

Introduction:

1-Matériel utilisé:

Arduino Uno R3: C'est une carte de développement à microcontrôleur basée sur le **ATmega328P**. Elle est largement utilisée dans les domaines de l'embarqué, de la robotique et de la domotique grâce à sa simplicité, sa communauté active et sa compatibilité avec une vaste bibliothèque d'extensions (notamment **mcp2515.h** pour le CAN). Elle dispose d'entrées/sorties numériques, d'interfaces de communication (SPI, I2C, UART), et permet un prototypage rapide en langage C/C++ via l'environnement Arduino IDE.

Dans le projet, l'Arduino Uno agit comme maître du système : il lit les trames CAN émises par les périphériques, commande dynamiquement les feux, active ou désactive des actionneurs (comme le buzzer), et gère l'affichage d'informations sur le moniteur série. Grâce à sa simplicité d'utilisation, sa stabilité, l'Arduino Uno constitue une plateforme robuste et pédagogique pour les systèmes embarqués en environnement multiplexé.

Caractéristiques principales:

- Microcontrôleur : ATmega328P (8 bits)
- Fréquence d'horloge : 16 MHz
- Mémoire Flash : 32 KB (dont 0.5 KB utilisés par le bootloader)
- RAM : 2 KB
- EEPROM : 1 KB
- Tensions : 5 V (logique), 7–12 V (entrée)
- 14 broches numériques (dont 6 PWM)
- 6 entrées analogiques
- Interface SPI, I2C, UART
- Broche d'interruption externe utilisée : D2 (INT)



CAN-BUS Shield v1.1: C'est une extension matérielle pour Arduino permettant d'interfacer la carte avec un réseau CAN (Controller Area Network). Il est basé sur deux composants principaux : le MCP2515, contrôleur CAN SPI, et le TJA1050, transceiver physique qui assure la conversion des signaux logiques en signaux différentiels compatibles avec le bus CAN. Ce shield permet à un Arduino de s'insérer dans un réseau de communication industriel ou automobile.

Dans le projet, il est utilisé pour échanger des trames CAN avec le commodo et les modules feux du VMD. Il reçoit les trames OM (Output Message) pour lire l'état des boutons du commodo, et envoie des trames IM (Instruction Message) pour piloter les feux et le klaxon. La broche INT est utilisée pour générer des interruptions sur l'Arduino (D2), garantissant une réactivité optimale

Caractéristiques principales:

- Contrôleur CAN : MCP2515 (Interface SPI jusqu'à 10 MHz)
- Transceiver CAN : TJA1050
- Implémente CAN V2.0B jusqu'à 1 Mb/s
- Alimentation : 5 V (directement via Arduino)
- Interface SPI avec l'Arduino (CS = D9 par défaut)
- Données standard (11 bits) et étendues (29 bits) et trames distantes
- Bornier à vis pour connexion CAN_H / CAN_L
- LED d'activité CAN



PicoScope 3204D MSO: C'est un oscilloscope numérique portable à mémoire (DSO) avec fonctionnalités mixtes analogiques et logiques (MSO : Mixed-Signal Oscilloscope). Il se connecte à un ordinateur via USB et offre des fonctionnalités avancées telles que les déclenchements conditionnels, le décodage de protocoles série (CAN, I2C, SPI), ou l'enregistrement longue durée. Il est particulièrement adapté pour le débogage de signaux numériques rapides dans des environnements embarqués.

Dans ce projet, le PicoScope a servi à visualiser les trames CAN ou SPI en temps réel (niveau physique), et à identifier des anomalies.

Caractéristiques principales:

- Canaux analogiques : 2
- Canaux logiques (MSO) : 16
- Bande passante : 70 MHz
- Échantillonnage max : 1 GS/s (analogique), 80 MS/s (logique)
- Résolution : 8 bits à 1 GS/s
- Mémoire tampon : 64 MS
- Interface : USB 3.0 (alimentation et données)
- Décodeurs intégrés : CAN, UART, SPI, etc.

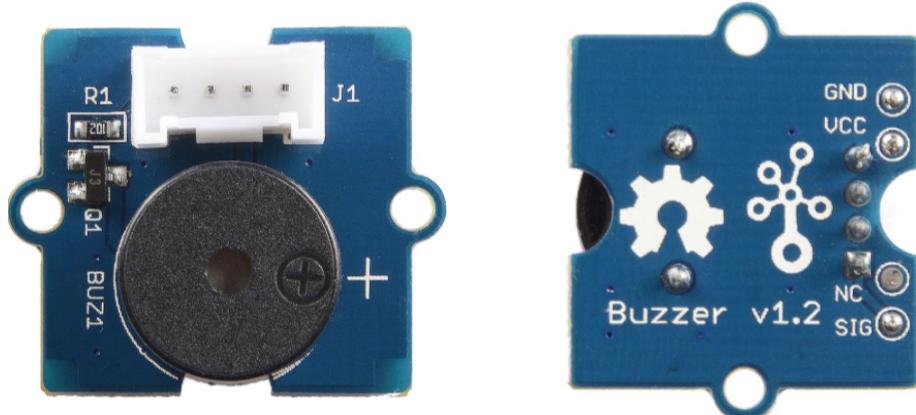


Buzzer V1.2: C'est un buzzer actif, c'est-à-dire qu'il émet un son de fréquence fixe (environ 2 kHz) lorsqu'un signal numérique haut lui est appliqué. Ce module est simple à utiliser, compact, et compatible avec les cartes Arduino via le connecteur Grove (ou en câblage classique avec VCC, GND et SIG). Il est souvent utilisé pour générer des alertes sonores ou des retours utilisateurs dans des systèmes embarqués.

Dans le cadre du projet, le buzzer est relié à la broche D3 de l'Arduino. Il est activé en temps réel lors de l'appui sur le bouton klaxon (GP7) du commodo, détecté via une trame CAN.

Caractéristiques principales:

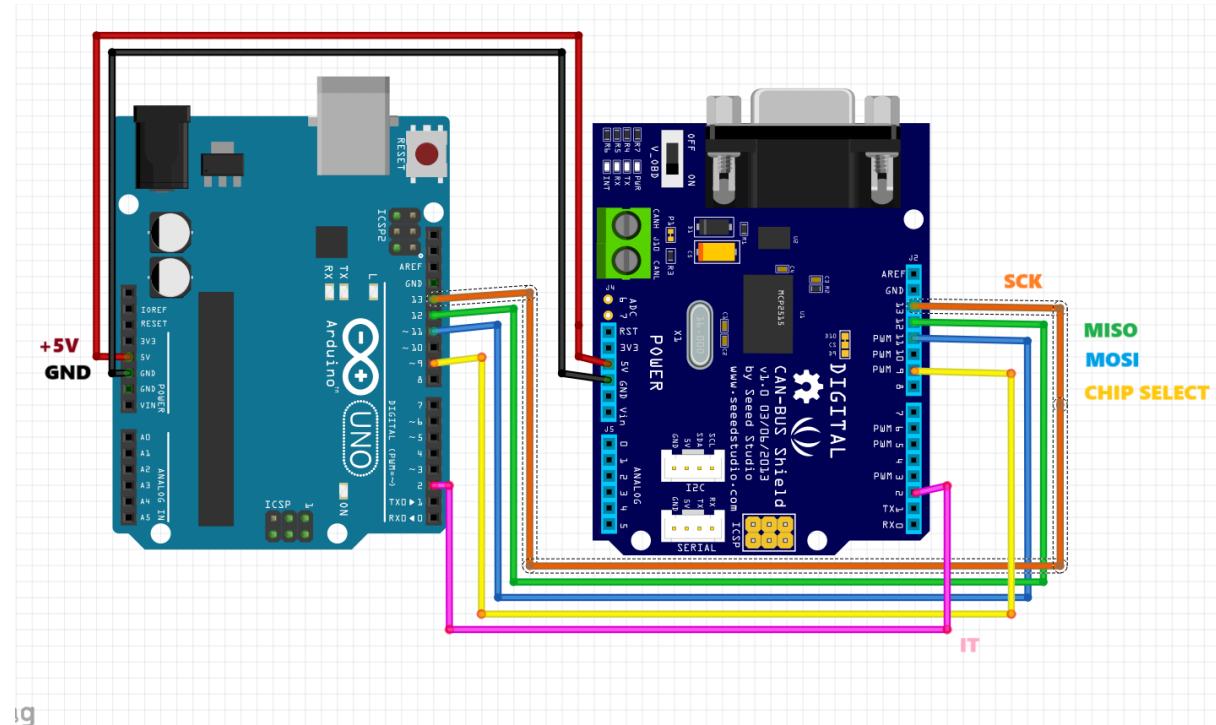
- Type : Buzzer actif (fréquence fixe)
- Tension de fonctionnement : 3.3 V à 5 V
- Broches :
 - **SIG** : signal d'activation (connectée à une sortie numérique)
 - **VCC**
 - GND
 - NC : non utilisée
- Consommation : faible (adapté à l'IoT)
- Fréquence typique : ~2 kHz



2-Connexions physiques et interface SPI

Lorsqu'on téléverse le code depuis l'IDE Arduino vers la carte Arduino Uno R3, le fichier binaire est transmis via le port USB en utilisant le convertisseur USB-série vers le microcontrôleur principal, qui exécute ensuite le bootloader avant de lancer le programme utilisateur, initialisant la communication SPI grâce à la fonction `SPI.begin()`. Cela configure automatiquement les broches D11 (MOSI), D12 (MISO), D13 (SCK) en mode maître SPI, et la broche D9 est utilisée comme CS (Chip Select) pour sélectionner le MCP2515 lors des transmissions.

a- Schéma de câblage:



b- Placement électrique : le shield se fige directement sur l'Arduino.

c- Fonctionnement de la communication SPI (CS, MOSI, MISO, SCK) avec interruption:

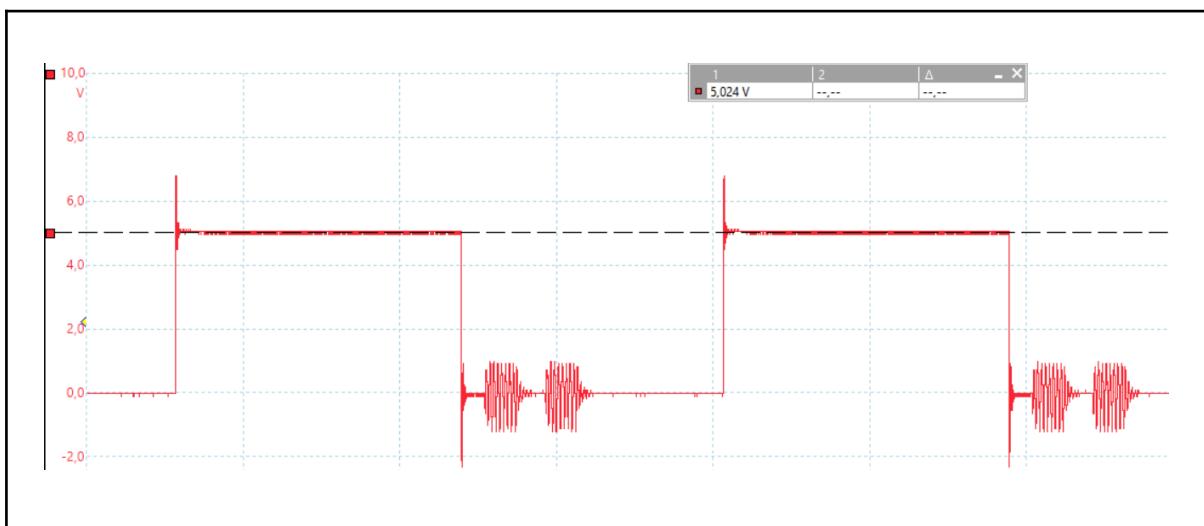
L'Arduino transmet les octets ci-dessus à travers le **bus SPI** vers le MCP2515, selon le protocole SPI standard :

1. Un message CAN est reçu par le contrôleur MCP2515 depuis le bus CAN.
2. Le MCP2515 génère une **interruption** matérielle en abaissant la broche **INT** (connectée à D2 de l'Arduino).

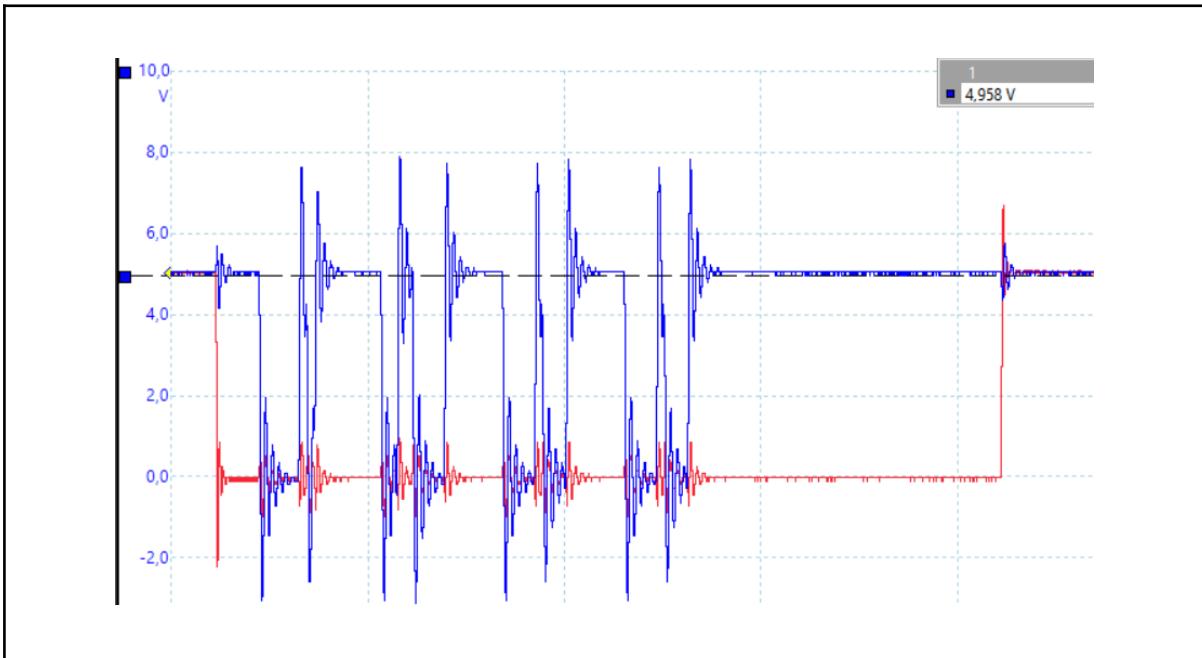


L'Arduino détecte cette interruption via INT0 et déclenche la fonction **interruptionCAN()**, qui met un flag **messageReçu** est mis à **true**

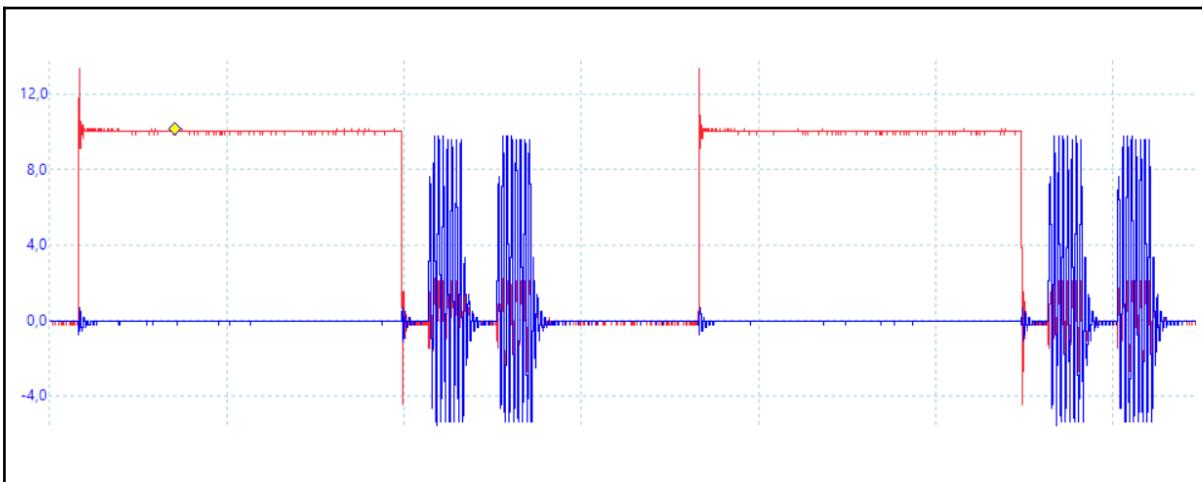
3. Le signal **CS (Chip Select)** est mis à LOW (**D9**) pour démarrer la communication SPI avec le MCP2515.



4. Les données sont transmises bit par bit de l'Arduino vers le MCP2515 via la broche **MOSI (D11)**.

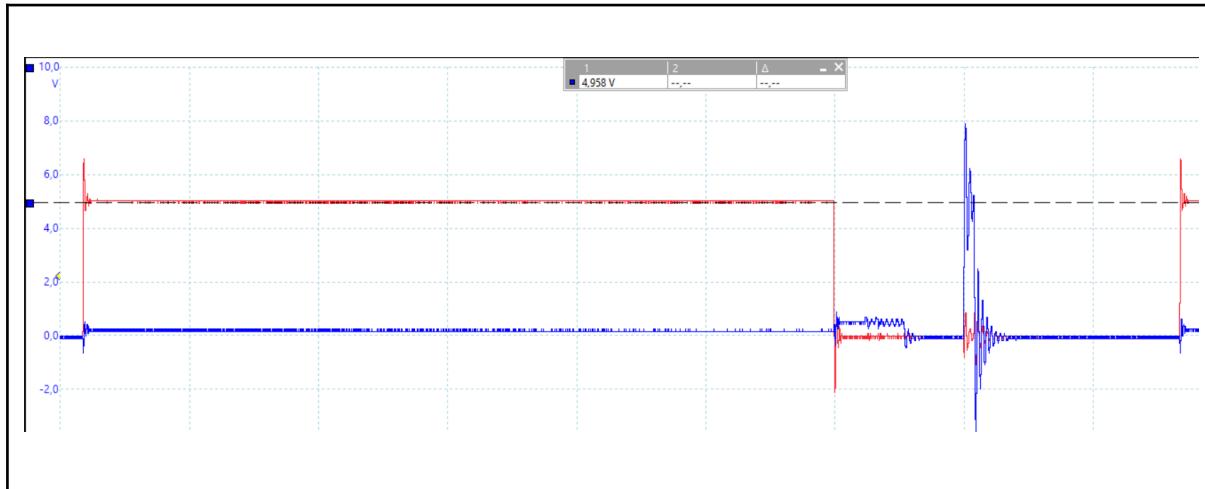


5. Le **signal d'horloge SCK (D13)** cadence chaque bit transmis



La fréquence de cette horloge est contrôlée par l'Arduino en fonction de la configuration SPI (de 1 MHz à 8 MHz).

6. Le MCP2515 répond, s'il y a lieu, en renvoyant des données via la broche MISO (D12).



7. La communication SPI se termine, le **CS est remis à HIGH**. L'Arduino peut maintenant traiter les données reçues.

Séquence complète : Arduino met CS à LOW → transmet les octets → MCP2515 répond → CS à HIGH & termine.

Le MCP2515, qui est un contrôleur CAN autonome, est interfacé avec l'Arduino à travers ces lignes SPI :

Broche Arduino	Fonction sur le shield CAN BUS v1.2	Rôle
D2	INT	Pour l'interruption en cas de message CAN entrant. (Le MCP2515 générant un interrupt sur INT, Arduino exécute “ <code>mcp2515.readMessage()</code> ” pour récupérer l'ACK ou les données.)
D9	CS (Chip Select SPI)	Sélection du MCP2515 pendant la communication SPI (active le MCP2515 lors des échanges)
D11 (MOSI)	SPI Master-Out, Slave-In	Données envoyées par l'Arduino vers le MCP2515 (données de configuration ou de trames)
D12 (MISO)	SPI Master-In, Slave-Out	Données reçues du MCP2515 vers l'Arduino
D13 SCK (via header ICSP)	SPI Clock	Horloge SPI
GND	Masse	Référence électrique commune
5V	Alimentation du shield	Alimente le MCP2515 et le TJA1050

Le programme configure ensuite le MCP2515 via SPI est initialisé pour fonctionner à 100 kbps avec un oscillateur de 16 MHz comme suit:

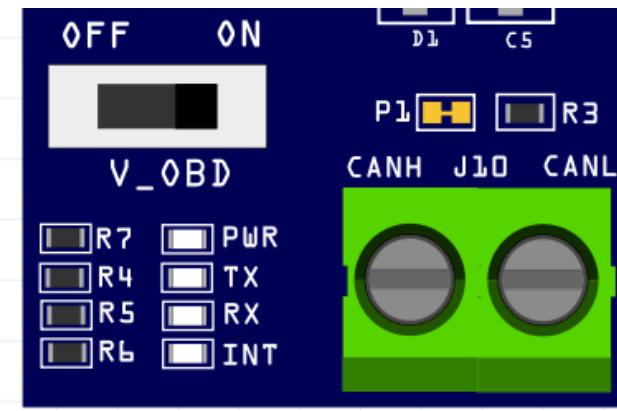
```
“mcp2515.reset();”
“mcp2515.setBitrate(CAN_100KBPS, MCP_16MHZ);”
```

puis le passage en mode de fonctionnement normal avec

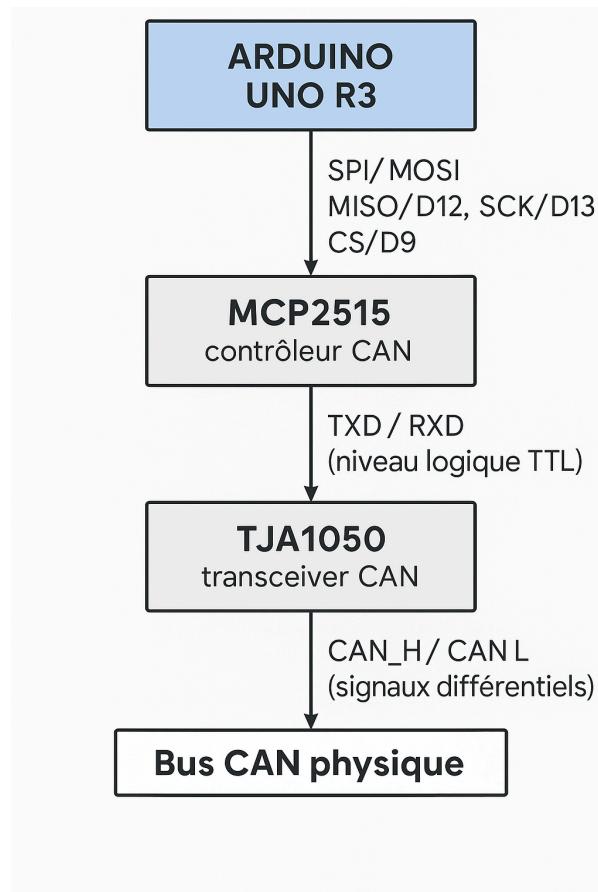
```
“mcp2515.setNormalMode();”
```

Le protocole SPI permet à l'Arduino (maître SPI) d'envoyer des commandes au MCP2515 (esclave SPI) pour écrire, lire ou envoyer des trames CAN.

d- Indicateurs LED sur le support CAN-BUS SHIELD indiquent l'état des transmissions : PWR (reste ON avec l'alimentation), TX, RX, INT



e-Séquence du flux SPI:



f- Décodage de la trame SPI (Envoyé de l'arduino):

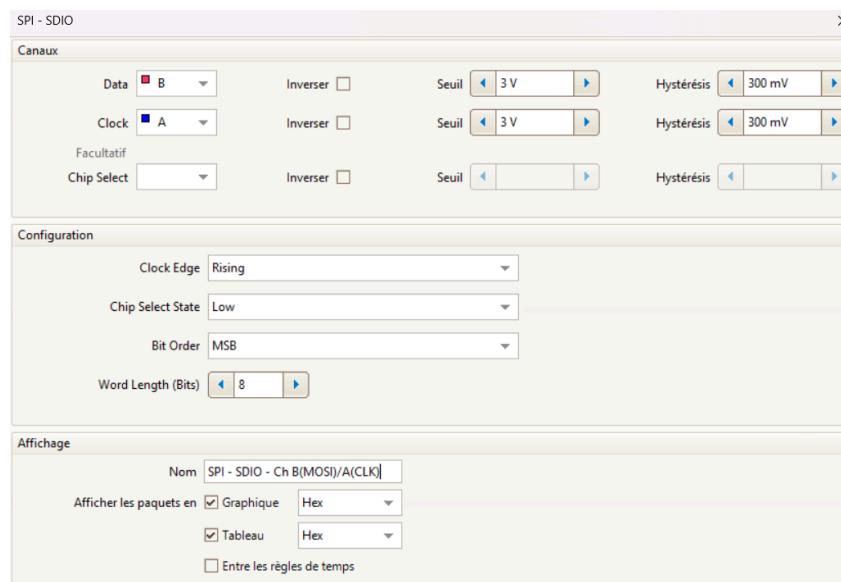


Trame complète pour allumer la led du code gp1:

A0 00 03 30 00 02 31 74 84 00 00 03 1E 0F 02 05 30 08 08 03 30 00 A0 00 A0 00 A0 00 A0 00 A0 00

g- Préparation et envoi de la trame SPI:

La trame SPI que tu as observée commence par l'instruction A0, qui est une commande d'écriture vers le MCP2515 permettant de charger un message dans le buffer de transmission. Dans cette trame SPI capturée, les données utiles principales sont les trois octets formant une commande destinée au module CAN01A (ex. feu avant droit), qui utilise un coprocesseur MCP25050. Le premier octet 1E correspond au registre GPLAT de ce module — il s'agit du registre de sortie logique qui contrôle directement l'état des broches GP0 à GP3 du module (donc les LEDs/feux dans le contexte VMD). Le second octet 0F est un masque de bits : il indique que seuls les bits de poids faible (GP0 à GP3) seront modifiés par cette commande (le masque 0x0F = 00001111 en binaire). Le troisième octet 02 est la valeur à écrire dans le registre GPLAT, en respectant le masque. Ici, 02 = 00000010 en binaire, donc seul le bit GP1 sera activé. GP1 correspond, selon la configuration du système VMD, au feu de croisement (Code). Ainsi, cette séquence a pour but d'allumer uniquement le feu de Code sur le bloc optique concerné, l'octet 03 indique que la trame CAN transportera 3 octets de données (DLC = 3)



3-Transmission sur le bus CAN

a- Construction d'une trame CAN avant transmission:

Lorsqu'on souhaite envoyer une commande de feu vers un module VMD (feux avant ou arrière), la fonction “*envoyerCommande()*” prépare une trame CAN. Lorsqu'on lance le code depuis l'Arduino Uno R3, celui-ci initialise le contrôleur MCP2515 via l'interface SPI, utilisant les 5 broches, le CS selon la déclaration “***MCP2515 mcp2515(9);***”. L'Arduino transmet donc une trame SPI au MCP2515 lui demandant de transmettre une trame CAN bien définie. -Cette trame est construite avec un ID étendu de 29 bits (ici ***0xE880000***), ce qui permet d'adresser de manière précise un module spécifique sur le réseau CAN (par exemple, ici, le module feux avant droit). -Vient ensuite le champ DLC (Data Length Code), qui indique ici la longueur des données (3 octets). Ensuite, les 3 octets de données utiles (Data) sont envoyés : ***0x1E, 0x0F, 0x02***. Ces octets ont une signification claire dans le contexte du système VMD (Véhicule Multiplexé Didactique).

- Le premier (***0x1E***) désigne le registre GPLAT du circuit MCP25050 du module feux, qui sert à écrire l'état des sorties (GP0 à GP7).
- Le second octet (***0x0F***) est un masque binaire ciblant les bits GP0 à GP3, soit les quatre sorties contrôlant les LED du bloc optique (électionne les bits modifiables veilleuse, code, phare, clignotant et évite de toucher aux GP4–GP7).
- Enfin, le dernier octet (***0x02***) signifie une valeur qui désigne l'état des sorties (LED). Dans ce cas, seulement la LED du code (GP1) qui s'allume (valeur binaire ***0010*** sur GP0 à GP3).

b- Traitement interne :

- Le **MCP2515** vérifie s'il a le droit d'émettre (arbitrage du bus)
- Si oui, il encode la trame CAN complète

Exemple de la fonction utilisé: “*envoyerCommande(REG_GPLAT, 0x0F, 0x02)*”

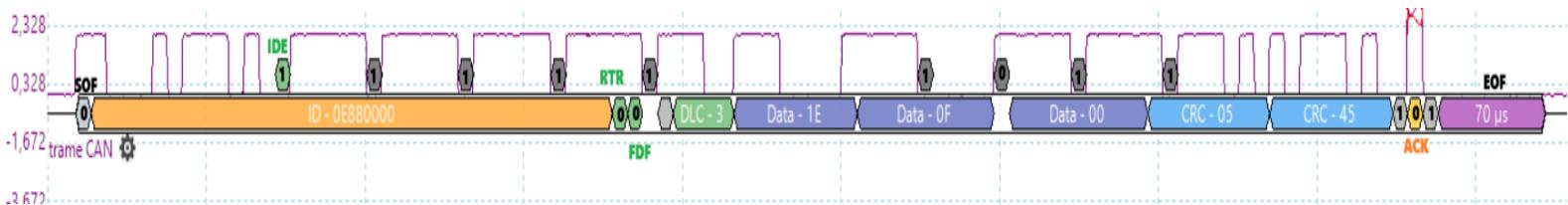
(allume le Code GP1)

La trame CAN créée à cette forme :

Champ	Valeur
can_id	<i>0xE880000</i> (module FAD) avec CAN_EFF_FLAG
can_dlc	<i>3</i> (3 octets de données)
data[0]	<i>0x1E</i> (adresse du registre GPLAT)
data[1]	<i>0x0F</i> (masque : 4 bits à modifier)
data[2]	<i>0x02</i> (commande à appliquer)

Une fois ces données SPI transférées, le MCP2515 génère une trame CAN à transmettre physiquement. Il ajoute un champ CRC automatiquement (**0x0545**) pour garantir l'intégrité de la trame. Cette valeur CRC est calculée à partir de tous les bits transmis précédemment, selon un polynôme standardisé (généralement **0x4599** pour CAN 2.0B).

Le bit d'acquittement (ACK) suit : chaque nœud connecté au bus CAN, s'il reçoit la trame sans erreur, doit envoyer un bit dominant (**0**) dans le slot ACK. Puis, la trame se termine avec le champ EOF (End of Frame), constitué de 7 bits récessifs (**1111111**) qu'on observe comme un segment de 70 µs, ce qui est parfaitement cohérent avec un débit de **100 kbps** (chaque bit = 10 µs).



Remarque: La ligne `#define CAN_CLOCK MCP_16MHZ` indique que le module CAN (MCP2515) utilise un oscillateur de 16 MHz comme horloge principale.

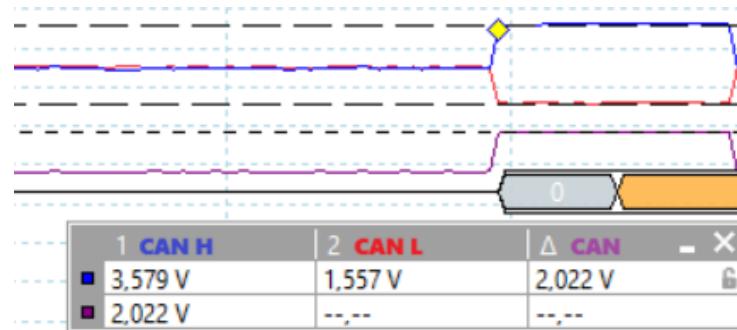
La ligne `#define CAN_SPEED CAN_100KBPS` signifie que l'on veut communiquer sur le bus CAN à une vitesse de 100 kilobits par seconde.

Pour que la communication fonctionne, le module doit diviser son horloge (16 MHz) pour générer des bits à la bonne vitesse (100 kbps). Cela doit se faire sans erreur, sinon la communication sera instable. Les deux valeurs sont donc cohérentes si l'horloge peut être divisée proprement pour obtenir exactement 100 000 bits par seconde.

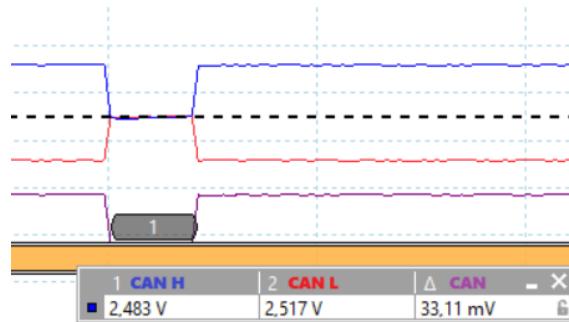
c- Conversion en signaux physiques (CAN_H et CAN_L)

Rôle du transceiver:

À ce moment-là, le transceiver CAN connecté au MCP2515 généralement un MCP2551 ou TJA1050 prend le relais. Son rôle est de convertir les signaux numériques du MCP2515 TXD, RXD (niveau TTL de 0 V / 5 V) en une différence de tension différentielle sur les lignes **CAN_H** et **CAN_L**. Lorsqu'un bit dominant (**0**) est transmis, le transceiver génère typiquement **CAN_H ≈ 3.5 V** et **CAN_L ≈ 1.5 V**, soit un écart de **~2 V** entre les deux lignes, comme visualisé sur le Picoscope:



Pour un bit récessif (1), les deux lignes sont ramenées à environ 2.5 V, et donc le différentiel devient nul : $\text{CAN_H} - \text{CAN_L} \approx 0 \text{ V}$. Cette modulation différentielle est extrêmement robuste face aux perturbations, ce qui rend le bus CAN si fiable en environnement bruyant (automobile, industrie).



État logique	CAN_H	CAN_L	Bus CAN
Récessif (1)	~2.5V	~2.5V	Aucune différence → bus au repos
Dominant (0)	~3.5V	~1.5V	Différence de potentiel = communication

d- Décomposition complète et continue de ta trame CAN:

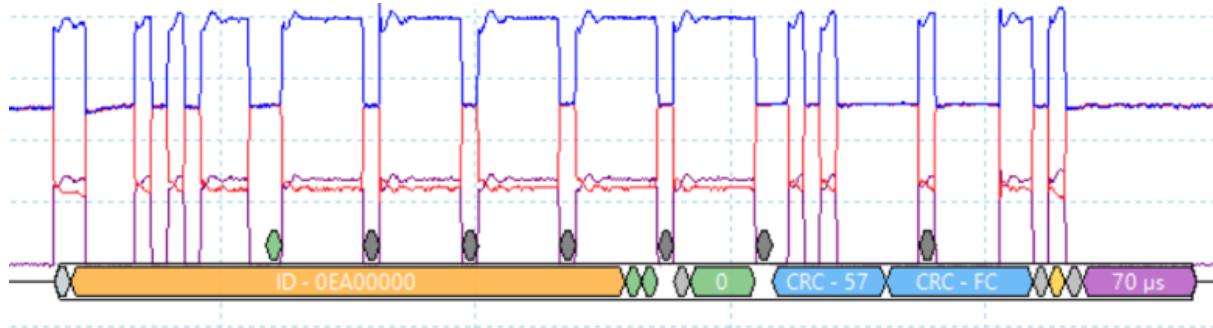
Champ	Longueur (bits)	Description
SOF	1	Start of Frame : début de la trame (toujours 0 = dominant)
ID (11 bits)	11	Bits d'identifiant prioritaire (partie haute)
SRR	1	Substitute Remote Request : 1 pour trames étendues
IDE	1	Identifier Extension : 1 → format étendu
ID (18 bits)	18	Bits d'identifiant secondaires (partie basse)

RTR	1	Remote Transmission Request : 0 si trame de données
FDF	1	Flexible Data Format, 1 si la trame est au format CANFD sinon CAN classique 2.0B
r1 (reserved)	1	Réservé (dominant : 0)
DLC	4	Data Length Code : nombre d'octets de données (0 à 8)
Data	0 à 64	Données utiles (jusqu'à 8 octets en CAN 2.0B)
CRC (15 bits)	15	Code de redondance cyclique
CRC Delimiter	1	Toujours récessif (1)
ACK Slot	1	Bit d'acquittement attendu à 0 si reçu correctement
ACK Delimiter	1	Toujours récessif (1)
EOF	7	End of Frame : toujours 1 (récessif)

Packet	Start Time	End Time	ID	RTR	FDF	DLC	Data	CRC	ACK	CRC Valid	Bit Stuffing Valid	Valid	IDE
1	120,2 ns	960 µs	0E880000	0	0	3	1E 0F 02	4B EE	0	✓	✓	✓	1
2	1,01 ms	1,71 ms	0EA00000	0	0	0		57 FC	0	✓	✓	✓	1

d- Réception & Acquittement

Le module cible reçoit une trame avec un ID correspondant à sa configuration, son propre transceiver re-convertit les signaux CAN_H/L en niveaux logiques TTL, que son contrôleur interne lit et interprète effectuant alors la commande sur les sorties GP0 à GP3 du module (représentant veilleuse, phare, code, clignotant, etc.). Il peut aussi générer une trame d'acquittement (ACK) qu'il renvoie sur le bus, le bit ACK est géré au niveau physique, le MCP2515 ne lit pas ce bit mais sait si l'émission a échoué ou non., ce retour est ensuite signalé à l'Arduino via une interruption déclenchée sur la broche INT



Ce type de trame AIM (Acknowledge Input Message) est envoyé automatiquement par le module VMD pour confirmer la bonne réception d'une commande de type IM (Input Message) provenant de l'Arduino.

C'est un acquittement explicite qui complète le mécanisme d'ACK implicite du protocole CAN, ce qui permet à l'Arduino :

- de s'assurer que le message a bien été traité par le bon module (grâce à l'ID unique)
- de détecter si un module ne répond pas, et donc de gérer les erreurs

4- Méthodes pour récupérer la trame AIM dans le MCP2515

- **Par interruption (INT)**

Le MCP2515 dispose d'une sortie d'interruption (broche INT), qui passe à l'état bas (LOW) dès qu'une nouvelle trame est reçue.

L'interruption INT est déclenchée lorsque le MCP2515 reçoit une trame dans son buffer RX. Elle permet d'éviter une boucle de polling constante, et rend le système plus réactif aux événements CAN

- **Par sondage (polling)**

L'Arduino vérifie régulièrement dans la boucle principale (loop()) si une trame CAN est disponible. Appels répétés à "`mcp2515.readMessage(&frame)`" pour tester la présence d'une trame, Si une trame est présente, elle est lue et traitée.

Ainsi, le flux de données suit une chaîne complète depuis le microcontrôleur (exécution du code), passant par la communication SPI, la construction d'une trame CAN via le MCP2515, la conversion électrique vers le bus CAN par le MCP2551, la réception par les modules de feux, et enfin la validation via trame d'ACK.

5- Les types de messages utilisées:

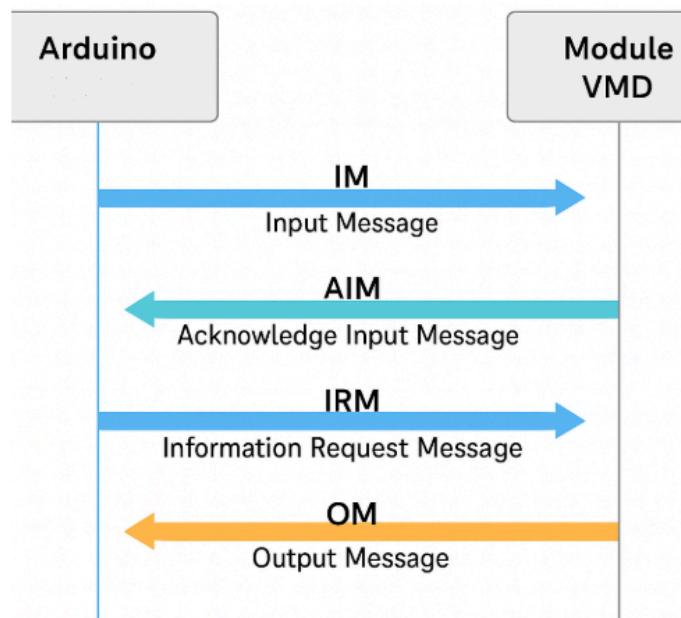
a- IRM (Information Request Message)

- **Émetteur : Arduino**

- **Récepteur** : Module VMD
- **Contenu** : Demande de lecture d'un registre (ex. GPLAT)
- **Utilisé pour** : Lire l'état d'un bouton ou d'une entrée du commodo
- **Exemple** : L'Arduino envoie une IRM toutes les 200 ms dans le TP4 pour lire l'état des BP du commodo.

b- OM (Output Message):

- **Émetteur** : Module VMD
- **Récepteur** : Arduino (ou tout noeud CAN)
- **Contenu** : Message spontané contenant l'état des entrées
- **Utilisé pour** : Notification d'événement (ex : changement d'état d'un bouton)
- **Remarques** :
 - Générés automatiquement par certains modules (ex : le commodo VMD)
 - Utilisé dans TP2 et TP4 : si un BP est pressé, un OM est émis sans requête IRM



Message	Déclenché par	Reçu automatiquement ?	Interruption utile ?	Contenu transmis
IM	Arduino	/	Oui (envoie)	Commande (ex : allumer feu)

AIM	Module VMD	OUI (après IM)	Oui	Acquittement : "commande bien reçue"
IRM	Arduino	/	Oui (envoie)	Requête d'état
OM	Module VMD	OUI (auto ou IRM)	Oui	État actuel du module (boutons, entrées)