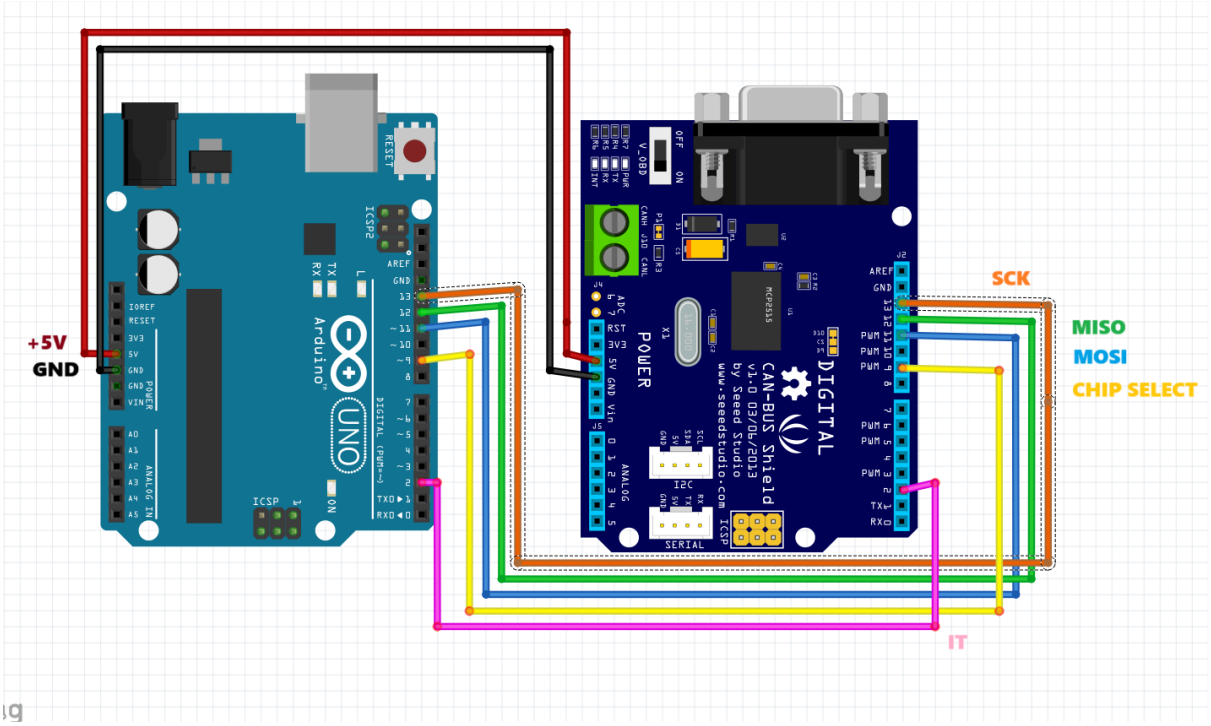
a-Assemblage matériel — Arduino Uno R3 + CAN-BUS Shield V1.2

Schéma de câblage:

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.



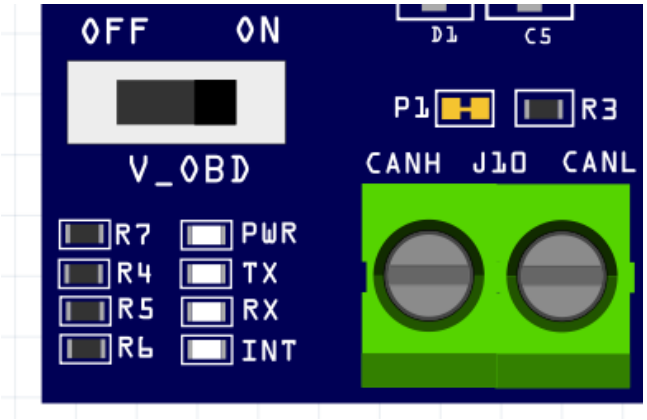
Placement électrique : le shield s'en fiche directement sur l'Arduino.

Le MCP2515, qui est un contrôleur CAN autonome, est interfacé avec l'Arduino à travers ces lignes SPI : **D11 (MOSI)** transmet les données de configuration ou de trames, **D12 (MISO)** lit les réponses du MCP2515, **D13** fournit l'horloge SPI (jusqu'à 16 MHz), et **D9** active le MCP2515 lors des échanges. Le programme configure ensuite le MCP2515 via SPI en envoyant successivement les commandes RESET, configuration du bitrate à 100 kbps puis le passage en mode de fonctionnement normal avec mcp2515.setNormalMode().

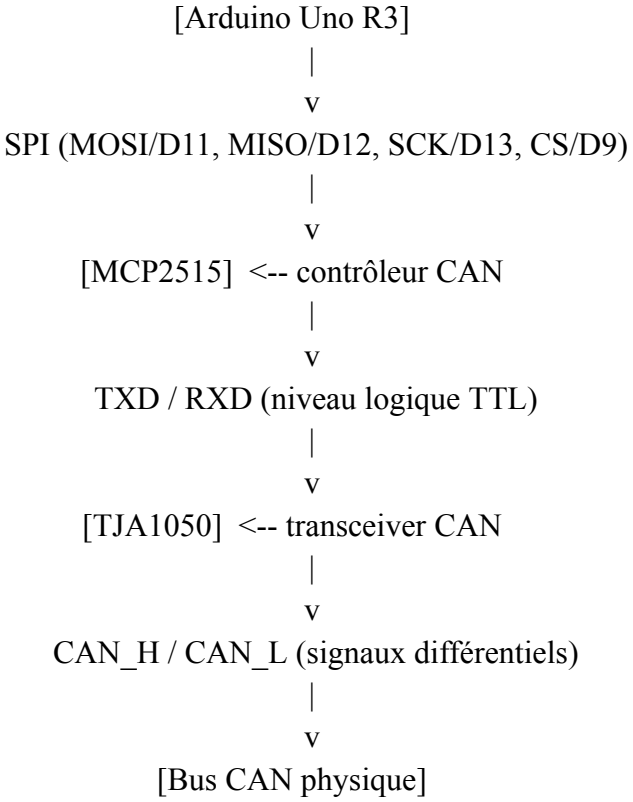
Broche Arduino	Fonction sur le shield CAN BUS v1.2	Rôle
D9	CS (Chip Select SPI)	Sélection du MCP2515 pendant la communication SPI
D11 (MOSI)	SPI Master-Out, Slave-In	Données envoyées par l'Arduino vers le MCP2515
D12 (MISO)	SPI Master-In, Slave-Out	Données reçues du MCP2515 vers l'Arduino
D13 SCK (via header ICSP)	SPI Clock	Horloge SPI
GND	Masse	Référence électrique commune
5V	Alimentation du shield	Alimente le MCP2515 et le TJA1050

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

Indicateurs LED : `PWR`, `TX`, `RX`, `INT` sur le shield indiquent l'état des transmissions.



b-Séquence du flux SPI:



Le MCP2515 est initialisé pour fonctionner à 100 kbps avec un oscillateur de 16 MHz comme suit:

```
mcp2515.reset();
mcp2515.setBitrate(CAN_100KBPS, MCP_16MHZ);
mcp2515.setNormalMode();
```

c-Construction d'une trame CAN avant transmission:

Lorsque l'utilisateur souhaite envoyer une commande de feu vers un module VMD (feux avant ou arrière), la fonction envoyerCommande() prépare une trame CAN en format étendu (29 bits), qui contient : un identifiant (ID) selon le module ciblé, trois octets de données CAN (dont le registre cible à écrire, un masque binaire, et la valeur à appliquer). Cette trame est transmise au MCP2515 via la liaison SPI, les octets étant sérialisés et horodatés grâce au signal SCK (généré par D13), pendant que D11 envoie les données et que D12 lit les éventuelles réponses.

Exemple : envoyerCommande(REG_GPLAT, 0x0F, 0x02) (allume le Code GP1)

La trame CAN créée à cette forme :

Champ	Valeur
can_id	0x0E880000 (module FAD) avec CAN_EFF_FLAG
can_dlc	3 (3 octets de données)
data[0]	0x1E (adresse du registre GPLAT)
data[1]	0x0F (masque : 4 bits à modifier)
data[2]	0x02 (commande à appliquer)

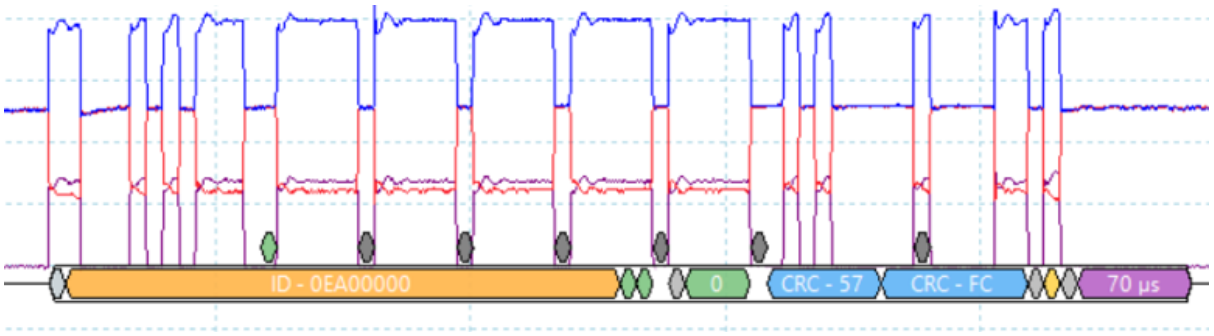
- GPDDR configure les directions (entrée/sortie)
- GPLAT contrôle les niveaux de sortie
- Le **masque** sélectionne les bits modifiables (évite de toucher aux GP4–GP7)

Dès réception complète de la trame SPI, le MCP2515 assemble une trame CAN standard en y ajoutant un SOF (Start of Frame), le champ d'arbitrage (ID étendu), le champ de contrôle (DLC), les octets de données, le CRC et un bit d'ACK, conformément au protocole CAN 2.0B. Cette trame est ensuite transmise électroniquement au transceiver MCP2551 via la broche TXCAN. Le MCP2551 convertit les signaux logiques TTL du MCP2515 en signaux différentiels analogiques sur les lignes CAN_H et CAN_L : lorsque la trame contient un bit dominant ('0'), CAN_H monte à environ 3,5 V tandis que CAN_L descend à 1,5 V ; en état récessif ('1'), les deux lignes sont équilibrées à environ 2,5 V.

Lorsqu'un 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, ce que le MCP2515 détecte via le bit ACK dans la trame CAN ; ce retour est ensuite signalé à l'Arduino via une interruption déclenchée sur la broche INT

d-Réception & Acquittement

Le programme Arduino, via readMessage(), lit alors cette trame d'acquittement pour valider le bon envoi. 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. Chaque broche joue un rôle précis : D11 pour l'envoi SPI (MOSI), D12 pour la réception SPI (MISO), D13 pour l'horloge SPI (SCK), D9 pour la sélection du MCP2515 (CS), et D2 pour l'interruption en cas de message CAN entrant.



1. Le **module réception** correspondant traite la trame (via filtre d'ID dans MCP2515).
2. Il génère un releay ACK sur le bit dédié dans la trame.
3. Le MCP2515 générant un interrupt sur INT (D2/D3), Arduino exécute mcp2515.readMessage() pour récupérer l'ACK ou les données.

e-Transmission via le MCP2515

Étape SPI :

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

1. Le signal **CS (D9)** est mis à LOW
2. Les données sont envoyées bit par bit via MOSI (D11)
3. Le SCK (D13) cadence la transmission
4. Le MCP2515 répond éventuellement via MISO (D12)
5. **can_id = ID 29 bits (ex. 0x0E880000) + Flag trame étendue**
6. **DLC = 3 octets**
7. **data = registre, masque, valeur.**

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

f-Données réellement échangées Arduino ↔ Shield

Données SPI échangées (via mcp2515.sendMessage) :

- Instruction d'écriture (ex. 0x40) + adresse + octets de la trame
- 3 octets de données CAN (ex. 0x1E, 0x0F, 0x04)
- Contrôle du statut du TX buffer, vérification de l'ACK, etc.

Données converties en signal CAN :

- Trame binaire complète encodée selon le format CAN 2.0B
- Émise par le TJA1050 vers les lignes CAN_H / CAN_L

g-Traitement interne :

- Le **MCP2515** vérifie s'il a le droit d'émettre (arbitrage du bus)
- Si oui, il **encode la trame CAN complète** (Start of Frame, champ ID 29 bits, champ DLC, données, CRC, ACK, EOF)

h-Conversion en signaux physiques (CAN_H et CAN_L)
Rôle du TJA1050 (ou équivalent sur le shield)

Le **transceiver** prend les signaux numériques du MCP2515 (TXD, RXD) et les convertit :

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

- Le bus **CAN fonctionne en différentiel** pour résister aux interférences.
- La trame est ainsi diffusée à tous les nœuds connectés (broadcast).