	0	0	4	x	xxxx xxxx	xxxx x111	RXF1SIDH	RXF1SIDL	RXF1EID8	RXF1EID0	n/a	n/a	n/a	n/a

Prenons un exemple pour bien comprendre comment compléter l'identifiant (ID) d'une trame CAN :

Récapitulons étape par étape pour l'envoi d'un **IRM (Information Request Message)** à la carte 8 entrées de Commodo Feux :

1. Début de l'ID :

- Le tableau d'identification des différents modules CAN nous donne **05 04 xx xx** pour un message IRM envoyé à ce nœud.
- On garde **05 04** et on complète les **xx xx** avec le tableau **de command messages**.

2. Choix de la bonne ligne dans le tableau de command messages :

- Un message **IRM** est une requête d'information, donc on se réfère à la ligne "**Read Register**".
- Cette ligne nous donne la structure de l'Extended ID : **addr / xxxx x111**.
- Les **deux premiers xx** doivent être remplacés par l'adresse du registre contenant l'état des entrées (**GPLAT, registre gère l'état des entrées et des sorties**).

3. Complétion des derniers octets :

- Les **deux derniers xx** correspondent à **xxxx x111**, on remplace **xxxx** par **0 → 07**.

ID final à envoyer : 05 04 [Adresse_Registre] 07

6.2.2 Remplissage du champ DLC :

Le champ DLC indique le nombre d'octets présents dans le champ DATA de la trame CAN. Il peut prendre une valeur de 0 à 8, selon la quantité de données envoyées.

Si on veut envoyer 3 octets dans le champ DATA, alors DLC = 0x03. Si on envoie 8 octets, alors DLC = 0x08.

6.2.3 Remplissage du champ DATA :

Si nous voulons envoyer un message de type IM, on est en mode **écriture dans un registre (on se réfère à la ligne "Write Register")** à la carte **Commodo Feux**, nous devons configurer les broches en mode **entrée** ou en mode **sortie**.

Pour cela, nous devons compléter l'**ID du message** et envoyer **3 octets** dans le champ **DATA** :

- L'**ID du message IM** pour le commodo feux est **05 08 xx xx**.
- D'après la structure du tableau IM, la suite de l'**ID** suit le format **xxxx xxxx / xxxx x000**.
- En remplaçant les **xxxx** par **0**, nous obtenons **00 00**.

ID final du message IM : 05 08 00 00

Ensuite, nous devons compléter le champ **DATA**, qui contient :

1. **Octet d'adresse** : L'adresse du registre responsable de la configuration des broches en entrée ou sortie (**GPDDR**).
2. **Octet de masque (Mask)** : Permet de sélectionner les bits à modifier.
3. **Octet de valeur (Value)** : Définit si une broche est en **entrée (1)** ou en **sortie (0)**.

6.2.3.1. Gestion des données avec le masque :

Lorsque la valeur d'un bit de **masque** est à **0**, nous ignorons la donnée correspondante dans **l'octet de valeur (Value)**. En d'autres termes :

- Si un bit de **masque** vaut **0**, la donnée associée dans **(Value)** n'est **pas prise en compte**.
- Si un bit de **masque** vaut **1**, la donnée associée est **appliquée**.

Exemple d'utilisation d'un masque :

Avec un **octet de valeur** de **0xFF** et un masque de **0x0F** :

- **Masque 0x0F :**

- Les bits **GP3 à GP0** sont pris en compte.
- Les bits **GP7 à GP4** sont ignorés.
- **Donnée : 0xFF :**
 - Les bits **GP7 à GP4** (qui valent 1) sont ignorés grâce au masque.
 - Les bits **GP3 à GP0** sont configurés comme suit :
 - GP3 : 1 (entrée).
 - GP2 : 1 (entrée).
 - GP1 : 1 (entrée).
 - GP0 : 1 (entrée).

Pour trouver les adresses des registres **GPDDR** (registre de direction) et **GPLAT** (registre de sortie/entrée), il faut consulter les tableaux d'affectation des registres dans la documentation du MCP25050 fournie par **didalab**.

Addr*	Name	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Value on POR	Value on RST
1Fh**	GPDDR	—	DDR6	DDR5	DDR4	DDR3	DDR2	DDR1	DDR0	-111 1111	-111 1111
18h	EFLG	ESCF	RBO	TXBO	TXEP	RXEP	TXWAR	RXWAR	EWARN	0000 0000	0000 0000
19h	TEC	Transmit Error Counters								0000 0000	0000 0000
1Ah	REC	Receive Error Counters								0000 0000	0000 0000
50h	ADRES3H	AN3.9	AN3.8	AN3.7	AN3.6	AN3.5	AN3.4	AN3.3	AN3.2	xxxx xxxx	uuuu uuuu
51h	ADRES3L	AN3.1	AN3.0	—	—	—	—	—	—	xx-- ----	uu-- ----
52h	ADRES2H	AN2.9	AN2.8	AN2.7	AN2.6	AN2.5	AN2.4	AN2.3	AN2.2	xxxx xxxx	uuuu uuuu
53h	ADRES2L	AN2.1	AN2.0	—	—	—	—	—	—	xx-- ----	uu-- ----
54h	ADRES1H	AN1.9	AN1.8	AN1.7	AN1.6	AN1.5	AN1.4	AN1.3	AN1.2	xxxx xxxx	uuuu uuuu
55h	ADRES1L	AN1.1	AN1.0	—	—	—	—	—	—	xx-- ----	uu-- ----
56h	ADRES0H	AN0.9	AN0.8	AN0.7	AN0.6	AN0.5	AN0.4	AN0.3	AN0.2	xxxx xxxx	uuuu uuuu
57h	ADRES0L	AN0.1	AN0.0	—	—	—	—	—	—	xx-- ----	uu-- ----

Addr	Name	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Value on POR	Value on RST
1Eh	GPPIN	GP7	GP6	GP5	GP4	GP3	GP2	GP1	GP0	0000 0000	0000 0000
34h	GPDDR *	—	DDR6	DDR5	DDR4	DDR3	DDR2	DDR1	DDR0	-111 1111	-111 1111
00h	IOINTEN	GP7TXC	GP6TXC	GP5TXC	GP4TXC	GP3TXC	GP2TXC	GP1TXC	GP0TXC	0000 0000	0000 0000
01h	IOINTPO	GP7POL	GP6POL	GP5POL	GP4POL	GP3POL	GP2POL	GP1POL	GP0POL	0000 0000	0000 0000
0Eh	ADCON0	ADON	T0PS2	T0PS1	T0PS0	GO/DONE	—	CHS1	CHS0	0000 0-00	0000 0-00
0Fh	ADCON1	ADCS1	ADCS0	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0	0000 0000	0000 0000
2Ch	ADCMP3H	AN3CMP.	AN3CMP.8	AN3CMP.7	AN3CMP.6	AN3CMP.5	AN3CMP.4	AN3CMP.3	AN3CMP2	xxxx xxxx	uuuu uuuu
2Dh	ADCMP3L	AN3CMP.	AN3CMP.0	—	—	Reserved			ADPOL	xx-- ----x	uu-- ----u
2Eh	ADCMP2H	AN2CMP.	AN2CMP.8	AN2CMP.7	AN2CMP.6	AN2CMP.5	AN2CMP.4	AN2CMP.3	AN2CMP2	xxxx xxxx	uuuu uuuu
2Fh	ADCMP2L	AN2CMP.	AN2CMP.0	—	—	Reserved			ADPOL	xx-- ----x	uu-- ----u
30h	ADCMP1H	AN1CMP.	AN1CMP.8	AN1CMP.7	AN1CMP.6	AN1CMP.5	AN1CMP.4	AN1CMP.3	AN1CMP2	xxxx xxxx	uuuu uuuu
31h	ADCMP1L	AN1CMP.	AN1CMP.0	—	—	Reserved			ADPOL	xx-- ----x	uu-- ----u
32h	ADCMP0H	AN0CMP.	AN0CMP.8	AN0CMP.7	AN0CMP.6	AN0CMP.5	AN0CMP.4	AN0CMP.3	AN0CMP2	xxxx xxxx	uuuu uuuu
33h	ADCMP0L	AN0CMP.	AN0CMP.0	—	—	Reserved			ADPOL	xx-- ----x	uu-- ----u
10h	STCON	STEM	STMS	STBF1	STBF0	STM3	STM2	STM1	STM0	0xxx xxxx	0uuuu uuuu

Remarque : le registre **GPPIN** est équivalent au **GPLAT**.

6.3. Exemple de trame du TP1 de VMD :

Cette trame configure les broches en mode **sortie (feux arrière droit)**.

```
T_IM_Feux.trame_info.registre=0x00;
T_IM_Feux.trame_info.champ.extend=1; // On travaille en mode étendu
T_IM_Feux.trame_info.champ.dlc=0x03; // Il y aura 3 données de 8 bits (3 octets envoyés)
T_IM_Feux.ident.extend.identificateur.ident=0xF880000; // C'est l'identificateur du bloc optique arrière droit
T_IM_Feux.data[0]=0x1F; // première donnée -> "Adresse" du registre concernée (GPDDR donne la direction des I/O)
T_IM_Feux.data[1]=0x7F; // deuxième donnée -> "Masque" -> les sorties sont sur les 4 bits de poids faibles
T_IM_Feux.data[2]=0xF0; // troisième donnée -> "Valeur" -> Les sorties sont sur les 4 bits lsb
```

Le nom de la trame : **T_IM_Feux**

- **T_IM_Feux.trame_info.registre=0x00** : initialise tous les bits de registre à zéro.
- **T_IM_Feux.trame_info.champ.extend=1** : Mode étendu activé.
- **T_IM_Feux.trame_info.champ.dlc=0x03** : Envoi de 3 données (car il s'agit d'un message **write register** en mode IM).
- **T_IM_Feux.ident.extend.identificateur.ident=0xF880000** : Identificateur spécifique au modèle arrière droit pour un message IM.

Contenu du champ champ data :

- **T_IM_Feux.data[0]=0x1F** : l'adresse de registre **GPDDR**.
- **T_IM_Feux.data[1]=0x7F** : Masque appliqué, seuls les 7 bits de poids faibles sont modifiés (le bit 7 reste inchangé car il est en mode entrée et non reprogrammable).
- **T_IM_Feux.data[2]=0xF0** : Les bits **GP7 à GP4** sont configurés en **entrée** et les bits **GP3 à GP0** sont configurés en **sortie**

Remarque : Les bits GP7 à GP4 du circuit MCP25050 sont déjà programmés en mode entrée et ne peuvent pas être reprogrammés, car le circuit n'est pas reprogrammable.

En conséquence, de notre travail sur le projet nous ne sommes pas intéressés par un masque comme de 0x7F, qui inclut des bits inutilisables (**GP7 à GP4**). À la place, nous pouvons utiliser un masque de 0x0F, qui cible uniquement les bits **GP3 à GP0**, configurables en mode sortie.

7. Installation du Logiciel EID210 :

Afin de mettre en œuvre les concepts expliqués dans ce chapitre, il est nécessaire d'installer le logiciel **EID210**. Ce logiciel permet de configurer et de tester la communication avec le module **CAN01A** et le **VMD** via la carte **EID210 000** et la carte **ATON CAN**.

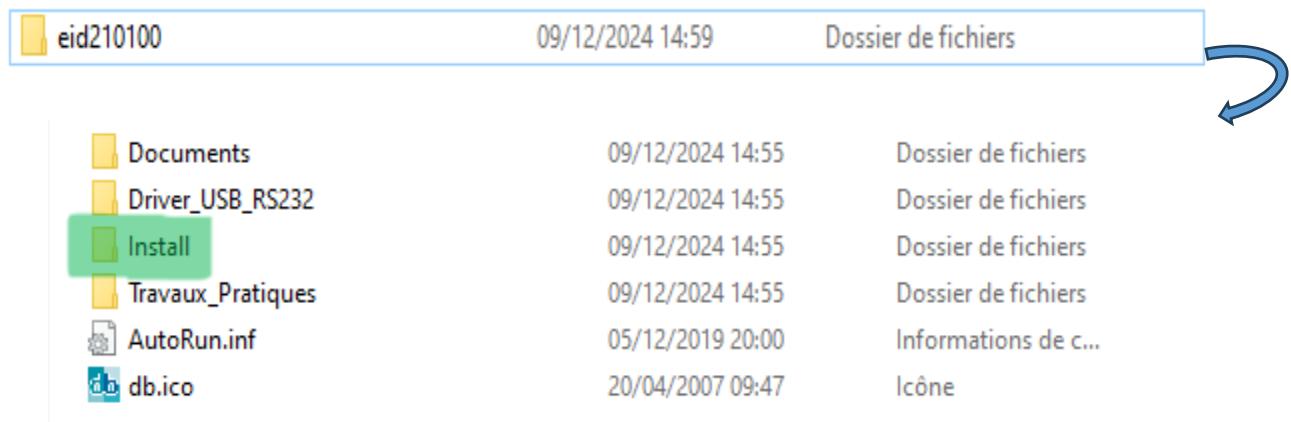
Introduction

Le logiciel **EID210** est un outil essentiel pour les professionnels et étudiants travaillant sur des systèmes embarqués. Ce guide propose une procédure détaillée pour installer, configurer et utiliser ce logiciel, garantissant ainsi une prise en main efficace et sans erreur. Que vous soyez débutant ou expérimenté, suivez ces étapes pour exploiter pleinement les fonctionnalités de l'**EID210**.

Étape 1 : Installation du logiciel

1. Accéder au dossier d'installation

- Ouvrez le dossier **eid210100**, puis accédez au sous-dossier **Install**.



📁	eid210100	09/12/2024 14:59	Dossier de fichiers
📁	Documents	09/12/2024 14:55	Dossier de fichiers
📁	Driver_USB_RS232	09/12/2024 14:55	Dossier de fichiers
📁	Install	09/12/2024 14:55	Dossier de fichiers
📁	Travaux_Pratiques	09/12/2024 14:55	Dossier de fichiers
📄	AutoRun.inf	05/12/2019 20:00	Informations de c...
.ico	db.ico	20/04/2007 09:47	Icône

2. Lancer l'installation

- Double-cliquez sur le fichier **Install_EID210.msi** pour démarrer l'installation.

📁	64bits	09/12/2024 14:55	Dossier de fichiers
📁	ACROBAT	09/12/2024 14:55	Dossier de fichiers
📁	Windows_Seven_Regedit	09/12/2024 14:55	Dossier de fichiers
Installer	Install_EID210.msi	16/01/2004 11:51	Package Windows... 17 536 Ko

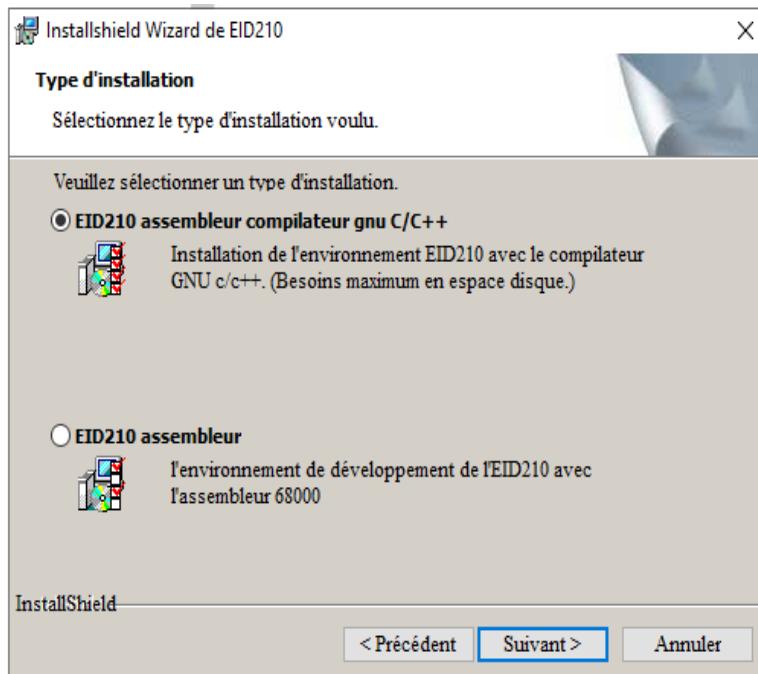
3. Suivre l'assistant InstallShield Wizard

- Une fenêtre s'ouvre. Cliquez successivement sur **Suivant > Suivant > Suivant**.



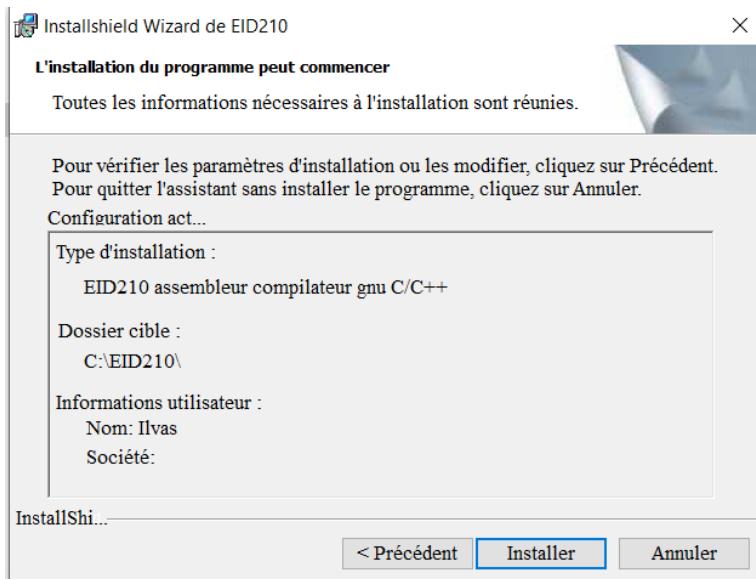
4. Sélectionner type d'installation

- Cochez l'option suivante : **EID210 assembleur compilateur GNU C/C++**



5. Finaliser l'installation

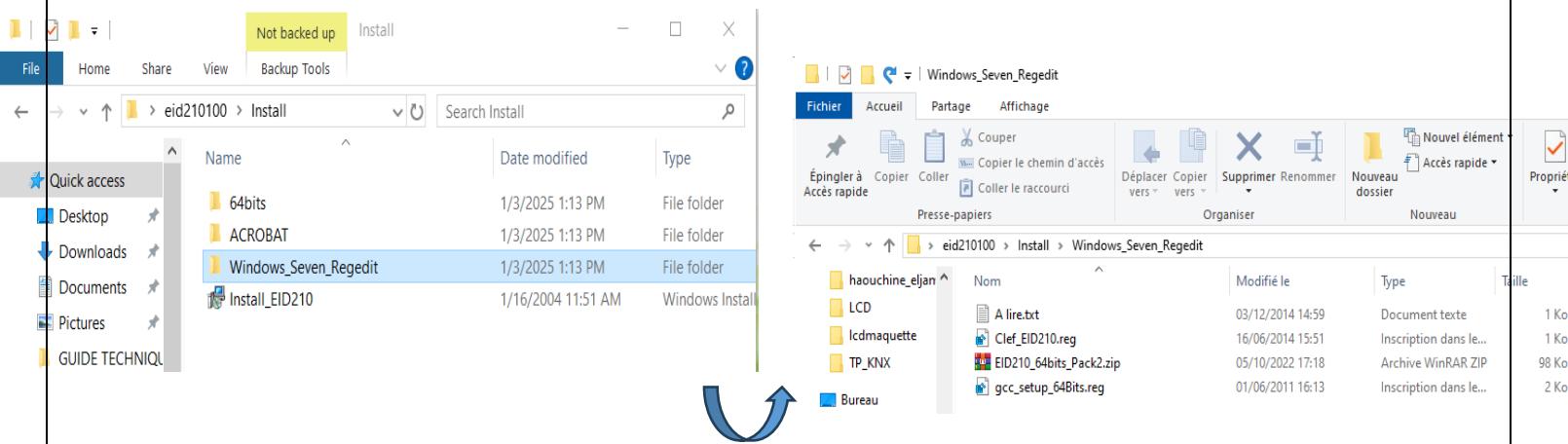
- Cliquez sur **Installer**, puis sur **Terminer** lorsque l'installation est achevée.



Étape 2 : Configuration après l'installation

1. Ouvrir Windows_Seven_Regedit

- Dans le dossier **Install**, double-cliquez sur le fichier **Windows_Seven_Regedit**

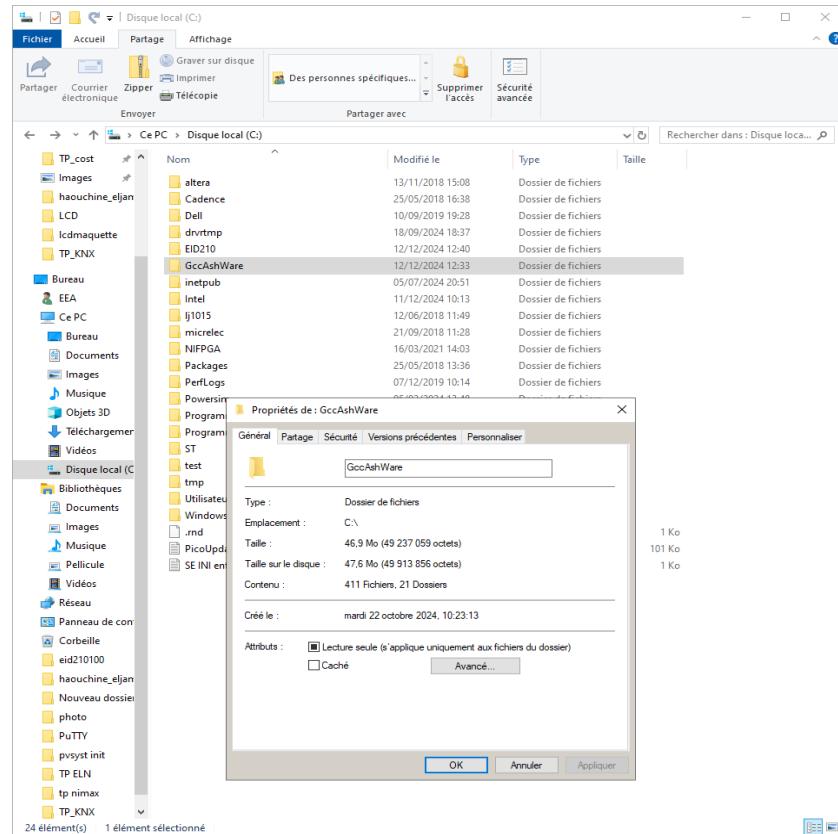


2. Vérification du GccAshWare

- Rechercher le dossier **GccAshWare** :

- Accédez au disque local (**C:**)

- Vérifiez que le dossier **GccAshWare** est présent.
- **Vérifier les propriétés :**
 - Cliquez avec le bouton droit sur **GccAshWare**.
 - Sélectionnez **Propriétés** pour confirmer que le fichier est correctement configuré



3. Accéder au dossier **EID210**

- Ouvrez le dossier **EID210** qui se trouve dans la fenêtre du **Disque local (C :)**

Astuce : Ouvrez deux fenêtres en parallèle :

- Une pour le **Disque local (C:)** (avec le dossier **EID210**).
- L'autre pour le chemin **eid210100 > Install > Windows_Seven_Regedit**.
Cela facilitera le transfert des fichiers à l'étape suivante.

4. Décompresser et transférer les fichiers

- **Dézipper le fichier**

- Dans la fenêtre ouverte sur **eid210100 > Install > Windows_Seven_Regedit**, sélectionnez le fichier ZIP.
- Effectuez un clic droit, puis choisissez **Extraire tout**.

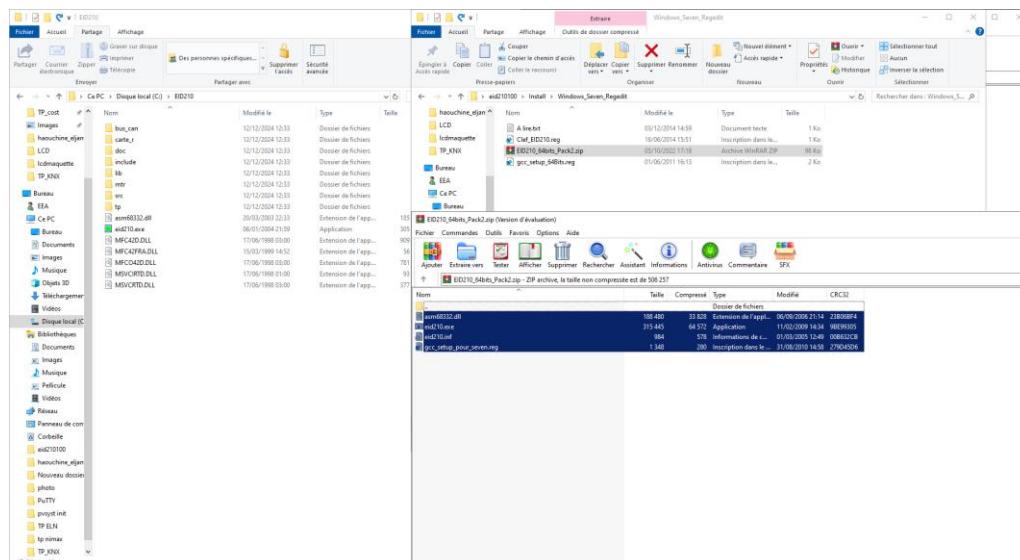
- **Sélectionner le contenu**

- Une fois le fichier décompressé, ouvrez le dossier extrait.
- Sélectionnez tous les fichiers à l'intérieur.

- **Transférer vers EID210**

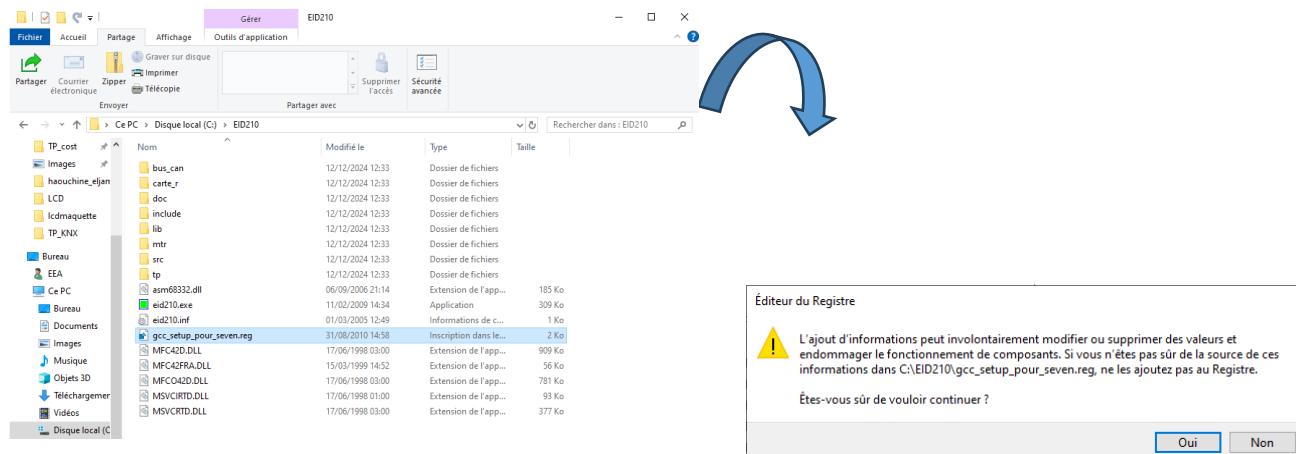
- Faites glisser les fichiers sélectionnés vers la fenêtre ouverte sur **Disque local (C:) > EID210**.

Astuce : Grâce aux deux fenêtres ouvertes en parallèle, le transfert est rapide et sans erreur.



5. Configurer les fichiers

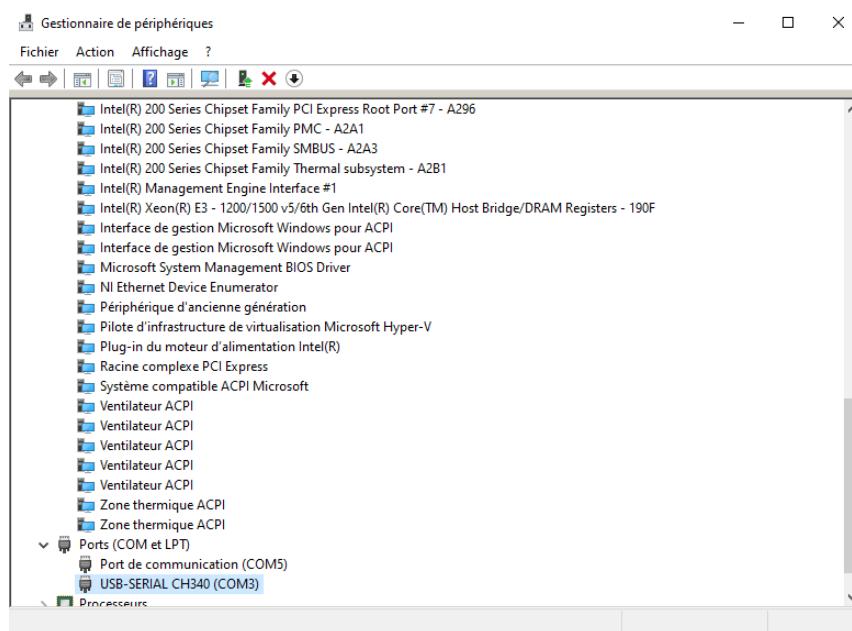
- Une fois les fichiers transférés, rendez-vous dans le dossier **EID210** sur le **Disque local (C:)**.
- Double-cliquez sur le fichier **gcc_setup_pour_seven.reg**. Une boîte de dialogue apparaîtra pour demander une autorisation : cliquez sur **Oui**, puis encore **Oui**, et enfin sur **OK** pour confirmer les modifications au registre.



Étape 3 : Configurer le port USB

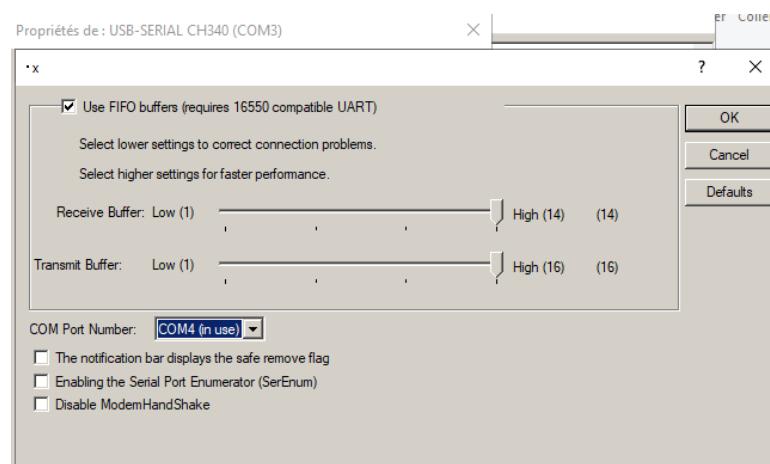
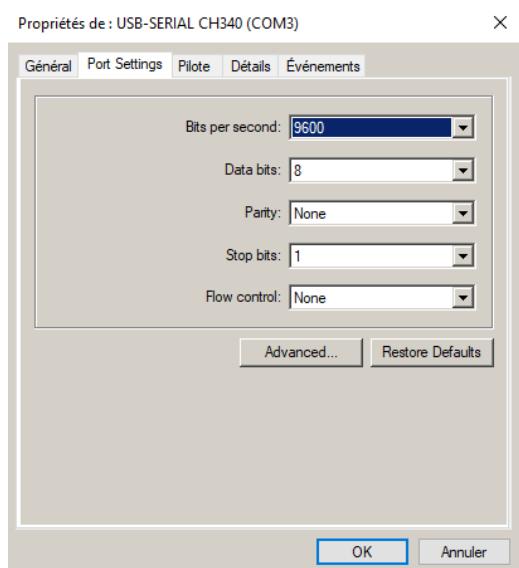
1. Accéder au gestionnaire de périphériques

- Appuyez sur les touches **Windows + R**, tapez **devmgmt.msc**, puis appuyez sur **Entrée**.
- Dans le **Gestionnaire de périphériques**, recherchez la section **Ports (COM et LPT)**.
- Si votre carte est connectée, elle devrait apparaître dans cette liste avec un numéro de port (par exemple, **COM3**).



2. Modifier les paramètres du port USB

- Faites un clic droit sur le port USB correspondant à votre carte et sélectionnez **Propriétés**.
- Allez dans l'onglet **Port Settings** (*Paramètres du port*).
- Vérifiez et configurez les paramètres comme suit :
 - Bits per second** : 9600
 - Data bits (Bits de données)** : 8
 - Parity (Parité)** : Aucun
 - Stop bits (Bits d'arrêt)** : 1
 - Flow control (Contrôle du flux)** : Aucun
- Cliquez sur **Advanced** (*Avancé*).
- Modifiez le numéro du port COM sous la section **COM Port Number**. Par exemple, changez **COM3** en **COM4 (in use)** ou tout autre numéro disponible.



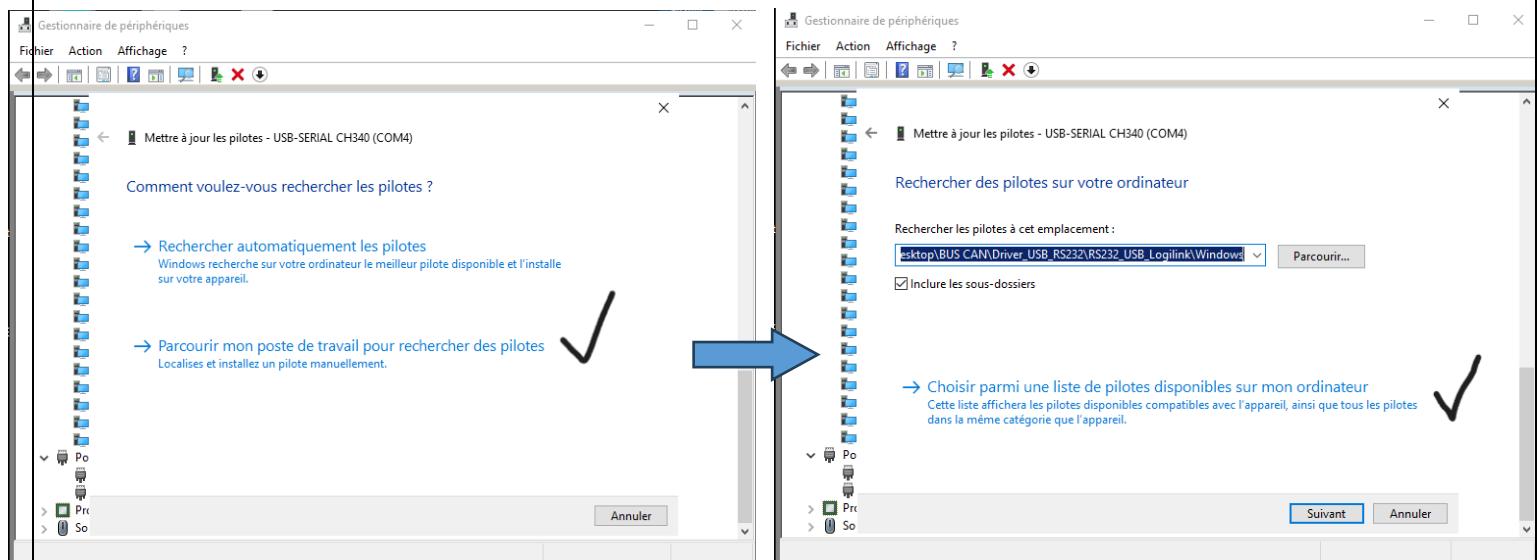
- Cliquez sur **OK** pour confirmer.

3. Valider la configuration

- Retournez dans le **Gestionnaire de périphériques** et assurez-vous que le port modifié apparaît correctement sans message d'erreur.

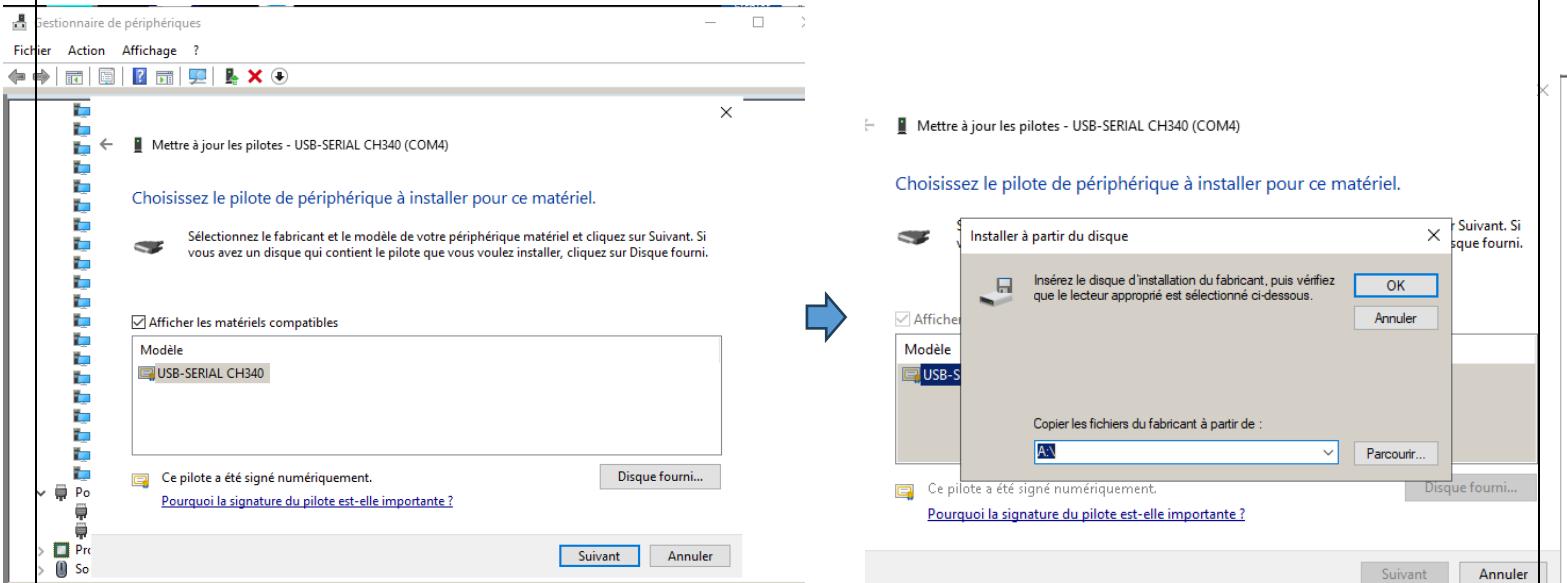
4. Mettre à jour les pilotes

- Faites un clic droit sur **USB-SERIAL CH340 (COM4)**.
- Sélectionnez **Mettre à jour le pilote**.
- Une fenêtre s'ouvre avec deux options :
 - **Rechercher automatiquement les pilotes**
 - **Parcourir mon poste de travail pour rechercher des pilotes :choisissez cette option.**
- Cliquez ensuite sur **Choisir parmi une liste de pilotes disponibles sur mon ordinateur**.



5. Sélectionner le pilote manuellement

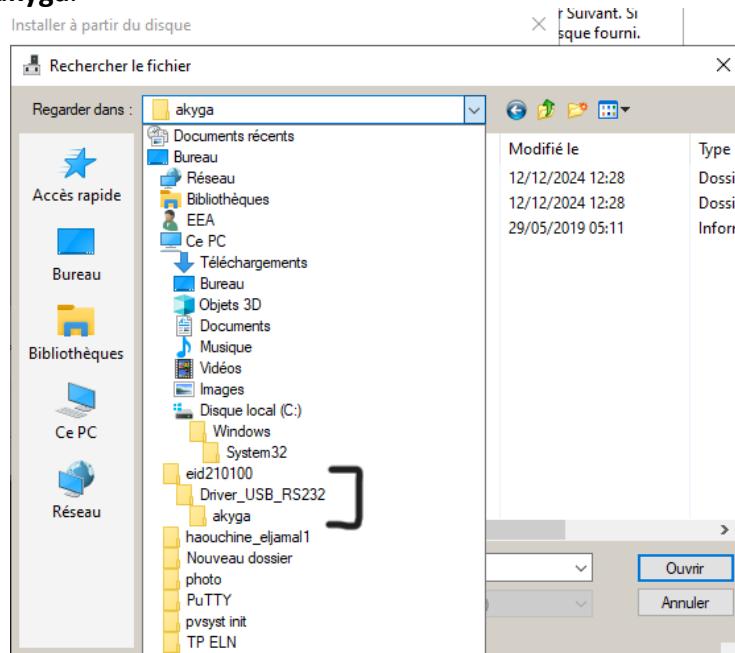
- Cliquez sur le bouton **Disque fourni**, puis sur **Parcourir**.



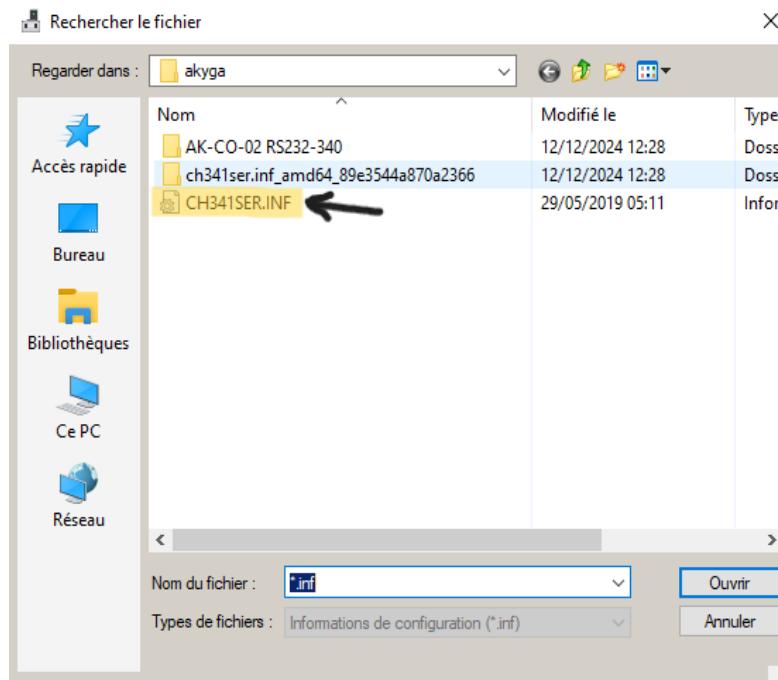
- Une fenêtre **Rechercher un fichier** s'ouvrira.

6. Naviguer jusqu'au fichier du pilote

- Accédez au chemin suivant pour localiser le fichier requis :
 - **Ce PC → Bureau ou Disque local (C:) → eid210100 → driver_USB_RS232 → akyga.**



- Double cliquez sur le fichier nommé **CH342SER.INF**.
- Cliquez sur **Ouvrir**, puis sur **OK**.



7. Installer le pilote

- Cliquez sur **Suivant** pour lancer l'installation du pilote.
- Si une fenêtre de sécurité s'affiche, cliquez sur **Installer quand même** ou **Autoriser**.

8. Confirmer l'installation

- Une fois l'installation terminée, vérifiez que le périphérique **USB-SERIAL CH340 (COM4)** n'affiche plus de symbole d'erreur dans le **Gestionnaire de périphériques**.
- Double-cliquez sur le périphérique pour vérifier que le statut dans l'onglet **Général** indique **Ce périphérique fonctionne correctement**.

9. Redémarrage si nécessaire

- Si votre carte ou logiciel ne fonctionne pas encore, redémarrez votre ordinateur pour appliquer les modifications.

Astuce : En cas de problème, assurez-vous que le fichier **CH342SER.INF** n'a pas été déplacé ou renommé dans le dossier **akyga**. Cela garantit une installation réussie.

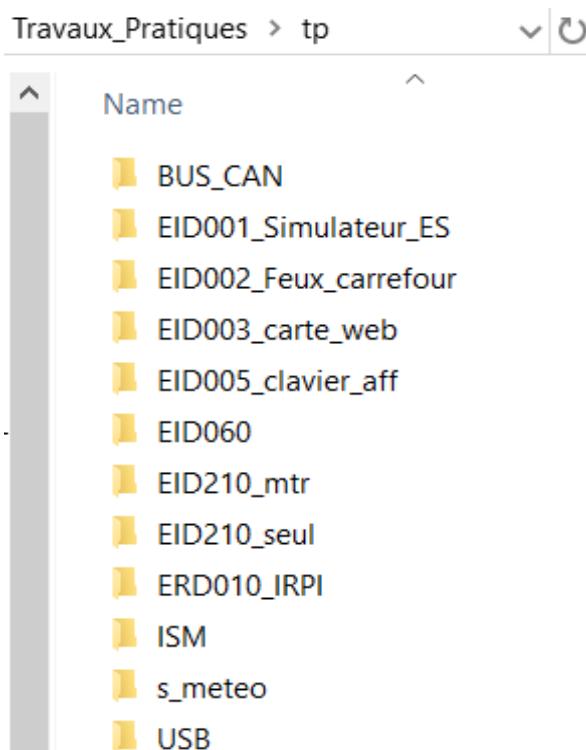
Étape 4 : Configurer le logiciel EID210 et choisir le port USB

1. Lancer le logiciel EID210

- Depuis le **bureau**, double-cliquez sur le logiciel **EID210** pour l'ouvrir.

2. Ouvrir un fichier de projet

- Cliquez sur **Fichier > Ouvrir** → eid210100 → Travaux_Pratiques→tp

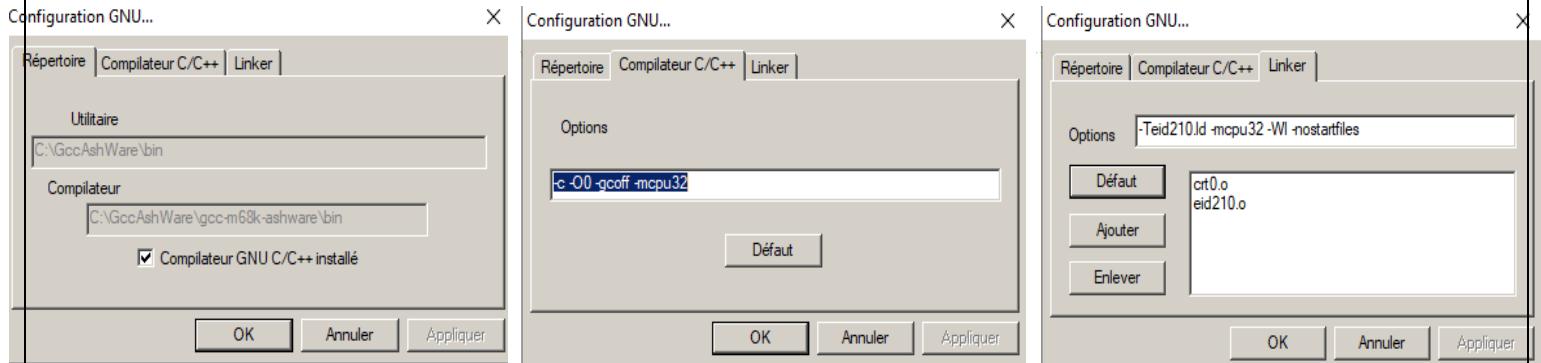


- Par exemple si vous voulez travailler sur **EID210 seul** : double cliquez sur **EID210_seul** puis **En_C** , prenez le premier exemple **Cheni1.c** .
- Le code s'affichera dans la fenêtre principale du logiciel.

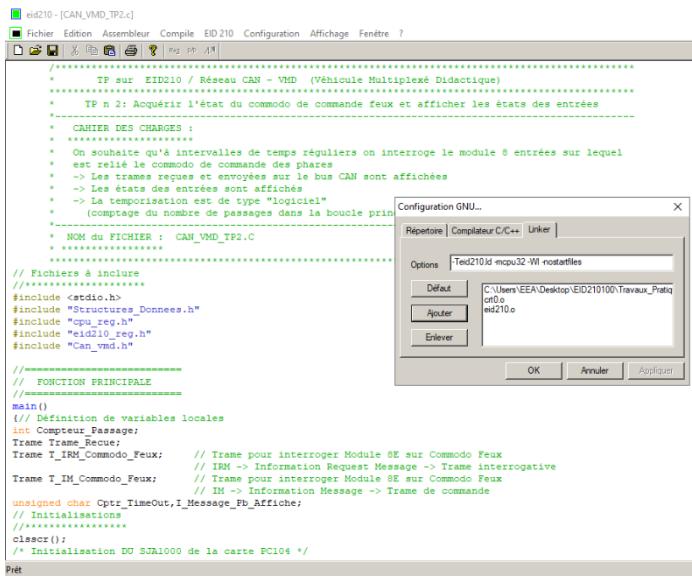
3. Configurer GNU C/C++

- Cliquez sur **Configuration > GNU C/C++**.
- Dans la fenêtre de configuration :

- Assurez-vous que tous les champs (**Répertoire**, **Compilateur C/C++**, **Linker**) sont définis sur **Par défaut**.



- **Pour tous les autres TP spécifiques** : Si vous travaillez sur des TP nécessitant le **CAN**, ajoutez le fichier **aton_can2.o** dans le champ **Linker**. Ce fichier est situé dans le dossier **BUS_CAN → CAN01A**.



```

//-----[CAN_VMD_TP2.c]
//-----TP sur EID210 / Réseau CAN - VMD (Véhicule Multiplexé Didactique)
//-----TP n° 2: Acquérir l'état du commode de commande feux et afficher les états des entrées
//-----CAHIER DES CHARGES :
//-----On souhaite qu'à intervalles de temps réguliers on interroge le module 8 entrées sur lequel
//-----est relié le commode de commande des phares
//------> Les trames reçues et envoyées sur le bus CAN sont affichées
//------> Les états des entrées sont affichés
//------> La temporisation est de type "logiciel"
//-----(comptage du nombre de passages dans la boucle principale)
//-----NOM du FICHIER : CAN_VMD_TP2.C
//-----*****[CODE]
//-----Fichiers à inclure
//-----*****
#include <stdio.h>
#include <epicd.h>
#include "Structures_Donnees.h"
#include "cpu_reg.h"
#include "eid210_reg.h"
#include "Can_vmd.h"

//----- FONCTION PRINCIPALE
//-----
main()
//----- Définition de variables locales
int Compteur_Passage;
Trame_T_IRE_Commande_Feu;
Trame_T_IM_Commande_Feu;
//----- Trame pour interroger Module 0E sur Commode Feux
//----- IIN -> Information Request Message -> Trame interrogative
Trame_T_IM_Commande_Feu;
//----- Trame pour interroger Module 0E sur Commode Feux
//----- IM -> Information Message -> Trame de commande
unsigned char Cptre_TimeOut,I_Message_Pb_Affiche;
//----- Initialisations
//-----*****
clscr();
/* Initialisation DU SJA1000 de la carte PC104 */
Pret

```

- Cliquez sur **OK** pour valider la configuration.

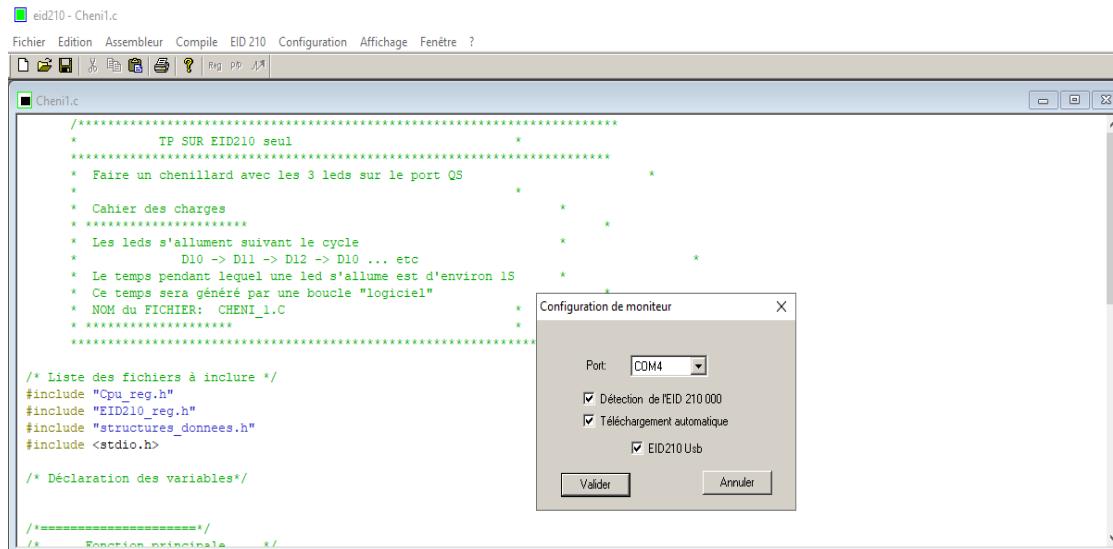
4. Configurer le moniteur

- Cliquez sur **Configuration > Moniteur**.

- Une fenêtre de configuration de moniteur s'affichera (comme dans l'image).

- **Port** : Choisissez le port **COM** configuré précédemment (par exemple, **COM4**).
- Cochez les options suivantes :
 - **Détection de l'EID 210 000**
 - **Téléchargement automatique**
 - **EID 210 USB**

- Cliquez sur **Valider**.



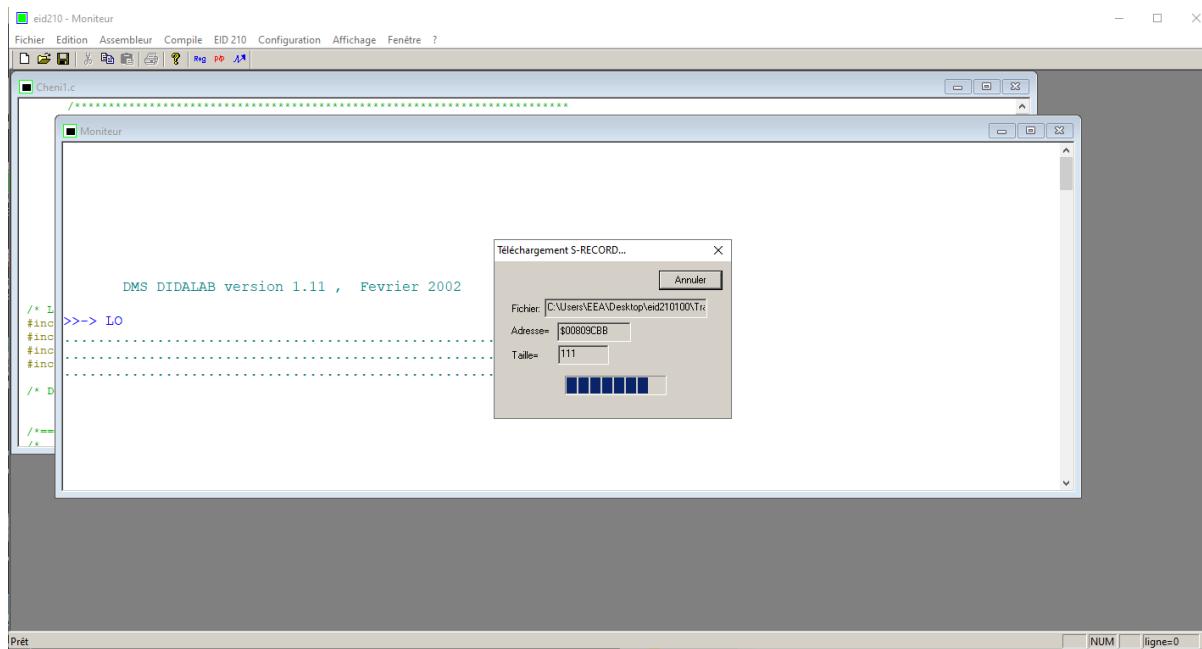
The screenshot shows the eid210 software interface with a configuration dialog box overlaid on the main window. The dialog box is titled "Configuration de moniteur" and contains the following settings:

- Port: COM4
- Détection de l'EID 210 000
- Téléchargement automatique
- EID210 Usb

At the bottom of the dialog box are two buttons: "Valider" (Validate) and "Annuler" (Cancel).

5. Vérification

- Assurez-vous que tout est bien configuré. Si des erreurs persistent, revérifiez les paramètres de port et la connexion USB.



The screenshot shows the eid210 monitor software interface with a download dialog box titled "Téléchargement S-RECORD...". The dialog box contains the following fields:

- Fichier: C:\Users\EEA\Desktop\eid210100\Trx
- Adresse: \$00809CB8
- Taille: 111

At the bottom of the dialog box are two buttons: "Annuler" (Cancel) and "Valider" (Validate). The background shows the assembly code for the program.

Conclusion

Vous avez maintenant terminé l'installation et la configuration du logiciel **EID210**. Si vous avez respecté toutes les étapes du guide, le code s'exécutera sans problème, et la communication entre le PC et les cartes sera établie avec succès.

Chapitre III :

*Conception et Implémentation du Système
Embarqué*

1. Introduction :

Dans le domaine des systèmes embarqués automobiles, l'intégration de solutions intelligentes pour améliorer la sécurité, l'efficacité énergétique et le confort des usagers est devenue une priorité. Ce projet s'inscrit dans cette dynamique en proposant un système embarqué capable de superviser des paramètres critiques, comme la température du moteur et la détection d'obstacles, tout en fournissant des retours d'information et des actions en temps réel.

Le projet repose sur l'utilisation d'un bus de communication CAN (Controller Area Network), largement utilisé dans l'industrie automobile, pour permettre une communication rapide et fiable entre plusieurs modules. Il est divisé en trois unités principales :

1. **Émetteur (unité 1)** : Mesure la température du moteur à l'aide d'un capteur DHT11 et détecte les obstacles via un capteur ultrasonique HC-SR04. Ces données sont transmises via le bus CAN pour un traitement ultérieur.
2. **Récepteur (unité 2)** : Utilise les données reçues pour piloter un moteur à courant continu (système de refroidissement de moteur de véhicule) en fonction de la température et active un buzzer pour signaler des obstacles, selon le mode de fonctionnement défini.
3. **Émetteur & Récepteur (unité 3)** : Sert d'interface utilisateur avec un écran LCD affichant la température ou la distance en fonction du mode sélectionné. Deux boutons simulent une boîte de vitesses, activant le mode "marche arrière" ou un mode "standard", et une LED signale les dépassements de température critique.

2. Matériel :

- **Capteur DHT11** : Mesure la température du moteur.
- **Capteur ultrasonique HC-SR04** : Mesure la distance pour la détection des obstacles.
- **Module CAN MCP2515** : Assure la communication entre les différentes unités via le bus CAN.
- **Moteur à courant continu** : Sert de système de refroidissement pour le moteur.
- **Buzzer** : Signale les obstacles en fonction de la distance.
- **Écran LCD (I2C)** : Affiche la température et la distance en fonction du mode de fonctionnement.
- **LED** : Indique une surchauffe du moteur.
- **Boutons poussoirs** : Simulent une boîte de vitesses.
- **Arduino Uno** : Utilisés pour le contrôle des différents modules.

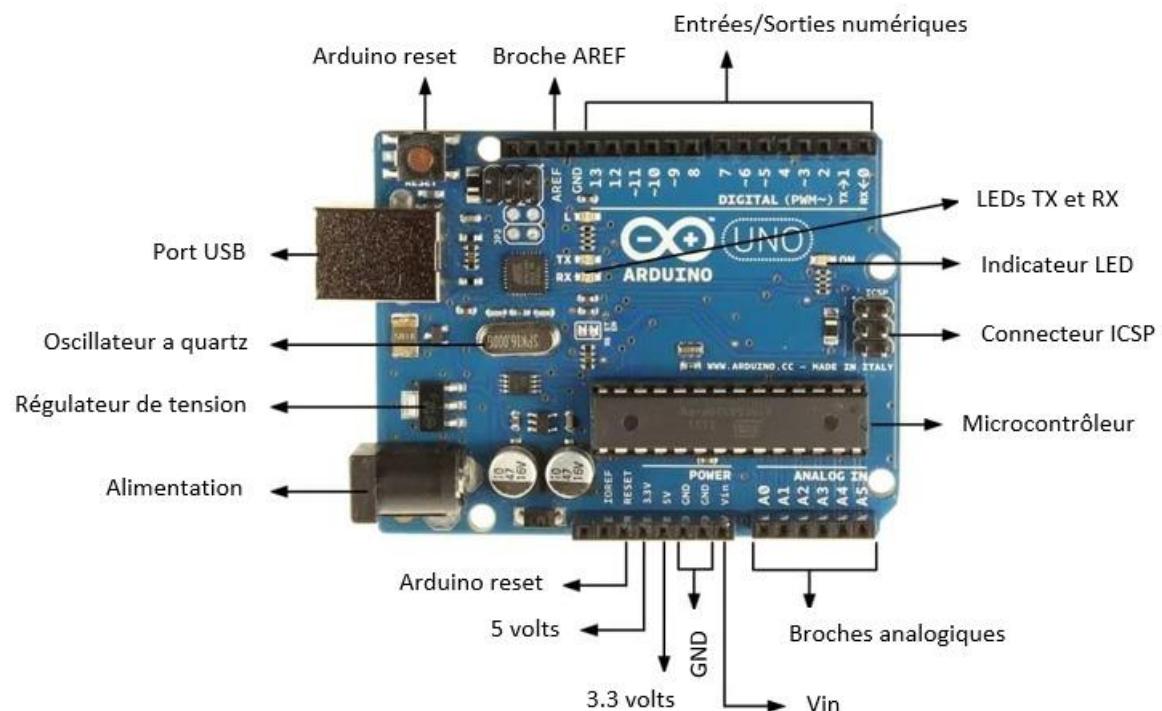
3. Explication de Matériel :

3.1. Arduino :

Arduino est une plateforme de développement de matériel et de logiciel open source. Cela signifie que les plans de la carte sont disponibles gratuitement et que les utilisateurs peuvent les modifier pour leur propre usage. Les cartes Arduino sont dotées d'un microcontrôleur, ce qui les rend capables de recevoir et d'exécuter des instructions décrites dans un code informatique. Les utilisateurs peuvent programmer ces instructions en utilisant un environnement de développement intégré (IDE) basé sur le langage de programmation de base. Et peut être utilisée pour contrôler des projets allant de simples objets connectés à des systèmes plus complexes tels que des robots ou des systèmes de contrôle de mouvement.



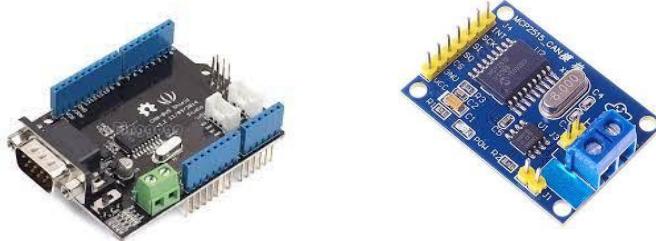
Les composants de la carte Arduino UNO :



3.2. Module CAN MCP2515 :

Définition :

Le module MCP2515 (CAN Shield) est un contrôleur CAN (Controller Area Network) externe qui permet à un microcontrôleur, tel qu'un Arduino, de communiquer via le protocole CAN. Il est souvent utilisé dans les applications automobiles, industrielles, et robotiques pour les réseaux de capteurs et d'actionneurs.



Principe de fonctionnement :

Le module MCP2515 convertit les messages du bus CAN en un format compréhensible pour le microcontrôleur et vice versa. Il agit comme une interface entre le bus CAN et le système embarqué.

Caractéristiques techniques :

- **Protocole supporté** : CAN 2.0A/B.
- **Vitesse maximale** : Jusqu'à 1 Mbps.
- **Interface microcontrôleur** : SPI (Serial Peripheral Interface).
- **Tension d'alimentation** : 5V.

Avantage :

- Communication rapide et fiable entre plusieurs modules ou capteurs.
- Compatible avec un large éventail de microcontrôleurs (Arduino, STM32, ESP32).

Applications :

- Systèmes de diagnostic automobile.
- Réseaux de capteurs industriels.
- Systèmes embarqués dans l'automobile.

Branchements du Module :

- **Alimentation du module** :

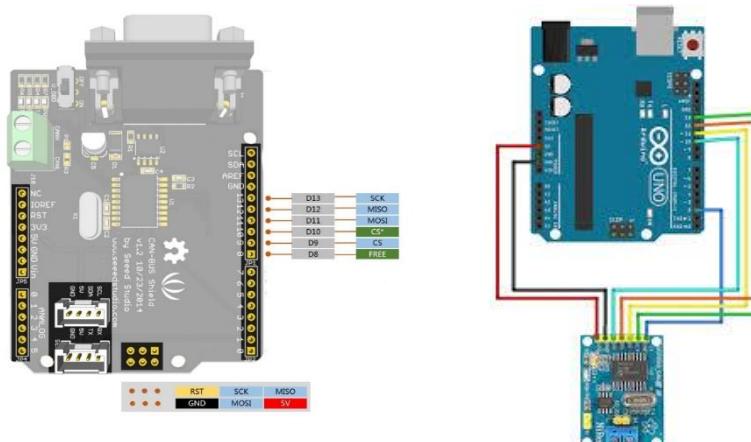
- Le module MCP2515 fonctionne généralement sous une tension de 5V. Connecter la broche **VCC** à la sortie 5V de l'Arduino.
- Relier la broche **GND** à la masse (GND) de l'Arduino.

- **Connexion SPI :**

- Le MCP2515 communique avec le microcontrôleur via le protocole SPI (Serial Peripheral Interface). Les broches suivantes doivent être connectées :
 - **CS (Chip Select)** : Connecter à une broche numérique configurable de l'Arduino (par exemple, D10).
 - **SCK (Serial Clock)** : Connecter à la broche SPI SCK de l'Arduino (D13 sur Arduino Uno).
 - **MOSI (Master Out Slave In)** : Connecter à la broche SPI MOSI de l'Arduino (D11 sur Arduino Uno).
 - **MISO (Master In Slave Out)** : Connecter à la broche SPI MISO de l'Arduino (D12 sur Arduino Uno).

- **Connexion au bus CAN :**

- Le module MCP2515 est souvent équipé d'un transceiver CAN (TJA1050) pour l'interface physique avec le bus CAN. Les broches **CAN_H** et **CAN_L** doivent être connectées au réseau CAN.



3.3. Capteur Ultrasonique HC-SR04 :

Définition :

Le capteur HC-SR04 est un dispositif de mesure de distance basé sur la technologie ultrasonique. Il est capable de détecter et de mesurer des distances en émettant des ondes sonores et en analysant leur réflexion. Ce capteur est largement utilisé dans les systèmes embarqués et robotiques en raison de sa fiabilité et de sa simplicité d'utilisation.



Principe de fonctionnement :

Le HC-SR04 fonctionne grâce à un émetteur et un récepteur ultrasoniques :

1. **Émission** : L'émetteur envoie une onde sonore haute fréquence (ultrason).
2. **Réflexion** : L'onde se réfléchit sur un obstacle.
3. **Réception** : Le récepteur capte l'onde réfléchie.
4. **Calcul de la distance** : Le capteur mesure le temps écoulé entre l'émission et la réception du signal, puis calcule la distance à l'aide de la formule :

$$\text{Distance} = \text{Temps} \times \text{Vitesse du son} / 2$$

Vitesse du son : ~ 343 m/s à 20°C .

Branchement du Capteur Ultrasonique HC-SR04 :

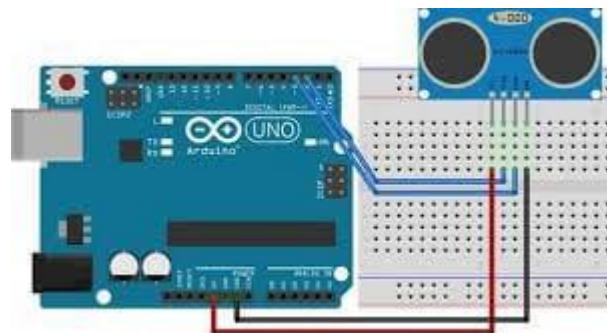
Le branchement du capteur HC-SR04 est simple et nécessite quatre connexions principales pour assurer son fonctionnement. Le capteur possède quatre broches : **VCC**, **Trig**, **Echo** et **GND**.

Alimentation (VCC et GND) :

- La broche **VCC** est connectée à une source d'alimentation de 5V, généralement fournie par le microcontrôleur (comme une carte Arduino).
- La broche **GND** est reliée à la masse (ground) du circuit.

Contrôle et Signal (Trig et Echo) :

- La broche **Trig** (Trigger) est connectée à une sortie numérique du microcontrôleur. Cette broche est utilisée pour envoyer un signal de déclenchement (un pulse de $10\ \mu\text{s}$) afin d'activer l'émetteur du capteur.
- La broche **Echo** est reliée à une entrée numérique du microcontrôleur. Cette broche transmet un signal de sortie proportionnel au temps de retour de l'onde sonore, qui sera utilisé pour calculer la distance.



3.4. Moteur à courant continu (DC) :

Un moteur à courant continu (DC) est un dispositif électromécanique qui convertit l'énergie électrique en énergie mécanique. Il est utilisé pour diverses applications nécessitant un contrôle de mouvement.



- **Principe de fonctionnement :**
 - Il fonctionne grâce à l'interaction entre un champ magnétique et un courant électrique dans des bobines, générant une force mécanique.
 - La vitesse et le sens de rotation peuvent être contrôlés via un signal PWM (Pulse Width Modulation).
- **Caractéristiques techniques :**
 - **Tension d'alimentation** : Varie selon le modèle (3V, 6V, 12V, etc.).
 - **Couple** : Dépend des spécifications du moteur.
 - **Courant** : Varie en fonction de la charge appliquée.
- **Pilotage :**
 - Nécessite un driver (ex. L298N, L293D) pour connecter à un microcontrôleur.
 - Contrôle possible de la vitesse, du démarrage/arrêt, et de l'inversion du sens.

3.5. Grove Base Shield :

Le **Grove Base Shield** est un module d'extension conçu pour simplifier la connexion de capteurs et d'actionneurs avec une carte Arduino. Il est spécialement conçu pour être compatible avec l'écosystème Grove, qui propose des capteurs et modules pré-câblés avec des connecteurs standardisés. Le shield s'enfiche directement sur une carte Arduino, offrant ainsi une facile à utiliser pour connecter rapidement des périphériques sans nécessiter de soudures ni de fils supplémentaires.



Ce shield intègre plusieurs types de connecteurs pour différents protocoles de communication :

- **Ports analogiques** pour les capteurs analogiques,
- **Ports numériques** pour les modules utilisant des signaux digitaux,
- **Ports UART** pour la communication série,
- **Ports I2C (inter-integrated circuits)**, pour le transfert de données entre un processeur central et plusieurs esclaves (afficheurs lcd, capteurs, encodeurs etc...) sur la même carte de circuit en utilisant seulement 4 fils communs.
 - Un signal de donnée (SDA) : ligne de données bidirectionnelle.
 - Un signal d'horloge (SCL) : ligne d'horloge de synchronisation.
 - La masse (GND) et l'alimentation (5V).

3.6. DHT11 :

Définition :

Le capteur de température et d'humidité DHT11 est un composant électronique utilisé pour mesurer la température et l'humidité relative de l'environnement dans lequel il est placé. Il est souvent utilisé dans des projets électroniques et des systèmes de surveillance pour obtenir des informations précises sur les conditions climatiques.

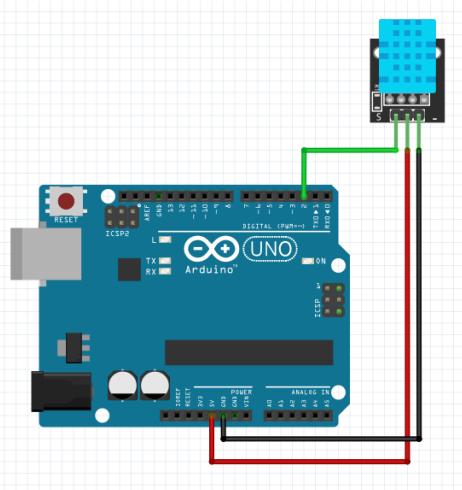


Caractéristiques :

Le capteur DHT11 est un capteur numérique basé sur un élément sensible à l'humidité et à la température. Il est capable de mesurer la température avec une précision de $\pm 2^{\circ}\text{C}$ et l'humidité relative avec une précision de $\pm 5\%$. Le capteur est alimenté en 3 à 5 volts et fournit une sortie numérique via une seule broche de données (DATA).

Fonctionnement :

Le capteur DHT11 utilise un élément sensible à l'humidité pour mesurer l'humidité relative de l'environnement et une thermistance pour mesurer la température. Il intègre un circuit électronique qui convertit les mesures en signaux numériques. Le capteur communique avec l'Arduino ou tout autre microcontrôleur via une simple connexion à une broche de données (DATA).



3.7. Buzzer actif :

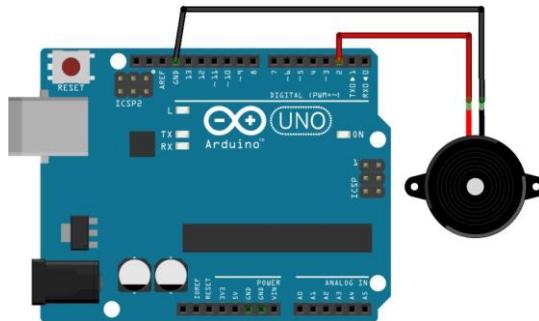
Définition :

Le buzzer actif d'Arduino est un composant électronique utilisé pour générer des sons et des signaux sonores dans les projets Arduino. Il est couramment utilisé pour ajouter des fonctionnalités audio, tels que des mélodies, des effets sonores ou des alarmes, à divers projets électroniques.



Branchements :

Pour utiliser un buzzer avec Arduino, vous devez le connecter correctement à la carte. Le buzzer est généralement composé de deux broches : une broche positive (+) et une broche négative (-). La broche positive est généralement connectée à une broche de sortie numérique de l'Arduino, tandis que la broche négative est connectée à la masse (GND) de la carte.



3.8. LED :

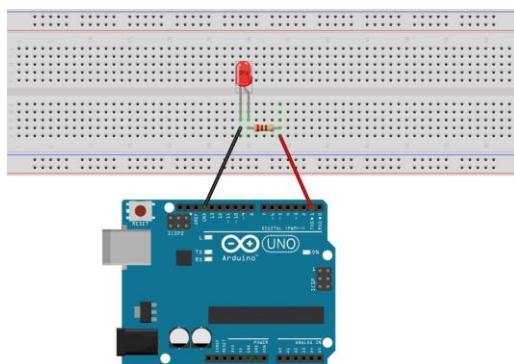
Définition :

La LED (Light Emitting Diode) d'Arduino est un composant électronique utilisé pour l'indication visuelle dans les projets basés sur Arduino. Il s'agit d'une diode électroluminescente qui émet de la lumière lorsqu'un courant électrique circule à travers elle.



Branchements :

Pour brancher une LED à une carte Arduino, il faut devez connecter la résistance en série avec la LED. L'une des extrémités de la résistance est reliée à la broche de sortie numérique de l'Arduino, tandis que l'autre extrémité de la résistance est connectée à l'anode (broche positive) de la LED. La cathode (broche négative) de la LED est ensuite connectée à la masse (GND) de l'Arduino. Il est essentiel d'inclure une résistance en série pour protéger la LED et la carte Arduino en limitant le courant traversant la LED. Il faut Choisisse une résistance appropriée en fonction de la tension d'alimentation et des spécifications de la LED.



3.9. L'afficheur LCD sans I2C :

Définition :

LCD est l'abréviation de "Liquid Crystal Display" c'est un type d'écran plat ce dernier est le plus utilisé dans les appareils électroniques comme les télévisions, les smartphones et ainsi de suite.

C'est un afficheur nous aide à afficher des caractères sur l'écran à l'aide d'une carte Arduino et un code, l'écran à cristaux liquides est un dispositif passif : il n'émet pas de lumière, seule sa transparence varie, et il doit donc disposer d'un éclairage

L'afficheur LCD utilise 6 à 10 broches de données ((D0 à D7) ou (D4 à D7) + RS + E) et deux d'alimentations (+5V et masse). La plupart des écrans possèdent aussi une entrée analogique pour régler le contraste des caractères.

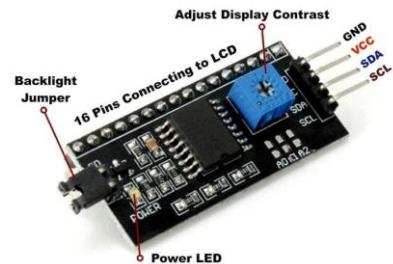


LCD avec I2C :

Définition :

Le terme I2C signifie (inter-integrated circuits). C'est un protocole de communication pour le transfert de données entre un processeur central et plusieurs esclaves (afficheurs lcd, capteurs, encodeurs etc...) sur la même carte de circuit en utilisant seulement 4 fils communs.

- Un signal de donnée (SDA) : ligne de données bidirectionnelle.
- Un signal d'horloge (SCL) : ligne d'horloge de synchronisation.
- La masse (GND).
- L'alimentation (5V).



3.10. Bouton poussoir :

Définition :

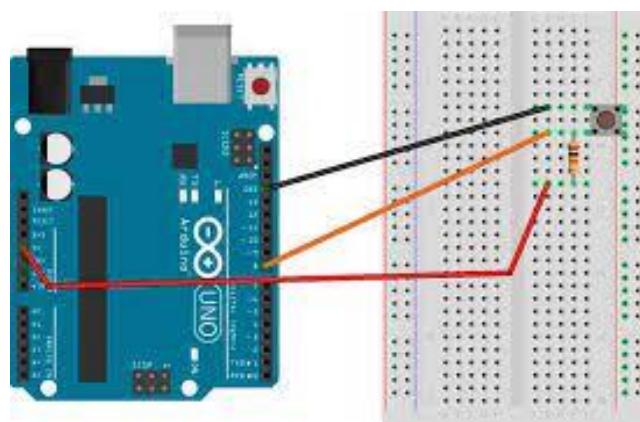
Le bouton-poussoir est un composant couramment utilisé avec les cartes Arduino. C'est un interrupteur momentané qui permet d'activer ou de désactiver une fonctionnalité en appuyant dessus. Le bouton-poussoir est généralement constitué de deux contacts métalliques qui se touchent lorsque le bouton est enfoncé. Lorsque le bouton est relâché, les contacts se séparent à nouveau. Lorsque les contacts sont en contact, le courant électrique peut circuler à travers le bouton-poussoir, ce qui permet à Arduino de détecter l'état du bouton.



Branchements :

Pour utiliser un bouton-poussoir avec Arduino, il faut le connecter correctement à la carte. Habituellement, l'un des contacts du bouton est connecté à une broche d'entrée numérique de l'Arduino, tandis que l'autre contact est connecté à la masse (GND) de la carte. Une résistance de pull-up (résistance de rappel) peut également être utilisée pour maintenir une valeur par défaut haute sur la broche d'entrée lorsque le bouton n'est pas enfoncé.

Une fois le bouton-poussoir connecté, on peut utiliser le code Arduino pour détecter l'état du bouton, on peut lire l'état de la broche d'entrée numérique associée au bouton en utilisant la fonction **digitalRead()**. Par exemple, si on a connecté le bouton à la broche 2 de l'Arduino, on peut utiliser la commande **digitalRead(2)** pour lire l'état du bouton (renvoie HIGH ou LOW).



4. Conception et Fonctionnement :

Unité 1 : Émetteur :

- **Fonction principale :** Lire la température du moteur et mesurer la distance à l'aide des capteurs, puis transmettre ces données via le bus CAN.
- **Structure des messages CAN :**
 - **Identificateur :** 0xFF
 - **Données :**
 - Byte 0 : Température (en °C).
 - Byte 1-2 : Distance (en cm).

Unité 2 : Récepteur :

- **Fonction principale :**
 - Commander le moteur en fonction de la température reçue.
 - Activer le buzzer en fonction de la distance et du mode (marche arrière).
- **Actions sur la température :**
 - **Température < 20°C :** Moteur éteint.
 - **20 < Température < 25°C :** Moteur activé à faible vitesse.
 - **Température > 25°C :** Moteur activé à pleine vitesse.
 - **Température > 30°C :** la LED s'allume.
- **Actions sur la distance :**
 - **Distance > 150 cm :** Signal faible.
 - **Distance entre 10 et 150 cm :** Signal variable.
 - **Distance < 10 cm :** Alerte continue.

Unité 3 : Émetteur & Récepteur :

- **Fonction principale :**
 - Afficher la température ou la distance sur l'écran LCD selon le mode choisi.
 - Permettre le changement de mode (à l'aide des boutons poussoirs).

- Transmettre la commande de mode (à travers une variable) à l'unité 2 (récepteur) via le bus CAN.

5. Communication Arduino avec le Shield CAN :

Pour mettre en œuvre un système embarqué basé sur le bus CAN avec une carte Arduino, il est indispensable d'ajouter un module CAN Shield. Les cartes Arduino, en elles-mêmes, ne possèdent pas les capacités matérielles pour générer ou traiter directement des trames CAN. Le module MCP2515, qui est un contrôleur CAN externe, remplit ce rôle en établissant une interface avec l'Arduino via le protocole SPI (Serial Peripheral Interface).

Pour utiliser le module MCP2515, il faut installer une bibliothèque dédiée, comme la bibliothèque "**mcp_can**". Celle-ci simplifie la gestion des trames CAN, qu'il s'agisse de leur envoi ou de leur réception.

6. Fonctionnement Réel du Système :

Dans une application réelle, ce système embarqué repose sur trois modules principaux : un émetteur (**unité 1**), un récepteur de contrôle (**unité 2**) et un tableau de bord simulé (**unité 3**) chacun ayant un rôle spécifique dans la gestion des informations et des actions critiques.

L'unité 1 ou L'émetteur est responsable de la mesure des données essentielles du véhicule. Il surveille en continu la température du moteur grâce à un capteur **DHT11**. Cette mesure est importante pour éviter les surchauffes qui pourraient endommager gravement le moteur. Parallèlement, un capteur ultrasonique **HC-SR04** détecte les obstacles à proximité du véhicule, notamment lors des manœuvres en marche arrière. Ces deux types d'informations, température et distance, sont envoyés via le bus CAN, grâce à un module **MCP2515**. Le module assure une transmission rapide et fiable des données à destination des récepteurs pour leur traitement.

L'unité 2 exploite ces données pour prendre des décisions automatiques liées à la sécurité et à la performance du véhicule. Il contrôle un moteur à courant continu, qui joue ici le rôle d'un système de refroidissement pour le moteur. Selon la température reçue, le moteur ajuste sa vitesse, augmentant l'intensité du refroidissement en cas de température élevée. Ce contrôle automatisé est indispensable pour protéger le moteur tout en maintenant une efficacité énergétique optimale. En outre, ce module active un buzzer pour avertir le conducteur de la présence d'obstacles détectés par le capteur ultrasonique.

L'unité 3 simule un tableau de bord interactif, servant de lien direct entre le conducteur et le véhicule. Un écran LCD y affiche en temps réel les informations collectées, comme la température du moteur ou la distance par rapport aux obstacles. Le mode d'affichage varie en fonction de l'état du véhicule : en mode standard, il affiche la température, tandis qu'en marche arrière, il met en avant la distance mesurée. Ce module permet également au conducteur de basculer manuellement entre les modes grâce à deux boutons. Une LED signale toute surchauffe critique du moteur, fournissant une alerte visuelle immédiate.

7. Architecture du Système :

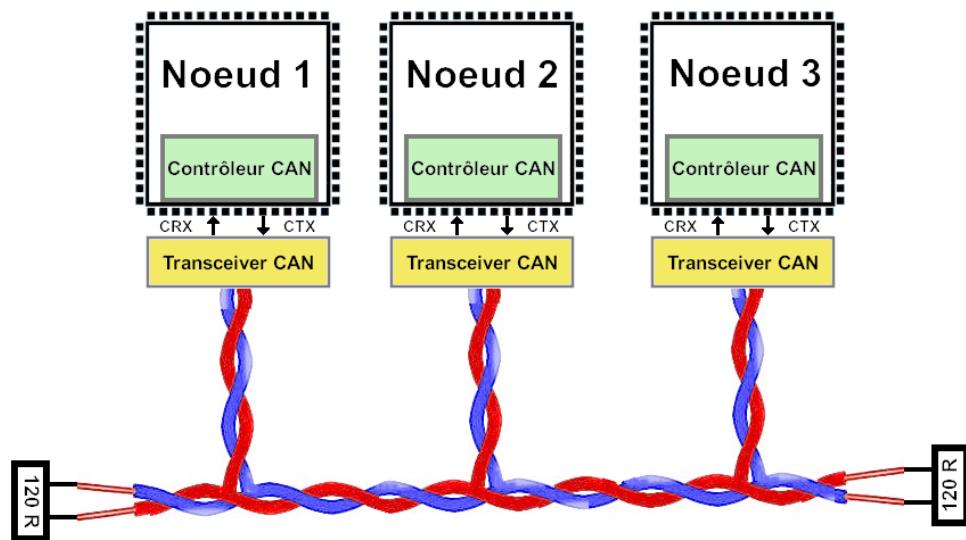
Pour implémenter notre système embarqué automobile, nous avons suivi l'architecture qui connecte les différentes unités du système, via une communication fiable et en temps réel. Chaque unité joue un rôle spécifique dans la supervision et le contrôle des paramètres critiques du véhicule.

Description des Nœuds ou des Unités :

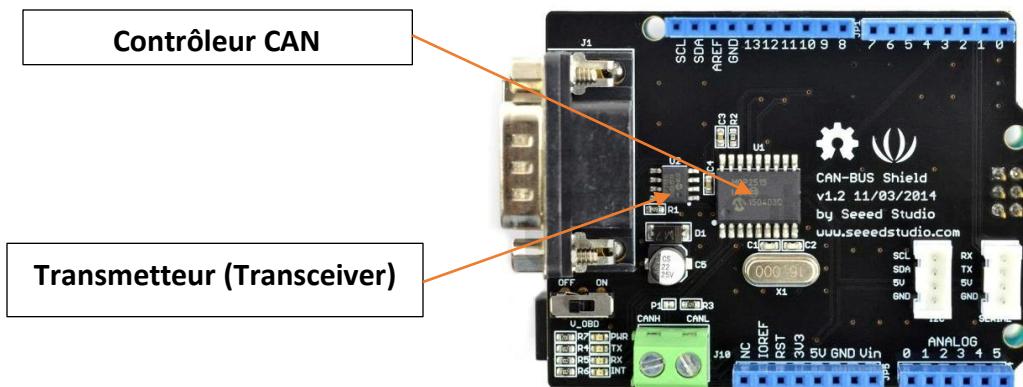
Le premier nœud est constitué le capteur **DHT11** et le capteur ultrasonique **HC-SR04**. Ces données sont traitées par la carte Arduino, qui envoie les informations sous forme de trames CAN à travers le module CAN Shield (MCP2515).

Le deuxième nœud est équipé le moteur à courant continu (système de refroidissement) et le buzzer. Il reçoit les données de température et de distance envoyées par l'émetteur via le bus CAN.

Le troisième nœud sert d'interface utilisateur. Il utilise la LED qu'indique la surchauffe, l'écran LCD pour afficher la température ou la distance en fonction du mode sélectionné et les boutons poussoirs qui permettent de changer le mode. Les données traitées sont également transmises au deuxième nœud pour coordonner les actions.



Le bus CAN relie les trois nœuds en utilisant des terminators de **120 Ω** pour garantir une transmission de signal stable et réduire les interférences. Chaque nœud est équipé d'un microcontrôleur (**Arduino**) et d'un module CAN Shield (**MCP2515**), qui inclut un **transmetteur** (Transceiver) et un **contrôleur CAN**.



Le fonctionnement de cette architecture repose sur un échange constant de trames CAN entre les nœuds. Par exemple :

- Lorsque le capteur de température détecte une surchauffe, l'émetteur envoie une trame CAN contenant cette information.
- Le récepteur, en analysant cette trame, active immédiatement le moteur de refroidissement.
- Simultanément, le deuxième récepteur met à jour l'affichage pour avertir l'utilisateur.

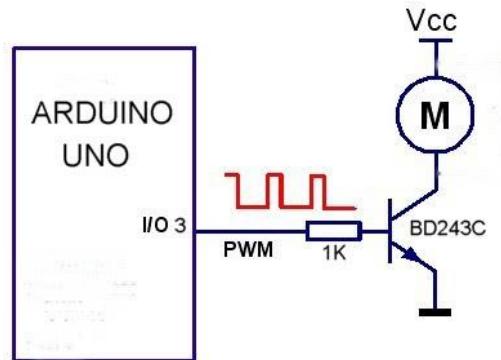
8. Partie réalisation :

Dans cette partie, nous décrivons le processus de mise en œuvre pratique de notre système. La réalisation se concentre sur l'assemblage des composants électroniques, leur connexion avec les modules Arduino, l'intégration du CAN Shield, et la mise en place des capteurs et actionneurs.

8.1. Commande du Moteur à Courant Continu :

Avant de commencer la réalisation de notre projet, il est essentiel de comprendre comment commander un moteur à courant continu. En effet, une carte Arduino ne peut fournir qu'un courant maximal de 40 mA, ce qui est insuffisant pour alimenter un moteur à courant continu, qui nécessite un courant bien supérieur. Pour résoudre ce problème, nous avons intégré un transistor NPN dans notre circuit. Ce composant agit comme un interrupteur électronique capable de fournir le courant nécessaire au moteur tout en étant contrôlé par l'Arduino. Grâce à cette configuration, nous pouvons facilement ajuster le fonctionnement du moteur en fonction des besoins de notre système.

Le choix du transistor NPN s'explique par sa capacité à connecter la borne positive du moteur à l'alimentation. Lorsque le signal de commande de l'Arduino est haut, le transistor se met en conduction, permettant ainsi au courant de traverser le moteur. Cette solution nous permet donc d'assurer un fonctionnement stable et efficace du moteur tout en protégeant notre Arduino d'un courant excessif.



Pour un moteur 5V, on connecte une borne du moteur directement à la broche **5V** de l'Arduino. L'autre borne du moteur est reliée au **collecteur** du transistor, tandis que l'**émetteur** du transistor est connecté à la masse commune (**GND**). La broche **PWM*** de l'Arduino est connectée à la base du transistor à travers une résistance de **1 kΩ**, qui limite le courant qui circule dans la base du transistor cela protège la sortie de l'Arduino contre un courant excessif. Enfin, la masse (**GND**) de l'Arduino est reliée à la masse commune.

- **PWM** : Une broche PWM (Pulse Width Modulation) sur une carte Arduino est une broche numérique capable de produire un signal modulé en largeur d'impulsion. Ce signal permet de simuler une sortie analogique tout en utilisant une sortie numérique.

8.2. Réalisation :

Après avoir clarifié les bases théoriques nécessaires, nous avons entamé la réalisation pratique de notre système.

Pour commencer, nous avons rassemblé les différents composants nécessaires, à savoir :

- Trois cartes Arduino.
- Trois modules CAN Shield (MCP2515).
- Deux boutons poussoirs.
- Un capteur de température (DHT).
- Un capteur ultrasonique (HC-SR04).
- Un moteur à courant continu (5V).
- Un afficheur LCD.
- Un buzzer et une LED

Pour les connexions ont été effectuées en suivant l'architecture que nous avons conçue. Chaque composant a été relié à la carte Arduino correspondante selon son rôle dans le système. Les

connexions du bus CAN (CAN High et CAN Low) ont été établies entre les différents CAN Shields pour garantir une communication fiable entre les nœuds.

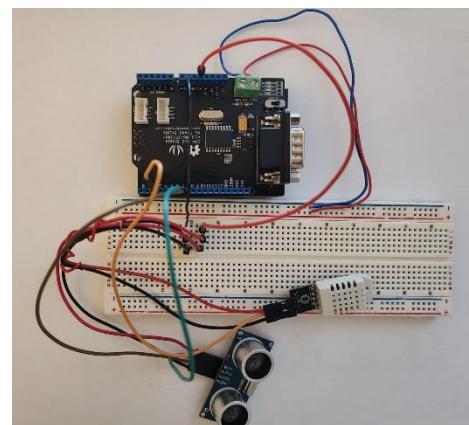
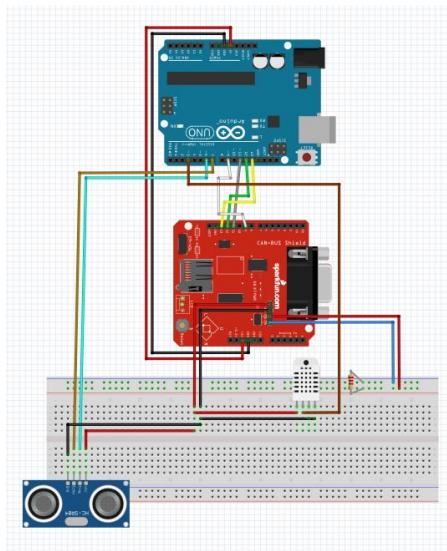
Pour le schéma de premier nœud :

Le **CAN Shield** est empilé directement sur la carte **Arduino**, permettant une communication via les broches **SPI** (MISO, MOSI, SCK) et la broche **CS** (broche 9), qui sont utilisées pour interagir avec le contrôleur **MCP2515** intégré au CAN Shield. Cette configuration est identique pour tous les nœuds du système.

Ensuite, les broches **5V** et **GND** de l'Arduino sont utilisées pour alimenter à la fois le CAN Shield et les capteurs connectés.

Les connexions spécifiques des capteurs sont les suivantes :

- Le **capteur DHT** est relié à la broche **3** de l'Arduino.
- Le **capteur ultrasonique HC-SR04** utilise la broche **6** pour le **Trigger** et la broche **7** pour l'**Echo**.

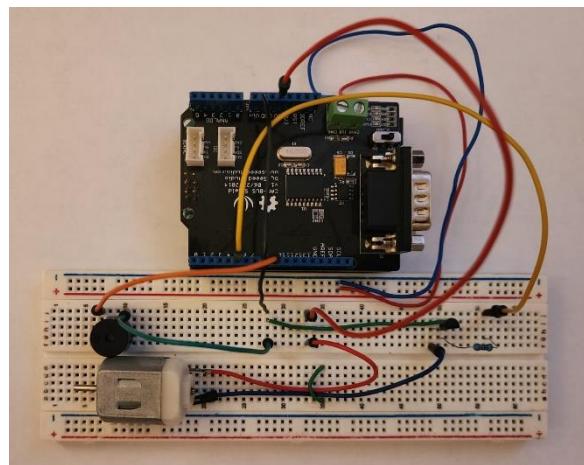
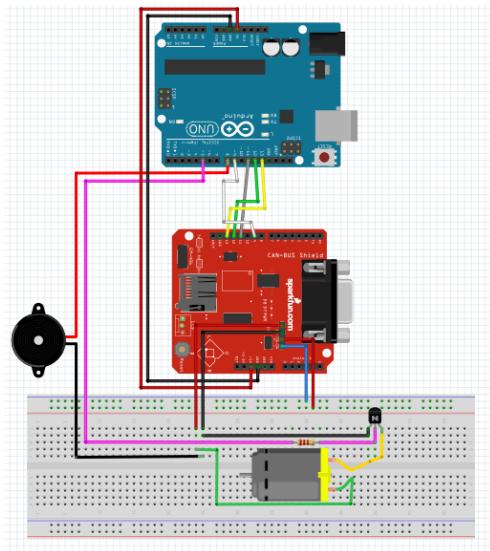


Ensuite, pour le deuxième nœud :

Le **moteur à courant continu (CC)** et le **buzzer** ont été ajoutés à ce nœud.

- Le **moteur CC** est connecté à la broche **PWM (broche 5)** de l'Arduino, permettant un contrôle précis de la vitesse.
- Le **buzzer** est relié à la broche **8**, permettant de produire des alertes sonores lorsque nécessaire.

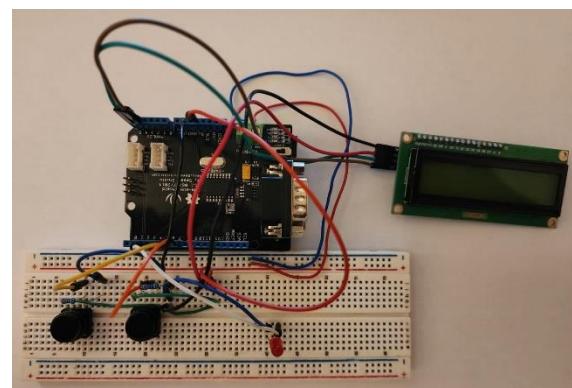
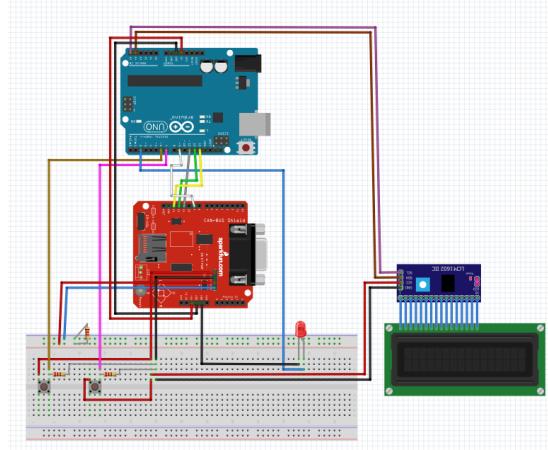
Ces composants reçoivent également leur alimentation via les broches **5V** et **GND** de l'Arduino.



Et pour le dernier nœud:

On a les deux boutons poussoirs (marche avant et marche arrière) qu'ils sont connecter a les broches 6 et 7 de l'Arduino, et la LED de surchauffe connectée à la broche 2.

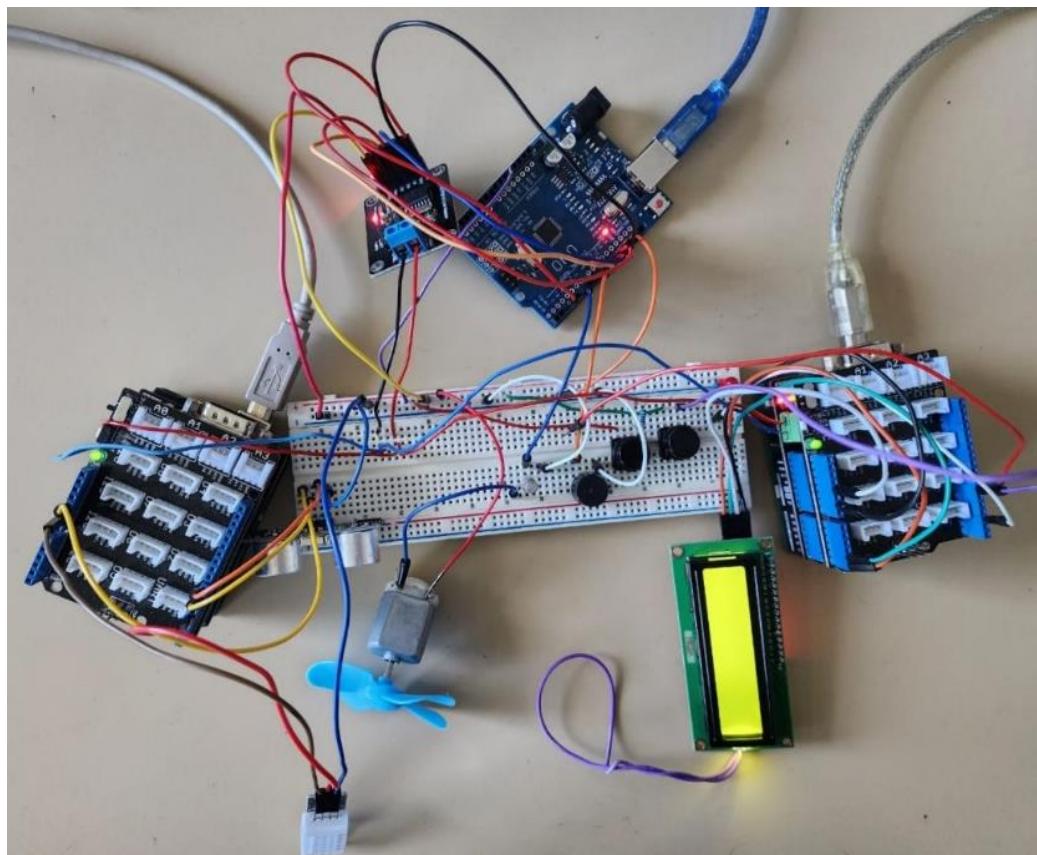
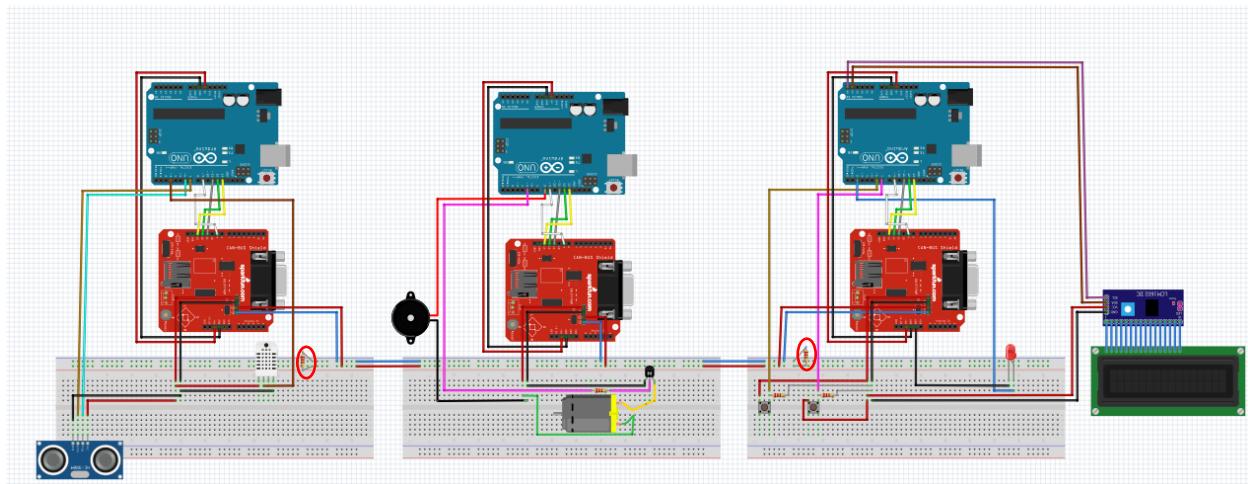
Pour le module LCD I2C, les connexions sont établies comme suit : la broche SDA est reliée à la broche A4 de l'Arduino, tandis que la broche SCL est connectée à la broche A5. Par ailleurs, les broches VCC et GND sont reliées respectivement aux broches 5V et GND de l'Arduino.



Interconnexion des Nœuds :

Après avoir expliqué le fonctionnement de chaque nœud individuellement, nous passons maintenant à l'interconnexion des différentes unités via le bus CAN. Ce réseau permet une communication bidirectionnelle rapide et fiable entre les modules. Chaque nœud est connecté aux lignes **CAN High (rouge)** et **CAN Low (bleu)**, assurant ainsi un échange de données structuré.

Des résistances de terminaison de **120Ω** sont placées aux extrémités du bus.



9. Explication des programmes :

Après avoir détaillé l'architecture matérielle et l'interconnexion des différents nœuds via le bus CAN, nous allons maintenant nous concentrer sur l'implémentation logicielle du système. Cette section vise à expliquer les différents codes programmés pour chaque unité, en mettant en évidence la gestion des capteurs, l'envoi et la réception des trames CAN, ainsi que le traitement des données pour assurer le bon fonctionnement du système embarqué. Nous détaillerons le rôle de chaque segment de code et son impact sur la communication et le contrôle des actions en temps réel.

9.1. Explication de l'utilisation de la fonction millis() dans nos codes :

Dans un système embarqué comme le nôtre, plusieurs processus doivent s'exécuter en parallèle (acquisition des capteurs, envoi de données, gestion de l'affichage, etc.). L'utilisation de **millis()** permet une exécution plus fluide et optimisée.

Nous avons utilisé la fonction **millis()** pour éviter le blocage du programme qui serait causé par **delay()**. Cette fonction permet de créer des temporisations sans interrompre l'exécution des autres tâches.

```
// Déclaration d'une variable pour stocker le temps du dernier déclenchement
unsigned long previousMillis = 0;

// Fonction permettant d'introduire une pause non bloquante avec millis()
void pause(unsigned long interval) {
    // Récupération du temps actuel en millisecondes
    unsigned long currentMillis = millis();

    // Vérification si l'intervalle spécifié s'est écoulé depuis la dernière mise à jour
    if (currentMillis - previousMillis >= interval) {
        // Mise à jour du temps du dernier déclenchement
        previousMillis = currentMillis;
    }
}
```

La fonction **millis()** est une fonction intégrée dans la programmation Arduino qui renvoie le nombre de millisecondes écoulées depuis le démarrage du programme. Elle est utilisée pour effectuer des mesures de temps et créer des retards sans bloquer l'exécution du reste du code. Dans nos programmes, la fonction **millis()** est utilisée pour gérer le temps entre certaines actions ou événements:

- La variable "**previousMillis**" est utilisée pour stocker le moment où une action particulière a été exécutée pour la dernière fois.
- Dans le bloc conditionnel (**if**), il y a une comparaison entre "**currentMillis**" (la valeur actuelle renvoyée par **millis()**) et la variable "**previousMillis**". Cette comparaison permet de vérifier si suffisamment de temps s'est écoulé depuis la dernière exécution d'une action spécifique.

- **Si le temps écoulé depuis la dernière exécution de l'action est supérieur ou égal à la valeur spécifiée ("interval"), alors l'action correspondante est considérée comme devant être exécutée à nouveau, et la variable "previousMillis" est mise à jour avec le temps actuel.**
- **En utilisant millis() de cette manière, le programme peut effectuer des tâches à des intervalles spécifiques sans bloquer l'exécution du reste du code, contrairement à delay().** Cela permet une meilleure réactivité et une utilisation plus efficace des ressources de la carte Arduino.

9.2. Programme de première Nœud :

```

emeteur

// Émetteur - Envoi des données de température et de distance via le bus CAN
#include <SPI.h>          // Bibliothèque pour la communication SPI
#include <mcp2515.h>        // Bibliothèque pour la gestion du contrôleur CAN MCP2515
#include <DHT.h>            // Bibliothèque pour la gestion du capteur DHT11

#define DHTPIN 3           // Définition de la broche utilisée pour le capteur DHT11
#define DHTTYPE DHT11       // Définition du type de capteur utilisé
DHT dht(DHTPIN, DHTTYPE); // Initialisation du capteur DHT11

int trigPin = 6; // Définition de la broche du capteur ultrasonique pour le déclenchement
int echoPin = 7; // Définition de la broche du capteur ultrasonique pour la réception
int distance;    // Variable pour stocker la distance mesurée

struct can_frame can_msg; // Déclaration d'une trame CAN pour l'envoi des données

MCP2515 mcp2515(9); // Initialisation du module CAN MCP2515 sur la broche CS (9)

void setup() {
    dht.begin();           // Initialisation du capteur DHT11
    SPI.begin();           // Initialisation de la communication SPI

    mcp2515.reset();      // Réinitialisation du module CAN
    mcp2515.setBitrate(CAN_100KBPS, MCP_8MHZ); // Configuration du débit CAN à 100 kbps
    mcp2515.setNormalMode(); // Activation du mode normal de communication CAN

    pinMode(trigPin, OUTPUT); // Configuration de la broche du capteur ultrasonique en sortie
    pinMode(echoPin, INPUT); // Configuration de la broche du capteur ultrasonique en entrée
}

void loop() {
    // Envoi d'une impulsion au capteur ultrasonique
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10); // Attente de 10 µs
    digitalWrite(trigPin, LOW);

    // Mesure du temps de retour de l'écho
    int pingTravelTime = pulseIn(echoPin, HIGH);
    // Conversion du temps de retour en distance (en cm)
    distance = (pingTravelTime * 0.0343) / 2;

    // Lecture de la température à partir du capteur DHT11
    int temperature = dht.readTemperature();

    // Construction du message CAN
    can_msg.can_id = 0xFF; // Identifiant de la trame CAN
    can_msg.can_dlc = 3;   // Définition de la taille des données (3 octets)
    can_msg.data[0] = temperature; // Stockage de la température dans la première case du tableau de données
    can_msg.data[1] = distance & 0xFF; // Stockage de l'octet de poids faible de la distance
    can_msg.data[2] = (distance >> 8) & 0xFF; // Stockage de l'octet de poids fort de la distance

    // Envoi du message via le bus CAN
    mcp2515.sendMessage(&can_msg);
}

```

9.3. Explication de programme de première Nœud :

On a commencé par inclure les bibliothèques nécessaires pour la gestion du capteur **DHT11**, du module **CAN MCP2515** et de la communication **SPI**.

Puis, on a défini les broches utilisées pour le capteur de température **DHT11** et le capteur ultrasonique **HC-SR04**. Une variable **distance** est déclarée pour stocker la distance mesurée par le capteur ultrasonique.

Ensuite, une structure **can_frame (can_msg)** est déclarée pour stocker les données qui seront envoyées sur le bus CAN. Enfin, on a procédé à l'initialisation du module **MCP2515** sur la broche **CS (9)**.

- la fonction **setup()** :

Pour la configuration du système dans **setup()** on a initialisé le capteur **DHT11** avec la fonction **dht.begin()**, puis activé la communication **SPI** avec **SPI.begin()**. Ensuite, on a réinitialisé le module **CAN** avec **mcp2515.reset()**, puis on a configuré son débit à **100 kbps** et activé son mode normal de communication. Enfin, on a défini **trigPin** comme **sortie** pour générer des impulsions et **echoPin** comme **entrée** pour capter le signal réfléchi par un obstacle.

- la boucle principale **loop()** :

Dans la fonction **loop()** on a commencé par envoyer une impulsion de **10 µs** sur **trigPin** pour déclencher la mesure du capteur ultrasonique. Ensuite, on a mesuré le temps d'aller-retour du signal avec **pulseIn()**, puis on l'a converti en distance à l'aide de la formule **(pingTravelTime * 0.0343) / 2** où **pingTravelTime** est la variable qui stocke la durée en microsecondes du trajet aller-retour de l'onde ultrasonique. Par la suite, on a récupéré la température du capteur **DHT11** avec **dht.readTemperature()**.

Ensuite, on a stocké les valeurs mesurées dans la trame CAN **can_msg**. L'identifiant CAN (**can_id**) a été défini à **0xFF**, ce qui permet aux récepteurs de reconnaître cette trame comme une donnée de capteur. Le champ **data** de la trame CAN contient trois octets :

- **data[0]** stocke directement la température mesurée.
- **data[1]** stocke l'octet de poids faible de la distance, obtenu avec **distance & 0xFF**.
- **data[2]** stocke l'octet de poids fort de la distance, obtenu avec **(distance >> 8) & 0xFF**.

L'encodage de la distance sur deux octets permet d'envoyer des valeurs supérieures à 255. Enfin, on a envoyé cette trame sur le bus CAN en utilisant **mcp2515.sendMessage(&can_msg)**.

- Conversion du temps d'écho en distance :

Le capteur ultrasonique HC-SR04 fonctionne en envoyant une onde sonore à l'aide de la broche **trigPin** et en écoutant son écho avec la broche **echoPin**. La durée entre l'émission et la

réception du signal est stockée dans la variable **pingTravelTime**, qui représente le temps aller-retour du son en microsecondes (μs).

Pour convertir ce temps en distance, nous utilisons la formule suivante :

$$\text{distance} = \frac{\text{pingTravelTime} \times 0.0343}{2}$$

- 0.0343 correspond à la vitesse du son dans l'air (343 m/s) convertie en cm/ μs .
- On divise par 2 car pingTravelTime représente le temps aller-retour, or nous avons besoin uniquement du trajet aller.

- **Encodage et stockage de la distance dans la trame CAN**

Une trame CAN permet d'envoyer des données sous forme d'un tableau de 8 octets (**data[8]**). Cependant, la variable distance est un entier qui peut dépasser 255, or un octet ne peut stocker que des valeurs de 0 à 255. Pour contourner cette limite, on décompose la valeur de la distance en deux octets :

- **L'octet de poids faible (LSB - Least Significant Byte) :**

$$\text{data}[1] = \text{distance} \& 0xFF$$

Cette opération extrait les 8 bits de droite de la variable distance pour les stocker dans **data[1]**.

- **L'octet de poids fort (MSB - Most Significant Byte) :**

$$\text{data}[2] = (\text{distance} \gg 8) \& 0xFF$$

Ici, on effectue un décalage de 8 bits vers la droite ($>> 8$), ce qui ramène les 8 bits de gauche dans la position du poids faible. Ensuite, on applique $\& 0xFF$ (comme un filtre) pour ne garder que ces 8 bits.

9.4. Programme de deuxième Nœud :

```

resepteur1

#include <SPI.h>           // Bibliothèque pour la communication SPI
#include <mcp2515.h>         // Bibliothèque pour gérer le module CAN MCP2515

#define PIN_MOTEUR 5        // Broche de commande du moteur
#define BUZZER 8             // Broche de commande du buzzer

struct can_frame canMsg1; // Trame contenant température et distance
struct can_frame canMsg2; // Trame contenant l'état de la marche arrière

bool c = 0;    // Variable pour stocker l'état de la marche arrière (0 = marche avant, 1 = marche arrière)

MCP2515 mcp2515(9); // Initialisation du module CAM MCP2515 sur la broche CS (9)

void setup() {
    SPI.begin(); // Initialisation de la communication SPI

    // Configuration du module MCP2515 (CAN)
    mcp2515.reset();                      // Réinitialisation du module
    mcp2515.setBitrate(CAN_100KBPS, MCP_8MHZ); // Configuration du débit du bus CAN (100 kbps, quartz 8 MHz)
    mcp2515.setNormalMode();               // Activation du mode normal (communication active)

    pinMode(PIN_MOTEUR, OUTPUT); // Définition de la broche moteur en sortie
    pinMode(BUZZER, OUTPUT);    // Définition de la broche buzzer en sortie
}

// Fonction pour activer le buzzer avec une temporisation non bloquante
void activerBuzzer(int interval) {
    static unsigned long previousMillis = 0; // Variable pour mémoriser le dernier déclenchement
    static bool buzzerState = false; // État du buzzer (éteint ou allumé)

    unsigned long currentMillis = millis(); // Récupération du temps actuel en millisecondes

    if (currentMillis - previousMillis >= interval) { // Vérification si l'intervalle est écoulé
        previousMillis = currentMillis; // Mise à jour du temps du dernier déclenchement

        // Inversion de l'état du buzzer
        buzzerState = !buzzerState;
        digitalWrite(BUZZER, buzzerState); // Mise à jour de l'état du buzzer
    }
}

void loop() {
    // Lecture de la trame CAN pour récupérer l'état de la marche arrière ('c')
    if (mcp2515.readMessage(&canMsg2) == MCP2515::ERROR_OK) { // Si une trame est reçue sans erreur
        if (canMsg2.can_id == 0xEE) { // Vérification que l'ID correspond à la trame contenant 'c'
            c = canMsg2.data[0]; // Récupération de la valeur de 'c'
        }
    }
}

```

```

// Lecture de la trame CAN pour récupérer la température et la distance
if (mcp2515.readMessage(&canMsg1) == MCP2515::ERROR_OK) { // Si une trame est reçue sans erreur
    if (camMsg1.can_id == 0xFF) { // Vérification que l'ID correspond à la trame contenant température et distance
        int valeurTP = canMsg1.data[0]; // Récupération de la température
        int distance = canMsg1.data[1] + (canMsg1.data[2] << 8); // Reconstruction de la distance en combinant les deux octets

        // Contrôle du moteur en fonction de la température
        if (valeurTP >= 20 && valeurTP < 25) {
            analogWrite(PIN_MOTEUR, 100); // Le moteur tourne à une vitesse moyenne
        } else if (valeurTP >= 25) {
            analogWrite(PIN_MOTEUR, 255); // Le moteur tourne à pleine vitesse
        } else {
            analogWrite(PIN_MOTEUR, LOW); // Le moteur est éteint si la température est inférieure à 20°C
        }

        // Contrôle du buzzer en fonction de la distance et de `c`
        if (c == 0) { // Si la voiture est en marche avant
            digitalWrite(BUZZER, LOW); // Désactivation du buzzer
        } else if (c == 1) { // Si la voiture est en marche arrière
            if (distance > 150) {
                activerBuzzer(420); // Son intermittent lent si distance > 150 cm
            } else if (distance <= 150 && distance > 110) {
                activerBuzzer(350); // Son un peu plus rapide entre 110 et 150 cm
            } else if (distance <= 110 && distance > 10) {
                activerBuzzer(distance * 2.5); // Fréquence du buzzer proportionnelle à la distance
            } else if (distance <= 10) {
                digitalWrite(BUZZER, HIGH); // Avertissement continu si distance < 10 cm
            }
        }
    }
}
}

```

9.5. Explication de programme de deuxième Nœud :

On a commencé par inclure les bibliothèques nécessaires pour la communication SPI et le module CAN **MCP2515**. Ensuite, on a défini les broches utilisées pour le **moteur** et le **buzzer**.

Deux structures **can_frame** ont été déclarées pour stocker les trames CAN reçues :

- **canMsg1** pour la température et la distance (trame envoyé par le premier nœud).
 - **canMsg2** pour l'état de la marche arrière **c** (trame envoyé par le deuxième nœud).

Le module **MCP2515** a été initialisé sur la broche **CS (9)**.

- La fonction setup() :

Dans la fonction `setup()`, nous avons commencé par **initialiser la communication SPI** en appelant `SPI.begin()`, ce qui permet la communication entre la carte et le module **MCP2515**. Ensuite, nous avons **réinitialisé le module CAN** avec `mcp2515.reset()`, puis configuré son **débit à 100 kbps** en utilisant `mcp2515.setBitrate(CAN_100KBPS, MCP_8MHZ)`. Une fois ces réglages appliqués, nous avons activé le mode normal de communication avec `mcp2515.setNormalMode()`, ce qui permet au module de fonctionner en mode standard et d'envoyer ou recevoir des messages CAN normalement. Après cela, nous avons défini les broches du moteur et le buzzer en sortie.

- Boucle principale loop() :

Avant d'exploiter les données reçues, nous effectuons une **vérification des trames** :

- Nous lisons les trames CAN et vérifions si elles ont été correctement reçues (**sans erreur**).
- Ensuite, nous nous assurons que l'**identifiant (ID) du message** correspond bien à la trame attendue avant de traiter les données.

Nous avons deux parties principales :

1. Lecture de la trame contenant c

- On récupère l'**ID 0xEE**, qui contient l'état de la marche arrière (**c**).
- Si **c == 0**, la voiture est en marche avant.
- Si **c == 1**, la voiture est en marche arrière.

2. Lecture de la trame contenant la température et la distance

- L'**ID 0xFF** contient la température et la distance.
- La température est stockée directement dans **data[0]**.
- La distance est encodée sur **deux octets** :
 - **data[1]** contient l'octet de poids faible.
 - **data[2]** contient l'octet de poids fort.
- On a reconstruit la valeur de la distance en combinant ces deux octets avec :

```
int distance = canMsg1.data[1] + (canMsg1.data[2] << 8)
```

Cette opération décale **data[2]** de 8 bits vers la gauche et l'ajoute à **data[1]** pour obtenir la valeur complète.

Après la réception **d'une trame valide** (après la vérification de l'**ID** et des données), nous contrôlons le **moteur** selon la température reçue :

- Si la **Température < 20°C** → Le moteur est éteint.
- Si la **Température entre 20°C et 25°C** → Le moteur tourne à vitesse moyenne (**analogWrite(PIN_MOTEUR, 100)**).
- Si la **Température ≥ 25°C** → Le moteur tourne à pleine vitesse (**analogWrite(PIN_MOTEUR, 255)**).

Le **buzzer** est allumé à l'aide de la valeur de **c**, reçue par la **trame envoyée par le troisième nœud**.

Si **c == 1** (marche arrière activée), le buzzer est contrôlé en fonction de la distance reçue par le **premier nœud** :

- **Distance > 150 cm** → Son intermittent lent (420 ms).
- **Distance entre 110 et 150 cm** → Son plus rapide (350 ms).
- **Distance entre 10 et 110 cm** → Fréquence du buzzer proportionnelle à la distance (**distance * 2.5**).
- **Distance ≤ 10 cm** → Buzzer allumé en continu (alerte critique).

La fonction **activerBuzzer(interval)** permet de générer un son intermittent **sans bloquer** l'exécution du programme (la fonction `millis()`).

9.6. Programme de troisième Nœud :

```

resepteur2

#include <SPI.h>           // Bibliothèque pour la communication SPI
#include <mcp2515.h>         // Bibliothèque pour la gestion du module CAN MCP2515
#include <LiquidCrystal_I2C.h> // Bibliothèque pour l'affichage LCD I2C

#define LED 2                 // Broche de la LED
#define BOUTON_ARRIERE 6      // Broche du bouton de marche arrière
#define BOUTON_MARCHE 7       // Broche du bouton de marche avant

struct can_frame canMsg1; // Trame contenant la température et la distance (reçue)
struct can_frame canMsg2; // Trame contenant l'état de la marche arrière (envoyée)

int etat_bouton_arriere = 0; // État du bouton de marche arrière
int etat_bouton_marche = 0; // État du bouton de marche avant
bool c = 0;                // Variable indiquant l'état de la marche arrière

MCP2515 MCP2515();
unsigned long previousMillis = 0; // Variable pour la gestion du temps avec millis()
/***
 * Fonction pour gérer une pause non bloquante avec millis()
 * Permet de temporiser sans bloquer l'exécution du programme (évite delay())
 */
void pause(unsigned long interval) {
    unsigned long currentMillis = millis();
    if (currentMillis - previousMillis >= interval) {
        previousMillis = currentMillis;
    }
}

void setup() {
    SPI.begin();           // Initialisation de la communication SPI
    lcd.init();            // Initialisation de l'affichage LCD
    lcd.backlight();       // Activation du rétroéclairage du LCD

    mcp2515.reset();      // Réinitialisation du module MCP2515
    mcp2515.setBitrate(CAN_100KBPS, MCP_8MHZ); // Configuration du débit CAN à 100 kbps
    mcp2515.setNormalMode(); // Activation du mode de communication normal du MCP2515

    pinMode(LED, OUTPUT); // Définition de la LED comme sortie
    pinMode(BOUTON_ARRIERE, INPUT); // Définition du bouton de marche arrière comme entrée
    pinMode(BOUTON_MARCHE, INPUT); // Définition du bouton de marche avant comme entrée
}

void loop() {
    // Vérification et lecture d'une trame CAN
    if (mcp2515.readMessage(&canMsg1) == MCP2515::ERROR_OK) {
        // Vérification de l'ID de la trame reçue
        if (canMsg1.can_id == 0xFF) {
            int temperature = canMsg1.data[0]; // Récupération de la température depuis data[0]
            // Reconstruction de la distance en combinant les octets de poids faible et fort
            int distance = canMsg1.data[1] + (canMsg1.data[2] << 8);
        }
    }
}

```

```

// Si la voiture est en marche avant, afficher la température moteur
if (c == 0) {
    lcd.clear(); // Efface l'affichage précédent
    lcd.setCursor(1, 0);
    lcd.print("Temperature de");

    lcd.setCursor(0, 1);
    lcd.print("moteur: ");
    lcd.print(temperature);
    lcd.print(" °C");
    pause(500); // Pause pour permettre l'affichage sans bloquer le programme

    // Allumer la LED si la température dépasse 30°C
    if (temperature > 30) digitalWrite(LED, HIGH);
    else digitalWrite(LED, LOW);
}

// Si la voiture est en marche arrière, afficher la distance d'un obstacle
else if (c == 1) {
    lcd.clear(); // Efface l'affichage précédent
    lcd.setCursor(1, 0);
    lcd.print("Obstacle a une");

    // Si distance >= 100 cm (1 mètre)
    lcd.setCursor(0, 1);
    if (distance >= 100) {
        float const_distance = distance / 100.0; // Conversion en mètres
        lcd.print("distance: ");
        lcd.print(const_distance);
        lcd.print(" m");
    }

    // Si la distance < a 1 mètre, afficher en centimètres
    else {
        lcd.print("distance: ");
        lcd.print(distance);
        lcd.print(" cm");
    }
    pause(500); // Pause pour l'affichage avant rafraîchissement
}
}

// Lecture des boutons pour définir l'état de la marche avant ou arrière
etat_bouton_arriere = digitalRead(BOUTON_ARRIERE); // Lire l'état du bouton arrière
etat_bouton_marche = digitalRead(BOUTON_MARCHE); // Lire l'état du bouton marche avant

// Mise à jour de l'état de la marche arrière en fonction des boutons
if (etat_bouton_arriere == HIGH) c = 1; // Si bouton arrière appuyé, activer marche arrière
if (etat_bouton_marche == HIGH) c = 0; // Si bouton marche avant appuyé, désactiver marche arrière

// Préparation et envoi d'une trame CAN contenant l'état de la marche arrière
canMsg2.can_id = 0xEE; // Définition de l'ID de la trame (identifiant unique)
canMsg2.can_dlc = 1; // Définition de la longueur des données (1 octet)
canMsg2.data[0] = c; // Stockage de l'état de la marche arrière dans la trame
mcp2515.sendMessage(&canMsg2); // Envoi de la trame CAN
}

```

9.7. Explication de programme de troisième Nœud :

Nous avons commencé par inclure les bibliothèques nécessaires pour la communication SPI et le module CAN MCP2515, ainsi que la bibliothèque LiquidCrystal_I2C pour gérer l'affichage LCD. Ensuite, nous avons défini les broches utilisées pour la LED et les boutons de commande de la boîte de vitesses (les boutons poussoirs).

Deux structures **can_frame** ont été déclarées pour stocker les trames CAN reçues et envoyées :

- **canMsg1** pour la température et la distance (trame envoyée par le premier nœud).
- **canMsg2** pour l'état de la marche arrière c (trame envoyée par ce nœud vers les autres).

Le module MCP2515 a été initialisé sur la broche CS (9) et l'écran LCD a été configuré pour l'affichage des informations.

- La fonction **setup()** :

Dans la fonction **setup()**, nous avons commencé par initialiser la communication **SPI** en appelant **SPI.begin()**, ce qui permet à la carte de communiquer avec le module **MCP2515**. Ensuite, nous avons réinitialisé le module CAN avec **mcp2515.reset()**, puis configuré son débit à **100 kbps** en utilisant **mcp2515.setBitrate(CAN_100KBPS, MCP_8MHZ)**.

Une fois ces réglages appliqués, nous avons activé le mode **normal** de communication avec **mcp2515.setNormalMode()**, ce qui permet au module de fonctionner en mode standard et d'envoyer ou recevoir des messages CAN normalement.

Ensuite, nous avons initialisé l'écran LCD avec **Lcd.init()**, activé son **rétroéclairage**, puis défini les broches de la **LED et des boutons** en entrée/sortie.

- Boucle principale **loop()** :

Avant d'exploiter les données reçues, nous effectuons une **vérification des trames CAN** :

- Nous **lisons les trames CAN** et vérifions si elles ont été correctement reçues (sans erreur).
- Ensuite, nous nous assurons que **l'ID du message** correspond bien à la trame attendue avant de traiter les données.

Nous avons **trois parties principales** dans la boucle principale :

1. Lecture de la trame contenant la température et la distance

- L'ID **0xFF** est utilisé pour la trame contenant la température et la distance.
- La température est stockée directement dans **data[0]**.
- La distance est encodée sur **deux octets** :

- **data[1]** contient l'octet de poids faible.
- **data[2]** contient l'octet de poids fort.

Nous reconstruisons la distance en combinant ces deux octets avec la formule suivante :

```
int distance = canMsg1.data[1] + (canMsg1.data[2] << 8)
```

2. Affichage des informations sur l'écran LCD

- **Si la voiture est en marche avant (c == 0)**
 - On affiche la température moteur sur le LCD.
 - Une LED d'alerte est allumée si la température dépasse 30°C.
 - Sinon, la LED reste éteinte.
- **Si la voiture est en marche arrière (c == 1)**
 - On affiche la distance d'un obstacle sur le LCD.
 - Si la distance est supérieure ou égale à 100 cm, elle est convertie en mètres (**distance / 100.0**).
 - Sinon, elle est affichée directement en centimètres.

3. Lecture des boutons et mise à jour de la marche arrière

Nous avons défini deux boutons pour simuler la boîte de vitesses de la voiture :

- **Bouton "Marche arrière"** : active la marche arrière (c = 1).
- **Bouton "Marche avant"** : désactive la marche arrière (c = 0).

Ces boutons sont lus en permanence et **l'état c est mis à jour** en fonction de leur état (HIGH ou LOW).

Une fois la mise à jour effectuée, nous préparons une **trame CAN** contenant l'état de la marche arrière :

- L'**ID** utilisé pour cette trame est **0xEE**.
- Nous remplissons la trame avec **1 octet** contenant c.
- Enfin, nous envoyons la trame avec `mcp2515.sendMessage(&canMsg2)`.

Cela permet aux **autres nœuds du réseau CAN** d'être informés de l'état de la marche arrière et d'agir en conséquence (activation du buzzer).

- Gestion de la fonction pause() :

Nous avons défini une fonction **pause()** pour introduire un délai dans l'affichage LCD sans bloquer l'exécution du programme.

Cette fonction utilise **millis()** ce qui permet de continuer à recevoir et envoyer des messages CAN en arrière-plan.

10. Visualisation des Trames sur le Réseau CAN :

Dans notre système, deux trames principales circulent sur le réseau CAN :

1. Trame envoyée par l'unité 1 (Noeud 1)
 - Contient la température du moteur et la distance détectée par le capteur ultrasonique.
2. Trame envoyée par l'unité 3 (Noeud 3)
 - Contient l'état des boutons simulant la boîte de vitesses (indiquant si le mode marche arrière est activé ou non).

Traitement des Trames par l'Unité 1 :

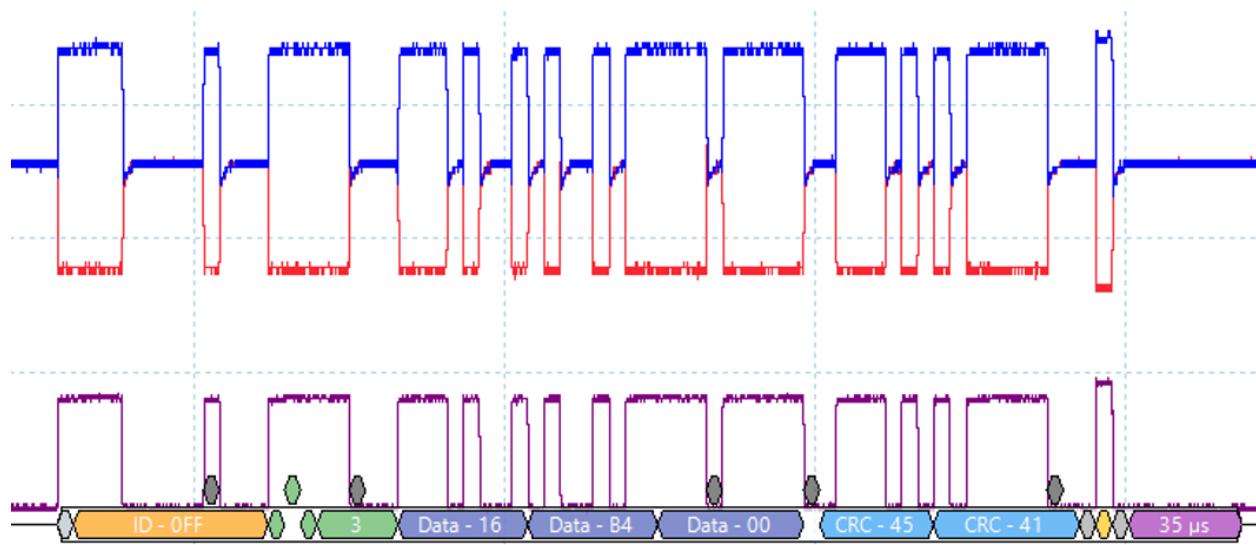
L'unité 1 joue un rôle clé dans l'acquisition des données et leur transmission sur le réseau CAN.

- Elle mesure la température du moteur à l'aide du capteur DHT.
- Elle détecte la distance avec un capteur ultrasonique.
- Elle envoie ces deux valeurs sous forme de trame CAN avec un ID spécifique (ex : 0xFF)

```
// Construction du message CAN
can_msg.can_id = 0xFF; // Identifiant de la trame CAN
can_msg.can_dlc = 3; // Définition de la taille des données (3 octets)
can_msg.data[0] = temperature; // Stockage de la température dans la première case du tableau de données
can_msg.data[1] = distance & 0xFF; // Stockage de l'octet de poids faible de la distance
can_msg.data[2] = (distance >> 8) & 0xFF; // Stockage de l'octet de poids fort de la distance

// Envoi du message via le bus CAN
mcp2515.sendMessage(&can_msg);
```

Si on analyse la visualisation de la trame suivante, on remarque plusieurs éléments importants :



- L'ID de la trame est le **même** pour les transmissions concernées. Cela signifie que toutes les unités qui reçoivent cette trame doivent être capables de l'interpréter correctement en fonction de son identifiant.
- Le champ "Data" contient trois valeurs distinctes. Ces valeurs correspondent aux données envoyées par l'unité 1 (ou une autre unité émettrice) :
 1. La **température du moteur** mesurée par le capteur DHT (22 C°).
 2. Et deux valeur pour la **distance détectée** par le capteur ultrasonique (**180 + 0**)cm.

Traitement des Trames par l'Unité 3 :

L'unité 3 reçoit la trame envoyée par l'unité 1, qui contient la température et la distance. Avant d'exploiter ces données, elle vérifie si l'ID du message reçu correspond bien à celui de l'unité 1 (ID = 0xFF).

```
// Vérification et lecture d'une trame CAN
if (mcp2515.readMessage(&canMsg1) == MCP2515::ERROR_OK) {
    // Vérification de l'ID de la trame reçue
    if (canMsg1.can_id == 0xFF) {
        int temperature = canMsg1.data[0]; // Récupération de la température depuis data[0]
        // Reconstruction de la distance en combinant les octets de poids faible et fort
        int distance = canMsg1.data[1] + (canMsg1.data[2] << 8);
```

Si l'identification est correcte :

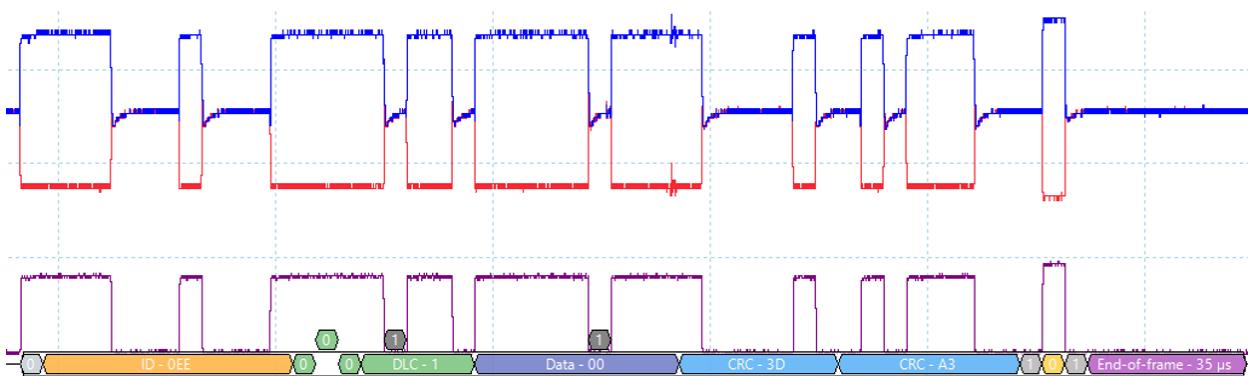
- Elle affiche la température et la distance sur l'écran LCD.
- Elle allume une LED si la température dépasse un seuil prédéfini.
- Elle affiche la distance uniquement si le mode marche arrière est activé.

En parallèle, l'unité 3 envoie l'état des boutons (mode marche arrière activé ou non) sous forme d'une autre trame CAN sous l'ID de 0xEE.

```
// Préparation et envoi d'une trame CAN contenant l'état de la marche arrière
canMsg2.can_id = 0xEE; // Définition de l'ID de la trame (identifiant unique)
canMsg2.can_dlc = 1; // Définition de la longueur des données (1 octet)
canMsg2.data[0] = c; // Stockage de l'état de la marche arrière dans la trame
mcp2515.sendMessage(&canMsg2); // Envoi de la trame CAN
```

On analyse la visualisation de la trame envoyée, on remarque que :

- L'ID de la trame est le **même** pour les transmissions concernées (0xEE).
- le champ "Data" indique l'état de bouton pour le mode marche arrière(ici il est à 0).



Traitement des Trames par l'Unité 2 :

L'unité 2 reçoit les **deux** trames envoyées par l'unité 1 et l'unité 3.

- Elle vérifie les ID de chaque trame afin de récupérer les bonnes informations.

```
// Lecture de la trame CAN pour récupérer l'état de la marche arrière (`c`)
if (mcp2515.readMessage(&canMsg2) == MCP2515::ERROR_OK) { // Si une trame est reçue sans erreur
    if (canMsg2.can_id == 0xEE) { // Vérification que l'ID correspond à la trame contenant `c`
        c = canMsg2.data[0]; // Récupération de la valeur de `c`
    }
}

// Lecture de la trame CAN pour récupérer la température et la distance
if (mcp2515.readMessage(&canMsg1) == MCP2515::ERROR_OK) { // Si une trame est reçue sans erreur
    if (canMsg1.can_id == 0xFF) { // Vérification que l'ID correspond à la trame contenant température et distance
        int valeurTP = canMsg1.data[0]; // Récupération de la température
        int distance = canMsg1.data[1] + (canMsg1.data[2] << 8); // Reconstruction de la distance en combinant les deux octets
    }
}
```

- Elle récupère l'état de la marche arrière pour déclencher un buzzer lorsque ce mode est activé.
- Elle utilise la température du moteur pour activer un moteur à courant continu servant de système de refroidissement.

- Elle ajuste l'intensité du buzzer en fonction de la distance détectée, afin d'avertir le conducteur de la proximité d'un obstacle.

11. Conclusion :

Ce chapitre a permis de présenter la conception et le déploiement d'un système embarqué basé sur le bus CAN, en mettant l'accent sur l'intégration des différents composants matériels et logiciels. À travers l'utilisation de cartes **Arduino**, du **module CAN MCP2515**, et de capteurs tels que le **DHT11** et le **HC-SR04**, nous avons pu mettre en place un système fonctionnel capable de collecter, transmettre et traiter des données en temps réel. L'architecture du système, décrite en détail, montre comment les différentes unités interagissent via le bus CAN pour assurer une communication fiable et efficace.

La réalisation pratique, incluant le câblage et l'implémentation du code source, a permis de concrétiser les concepts théoriques abordés dans les chapitres précédents. Les explications détaillées du code, notamment l'utilisation de la fonction **millis()** pour gérer les temporisations sans bloquer l'exécution du programme, illustrent l'optimisation des ressources et la réactivité du système.

CONCLUSION GENERALE

Ce projet nous a permis d'approfondir notre compréhension du bus CAN, un protocole de communication essentiel dans les systèmes embarqués. À travers une étude détaillée de son architecture, de ses mécanismes de transmission et de ses différentes applications, nous avons pu explorer son intégration dans des systèmes industriels et automobiles.

L'implémentation du bus CAN dans le Véhicule Multiplexé Didactique (VMD) a été une étape clé, mettant en avant l'importance de la communication entre les différents modules via la carte EID210 et la carte ATON CAN. Nous avons analysé et configuré les trames CAN pour assurer un échange efficace des informations entre les différents composants, garantissant ainsi un contrôle optimal du système.

En complément, la conception et la réalisation d'un système embarqué basé sur le bus CAN ont renforcé nos compétences en programmation, en configuration matérielle et en gestion des protocoles de communication. L'intégration de plusieurs capteurs (DHT11, HC-SR04), d'un moteur à courant continu, d'un écran LCD et d'un buzzer a permis de mettre en place une architecture fonctionnelle capable de traiter et de réagir aux informations en temps réel.

Ce projet nous a offert une expérience complète, combinant théorie et pratique, et nous a permis d'acquérir une expertise précieuse en systèmes embarqués et en automatisation. Pour aller plus loin, des perspectives d'amélioration pourraient inclure l'optimisation de la gestion des priorités des trames CAN, l'intégration de nouvelles technologies pour une meilleure supervision des systèmes embarqués dans les communications industrielles.

Sources d'informations et Références techniques :

Réponse	ADDR1 (m)	ADDR2 (m)	ADDR3 (m)	ADDR4 (m)	Identificateur	Labels définis dans le filtre
Nom "Commandes feu"						
E001 -> E001	0000 0000	000 100	000 000	000 000	E001 ou se	E001_RSI_Condition_Fine
E001 -> E002	0000 0000	000 100	000 000	000 000	E001 ou se	E001_RSI_Condition_Fine
E001 -> E003	0000 0000	000 100	000 000	000 000	E001 ou se	E001_RSI_Condition_Fine
E001 -> E004	0000 0000	000 100	000 000	000 000	E001 ou se	E001_RSI_Condition_Fine
E001 -> E005	0000 0000	000 100	000 000	000 000	E001 ou se	E001_RSI_Condition_Fine
Nom "Feu avant gauche"						
E002 -> E001	0000 0000	000 100	000 000	000 000	E002 ou se	E002_RSI_Condition_Fine
E002 -> E002	0000 0000	000 100	000 000	000 000	E002 ou se	E002_RSI_Condition_Fine
E002 -> E003	0000 0000	000 100	000 000	000 000	E002 ou se	E002_RSI_Condition_Fine
E002 -> E004	0000 0000	000 100	000 000	000 000	E002 ou se	E002_RSI_Condition_Fine
E002 -> E005	0000 0000	000 100	000 000	000 000	E002 ou se	E002_RSI_Condition_Fine
Nom "Feu avant droit"						
E003 -> E001	0000 0000	000 100	000 000	000 000	E003 ou se	E003_RSI_Condition_Fine
E003 -> E002	0000 0000	000 100	000 000	000 000	E003 ou se	E003_RSI_Condition_Fine
E003 -> E003	0000 0000	000 100	000 000	000 000	E003 ou se	E003_RSI_Condition_Fine
E003 -> E004	0000 0000	000 100	000 000	000 000	E003 ou se	E003_RSI_Condition_Fine
E003 -> E005	0000 0000	000 100	000 000	000 000	E003 ou se	E003_RSI_Condition_Fine
Nom "Feu arrière gauche"						
E004 -> E001	0000 0000	000 100	000 000	000 000	E004 ou se	E004_RSI_Condition_Fine
E004 -> E002	0000 0000	000 100	000 000	000 000	E004 ou se	E004_RSI_Condition_Fine
E004 -> E003	0000 0000	000 100	000 000	000 000	E004 ou se	E004_RSI_Condition_Fine
E004 -> E005	0000 0000	000 100	000 000	000 000	E004 ou se	E004_RSI_Condition_Fine
E004 -> E006	0000 0000	000 100	000 000	000 000	E004 ou se	E004_RSI_Condition_Fine

EID210100 >> Documents >> Travaux Pratiques >> PDF >> CAN
>> EID056010 Notice_Technique_CAN01A

Page 15.

6.2.2 Carte 4 sorties TOR							
Le port 8 bit du can expander MCP25050 est configuré :							
GPI7	GPI6	GPI5	GPI4	GPI3	GPI2	GPI1	GPI0
E	E	E	E	S	S	S	S
Avec : E : entrée TOR, S : sortie TOR							
6.2.2.1 Feux avant							
L'affectation des entrées sur la carte entrée est la suivante :							
GPI7	GPI6	GPI5	GPI4	GPI3	GPI2	GPI1	GPI0
Etat dipolant	Etat place	Etat code	Etat veilleuse	dipolant	place	code	Vérlace
6.2.2.2 Feux arrières							
L'affectation des entrées sur la carte entrée est la suivante :							
GPI7	GPI6	GPI5	GPI4	GPI3	GPI2	GPI1	GPI0
Etat EPS	dipolant	Etat code	Etat veilleuse	(blason)	dipolant	code	Vérlace
Remarques : la commande klaxon est active sur le module feux arrières gauche.							

EID210100 >> Documents >> Travaux Pratiques >> PDF >> CAN
>> EID056010 Notice_Technique_CAN01A

Page 21.

TABLE 4-3: COMMAND MESSAGES (EXTENDED IDENTIFIER)															
Message ID		Data Length		Data		Priority		Type		Source Address		Destination Address		Length	
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x00000000	00000000	00000000	000000												

BIBLIOGRAPHIE

- ❖ www.webfleet.com
- ❖ www.how2electronics.com
- ❖ www.medium.com
- ❖ www.csselectronics.com
- ❖ www.didalab-didactique.fr