Systèmes embarqués automobiles

Master D'Université Spécialisé en
Ingénierie des Systèmes Embarqués

Chapitre 2: Le Bus LIN

Année universitaire 2024/2025

Dr. Ing. LASSIOUI Abdellah

Introduction

- Le bus de données LIN (*Local Interconnect Network*) est un bus multipoint, de faible coût facilement déployé dans un véhicule.
- Il fonctionne souvent comme un sous-nœud du bus CAN.

- Un bus **multipoint** est un type de bus de communication dans lequel plusieurs dispositifs (ou nœuds) partagent la même ligne de communication physique.

- Contrairement aux bus **point-à-point,** où il y a une connexion directe entre deux dispositifs, un bus multipoint permet à plusieurs dispositifs de se connecter et de communiquer sur le même canal ou ligne de données.

Pourquoi on a besoin du bus LIN

- Le **bus LIN** (Local Interconnect Network) est un type de bus de communication série multipoint, principalement utilisé dans l'industrie automobile pour la communication entre les composants électroniques de faible coût et de faible vitesse. Il complète le bus CAN dans des applications moins critiques et avec des exigences de communication plus modestes.

- **1- Réduction des coûts** : Le bus LIN est conçu pour être moins coûteux que le bus CAN, avec un protocole plus simple et une interface matérielle minimaliste. Cela permet de connecter des capteurs, des actionneurs et des modules électroniques simples sans recourir à des contrôleurs CAN plus onéreux.

- **2- Applications de faible vitesse** : Le bus LIN est optimisé pour des applications ne nécessitant pas de transmission à haute vitesse. Il est donc adapté pour des communications en dessous de **20 kbps**, comme le contrôle des sièges, des fenêtres électriques, des rétroviseurs, de l'éclairage intérieur, et d'autres fonctions non critiques.

Pourquoi on a besoin du bus LIN (Suite)

3- **Simplicité de communication** : Contrairement au bus CAN, le bus LIN utilise une communication **maître-esclave**, où un seul maître coordonne les échanges entre les esclaves. Cette architecture simplifie la gestion du bus et permet de réduire le besoin de traitement complexe dans les dispositifs esclaves.

4- Tolérance aux erreurs moins stricte : Les applications utilisant le bus LIN ne nécessitent pas une tolérance aux erreurs aussi élevée que celles sur le bus CAN. Le LIN est tolérant aux erreurs mineures, ce qui le rend suffisant pour des applications où une redondance complète n'est pas nécessaire.

Le bus LIN: dates clés

- **1999** : Création du standard LIN par un consortium de constructeurs européens (BMW, Volvo, VW, Audi, etc.), visant une solution économique pour les applications non critiques dans les véhicules.

- **2000** : Publication de la première version LIN 1.0, définissant le protocole maître-esclave et un débit maximal de 20 kbps avec câblage simple.

- **2002** : Sortie de LIN 1.3, avec des améliorations de fiabilité ; cette version est adoptée pour les premières applications commerciales comme le contrôle des fenêtres et sièges.

- **2003** : Lancement de LIN 2.0, qui améliore la gestion des erreurs, ajoute des diagnostics robustes, et facilite l'intégration avec d'autres réseaux de véhicules.

- **2006** : Publication de LIN 2.1, introduisant des fonctionnalités d'auto-adressage et une meilleure gestion énergétique pour réduire la consommation.

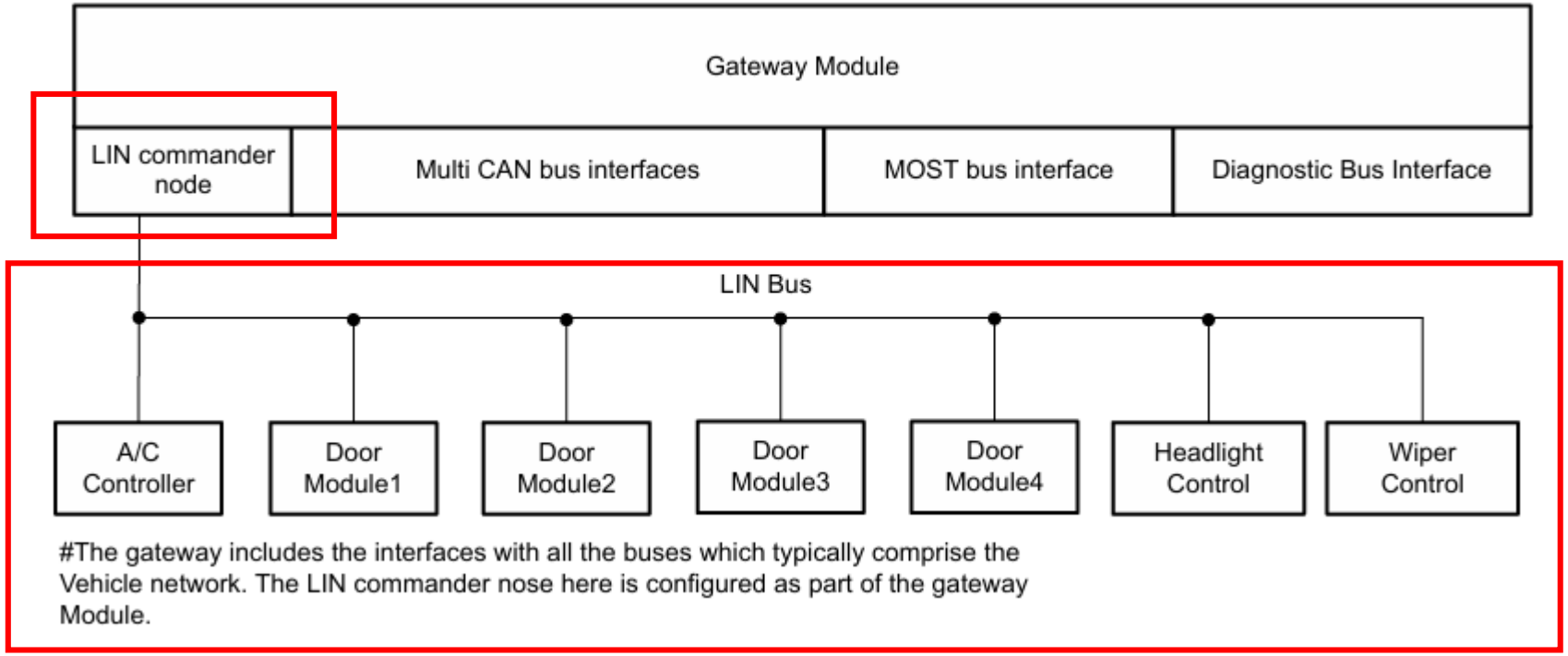
Le bus LIN: dates clés (suite)

- **2008** : LIN est standardisé sous **ISO 17987**, ce qui renforce son adoption internationale dans l'industrie automobile.
- **2010** : Introduction de LIN 2.2A, la version finale qui corrige des erreurs mineures et optimise les performances, devenant la norme ISO pour le bus LIN.
- **Années 2010** : LIN devient le standard pour les systèmes de confort (rétroviseurs, éclairage intérieur, etc.), complémentaire au bus CAN pour les fonctions critiques.
- **Depuis 2020** : Avec l'essor des véhicules électriques, LIN continue de servir pour des fonctions basiques, tandis que les constructeurs adoptent des architectures réseaux plus avancées pour les systèmes critiques.

Le gateway module comme interface entre les réseaux du véhicule

La figure suivante illustre la connexion du bus LIN au module Gateway (passerelle) qui joue un rôle crucial en assurant la communication entre le bus LIN et d'autres réseaux du véhicule, comme le bus CAN ou d'autres réseaux LIN. Étant donné que le bus LIN est conçu pour des communications simples, locales et de faible vitesse, un module passerelle est souvent nécessaire pour relier ces sous-réseaux aux systèmes de communication principaux du véhicule.

Le bus LIN est utilisé pour contrôler les fonctions secondaires dans la voiture comme la fermeture des portes, la ventilation, le control de la lumière interne, le control de l'essuie-glace…

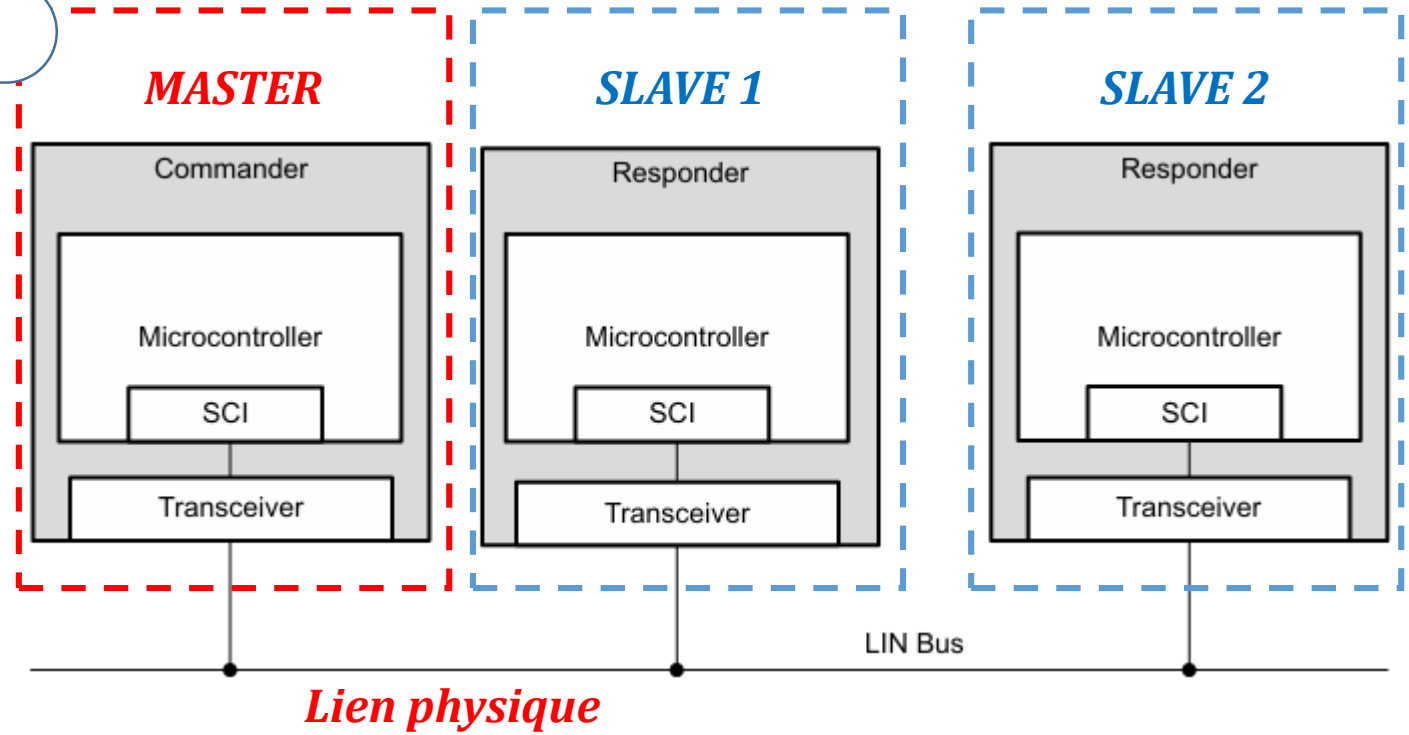


Architecture du bus LIN

- On définit le cluster LIN comme étant un ensemble de nœuds connectés à travers un lien physique
- Il existe deux types de nœuds dans un cluster : un nœud master and un nombre d'esclaves pouvant atteindre 16
- Le nœud master gère la communication entre les différents nœuds esclaves

1-

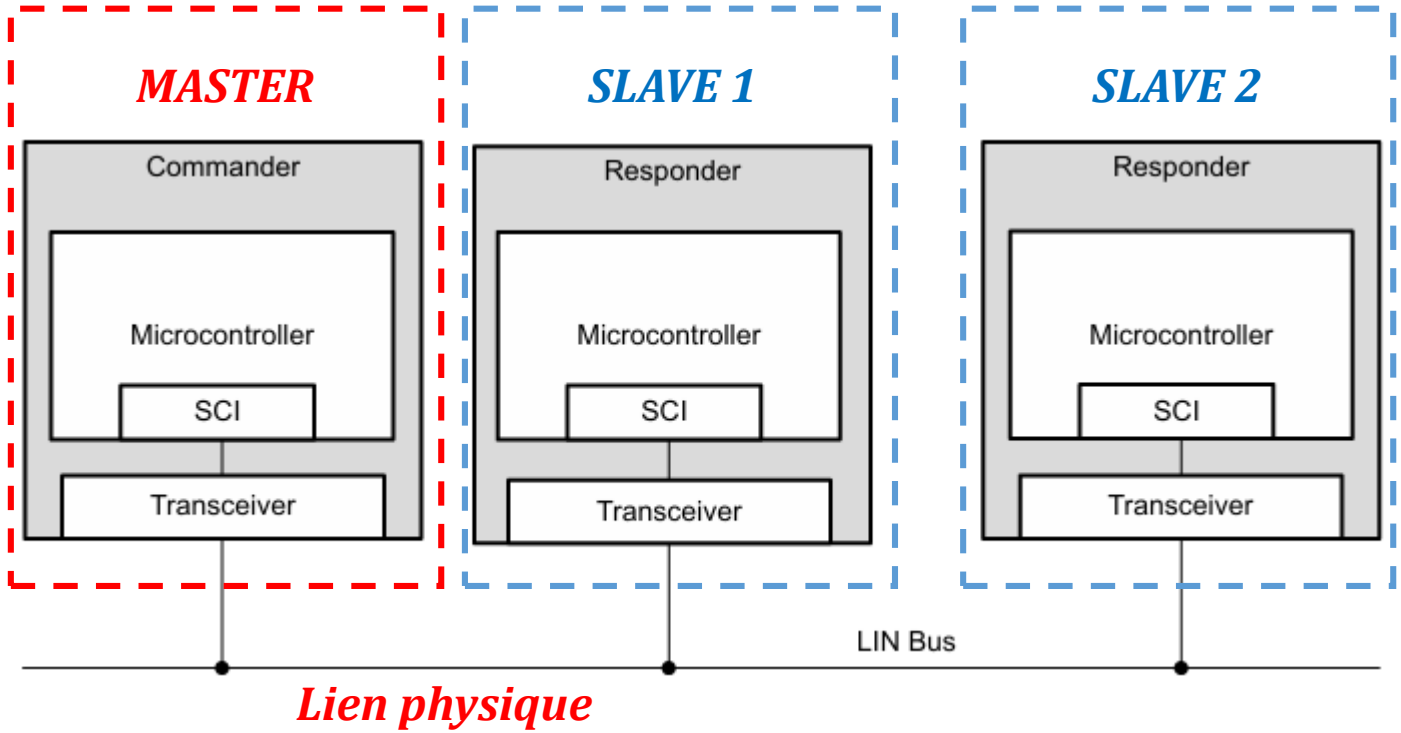
*La transmission du bus LIN utilise **un seul fil + fil masse** pour la communication entre les dispositifs connectés. Ce fil unique transporte les données entre le maître (contrôleur central) et les esclaves (modules périphériques). Cette simplicité permet de réduire le câblage et les coûts de mise en œuvre.*



Architecture du bus LIN

2-

Le bus LIN utilise une **vitesse de communication relativement faible** (environ 20 kbps maximum). Cette vitesse réduite est intentionnelle pour **minimiser les émissions électromagnétiques rayonnées**. À des vitesses plus élevées, les câbles peuvent émettre des interférences qui affectent d'autres composants électroniques.



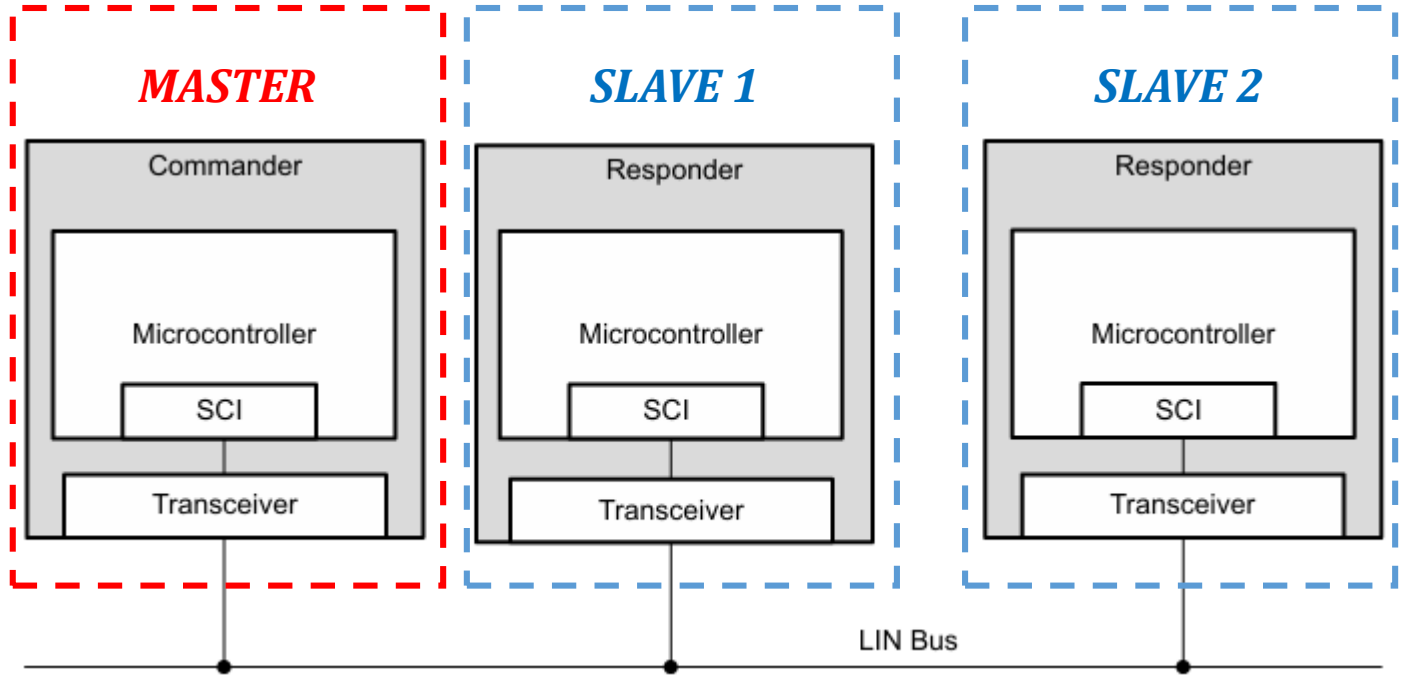
Lien physique

Cela permet également d'utiliser des microcontrôleurs moins performant de faible coût.

Architecture du bus LIN

3-

Tous les **nœuds** connectés au bus LIN sont connectés de manière **passive**. Cela signifie qu'ils n'interfèrent pas activement avec le signal de communication lorsqu'ils ne communiquent pas. Ils écoutent le bus sans générer d'activité tant qu'ils n'ont pas reçu d'instruction du maître.

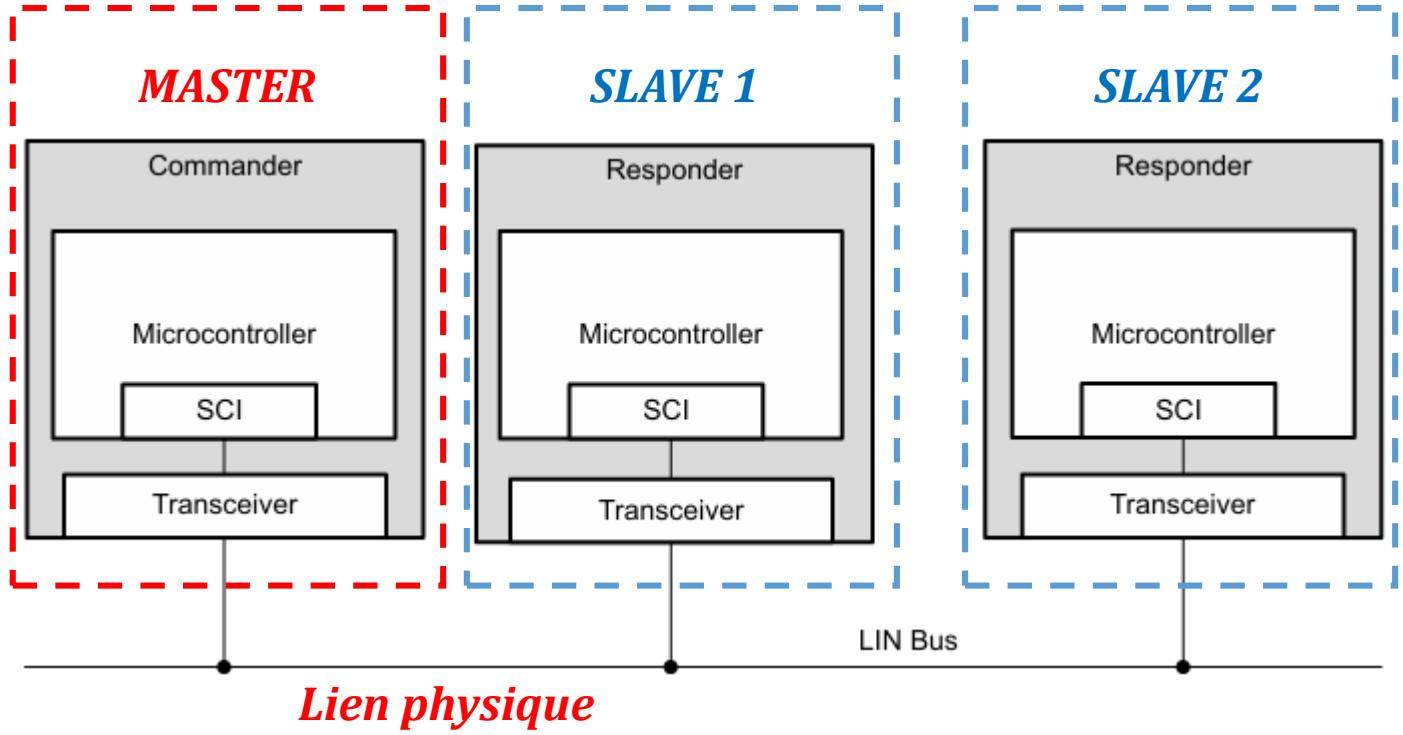


Lien physique

Architecture du bus LIN

4-

Une **résistance de pull-up** est utilisée pour maintenir le **niveau de tension du bus à la tension d'alimentation** lorsque tous les nœuds sont inactifs Cette résistance garantit que le bus LIN reste dans un état défini lorsqu'il n'y a pas de signal actif, ce qui empêche des fluctuations de signal indésirables et assure une lecture stable pour les dispositifs connectés.



Lien physique

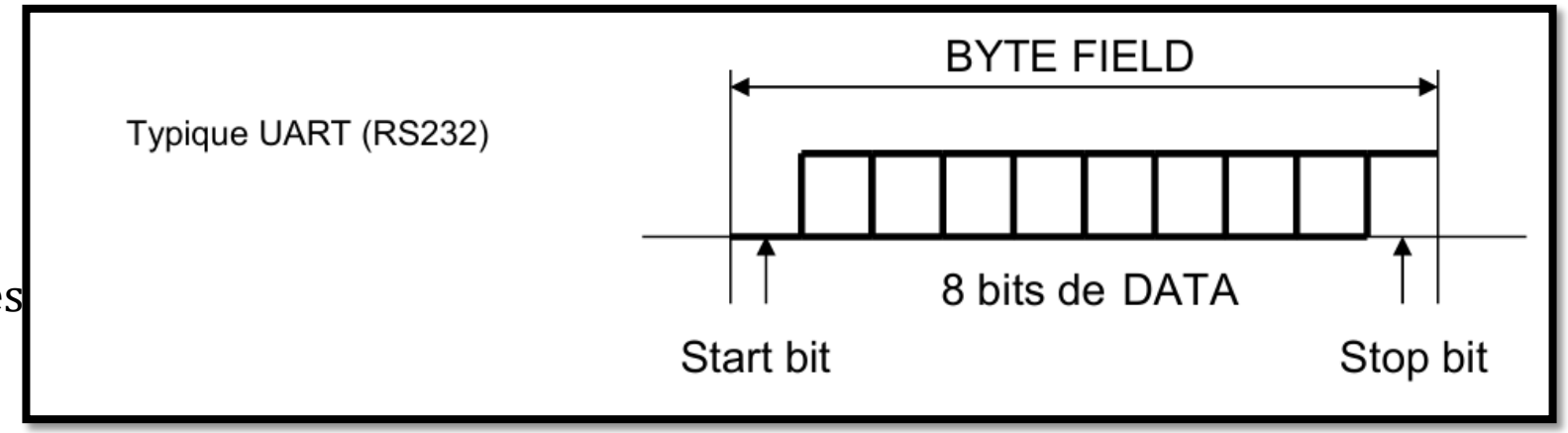
Rôle du bloc SCI: Serial Communication Interface

- La communication série est la méthode la plus utilisée pour communiquer entre les microcontrôleurs du master et des esclaves
- Le protocole UART: Universal Asynchronous Receiver Transmitter est la solution la plus utilisée pour assurer cette communication

1. Le µC transmis les trames, en commençant par le bit dominant (**Start Bit**). Cela permet de synchroniser tous les récepteurs dans le bus.
2. Après, les autres bits de poids le plus faible vers le poids le plus fort sont envoyés
3. Après un bit de stop **(Stop Bit)** est envoyé
4. Cela constitue une trame de donnée

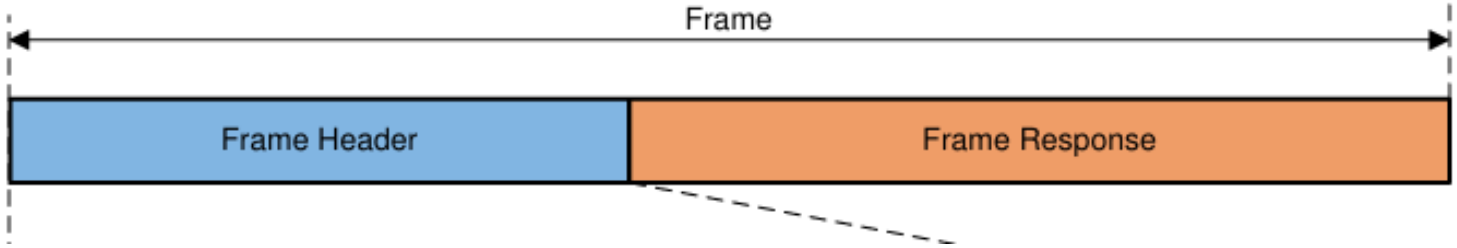
de bus LIN

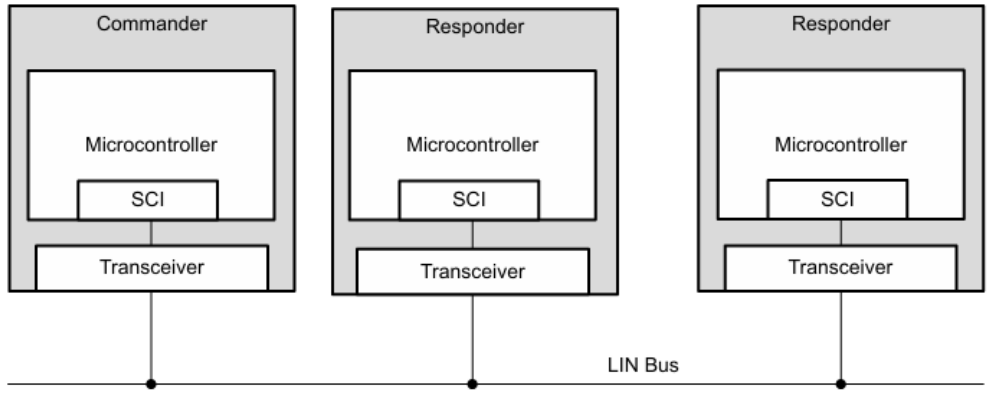
5. Le message est composé de plusieurs trames



Principe de communication maître-esclave

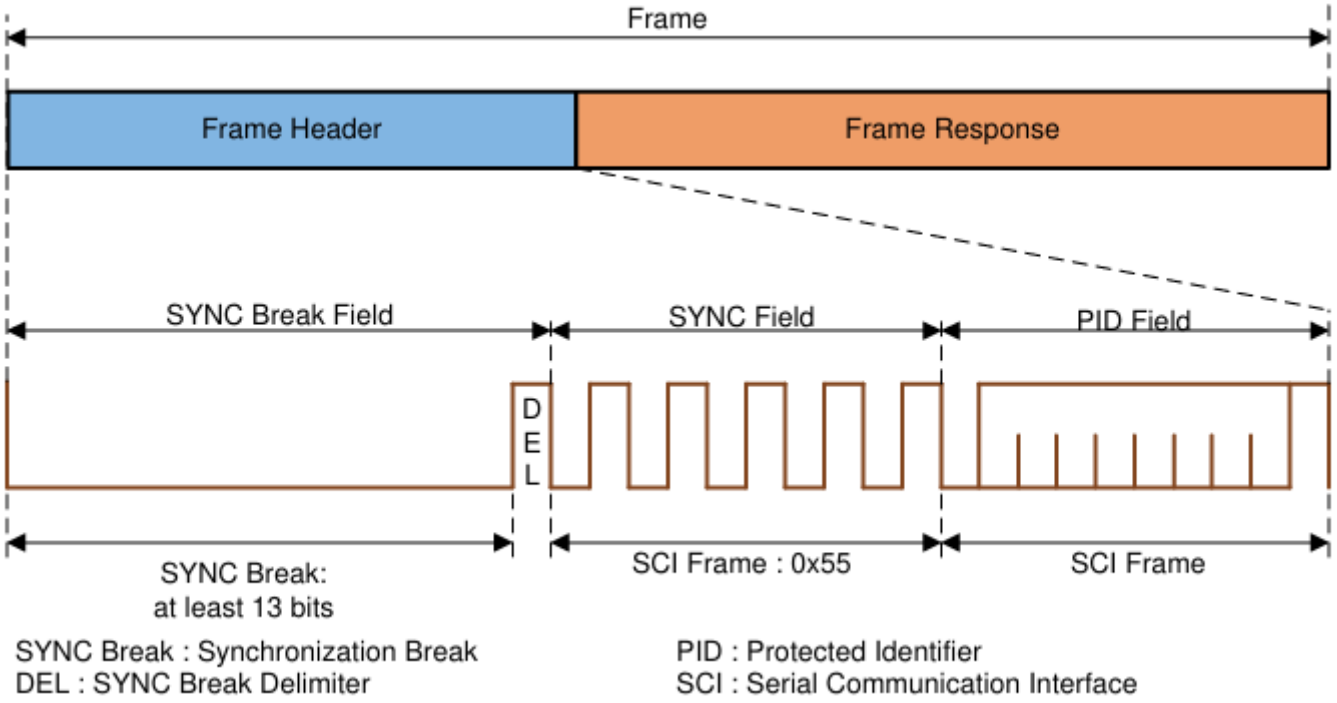
- Les esclaves ne peuvent pas communiquer entre eux. Ils peuvent répondre seulement au maître
- Le maitre envoie une requête à l'une des esclaves en définissant l'entête de la trame (Frame Hedear)
- L'esclave renvoie la réponse en tant que Frame Response
- Il existe des cas ou le maître envoie l'entête et aussi la réponse
- Dans ce dernier cas, l'esclave concerné ne vas pas répondre mais seulement il écoute
- L'inconvénient principale de ce type d'architecture est que si le maître tombe en panne, tous le cluster tombe en panne ce qui limite l'application du bus LIN pour les systèmes non critiques dans le véhicule





Format de la trame LIN

- Chaque message du bus LIN est constitué de deux parties: l'entête et la donnée
- L'entête est souvent transmis par le maître et il est composé de trois champs: Sync Breake, Sync Field et le PID
- Le Sync Field et le Sync Breake sont utilisés pour synchroniser tous les esclaves sur le bus
- Le champ PID contient l'identifiant de l'esclave qui va répondre, ce qui permet aux autres esclaves d'ignorer le message envoyé par le maître

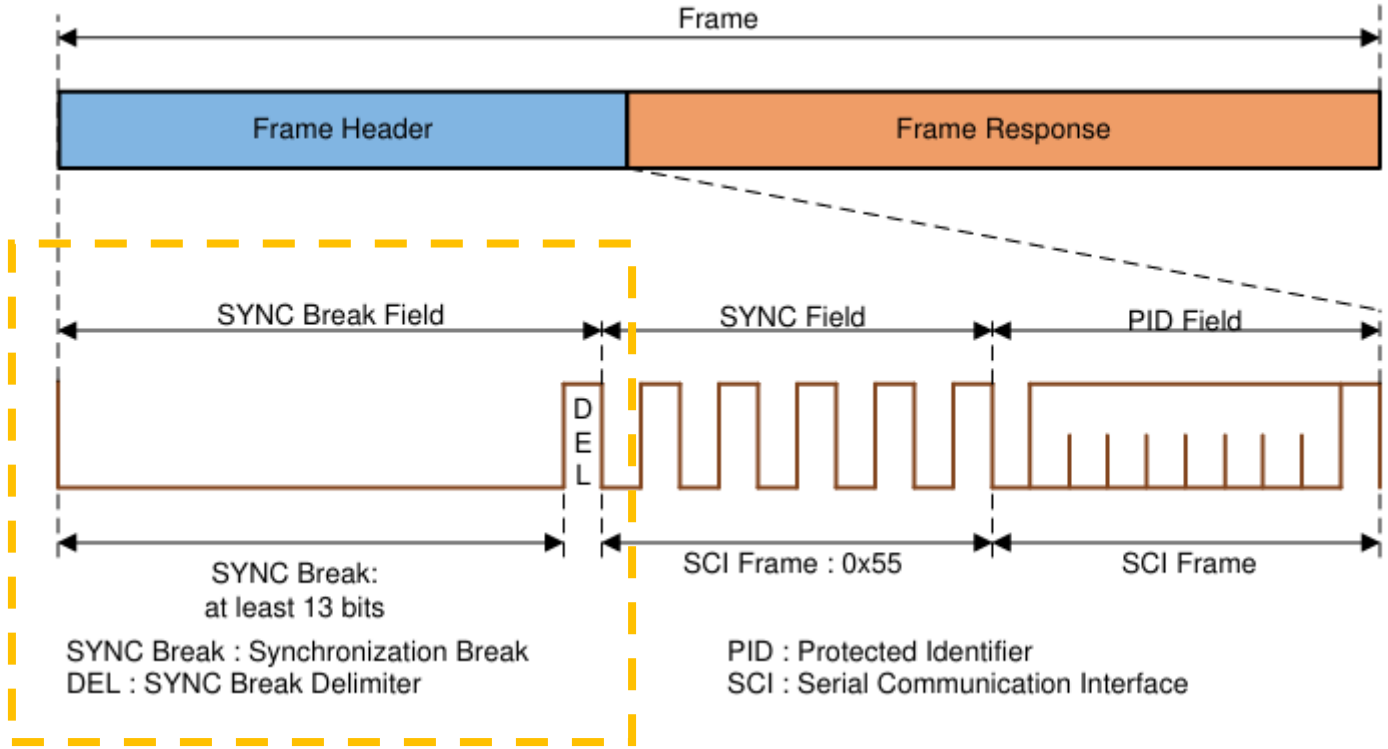


Format de la trame LIN (Suite)

- Le champ Sync Break Field est un champ permettant d'informer les esclave que le maitre est entrains de préparer l'envoi d'un message

- Il est constitué de 13 bits dominants (0)

- Après ces bits dominants, un bit récessif (1) est envoyé pour indiquer la fin du champ Sync Breake Field

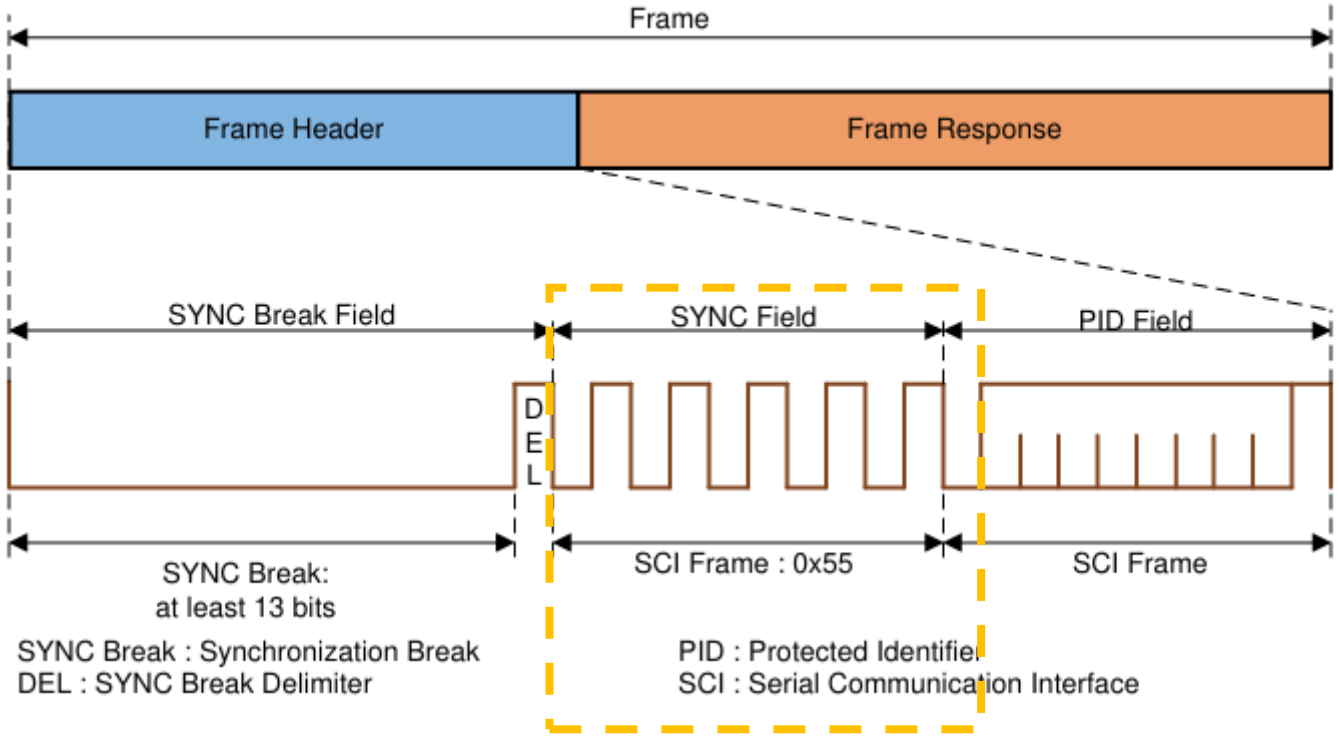


Remarque

- L'utilisation d'un signal faible (0V) permet à tous les esclaves de le distinguer par rapport aux données

Format de la trame LIN (Suite)

- Le champ Sync Field est un champ permettant de synchroniser tous les vitesses de transmissions (Baude Rate en Bit/s) entre le master et les autres esclaves. Puisque la communication est asynchrone, il est judicieux d'avoir les mêmes vitesses sinon on aura des pertes de données.
- Il est constitué de 8 bits alternés 0 et 1
- 0b01010101 soit (0x55 en Hexadécimal)

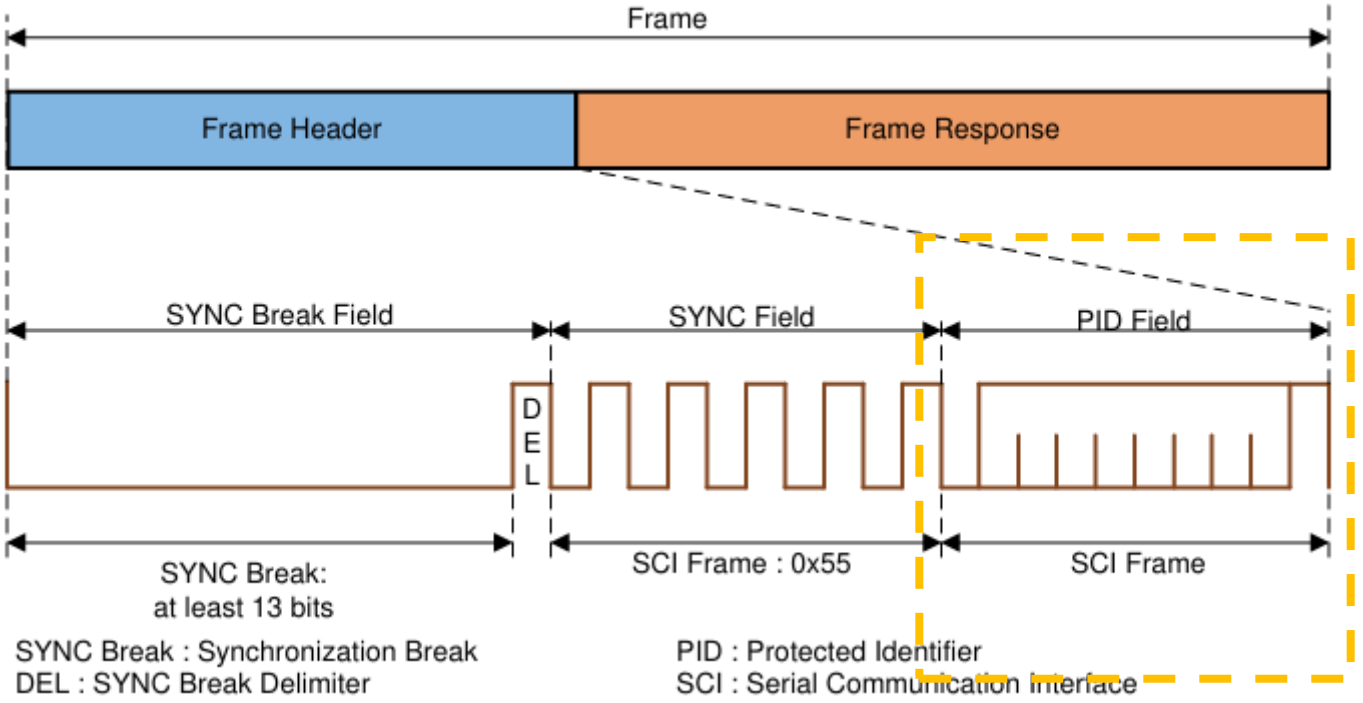


Remarque

- Ce champ crée un signal d'horloge de forme carrée que les esclaves utilisent pour calibrer leurs horloges internes
- Cette synchronisation de type software permet de minimiser le matériel nécessaire pour la génération du signal d'horloge

Format de la trame LIN (Suite)

- Le champ PID Protected Identifier est un champ permettant d'identifier le nœud esclave concerné par la requête
- Il constitué de 10 Bits:
  - 1 Start Bit,
  - 6 Bits pour l'identifiant du nœud et du type de message,
  - 2 Bits pour la parité et 1 Bits de Stop



Remarque

- Chaque ID correspond à une donnée spécifique d'un capteur ou d'une commande d'un actionneur
- Avec 6 bits on peut avoir 63 messages au total sauf que certain ID sont réservés pour des fonctions de diagnostique ce qui laisse à peu prés 60 différents messages dans le bus LIN

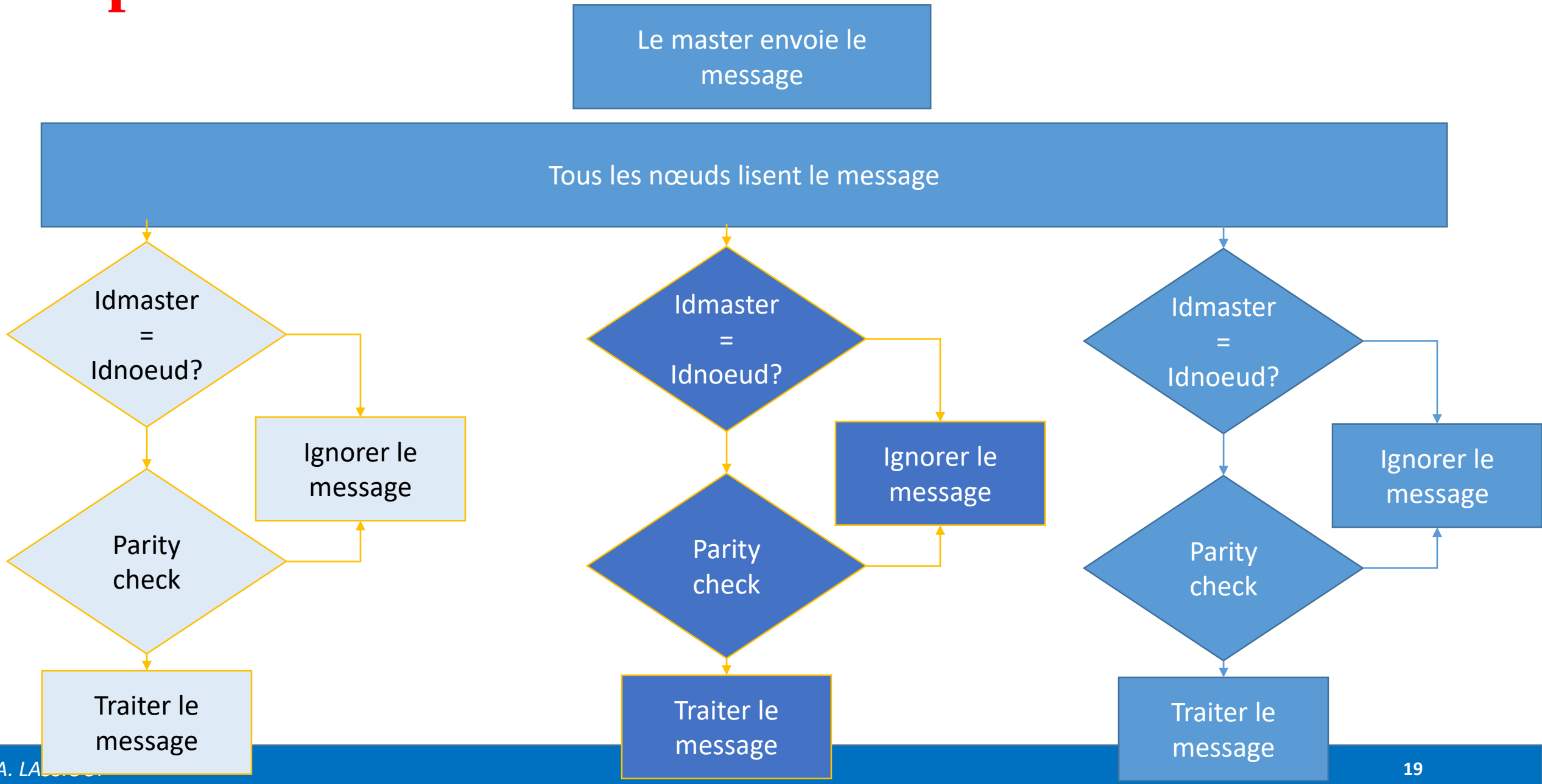
Format de la trame LIN (Suite)

- **Le rôle des deux bits de parités**
- Les deux bits de parités permettent de s'assurer de l'intégrité du signal dans le bus LIN et de détecter d'éventuelles erreurs de transmission.
- Ils sont calculer de la manière suivante:
- **P0** = ID0 **XOR** ID1 **XOR** ID2 **XOR** ID4 (à la sortie on aura soit 1 soit 0 en fonction de nombre des 1 et des 0)
- **P1** = ID1 **XOR** ID3 **XOR** ID4 **XOR** ID5
- Les bits ID1 et ID4 sont utilisés dans les deux séquences de calculs ce qui ajoutent un niveau de redondance et améliore la détection des erreurs

Exemple:

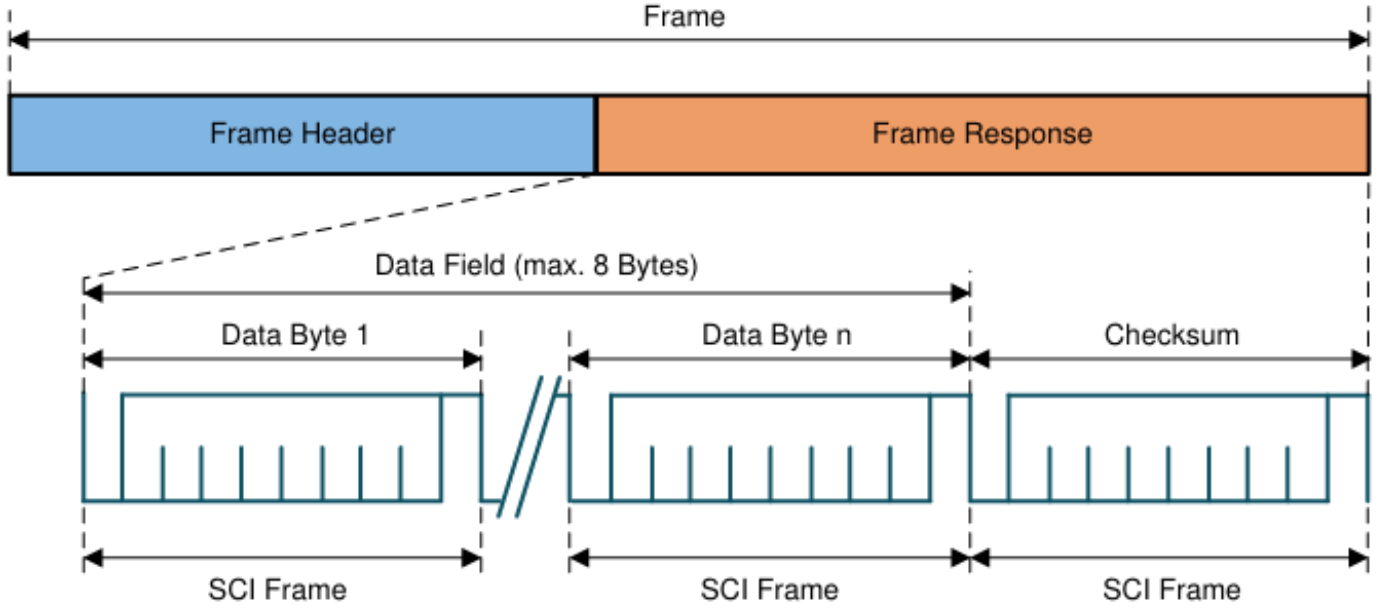
- Soit ID = 0b001010
- P0 = (0 xor 0) xor 1 xor 1 = 0
- P1 = 0 xor 0 xor 1 xor 0 = 1

Le processus de fonctionnement du bus LIN



Format de la trame LIN: le champ de donnée

- Le champ de donnée est envoyé soit par le master soit par l'esclave
- Le champ de donnée est subdivisé en plusieurs octets (Bytes: 8 Bits) maximum 8
- Chaque octet de donnée est séparé par apport aux autres à travers un bit de Start et un bit de Stop
- Finalement un champ de checksum est utilisé pour vérifier que le message envoyé est bien celui reçu sans perte de données.



- Le champ de checksum est calculé en sommant tous les octets des données et en inversant le résultat. Il est constitué en générale de 10 Bits (un Bit de Start, 8 Bits de Checksum et un Bit de Stop)

Format de la trame LIN: le checksum

- Exemple de calcul de checjsum
- Soit : Data bytes: 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE, 0xF0
- 0x12 + 0x34 = 0x46
- 0x46 + 0x56 = 0x9C
- 0x9C + 0x78 = 0x114 (On peut supprimer la retenue et garder seulement 0x14)
- 0x14 + 0x9A = 0xAE
- 0xAE + 0xBC = 0x16A (On garde seulement 0x6A)
- 0x6A + 0xDE = 0x148 (On garde 0x48)
- 0x48 + 0F0 = 0x38

- Après on inverse le résultat obtenue: Inv(0x38) = 0xC7

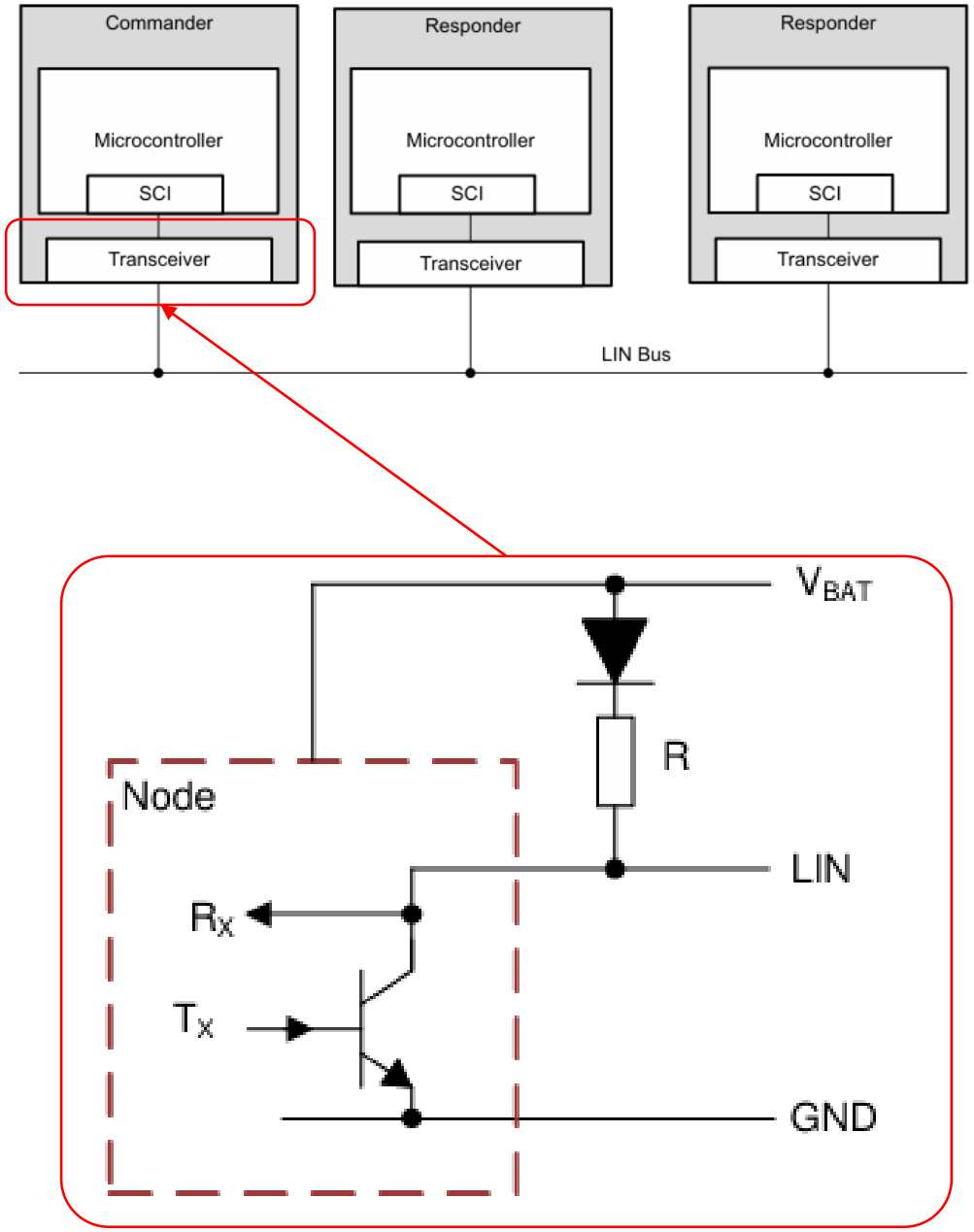
- Le checksum est alors 0xC7

- 0x9C = 1001 1100
- 0x78 = 0111 1000

```
  1001 1100
+ 0111 1000
-----------
 1 0001 0100
```

La couche physique du bus LIN

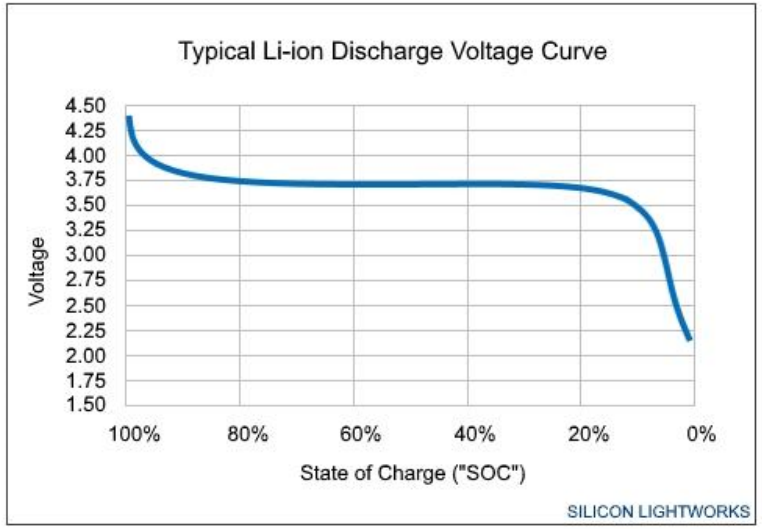
- La figure suivante illustre le schéma simplifié d'un driver du bus LIN (tranceiver).
- Il permet de convertir les signaux du niveau du microcontrôleur (souvent 3.3V ou 5V) vers des tensions élevées (niveau batterie, souvent 12V en automobile).
- Quand un nœud envoie un message à travers la pin TX, cette dernière contrôle l'état du transistor NPN.
- Si TX =1 alors LIN = 0 (0V)
- Si TX = 0 alors LIN = 1 (12V ou 24V)
- La pin RX est utilisée pour surveiller le réseau LIN à travers un diviseur de tension permettant d'adapter la tension au niveau du microcontrôleur
- **Remarque:**
- Quand aucun nœud ne transmet, le réseau LIN est à l'état récessif



Les tensions de seuils des niveaux récessifs et dominants

- La tension de la batterie du véhicule varie en fonction de l'état de charge
- Par exemple une batterie chargée à 100% peut avoir une tension de 14V
- Lors de sa charge sa tension peut être de 14.5V
- Une batterie déchargée à 20% peut avoir une tension de 12.2V
- Les résistances associées aux câbles peuvent également introduire des chutes de tensions au niveau du bus LIN.

- Il est donc judicieux de spécifier les marges de tensions dans un bus LIN (à partir de quel seuil on peut considérer que c'est 1 ou bien c'est 0)

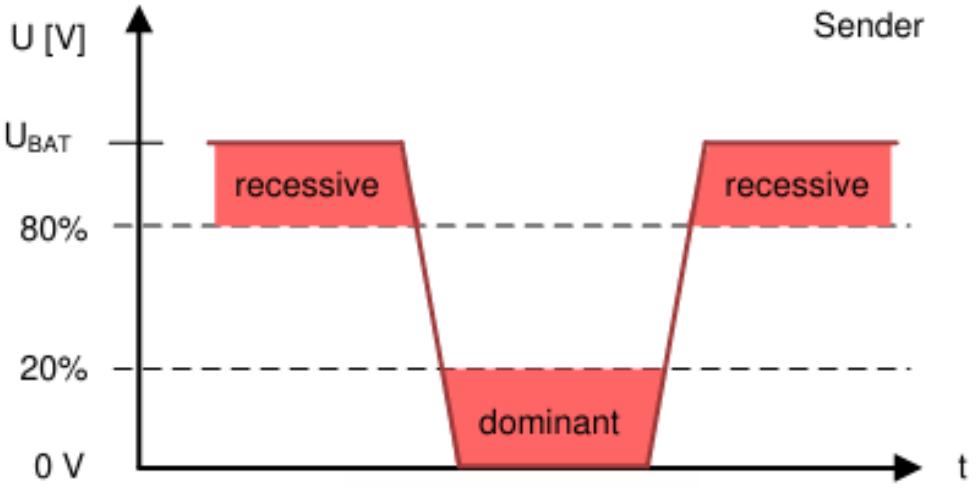


Cleversolarpower.com

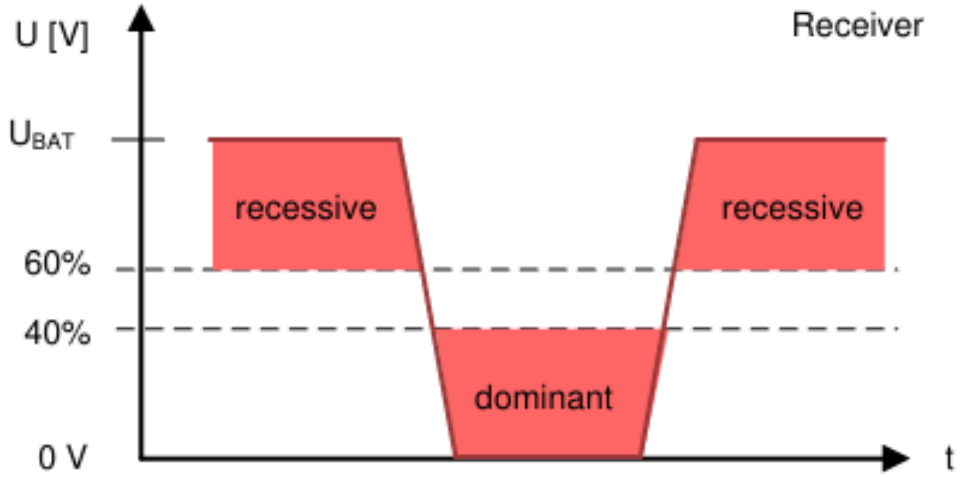
Percentage (SOC)	1 Cell	12V	24V	48V
100% Charging	3.65	14.6	29.2	58.4
100% Rest	3.40	13.6	27.2	54.4
90%	3.35	13.4	26.8	53.6
80%	3.32	13.3	26.6	53.1
70%	3.30	13.2	26.4	52.8
60%	3.27	13.1	26.1	52.3
50%	3.26	13.0	26.1	52.2
40%	3.25	13.0	26.0	52.0
30%	3.22	12.9	25.8	51.5
20%	3.20	12.8	25.6	51.2
10%	3.00	12.0	24.0	48.0
0%	2.50	10.0	20.0	40.0

Les tensions de seuils des niveaux récessifs et dominants

- **Du côté de l'émetteur (Sender)**
  - le bus est dit à l'état récessif si et seulement si la tension du bus est supérieure ou égale à 80% de la tension de la batterie.
  - Le bus est dit à l'état dominant si et seulement si la tension du bus est inférieure ou égale à 20% de la tension de la batterie

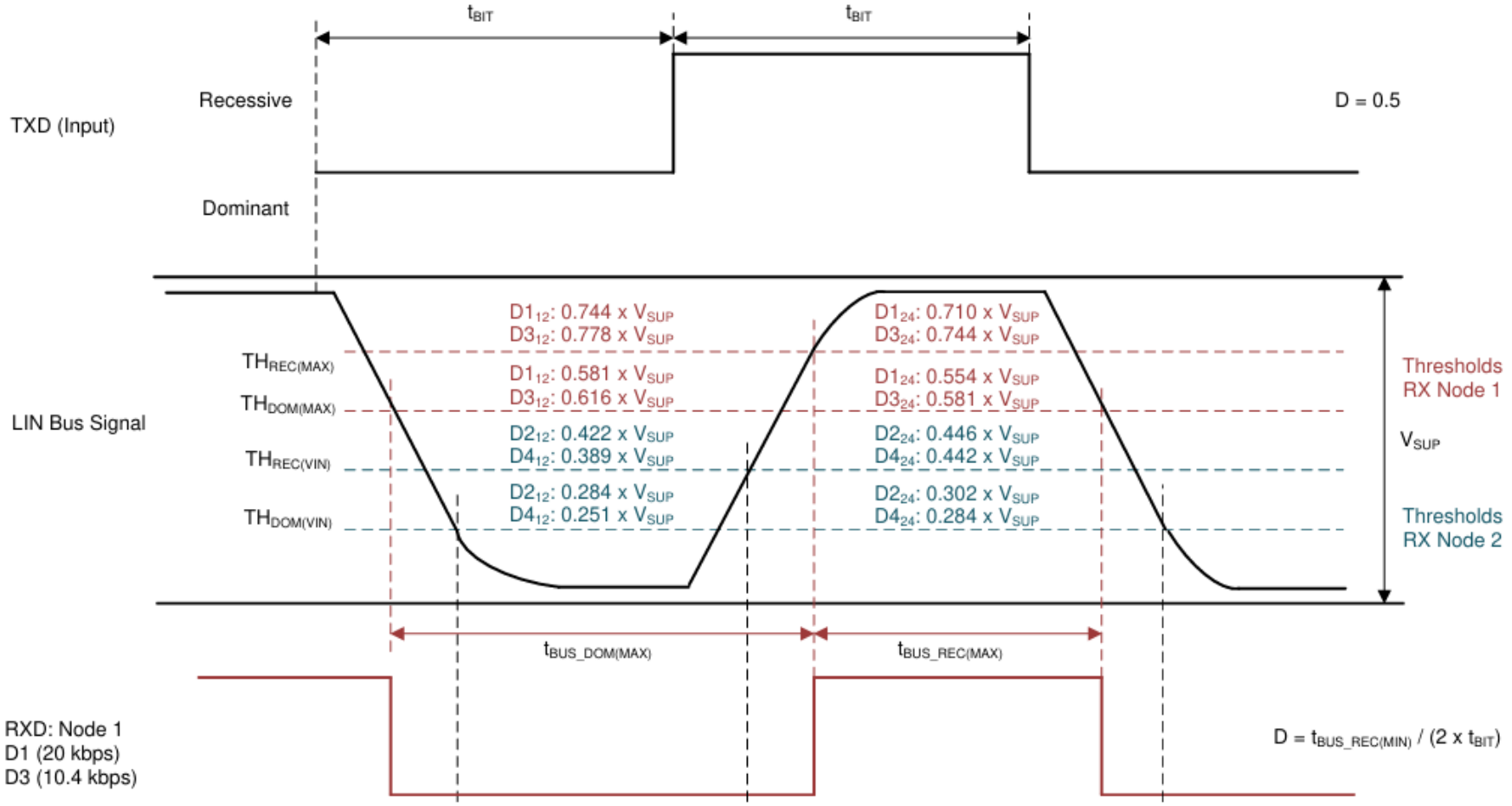


- **Du côté de récepteur (Receiver)**
  - le bus est dit à l'état récessif si et seulement si la tension du bus est supérieure ou égale à 60% de la tension de la batterie.
  - Le bus est dit à l'état dominant si et seulement si la tension du bus est inférieure ou égale à 40% de la tension de la batterie



Les délais entre la transmission et la réception

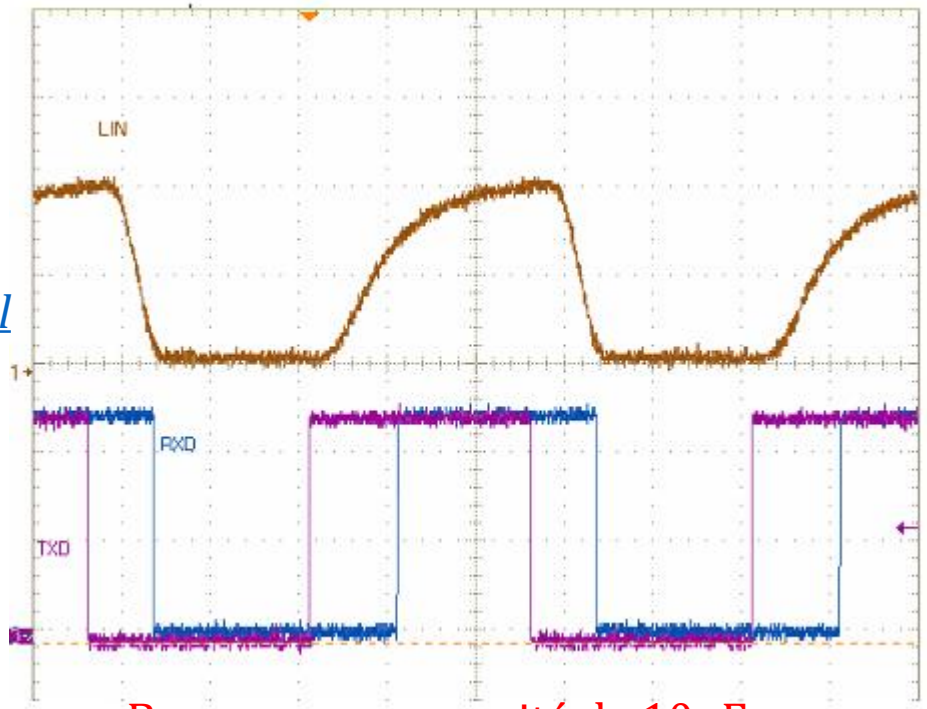
- Les lignes de transmissions de données ont un comportement capacitif ce qui ajoute des délais lors des passages de 1 à 0 et inversement.
- Ce qui engendre des retards entre l'émission et la réception du message comme le montre la figure suivante



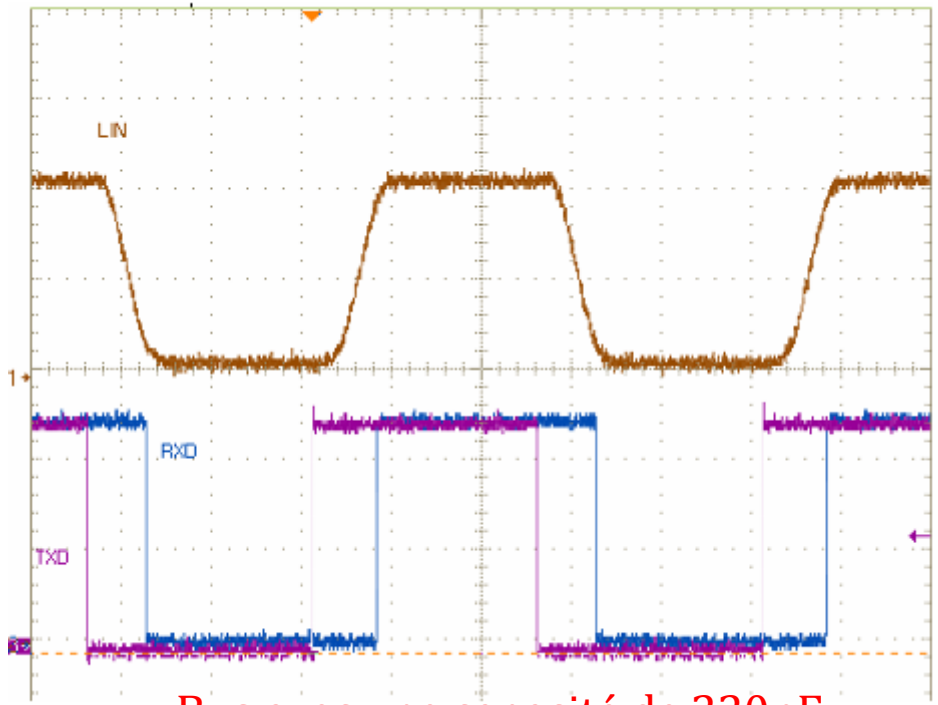
Les délais en fonction de la capacité du bus LIN

- Les figures suivantes illustrent le comportement du bus LIN en fonction de la capacité du bus.
- En chargeant le bus par une capacité de 220pF on remarque des temps de montés et de descentes faibles et des délais faibles aussi entre la transmission et la réception
- En chargeant le bus par une capacité de 10nF on remarque des temps de montés et de descentes élevés ainsi qu'un retard important entre la transmission et la réception.

Source: LIN Protocol and Physical Layer Requirements (Rev. A)



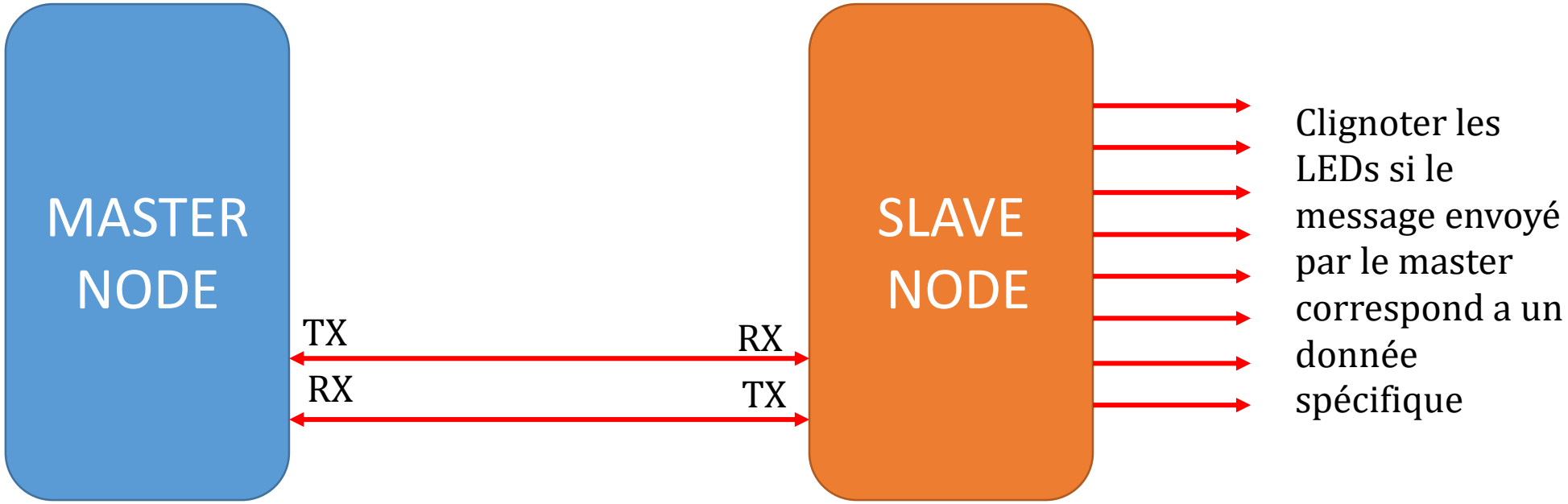
Bus avec une capacité de 10nF



Bus avec une capacité de 220pF

Activité pratique 1

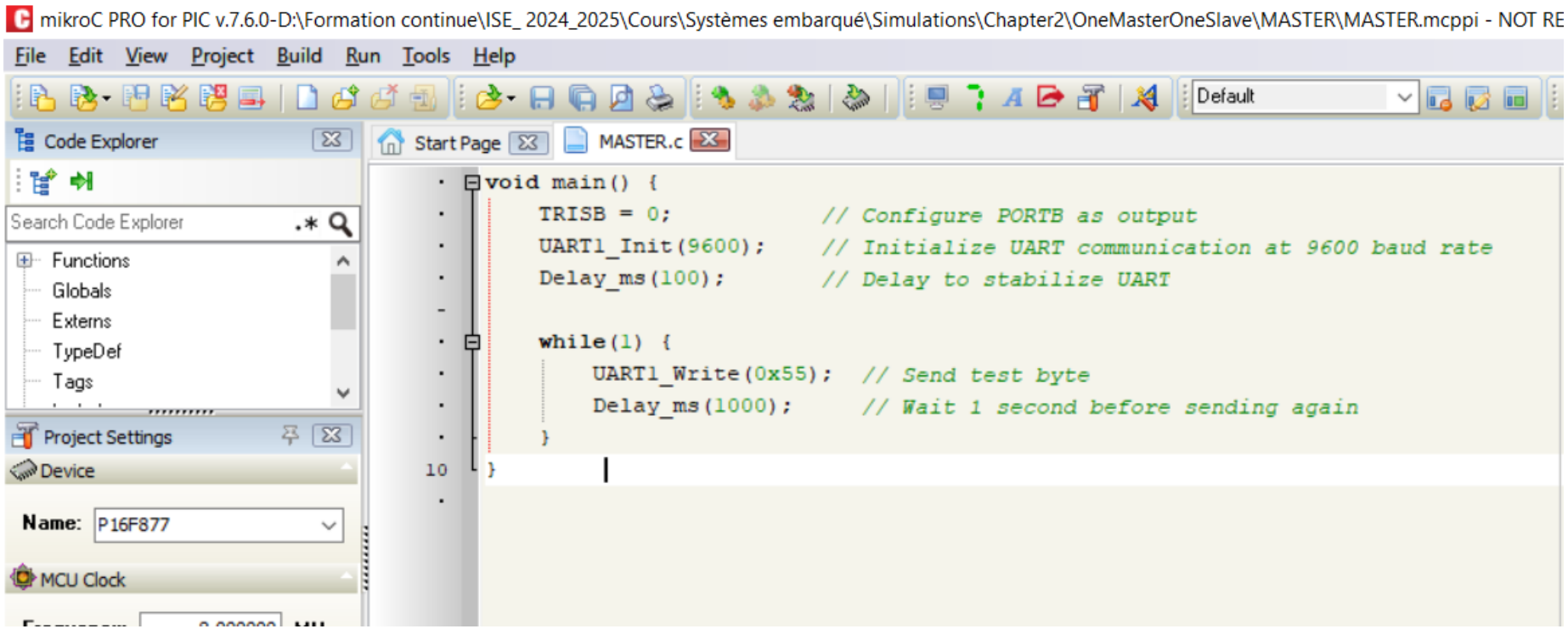
- L'objectif de cette première activité pratique est de tester la communication série à travers le protocole UART entre un nœud Master et un Nœud Esclave. Le nœud ESCLAVE doit clignoter les LEDs si une donnée spécifique est envoyée par le MASTER. Les deux nœuds sont basées sur des microcontrôleurs de type PIC 16F877.



Activité pratique 1 (Solution)

Le programme du nœud MASTER est illustré dans la figure suivante.

Le master envoi un message à travers la fonction UART1_Write(0x55) au nœud esclave à travers le bus UART

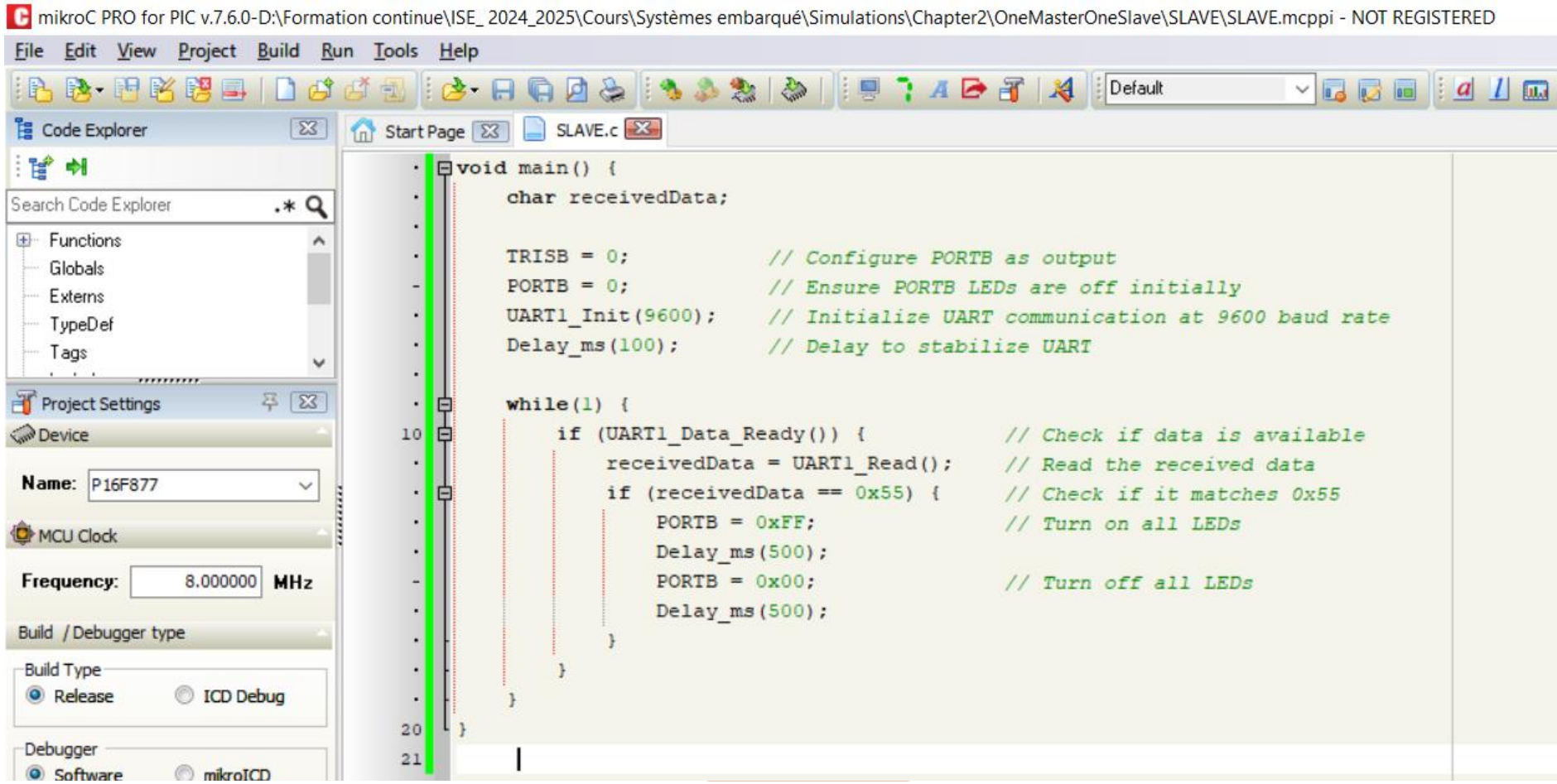


Activité pratique 1 (Solution)

Le programme du nœud esclave est illustré dans la figure suivante:

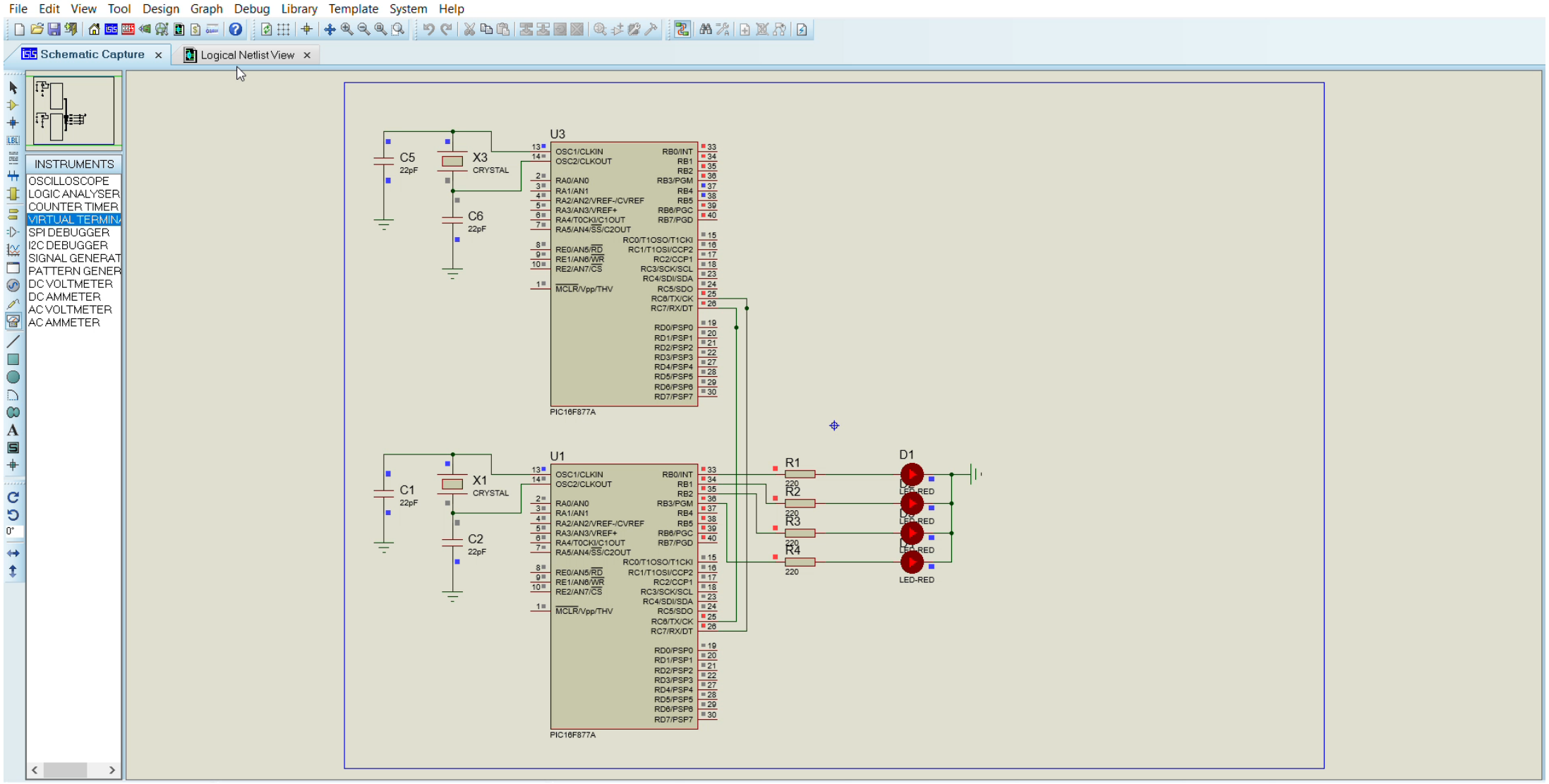
Les LEDs sont connectées au port B du PIC

Si la donnée reçue à travers le bus UART est égale à 0x55 alors le nœud esclave clignote les LEDs connectées à son port B



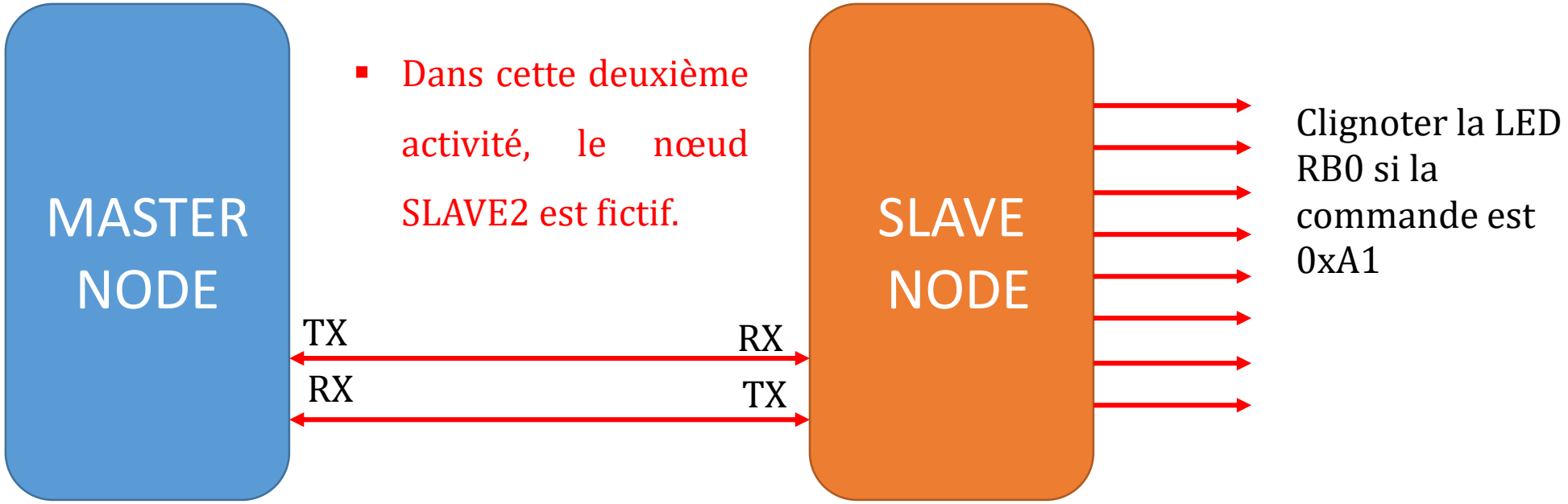
Activité pratique 1 (Solution)

Simuler l'activité dans Isis Proteus et vérifier le bon fonctionnement des deux programmes.



Activité pratique 2

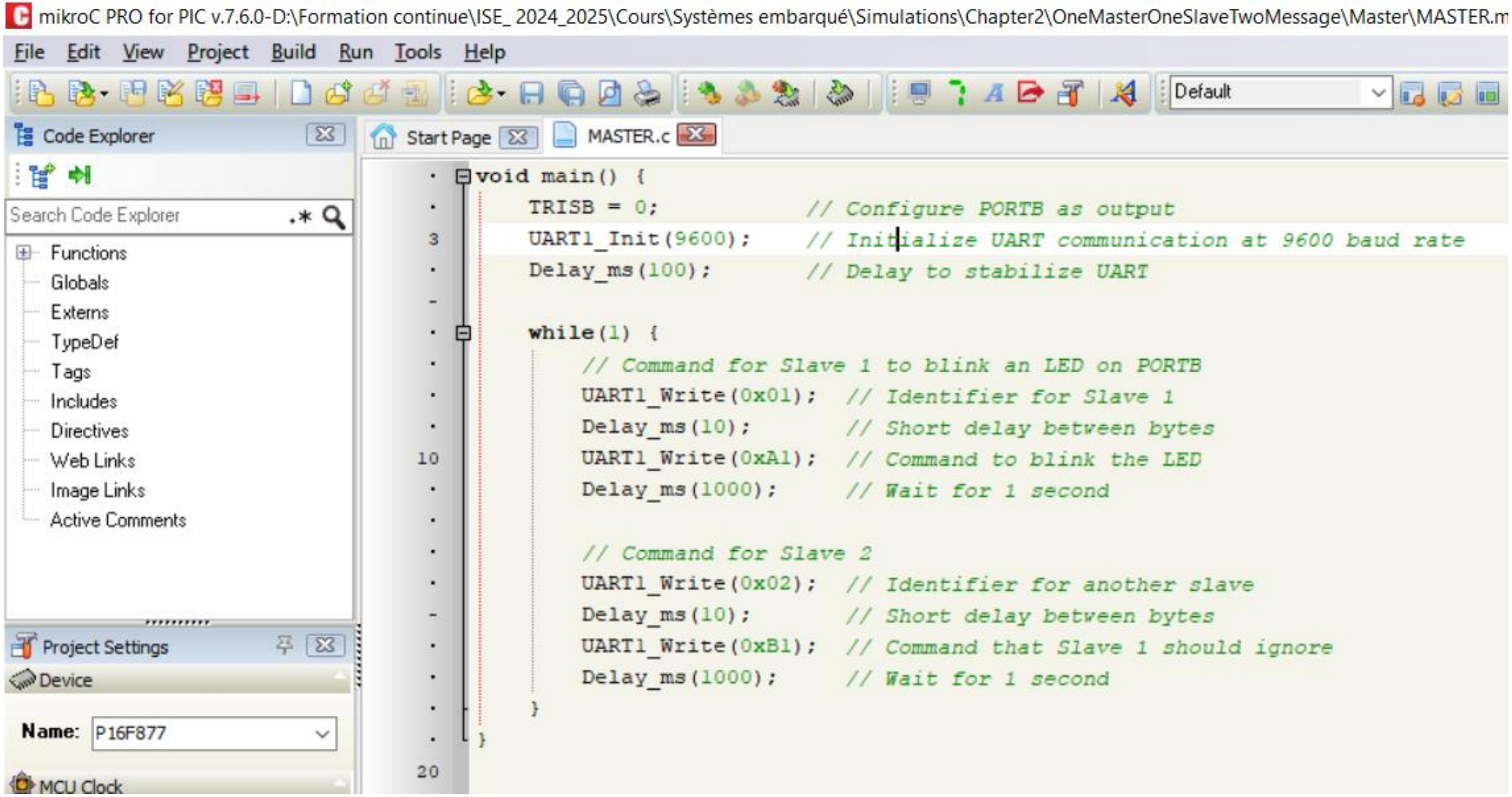
- L'objectif de cette deuxième activité est de modifier le code du nœud Master de telle sorte à ce qu'il envoie deux commande avec deux identifiants différents:
- On suppose que l'identifiant du nœud SLAVE1 est 0x01 et la commande 0x0A permet de clignoter une seule LED connectée au port B0.
- On suppose que le MASTER envoie aussi une deuxième commande (0xB1) pour un autre nœud dont l'identifiant est 0x02.



- Dans cette deuxième activité, le nœud SLAVE2 est fictif.

Activité pratique 2 (Solution)

- La figure suivante illustre le code du nœud MASTER

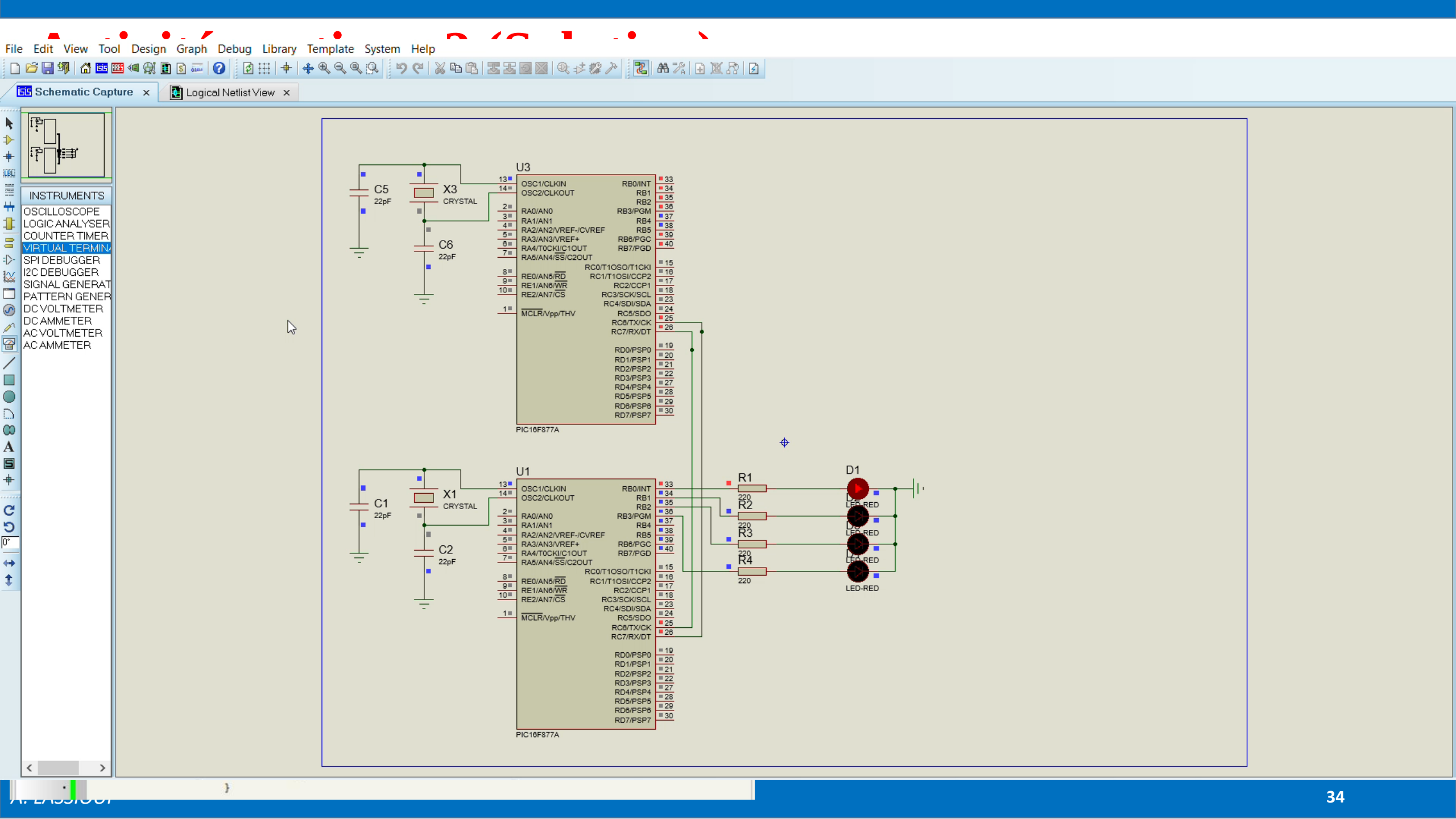


Activité pratique 2 (Solution)

- La figure suivante illustre le code du nœud SLAVE1

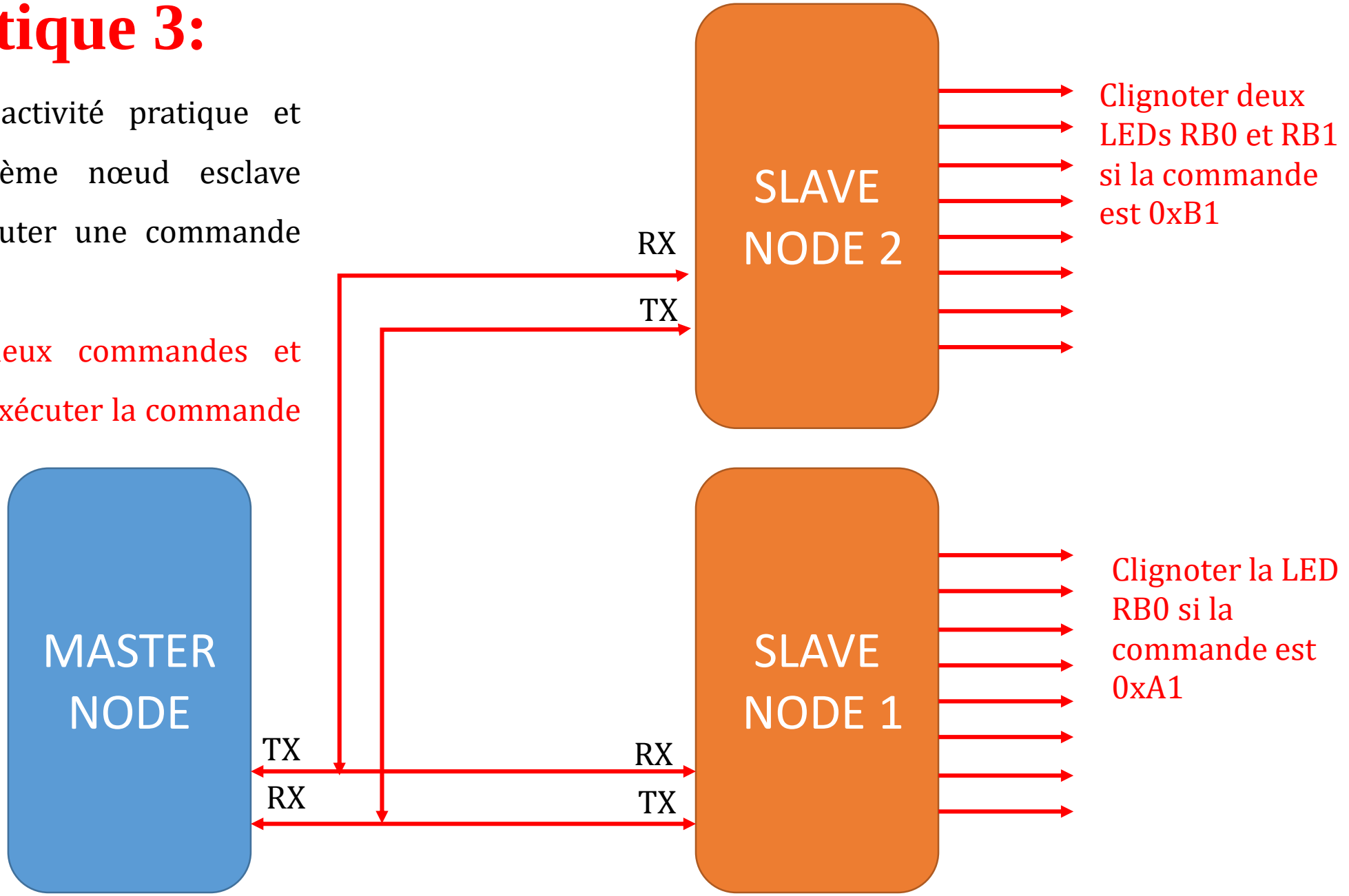
Start Page | SLAVE.c

```
void main() {
    char identifier;
    char command;
    int blinking = 0;  // Flag to control continuous blinking

    TRISB = 0;           // Configure PORTB as output
    PORTB = 0;           // Ensure LEDs are initially off
    UART1_Init(9600);    // Initialize UART communication at 9600 baud rate
    Delay_ms(100);       // Delay to stabilize UART

    while(1) {
        // Check for new data only when UART data is ready
        if (UART1_Data_Ready()) {
            identifier = UART1_Read();   // Read the identifier

            // Check if the identifier matches this slave (0x01)
            if (identifier == 0x01) {
                while (!UART1_Data_Ready());  // Wait for the command byte
                command = UART1_Read();       // Read the command

                // Execute action based on command
                if (command == 0xA1) {        // Start blinking command
                    blinking = 1;             // Set blinking flag
                } else if (command == 0xA0) { // Stop blinking command
                    blinking = 0;             // Clear blinking flag
                    PORTB = 0x00;             // Turn off LED
                }
            } else {
                // If identifier doesn't match, discard any extra bytes
                while (UART1_Data_Ready()) {
                    UART1_Read();
                }
            }

            // If blinking flag is set, keep LED blinking
            if (blinking) {
                PORTB = 0x01;    // Turn on LED connected to RB0
                Delay_ms(500);   // Keep LED on for 500 ms
                PORTB = 0x00;    // Turn off LED
                Delay_ms(500);   // Keep LED off for 500 ms
            }
        }
    }
}
```


File Edit View Tool Design Graph Debug Library Template System Help
Schematic Capture
Logical Netlist View
INSTRUMENTS
OSCILLOSCOPE
LOGIC ANALYSER
COUNTER TIMER
VIRTUAL TERMIN
SPI DEBUGGER
I2C DEBUGGER
SIGNAL GENERAT
PATTERN GENER
DC VOLTMETER
DC AMMETER
AC VOLTMETER
AC AMMETER
U3
C5
22pF
X3
CRYSTAL
C6
22pF
OSC1/CLKIN
OSC2/CLKOUT
RA0/AN0
RA1/AN1
RA2/AN2/VREF-/CVREF
RA3/AN3/VREF+
RA4/T0CKI/C1OUT
RA5/AN4/SS/C2OUT
RE0/AN5/RD
RE1/AN6/WR
RE2/AN7/CS
MCLR/Vpp/THV
RB0/INT
RB1
RB2
RB3/PGM
RB4
RB5
RB6/PGC
RB7/PGD
RC0/T1OSO/T1CKI
RC1/T1OSI/CCP2
RC2/CCP1
RC3/SCK/SCL
RC4/SDI/SDA
RC5/SDO
RC6/TX/CK
RC7/RX/DT
RD0/PSP0
RD1/PSP1
RD2/PSP2
RD3/PSP3
RD4/PSP4
RD5/PSP5
RD6/PSP6
RD7/PSP7
PIC16F877A
U1
C1
22pF
X1
CRYSTAL
C2
22pF
R1
220
R2
220
R3
220
R4
220
D1
LED-RED
LED-RED
LED-RED
LED-RED

Activité pratique 3:

- L'objectif de cette activité pratique et d'ajouter un deuxième nœud esclave (SLAVE2) pour exécuter une commande donnée.
- Le master envoi deux commandes et chaque esclave doit exécuter la commande lui concernant



Activité pratique 3: SOLUTION

- Le code du master est le suivant:

Start Page | MASTER.c

```
void main() {
    TRISB = 0;            // Configure PORTB as output
    UART1_Init(9600);     // Initialize UART communication at 9600 baud rate
    Delay_ms(100);        // Delay to stabilize UART

    while(1) {
        // Command Slave 1 to blink one LED on PORTB
        UART1_Write(0x01);  // Identifier for Slave 1
         Delay_ms(10);
        UART1_Write(0xA1);  // Command to blink the LED
        Delay_ms(2000);     // Wait for 2 seconds before next command

        // Command Slave 2 to blink two LEDs on PORTB
        UART1_Write(0x02);  // Identifier for Slave 2
        Delay_ms(10);
        UART1_Write(0xB1);  // Command to blink two LEDs
        Delay_ms(2000);     // Wait for 2 seconds before next command
```

Activité pratique 3: SOLUTION

- Le code du SLAVE1 est le suivant:

Page | SLAVE.c

```
void main() {
    char identifier;
    char command;
    int blinking = 0;  // Flag to control continuous blinking

    TRISB = 0;           // Configure PORTB as output
    PORTB = 0;           // Ensure LEDs are initially off
    UART1_Init(9600);    // Initialize UART communication at 9600 baud rate
    Delay_ms(100);       // Delay to stabilize UART

    while(1) {
        // Check for new data only when UART data is ready
        if (UART1_Data_Ready()) {
            identifier = UART1_Read();   // Read the identifier

            // Check if the identifier matches this slave (0x01)
            if (identifier == 0x01) {
                while (!UART1_Data_Ready());  // Wait for the command byte
                command = UART1_Read();       // Read the command

                // Execute action based on command
                if (command == 0xA1) {        // Start blinking command
                    blinking = 1;             // Set blinking flag
                } else if (command == 0xA0) { // Stop blinking command
                    blinking = 0;             // Clear blinking flag
                    PORTB = 0x00;             // Turn off LED
                }
            } else {
                // If identifier doesn't match, discard any extra bytes
                while (UART1_Data_Ready()) {
                    UART1_Read();
                }
            }
        }
        // If blinking flag is set, keep LED blinking
        if (blinking) {
            PORTB = 0x01;    // Turn on LED connected to RB0
            Delay_ms(500);   // Keep LED on for 500 ms
            PORTB = 0x00;    // Turn off LED
            Delay_ms(500);   // Keep LED off for 500 ms
        }
    }
}
```

Activité pratique 3: SOLUTION

- Le code du nœud SLAVE2 est le suivant:

Start Page | Slave2.c

```
void main() {
    char identifier;
    char command;
    int blinking = 0;  // Flag to control continuous blinking

    TRISB = 0;           // Configure PORTB as output
    PORTB = 0;           // Ensure LEDs are initially off
    UART1_Init(9600);    // Initialize UART communication at 9600 baud rate
    Delay_ms(100);       // Delay to stabilize UART

    while(1) {
        // Check for new data only when UART data is ready
        if (UART1_Data_Ready()) {
            identifier = UART1_Read();   // Read the identifier

            // Check if the identifier matches this slave (0x02)
            if (identifier == 0x02) {
                while (!UART1_Data_Ready());  // Wait for the command byte
                command = UART1_Read();       // Read the command

                // Execute action based on command
                if (command == 0xB1) {      // Start blinking command
                    blinking = 1;           // Set blinking flag
                } else if (command == 0xB0) { // Stop blinking command
                    blinking = 0;           // Clear blinking flag
                    PORTB = 0x00;           // Turn off LEDs
                }
            } else {
                // If identifier doesn't match, discard any extra bytes
                while (UART1_Data_Ready()) {
                    UART1_Read();
                }
            }
        }

        // If blinking flag is set, keep LEDs blinking
        if (blinking) {
            PORTB = 0x03;   // Turn on both LEDs connected to RB0 and RB1
            Delay_ms(500);   // Keep LEDs on for 500 ms
            PORTB = 0x00;    // Turn off both LEDs
            Delay_ms(500);   // Keep LEDs off for 500 ms
        }
    }
}
```

Activité pratique 3 (Solution)

- La figure suivante illustre le schéma sous ISIS Proteus