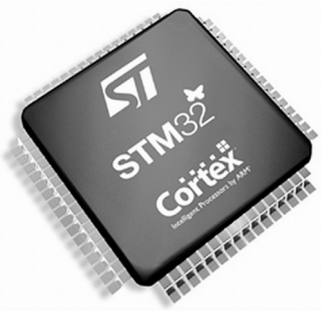
IUT
CERGY-PONTOISE



Périphériques

BUS
UART, I2C et SPI

J.Lorandel & O.Romain
Maj: F. Ghaffari & S.Zuckerman – 2020

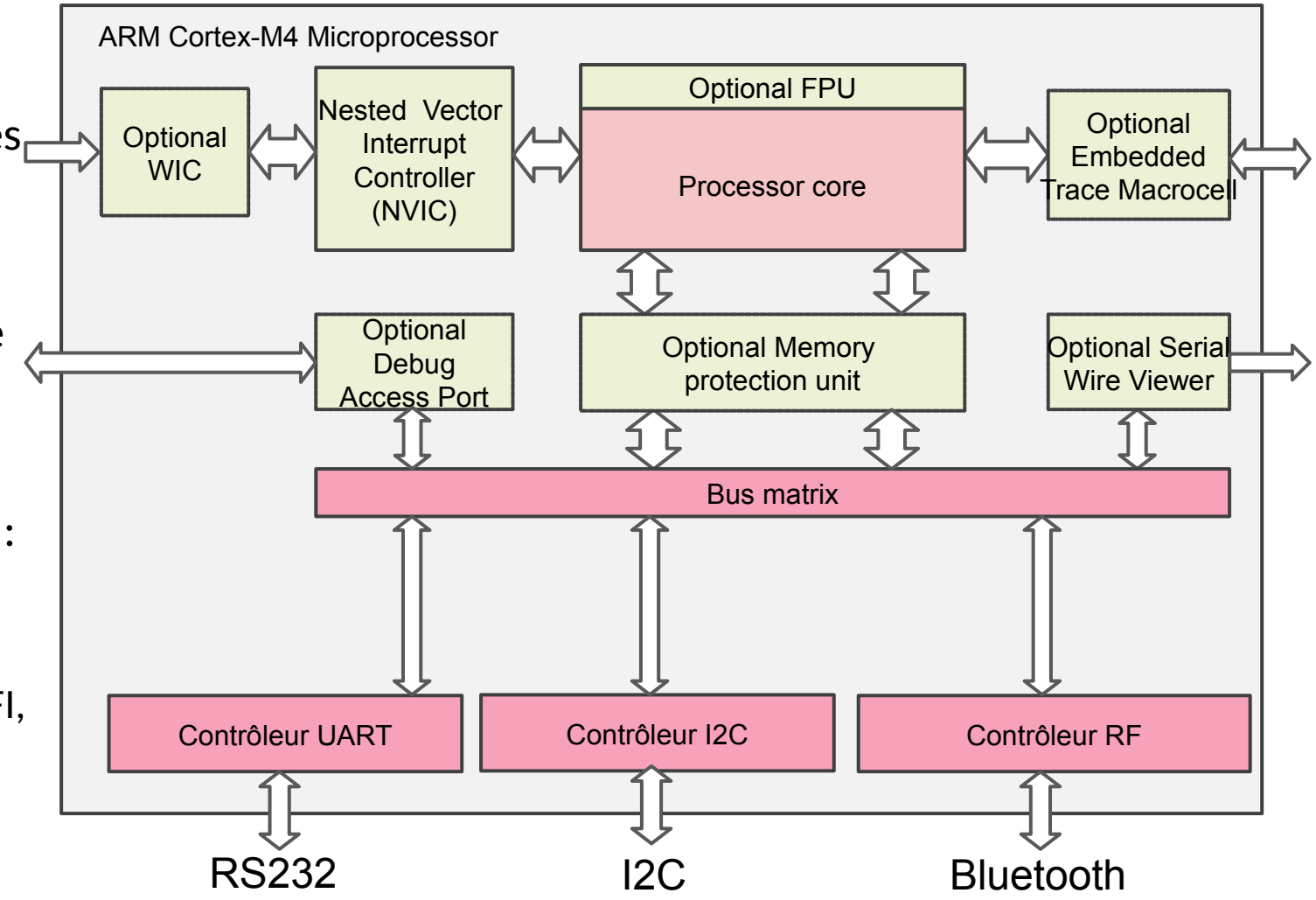
Sommaire

- **Généralités sur les bus**
- **Partie 1 : UART**
  - Fonctionnalité
  - Registres de configurations
  - Exemple de programmation
- **Partie 3 : I2C**
  - Fonctionnalité
  - Registres de configuration
  - Exemple de programmation
- **Partie 2 : SPI**
  - Fonctionnalité
  - Registres de configuration
  - Exemple de programmation

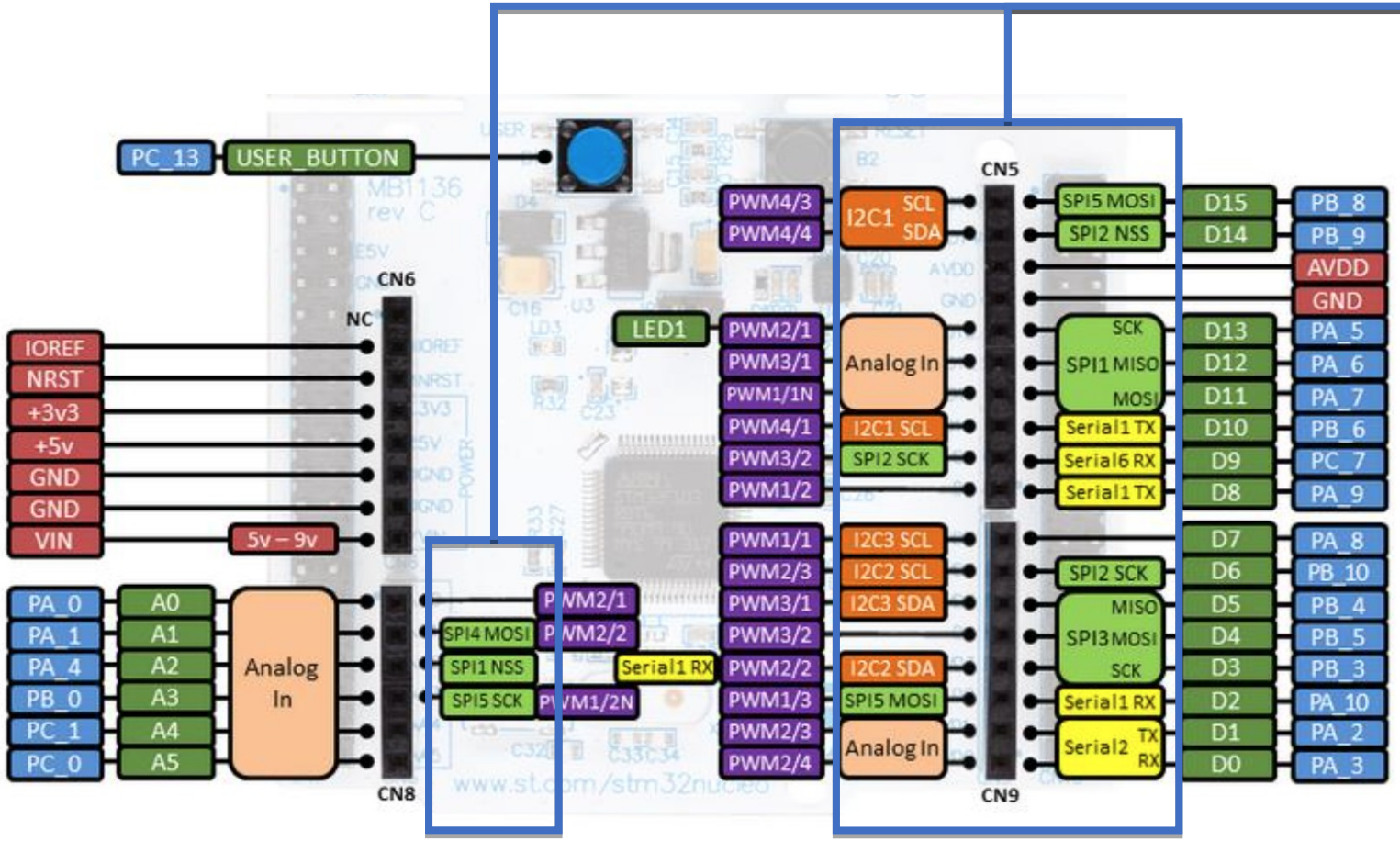
Séance 1

Mode de communication

- Aujourd'hui, il existe plusieurs modes de communication entre un processeur et un/des périphérique(s)
- Généralement, un périphérique communique avec le processeur via un bus système
- Le périphérique de communication peut être :
  - Une interface de communication série
  - Une interface de communication parallèle
  - Une interface de communication RF (BLE, WIFI, 4G, etc.)



Les bus du STM32



La carte Nucléo STM32 possède :

- ❑ 3 UARTS
- ❑ 3 I2C
- ❑ 5 SPI

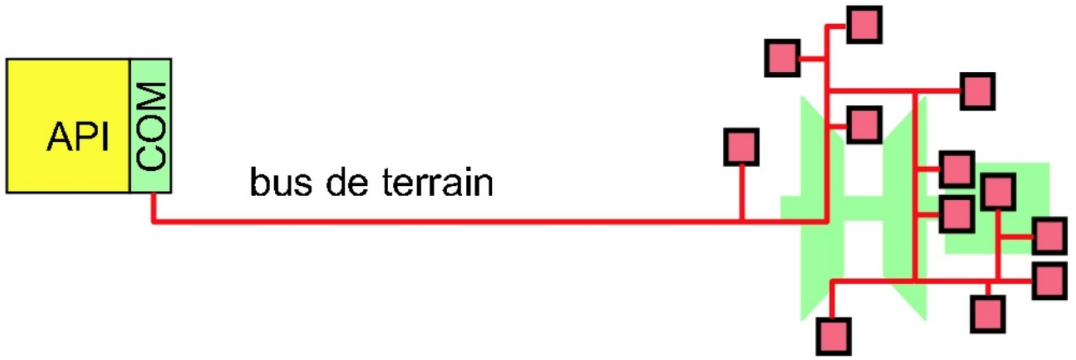
Mode de communication

- Un bus est un protocole, composé de règles qui gouvernent les modes de communication entres plusieurs dispositifs connectés.

- Un bus est composé de signaux de donnée et de contrôle

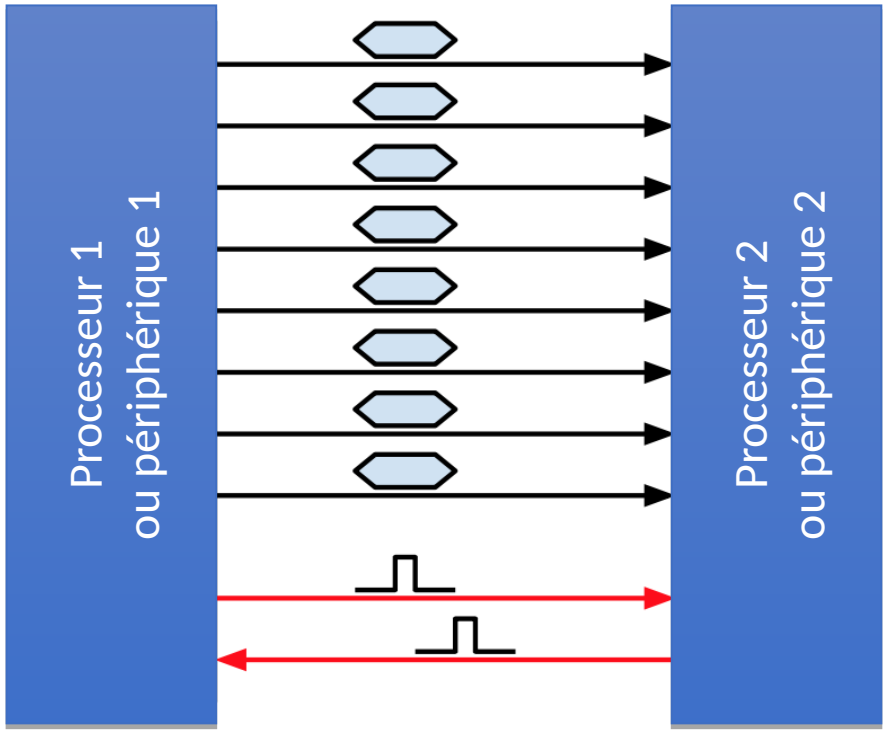
- Un bus peut être asynchrone ou synchrone
  - Dans un bus synchrone, l'horloge est distribuée avec les données
  - Dans un bus asynchrone, l'horloge est récupérée à travers les données envoyées

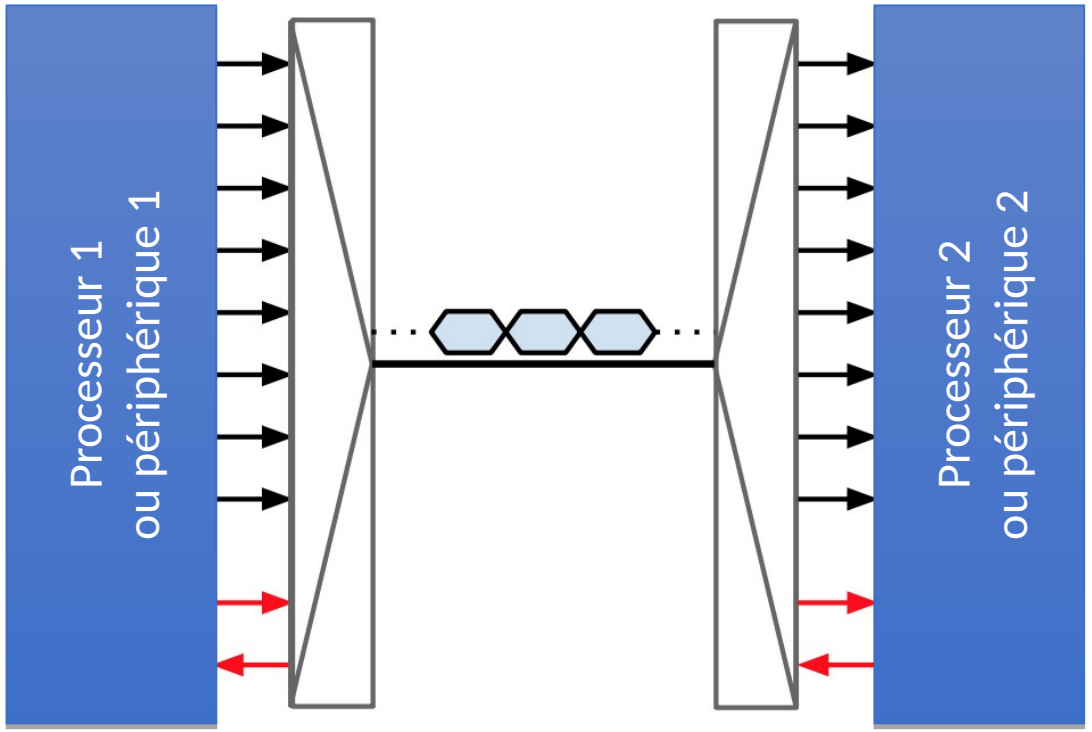
- Suivant le protocole, plusieurs appareils peuvent être connectés sur le même support physique
  - Bus de terrain



Mode de communication

- **Parallèle**
  - Donnée sur N fils en //
- **Série**
  - Donnée sérialisée/désérialisée sur 1 fil





Mode de communication

Parallèle

- Bus de N fils en parallèle
- Tous les signaux doivent arriver en même temps (synchrone)
- Limité par le nombre d'IO du processeur
- Simplicité de mise en œuvre
- Débit important
- Courte distance
- Tend à être abandonné

Série

- Bus de 2 à 3 fils en parallèle
- Signaux asynchrone
- Permet plus de bus en parallèle
- Débit important
- Longue distance
- Complexité dépend du protocole
- Beaucoup de nvx protocoles (USB3.0, USBC, Thunderbolt, etc.)

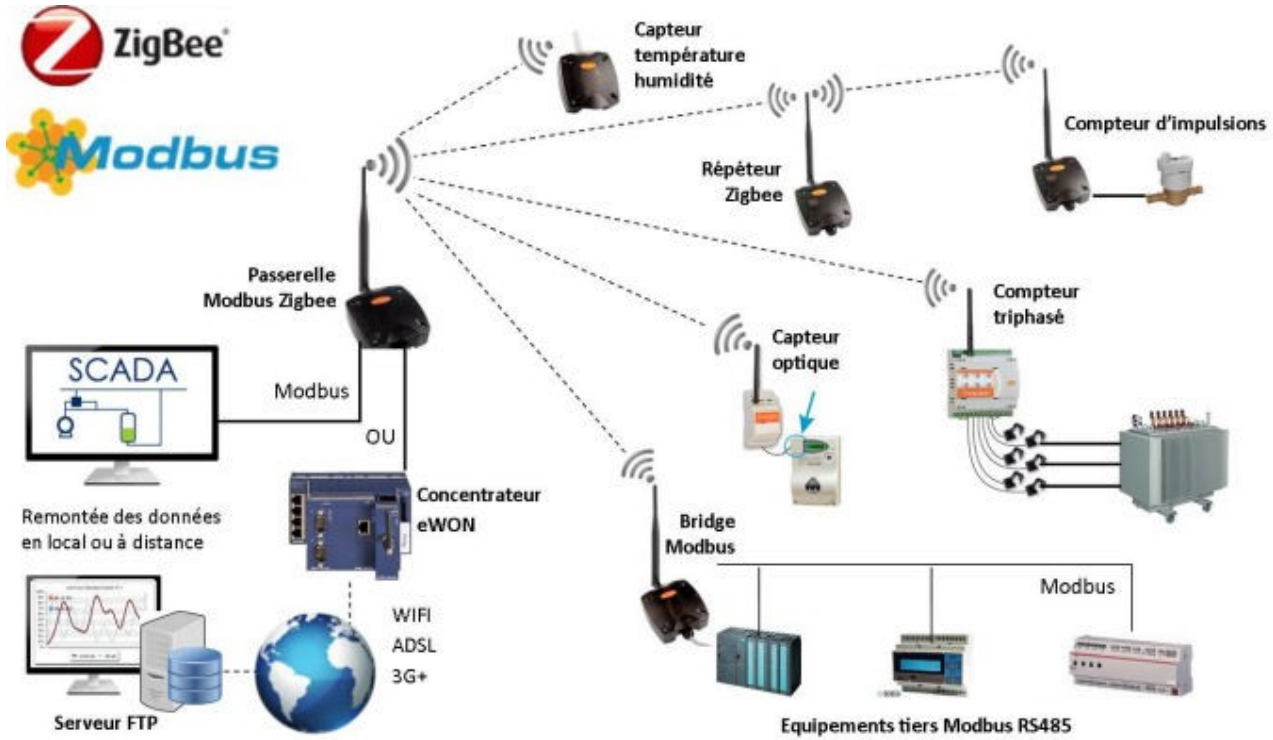
Horloge de transmission

- La transmission se fait :
  - Soit en **synchronisme** avec **une horloge de référence** commune au 2 systèmes et transmise sur une ligne supplémentaire :
    - Exemple : liaison SSP du PIC (Synchronous Serial Port).
  - Soit de façon indépendante sans horloge de référence :
    - La vitesse de transmission doit être identique sur une même ligne qui relie les circuits d'émission et de réception. Par contre elle n'est pas forcément la même sur les 2 lignes :
      - Exemple liaison USART du PIC (Asynchronous Synchronous Receiver Transmitter)

Bus de communication

- Pas de bus universel pour toutes les configurations de communication

- Comment choisir ?
  - Le choix dépend :
    - De la distance entres les équipements,
    - Le nombre d'équipements,
    - Du support physique de transmission,
    - Du débits,
    - De l'immunité au bruit,
    - De la consommation,
    - Du choix de modulation,
    - Du coût,
    - Etc….



IUT
CERGY-PONTOISE

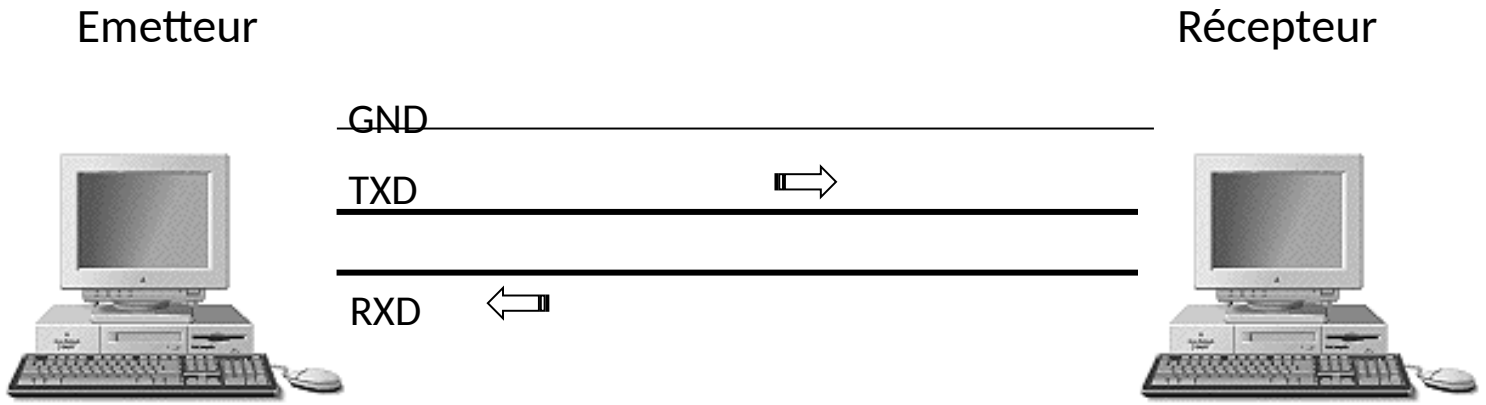
Partie 1

UART

Bus de communication série

Principe de fonctionnement

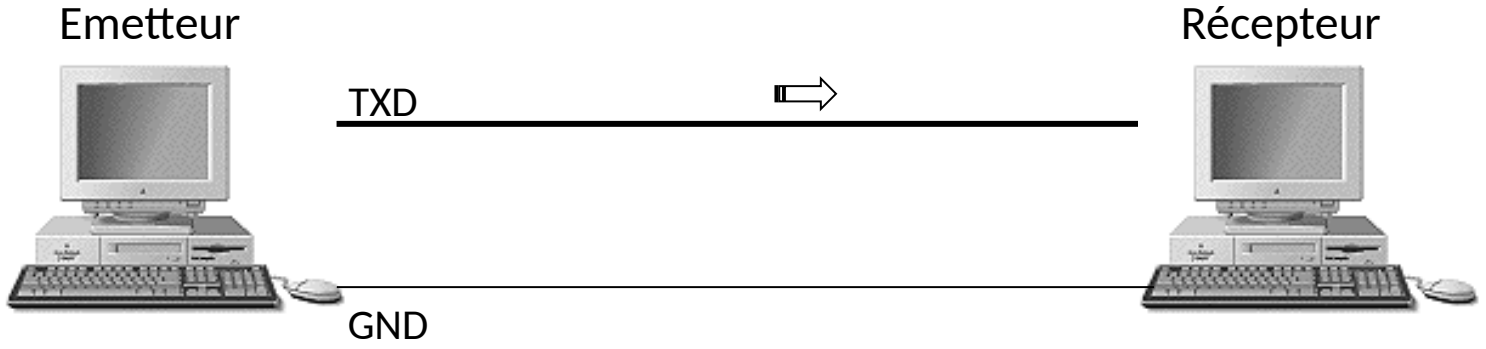
- A la différence des liaisons parallèles la transmission série consiste à transmettre des informations binaires bit par bit sur un fil électrique.



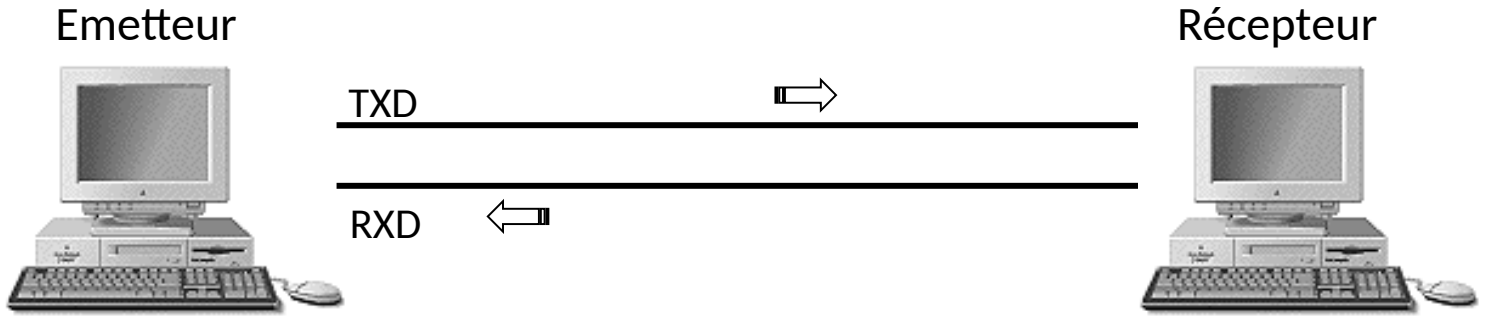
- La communication peut se faire en même temps dans les deux sens : Full-duplex

Principe de fonctionnement

- Communication Half Duplex



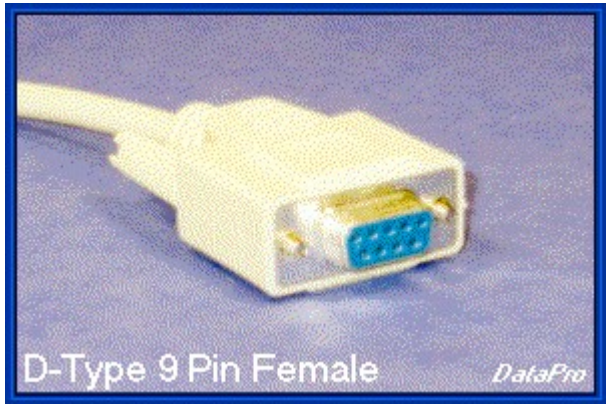
- Communication Full-duplex

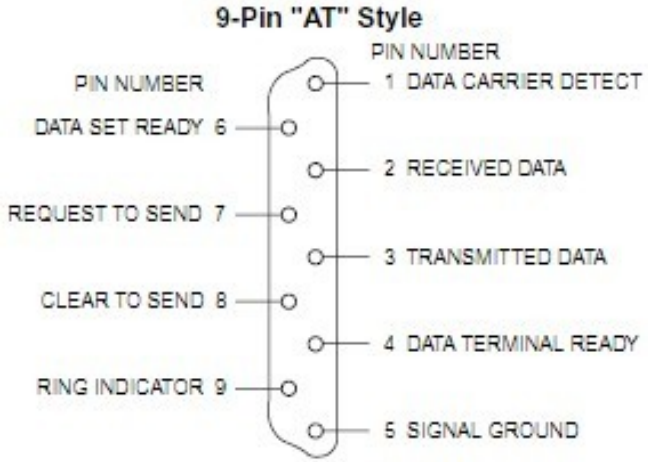


UART – RS232

UART – RS232 : Universal Asynchronous Receiver/ Transmitter

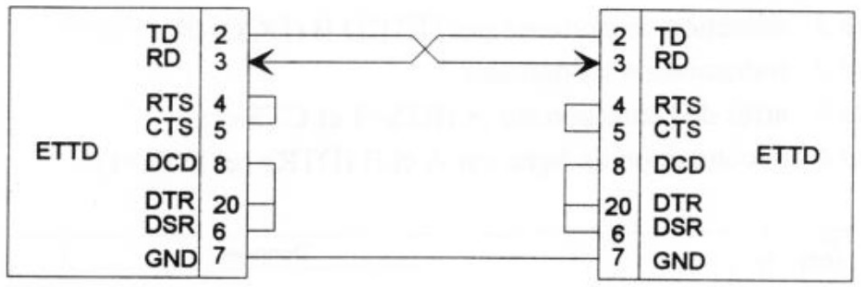
- Débit <1Mbit/s
- Protocole Full-Duplex
- Synchronisation via des signaux de handshake : CTS et RTS
- Liaison point à point
- Trame composée d'un bit de start, de 7 bits de données et 1 bit d stop
- La parité est optionnelle : 1 bit supplémentaire
- 3 fils : TX, RX et GND
- UART : niveaux entre 0 et 5V
- RS232 : niveau 1 entre -25 et -5v et niveau 0 entre 5 et 25v



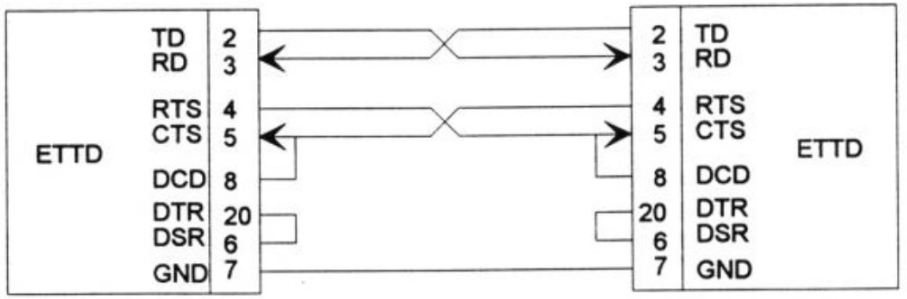


UART – RS232

- Exemple de configuration



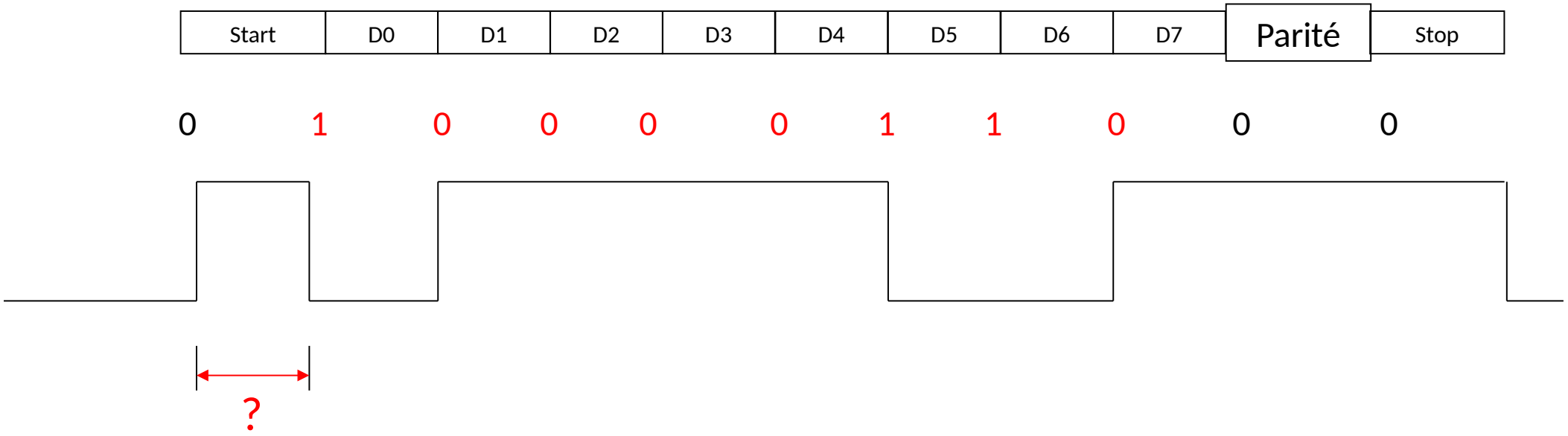
Cablâge Null-modem



Avec flux matériel (signaux de contrôles RTS,CTS)

UART – RS232

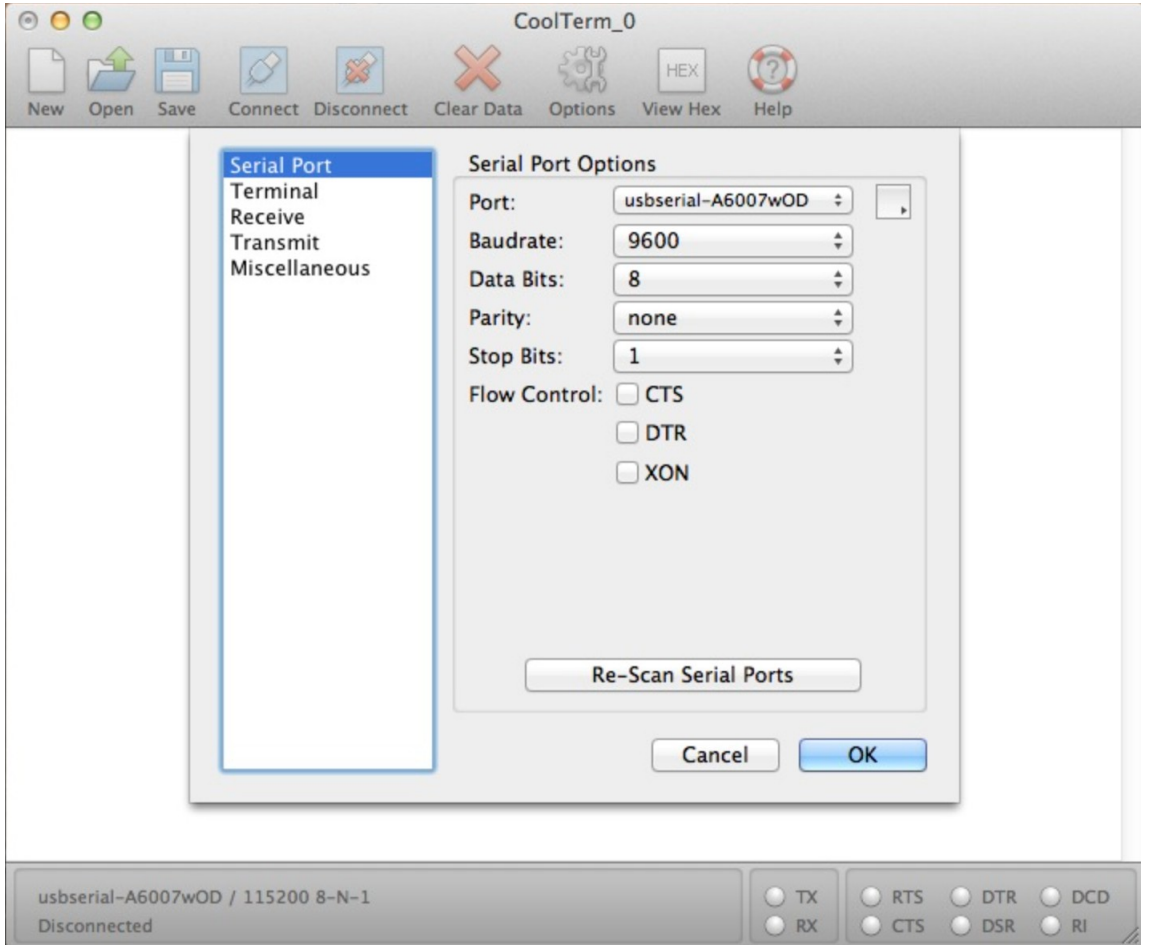
- Exemple d'une trame



Période d'un bit : débit binaire en baud

UART – RS232

- Vitesse de transmission
  - 1200, 2400, 4800, 9600, 14400, 19200, 28800, 33600, 56000, 115000 …
  - La vitesse doit être identique pour l'émetteur et le récepteur. Elle est prédéfinie par configuration



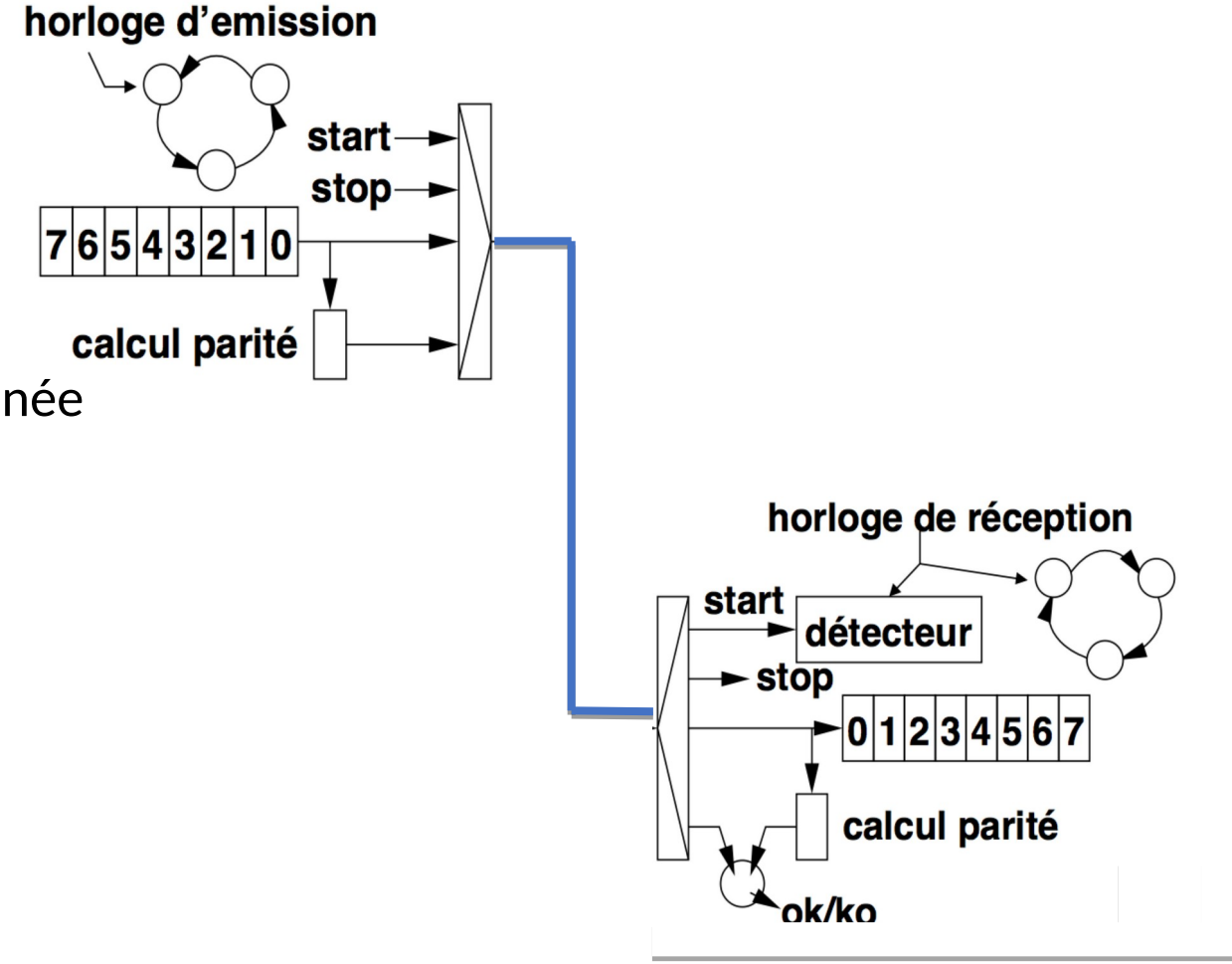
UART – RS232

Emetteur

- La donnée est remplie dans un registre à décalage à chargement parallèle
- Principe de la sérialisation
- Un bit supplémentaire est rajoutée pour signer la donnée (bit à 1 pour les données paires)

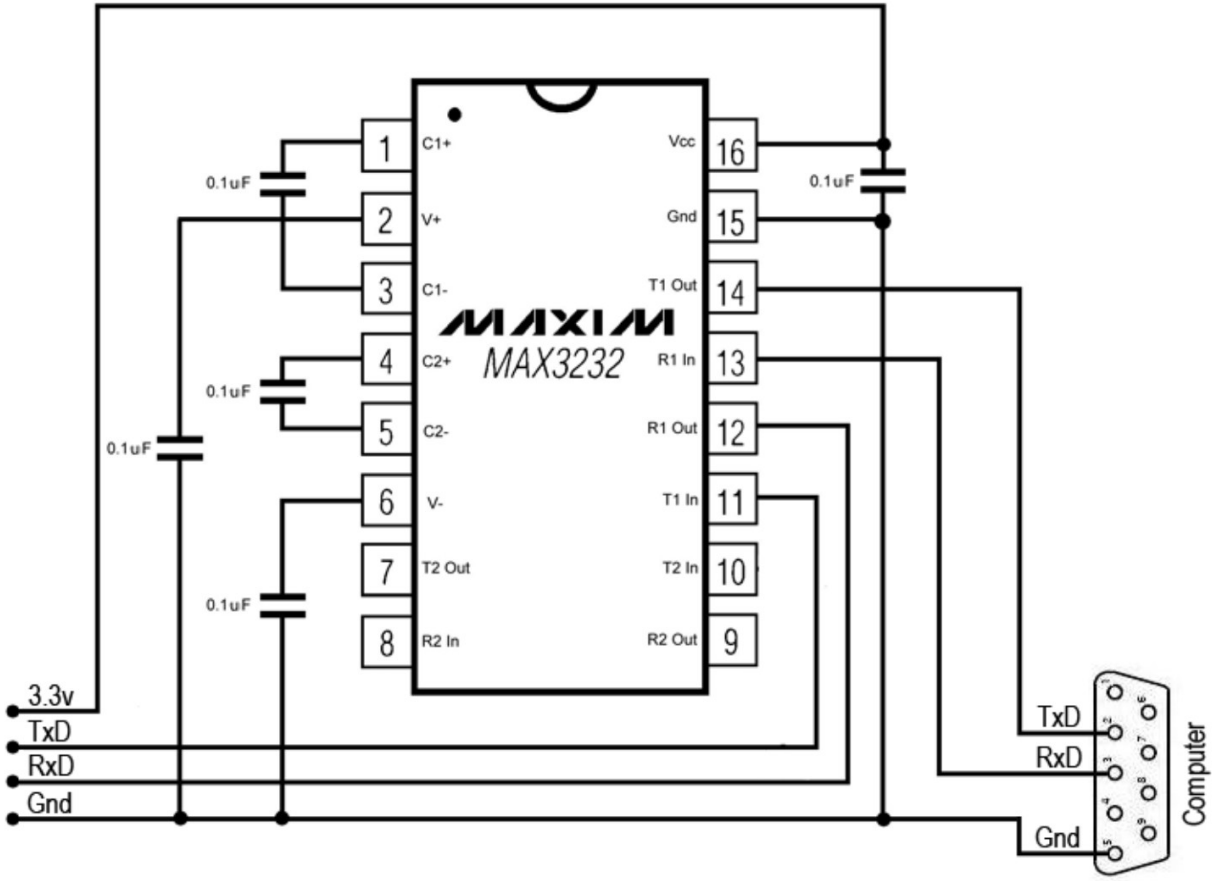
Récepteur

- La donnée lue sur la ligne après la détection du bit de start est remplie dans un registre à décalage, à chargement série
- Les données sont disponibles ensuite en parallèle
- Principe de la sérialisation



UART – RS232

- Adaptation des niveaux



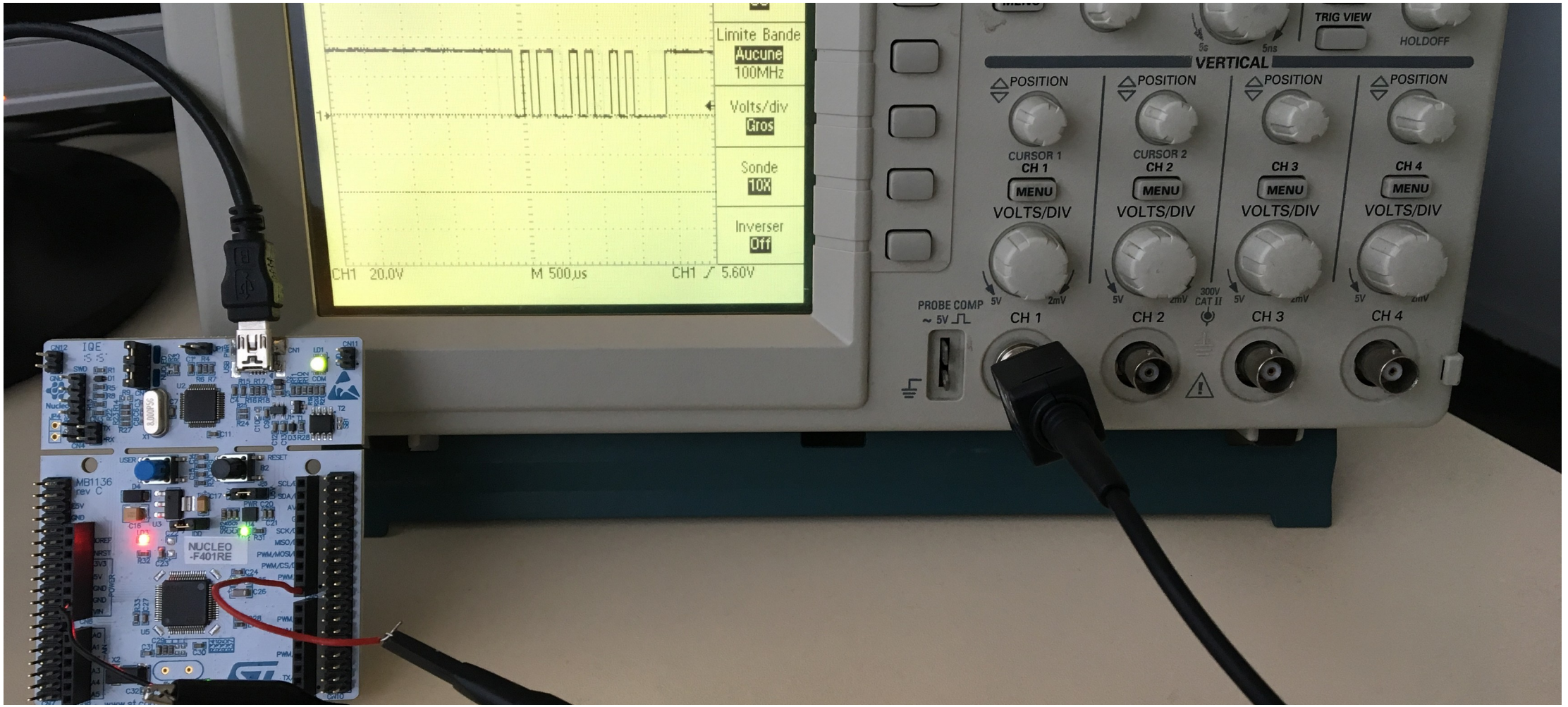
UART – RS232

Exemple sous MBED

```
2  #include "mbed.h"
3
4  Serial uart(D8,D2);
5  DigitalOut myled(LED1);
6
7  int main()
8  {
9      int i=48;
10     uart.baud(9600);
11     while(1)
12         {
13         uart.printf("%c\n",i);
14         if (i>100) i=48;
15         myled = 1;
16         wait(0.5);
17         myled = 0;
18         wait(0.5);
19         i++;
20         }
21 }
22
```

	Serial (PinName tx, PinName rx, const char *name=NULL, int baud=MBED_CONF_PLATFORM_DEFAULT_SERIAL_BAUD_RATE) Create a **Serial** port, connected to the specified transmit and receive pins.
	Serial (PinName tx, PinName rx, int baud) Create a **Serial** port, connected to the specified transmit and receive pins, with the specified baud.
void	**baud** (int baudrate) Set the baud rate of the serial port.
void	**format** (int bits=8, Parity parity=SerialBase::None, int stop_bits=1) Set the transmission format used by the serial port.
int	**readable** () Determine if there is a character available to read.
int	**writeable** () Determine if there is space available to write a character.
void	**attach** (Callback< void()> func, IrqType type=RxIrq) Attach a function to call whenever a serial interrupt is generated.
template<typename T >	**MBED_DEPRECATED_SINCE** ("mbed-os-5.1","The attach function does not support cv-qualifiers. Replaced by ""attach(callback(obj, method), type).") void attach(T *obj Attach a member function to call whenever a serial interrupt is generated.
void	**set_flow_control** (Flow type, PinName flow1=NC, PinName flow2=NC) Set the flow control type on the serial port.
int	**write** (const uint8_t *buffer, int length, const event_callback_t &callback, int event=SERIAL_EVENT_TX_COMPLETE) Begin asynchronous write using 8bit buffer.
int	**write** (const uint16_t *buffer, int length, const event_callback_t &callback, int event=SERIAL_EVENT_TX_COMPLETE) Begin asynchronous write using 16bit buffer.
void	**abort_write** () Abort the on-going write transfer.
int	**read** (uint8_t *buffer, int length, const event_callback_t &callback, int event=SERIAL_EVENT_RX_COMPLETE, unsigned char char_match=SERIAL_RESERVED_CHAR_MATCH) Begin asynchronous reading using 8bit buffer.
int	**read** (uint16_t *buffer, int length, const event_callback_t &callback, int event=SERIAL_EVENT_RX_COMPLETE, unsigned char char_match=SERIAL_RESERVED_CHAR_MATCH) Begin asynchronous reading using 16bit buffer.

UART – RS232


Limite Bande
Aucune
100MHz
Volts/div
Gros
Sonde
10X
Inverser
Off
CH1 20.0V
M 500µs
CH1 5.60V
TRIG VIEW
HOLDOFF
VERTICAL
POSITION
CURSOR 1
CH 1
MENU
VOLTS/DIV
CURSOR 2
CH 2
CH 3
CH 4
PROBE COMP
~ 5V
NUCLEO
-F401RE

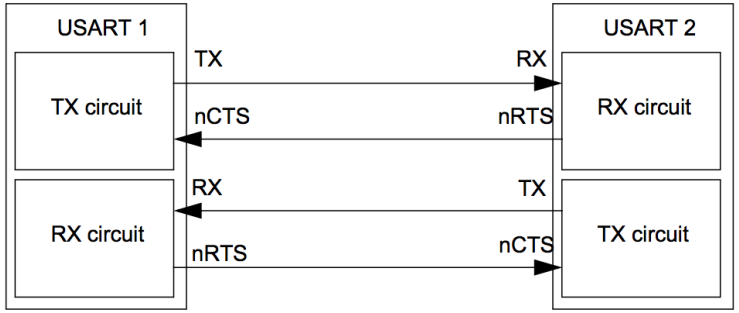
UART – STM32

- STM32
  - Possède 3 USART
    - Universal Synchronous Asynchronous Receiver Transmitter
  - Chaque USART
    - 9 modes de communication
    - Les plus usuels sont les modes asynchrone et HW control flow

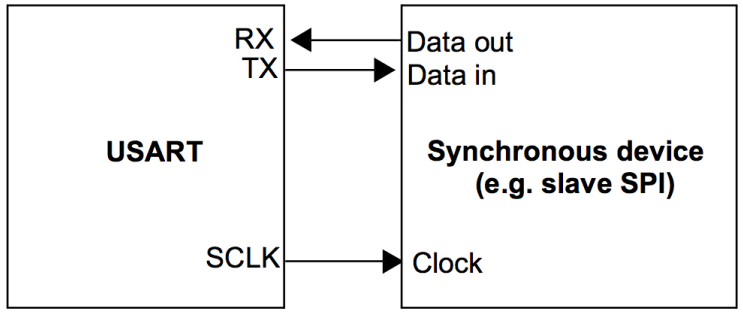
USART modes	USART1	USART2	USART6
Asynchronous mode	X	X	X
Hardware flow control	X	X	X
Multibuffer communication (DMA)	X	X	X
Multiprocessor communication	X	X	X
Synchronous	X	X	X
Smartcard	X	X	X
Half-duplex (single-wire mode)	X	X	X
IrDA	X	X	X
LIN	X	X	X

UART – STM32

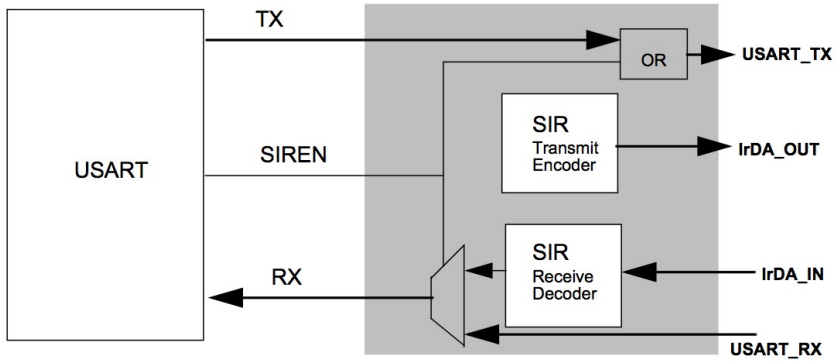
HW control flow

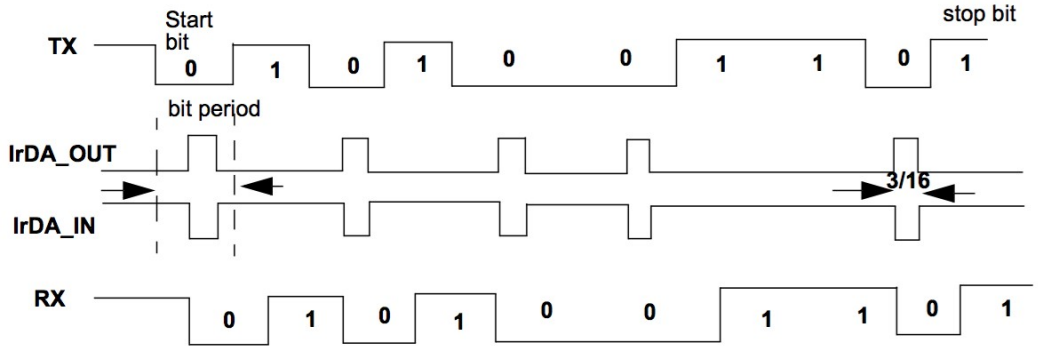


Mode synchrone



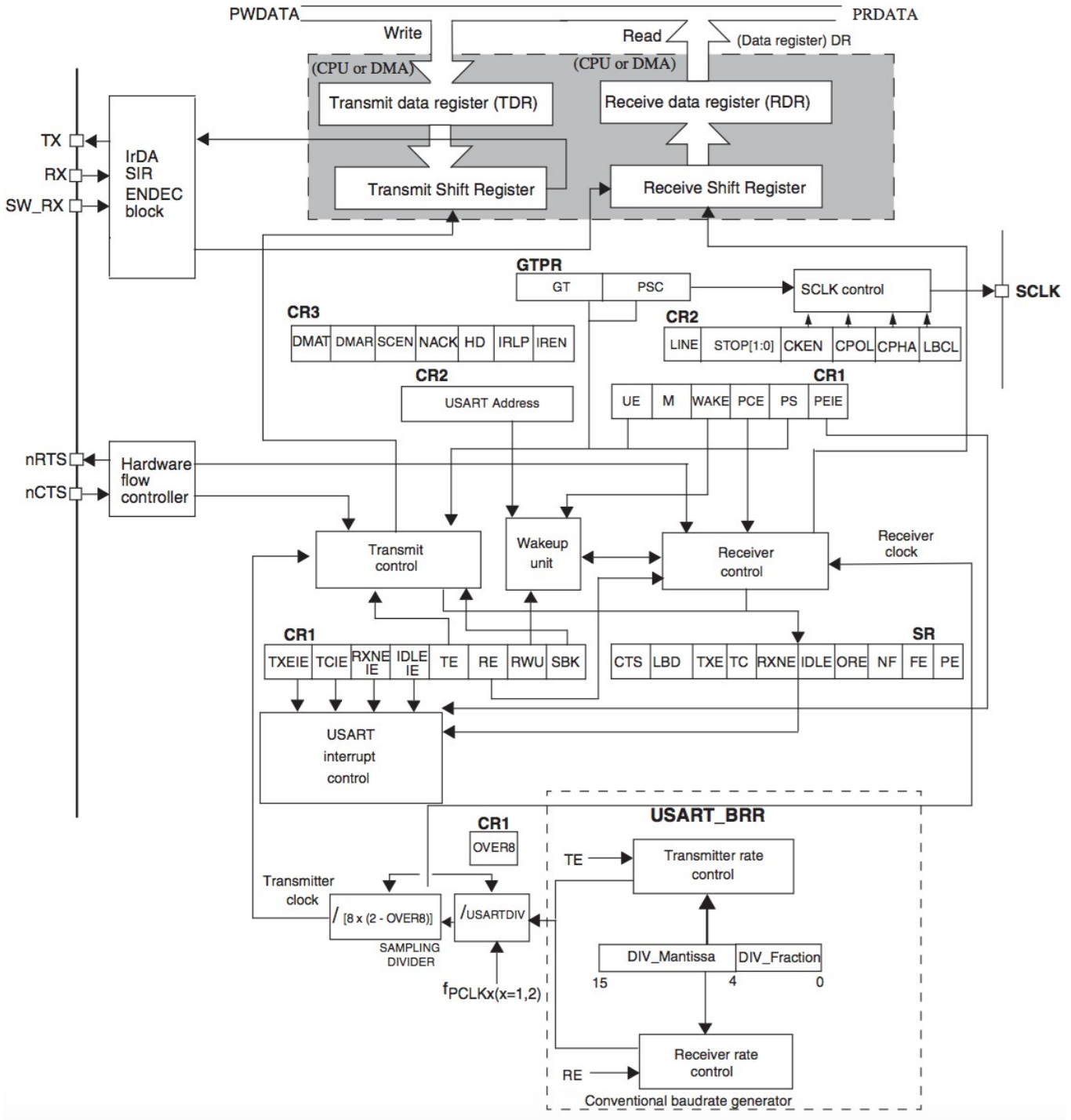
IRDA





UART – STM32

- Registres du STM32
  - `USART_BRR` : baud rate
  - `SR` : Status Register
  - `CR1`, `CR2`, `CR3` et `CR4` : registres de contrôle

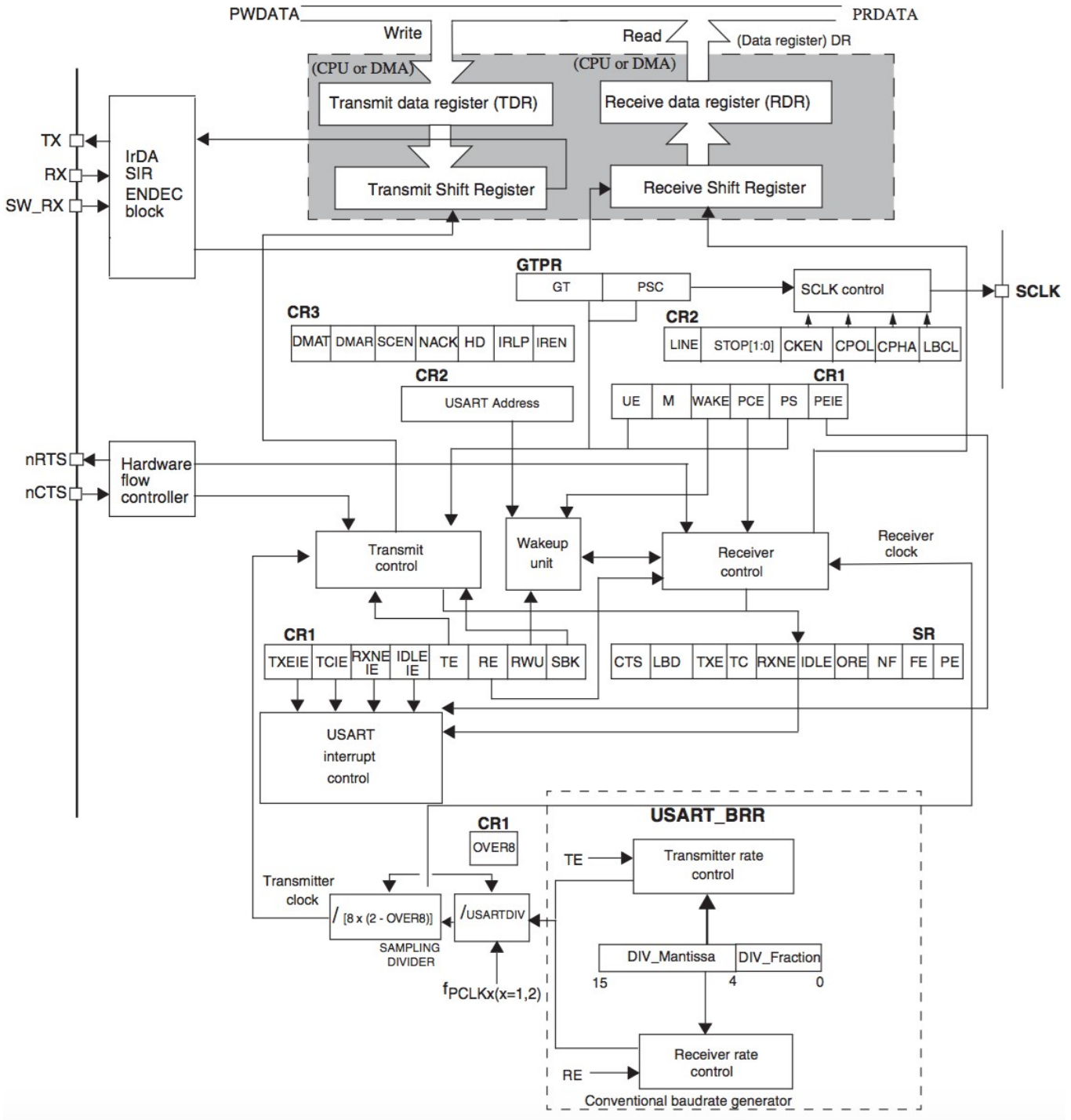


UART – STM32

- Registres du STM32
  - `USART_BRR` : baud rate
  - `SR` : Status Register
  - `CR1`, `CR2`, `CR3` et `CR4` : registres de contrôle

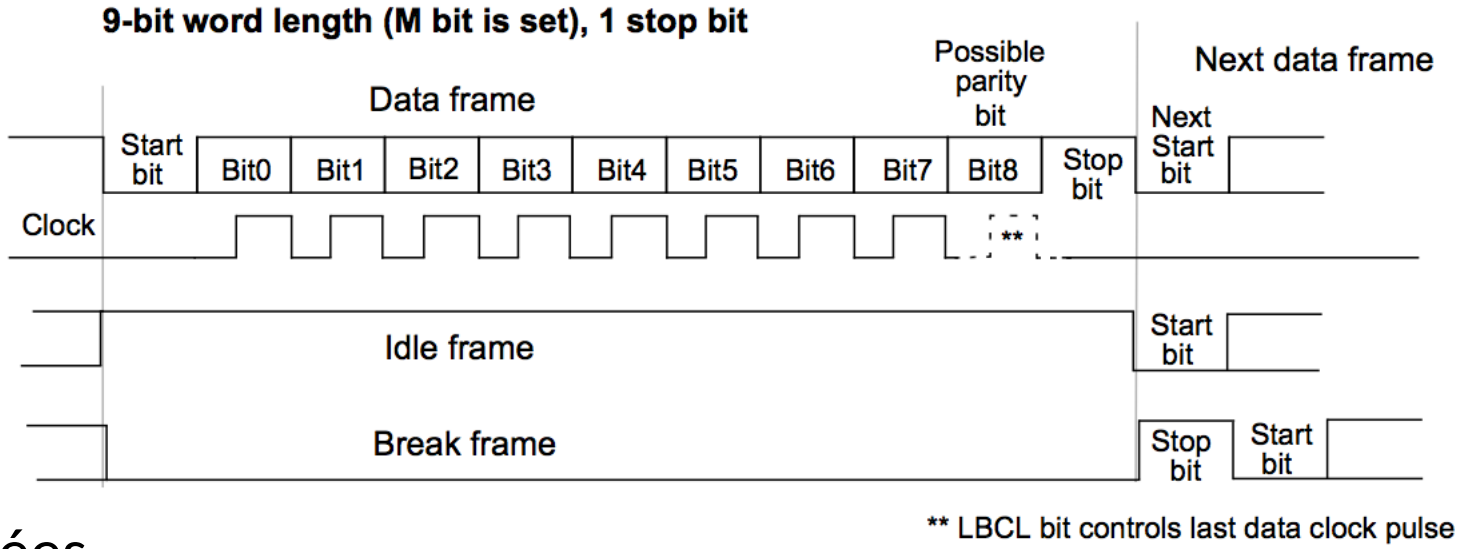
  - `USART_CR1`
    - Taille des données
    - Contrôle de la parité

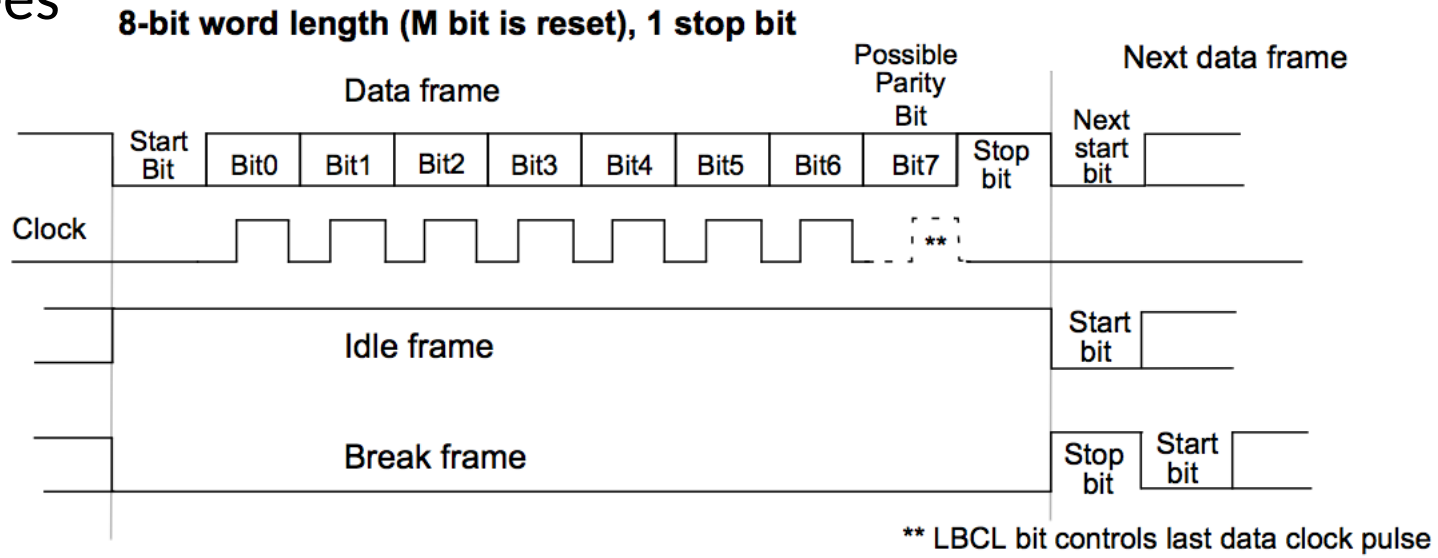
  - `USART_CR2`
    - nombre de bit de contrôle



UART – STM32

- USART_CR1
  - Configuration de la taille des données
  - 7 ou 8 bits





UART – STM32 – interruptions

- Source d'interruption
  - 9 sources

Interrupt event	Event flag	Enable control bit
Transmit Data Register Empty	TXE	TXEIE
CTS flag	CTS	CTSIE
Transmission Complete	TC	TCIE
Received Data Ready to be Read	RXNE	RXNEIE
Overrun Error Detected	ORE	
Idle Line Detected	IDLE	IDLEIE
Parity Error	PE	PEIE
Break Flag	LBD	LBDIE
Noise Flag, Overrun error and Framing Error in multibuffer communication	NF or ORE or FE	EIE

UART – STM32

Vitesse de transmission

$$\text{Tx/Rx baud} = \frac{f_{CK}}{8 \times (2 - \text{OVER8}) \times \text{USARTDIV}}$$

19.6.3 Baud rate register (USART_BRR)

Note: The baud counters stop counting if the TE or RE bits are disabled respectively.

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value

Bits 15:4 **DIV_Mantissa[11:0]**: mantissa of USARTDIV
These 12 bits define the mantissa of the USART Divider (USARTDIV)

Bits 3:0 **DIV_Fraction[3:0]**: fraction of USARTDIV
These 4 bits define the fraction of the USART Divider (USARTDIV). When OVER8=1, the DIV_Fraction3 bit is not considered and must be kept cleared.

UART – STM32

- Procédure à respecter pour réaliser une transmission
  1. Activer l'USART en écrivant le bit UE à 1 dans le registre USART_CR1
  2. Définir la taille des données dans le registre USART_CR1
  3. Définir le nombre de bit de stop dans le registre USART_CR2
  4. Définir la vitesse de transmission dans le registre USART_BRR
  5. Ecrire les données à transmettre dans le registre USART_DR

UART – STM32

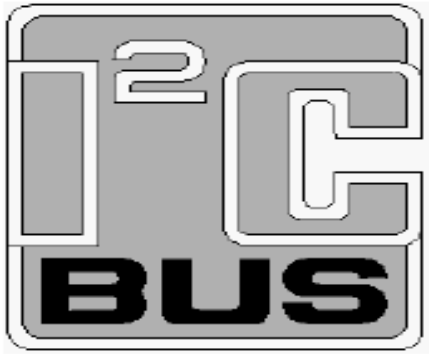
- Procédure à respecter pour réaliser une réception
  1. Activer l'USART en écrivant le bit UE à 1 dans le registre USART_CR1
  2. Définir la taille des données dans le registre USART_CR1
  3. Définir le nombre de bit de stop dans le registre USART_CR2
  4. Définir la vitesse de transmission dans le registre USART_BRR
  5. Mettre le RE à 1 dans le registre USART_CR1 pour autoriser l'attente d'un bit de start

- Quand un caractère arrive
  1. Le bit RXNE est à 1 pour indiquer d'une donnée est réceptionnée.
  2. La donnée est contenue dans le registre RDR
  3. Une interruption est générée (RXNEIE) si elle a été autorisée au préalable

IUT
CERGY-PONTOISE

Séance 2

IUT
CERGY-PONTOISE



Partie 2

I2C

Bus de terrain

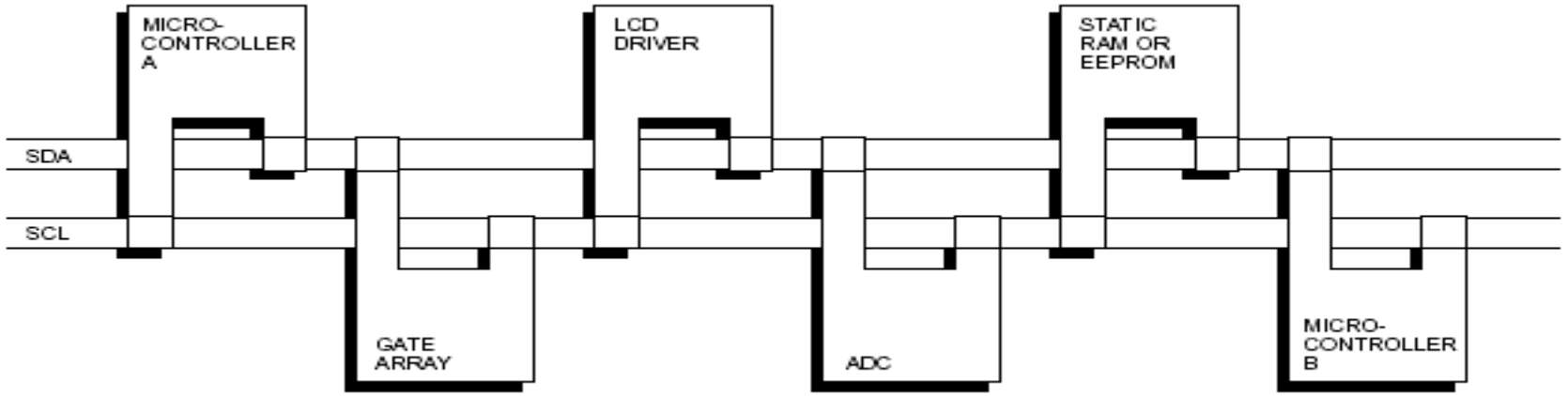
Série vs Parallèle

- A la différence des liaisons parallèles la transmission série consiste à transmettre des informations binaires bit par bit sur un fil électrique.

- La transmission se fait :
  - Soit en **synchronisme** avec **une horloge de référence** commune au 2 systèmes et transmise sur une ligne supplémentaire :
    - Exemple : liaison SSP du PIC (Synchronous Serial Port).
  - Soit de façon indépendante sans horloge de référence :
    - La vitesse de transmission doit être identique sur une même ligne qui relie les circuits d'émission et de réception. Par contre elle n'est pas forcément la même sur les 2 lignes :
      - Exemple liaison USART du PIC (Asynchronous Synchronous Receiver Transmitter)

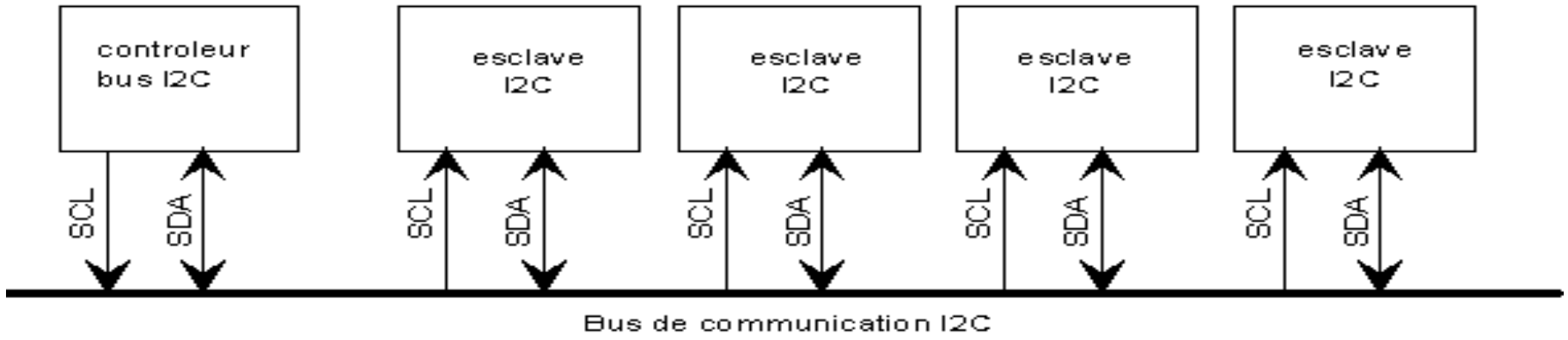
Le bus I2C

- Le bus I2C ou Inter-IC-Communication est un bus de communication série synchrone.
- Il a été conçu pour réaliser des liaisons séries sur 2 fils entre plusieurs périphériques connectés sur le bus.
- Philips est à l'origine de ce bus en 1982.
- Il équipe des appareils électroniques destinés au grand public (appareils TV et radio, systèmes audio et radio, postes téléphoniques, systèmes électrique automobile, appareils électroménager…)



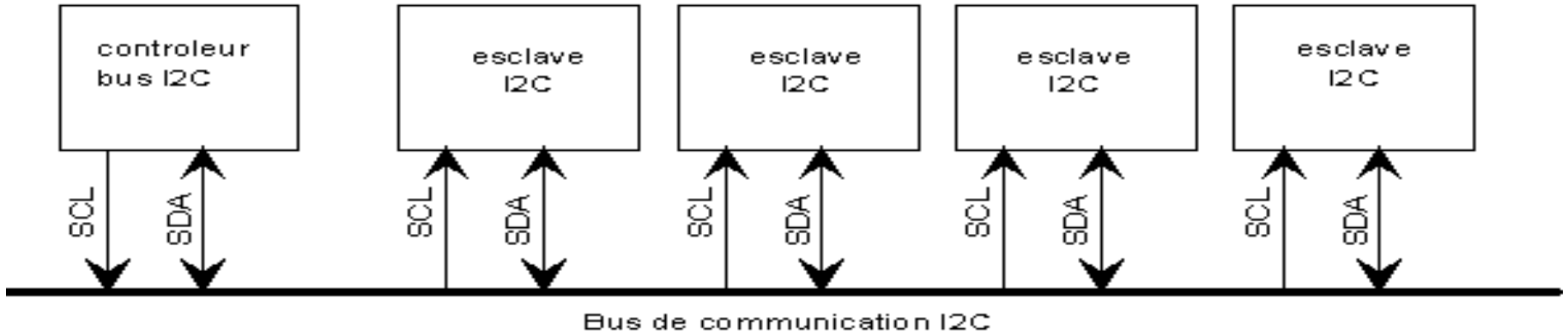
Le bus I2C

- Les informations sont échangées au moyen de deux lignes :
  - SDA (Serial Data) bidirectionnel
  - SCL (Serial Clock) unidirectionnel
- Chaque circuit intégré possède une adresse matérielle unique qui le distingue des autres sur 7 bits : 128 périphériques
- Chaque composant peut émettre ou recevoir des informations suivant sa fonction avec un débit standard de 100Kb/s.
- Le bus I2C est un bus de communication série dit "2 fils" (2 wire BUS) mais en fait il est nécessaire de rajouter la masse pour référencer les deux signaux d'information (SDA) et d'horloge (SCL).
- C'est un bus dit Maitre/esclave
  - Tout échange sur le bus ne peut se faire qu'à l'initiative du maitre.
  - Les esclaves ne font qu'y répondre par un mécanisme d'appel/réponse. Le maitre est propriétaire de SCL



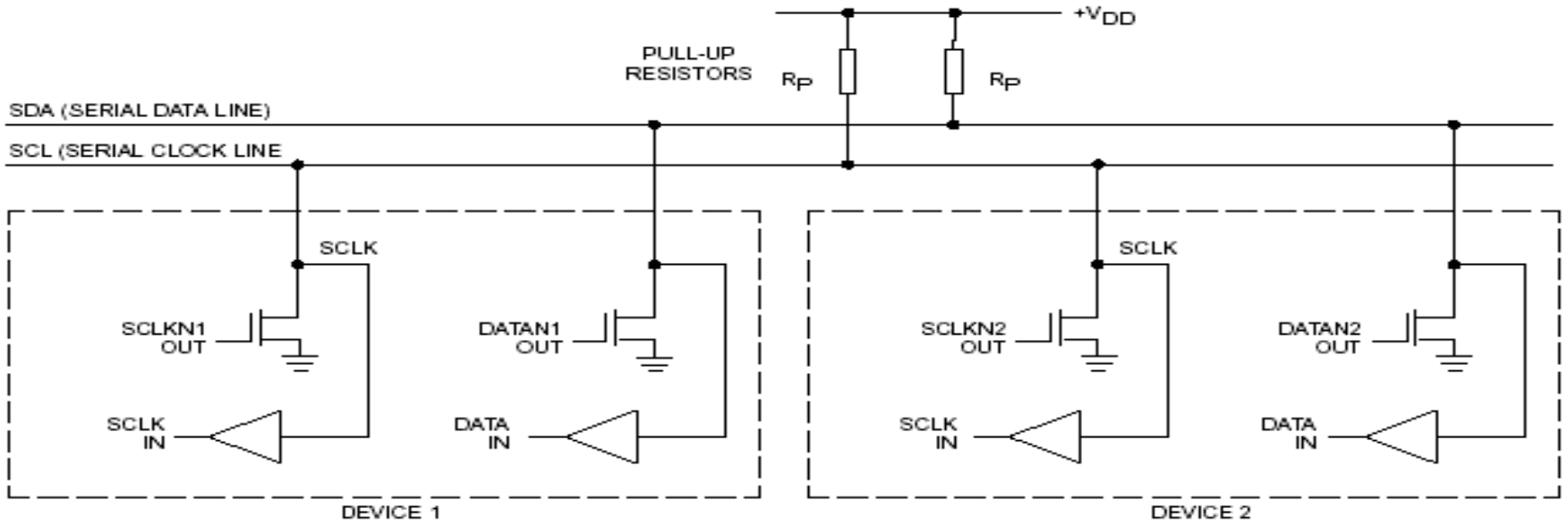
Le bus I2C

- Il existe cinq vitesses de transmission :
  - Standard mode : 100 kbit/s,
  - Fast mode : 400 kbit/s,
  - Fast plus mode : 1 Mbit/s,
  - High-speed mode (Hs-mode) : jusqu'à 3,4 Mbit/s,
  - Ultra-fast mode (UFm) : jusqu'à 5 Mbit/s en mode unidirectionnel



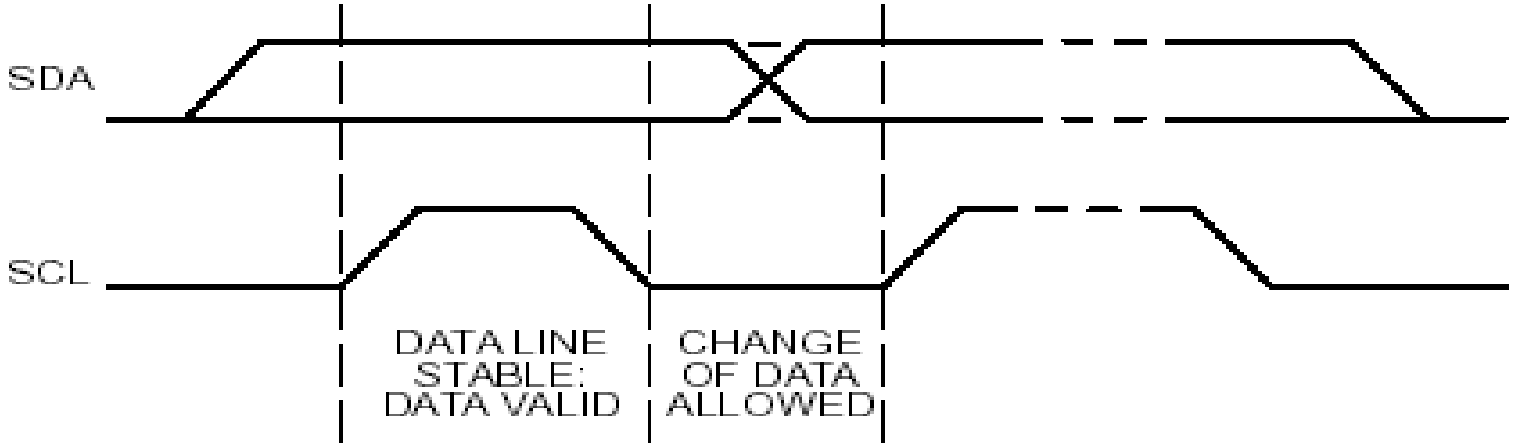
Le bus I2C : accès et niveaux

- Pour le bus I2C, le niveau électrique dominant est l'état bas ₂ masse commune
- Les deux lignes SDA et SCL sont donc maintenues au niveau haut tant que le bus est libre.
- Les étages de sortie des circuits connectés au bus doivent avoir un drain ouvert ou un collecteur ouvert pour remplir la fonction « ET câblé ».



Le bus I2C : protocole

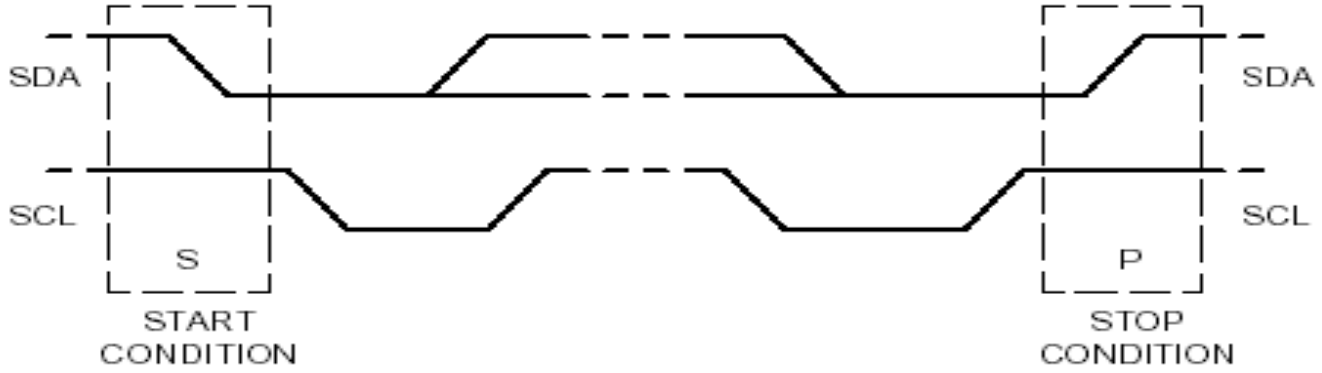
- Le protocole de communication I2C est basé sur les changements d'état de SDA cadencé par SCL
- Le bus I2C appartient à la catégorie des bus série. Les données sont ici envoyées bit par bit par groupe d'octets sur la ligne SDA. On appelle ces groupes des trames.
- La ligne SCL fonctionne comme une horloge sérielle d'un registre à décalage. Lorsque que la ligne SCL est à l'état haut, les données de la ligne SDA doivent être stables.
- Lorsque la ligne SCL est à l'état bas, le circuit qui émet les données, peut modifier l'état de la ligne SDA.
- la modification de SDA par le contrôleur ou l'esclave n'est possible que si SCL=0



Le bus I2C : protocole

- Le début et la fin d'un échange doivent respecter les conditions de start et les conditions de stop
- Certaines combinaisons particulières de niveaux et de fronts des deux lignes déterminent la condition de départ (START) ou d'arrêt (STOP) de la transmission de la trame constituant le message.
  - ► Condition de départ : un front descendant sur SDA alors que SCL est à l'état haut.
  - ► Condition d'arrêt : un front montant sur SDA alors que SCL est à l'état haut.
- Le front montant de la ligne SCL sert généralement pour maintenir la donnée présente sur la ligne SDA coté récepteur.

Remarque : les conditions de départ et d'arrêt sont toujours générées par le maître du bus.

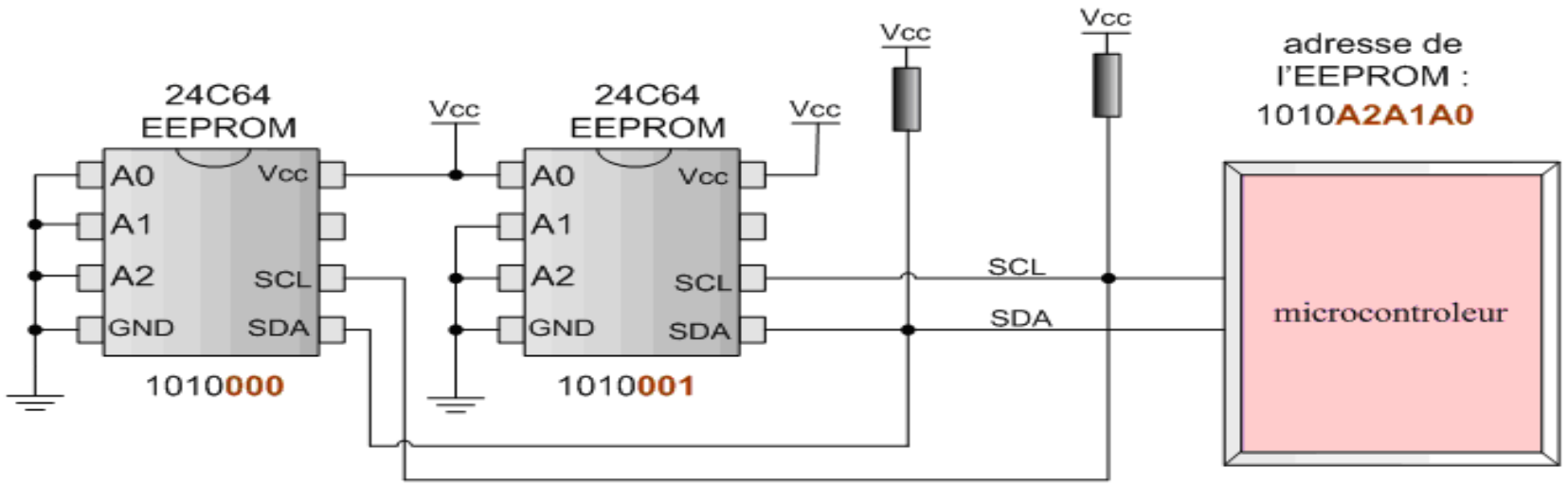


Le bus I2C : protocole

- La trame est composée
  - La condition de start est suivie par l'adresse du composant appelé par le maître.
  - Chaque composant possède une adresse unique codée sur sept bits ₂ₑ 128 périphériques
  - Le huitième bit (R/W) émis indique la direction du transfert
  - R/W+0 = écriture ₂ₑ maitre écrit et l'esclave lit : une donnée est envoyée à un esclave
  - R/W=1 = lecture ₂ₑ maitre lit et esclave écrit : l'esclave revoie une donnée au maitre.
- Seuls les esclaves qui auront reconnu leur adresse sur le bus participeront à la suite de l'échange. La transmission se fait MSB first.

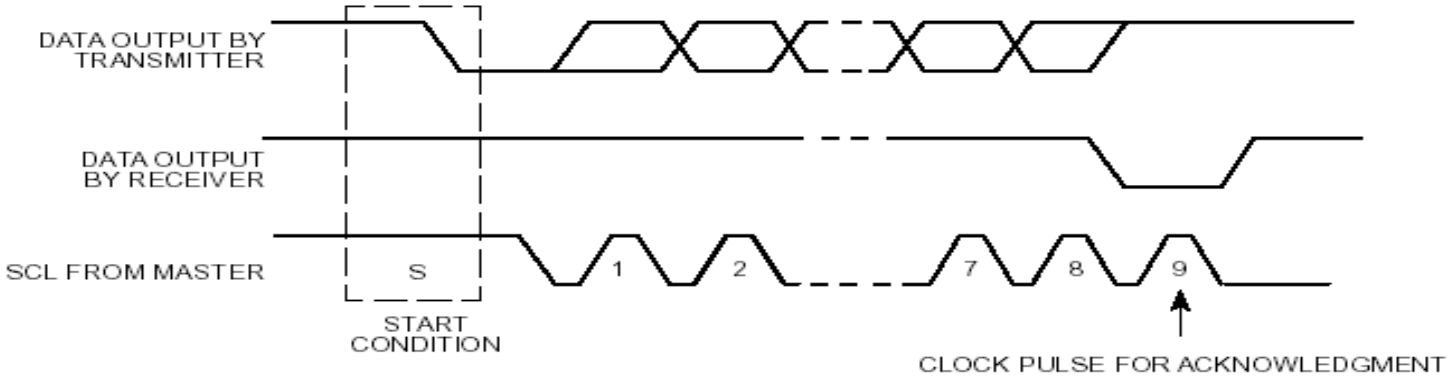
Le bus I2C : protocole

- Exemple d'échange entre un uC et deux mémoires I2C :
  - En général chaque boîtier possède 3 broches A2, A1 et A0 de sélection de l'adresse basse.
  - Ces broches de sélection permettent de placer jusqu'à 8 boîtiers de fonction identique sur le bus.
    - Dans l'exemple ci-dessous, l'EEPROM I2C 64kbits 24C64 possède l'adresse fixe 1010XXX.
  - L'adresse complète est choisie en fonction des niveaux logiques placés sur les broches A2,A1,A0.
    - En plaçant la broche A0 à 5V et les broches A1 et A2 à 0V, le boîtier répondra à l'adresse I2C 1010001.

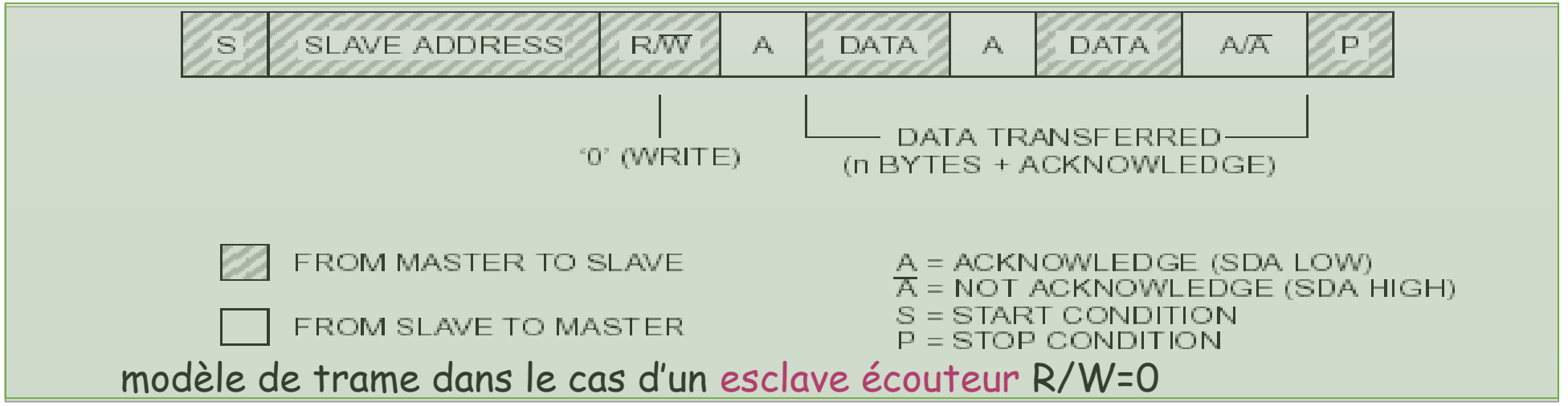


Le bus I2C : protocole

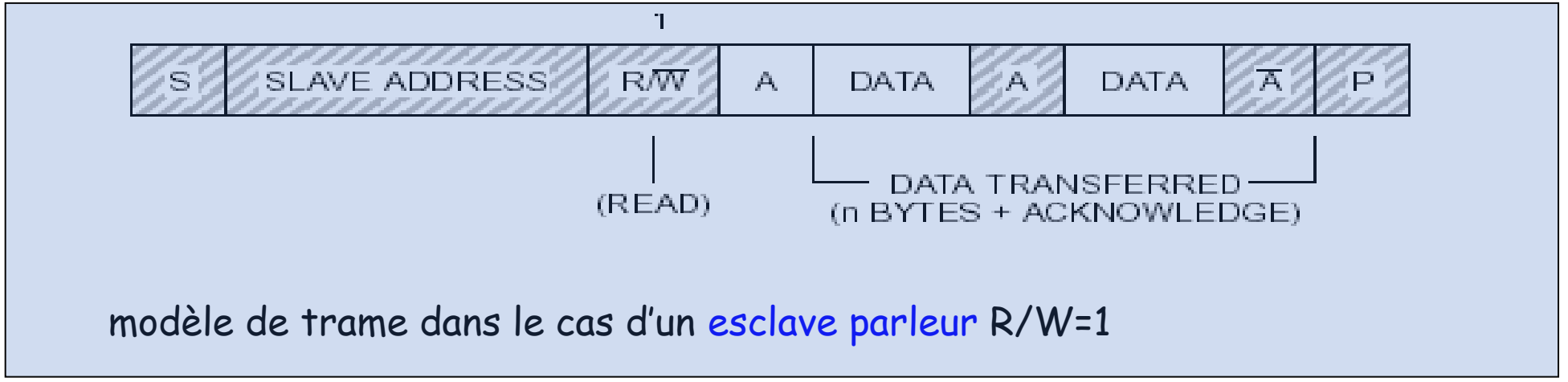
- Comment sait on que le périphérique à reçu une information ? Mécanisme d'acquittement
  - Chaque octet de donnée transmis (contenant @I2C et R/W=) est suivi par l'émission d'un bit d'acquittement (ou ACK).
  - Le récepteur de l'échange doit procéder à l'acquittement pour indiquer à l'émetteur (qu'il soit maître ou esclave) que l'information a bien été reçue.
  - C'est toujours le circuit (maître ou esclave) qui reçoit les données qui doit générer l'accuse réception de l'octet envoyé.
    - Dans le cas du premier octet d'@ I2C transmis c'est forcément l'esclave.
  - Lorsque le 8ème coup d'horloge est passé (SCL repasse alors à 0) l'émetteur de la donnée relâche la ligne SDA (release à 1), le récepteur doit alors faire passer SDA à 0(c'est l'ACK) .



Le bus I2C : format des trames



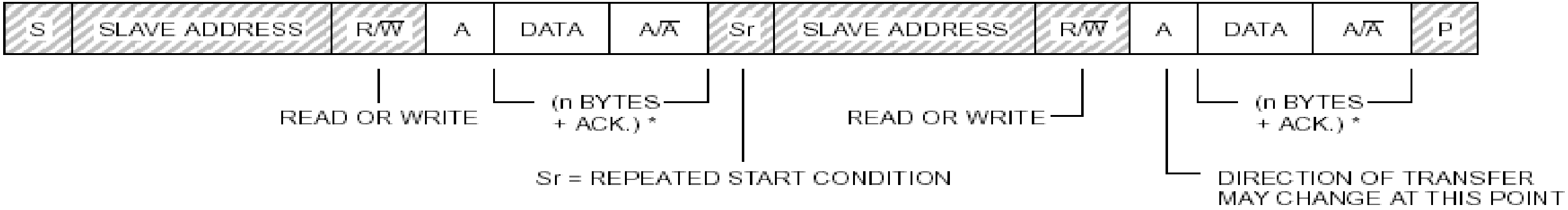
modèle de trame dans le cas d'un esclave écouteur R/W=0



modèle de trame dans le cas d'un esclave parleur R/W=1

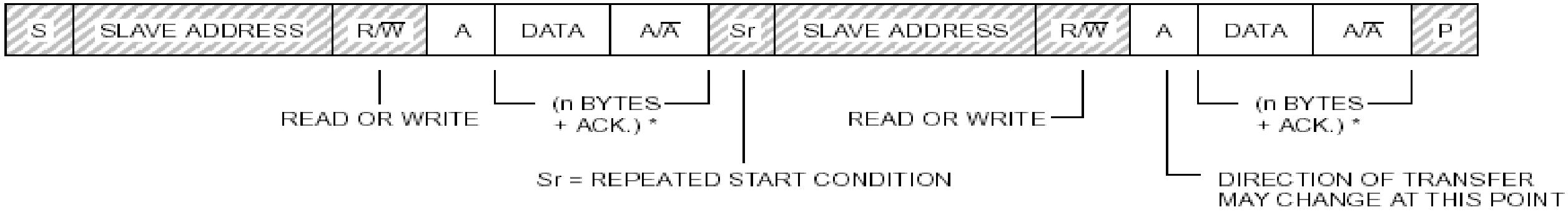
Le bus I2C : protocole

- Pour certains périphériques évolués (mémoires EEPROM) il peut être nécessaire d'abord de dire à l'esclave quelle case mémoire on veut lire (envoie de l'adresse de la case à lire). L'esclave est alors en mode lecture.
- Puis dans une deuxième partie de l'échange le maitre vient lire la valeur de la case mémoire pointée. Le maitre est alors écouteur.
- Le changement de mode s'opère en milieu de trame par une condition de Restart.

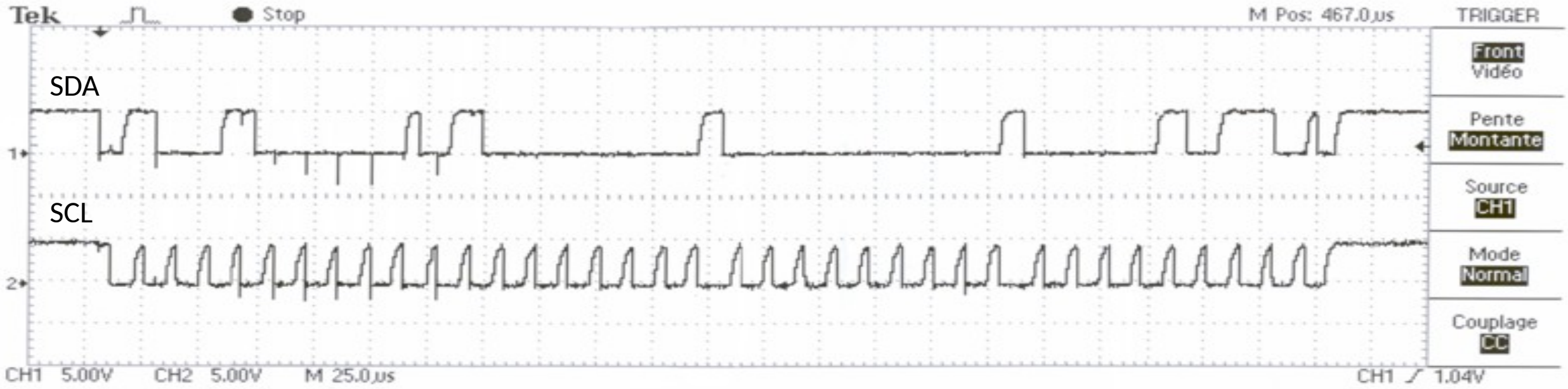


Le bus I2C : protocole

- Le nombre d'octet transmis dépend de la transaction.
  - Pas de limitation sur la durée
  - Echange continu d'information tant que l'échange procèdent à l'acquittement
    - Exemple du cas de la lecture d'une plage mémoire dans un composant FLASH I2C
- Seul un non acquittement volontaire de la part du maitre ou une condition de STOP peut mettre fin à l'échange.
- Exemple : chronogramme d'échange type entre un maitre et un esclave. Au cours de cette trame 2 octets de donnée sont envoyés à l'esclave avant que le maitre ne mette fin à la communication

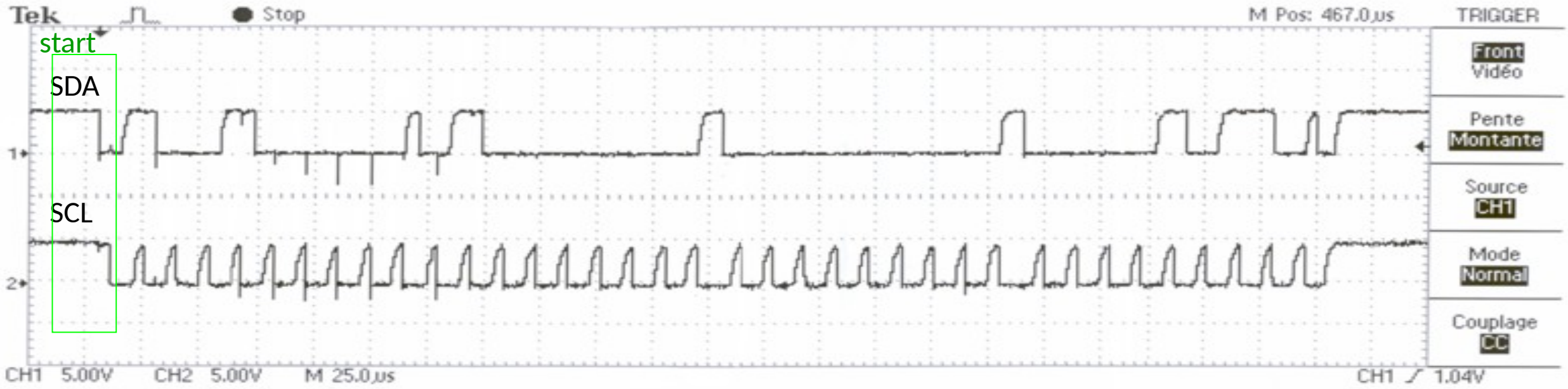


Le bus I2C : exemple



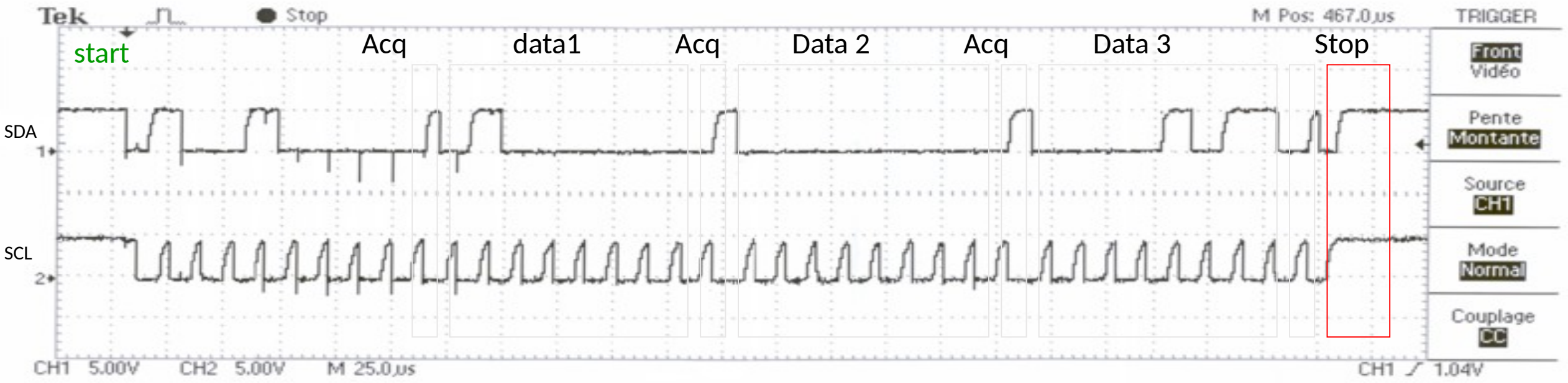
- Pour analyser cette trame , il faut compter les coups d'horloge générés par le maitre.
- SCL est en voie 2 et SDA en voie 1.
- Ils correspondent à l'envoie d'une adresse et de trois octets de données vers l'esclave.

Le bus I2C : exemple



- L'échange débute par la condition de START
- Puis le maitre envoie l'octet d'adresse sur 7 bits + 1 bit R/W
- Ainsi, on note que l'octet d'adresse (8 premier coups d'horloge de SCL) vaut 0x90 soit une @I2C=0x48 codée sur 7 bits et un R/W=0. Le maitre est donc parleur pour cet échange.

Le bus I2C : exemple



- l'acquittement Esclave > Maître s'est bien passé => bit d'acquittement => on continue l'envoie des données
- Le deuxième octet (M>E) vaut 0x40 : C est un octet de donnée. L'acquittement E>M s'est bien passé.
- Le troisième octet (M>E) de donnée vaut 0x00. L'acquittement E>M s'est bien passé.
- Le quatrième octet (M>E) de donnée vaut 0x0B. L'acquittement E>M s'est bien passé .
- Puis le maitre a mis fin à l'échange par une condition de stop.

I2C – exemple

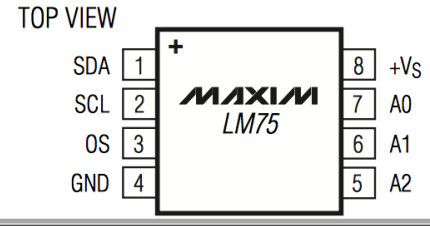
- Exemple sous MBED : lecture de la température sur un LM75

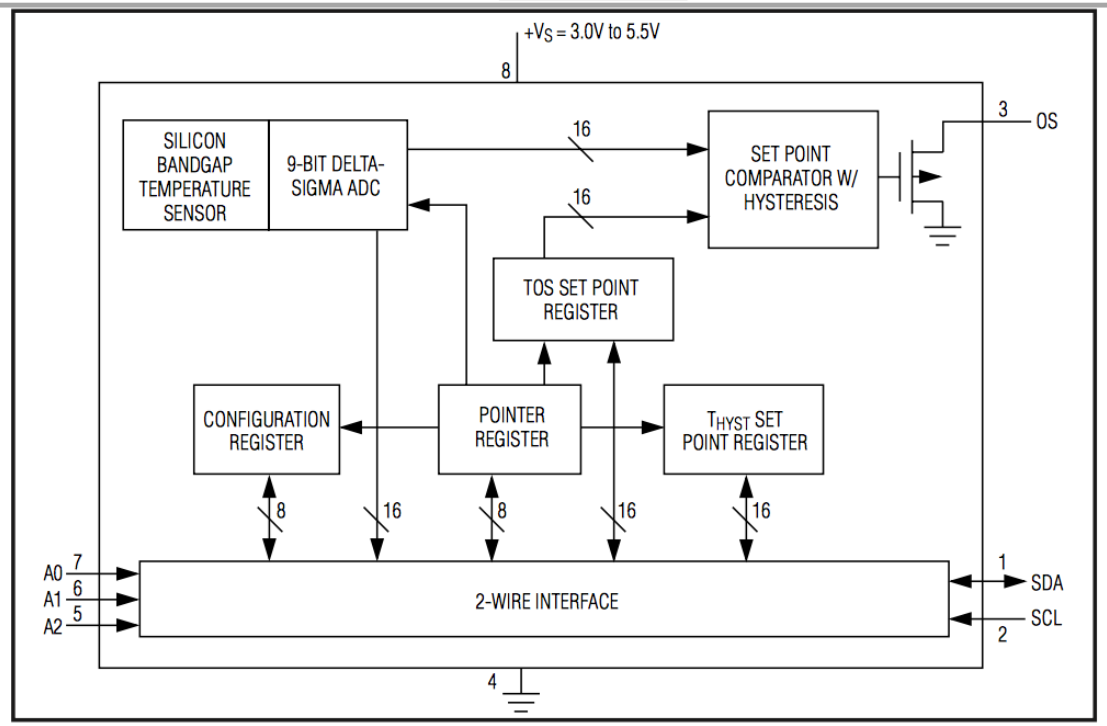
```
#include "mbed.h"

#define LM75_REG_TEMP (0x00) // Temperature Register
#define LM75_REG_CONF (0x01) // Configuration Register
#define LM75_ADDR     (0x90) // LM75 address

I2C i2c(I2C_SDA, I2C_SCL);
DigitalOut myled(LED1);
Serial pc(SERIAL_TX, SERIAL_RX);

volatile char TempCelsiusDisplay[] = "+abc.d C";

int main()
{

    char data_write[2];
    char data_read[2];

    data_write[0] = LM75_REG_CONF;
    data_write[1] = 0x02;
    int status = i2c.write(LM75_ADDR, data_write, 2, 0);
    if (status != 0) { // Error
        while (1) {
            myled = !myled;
            wait(0.2);
        }
    }

    while (1) {
        // Read temperature register
        data_write[0] = LM75_REG_TEMP;
        i2c.write(LM75_ADDR, data_write, 1, 1); // no stop
        i2c.read(LM75_ADDR, data_read, 2, 0);

        // Calculate temperature value in Celcius
        int tempval = (int)((int)data_read[0] << 8) | data_read[1];
        tempval >>= 7;
        if (tempval <= 256) {
            TempCelsiusDisplay[0] = '+';
        } else {
            TempCelsiusDisplay[0] = '-';
            tempval = 512 - tempval;
        }

```

I2C – exemple

- Exemple sous MBED : lecture de la température sur un LM75

```
45          // Decimal part (0.5°C precision)
46          if (tempval & 0x01) {
47              TempCelsiusDisplay[5] = 0x05 + 0x30;
48          } else {
49              TempCelsiusDisplay[5] = 0x00 + 0x30;
50          }
51
52          // Integer part
53          tempval >>= 1;
54          TempCelsiusDisplay[1] = (tempval / 100) + 0x30;
55          TempCelsiusDisplay[2] = ((tempval % 100) / 10) + 0x30;
56          TempCelsiusDisplay[3] = ((tempval % 100) % 10) + 0x30;
57
58          // Display result
59          pc.printf("temp = %s\n", TempCelsiusDisplay);
60          myled = !myled;
61          wait(1.0);
62      }
63  }
64
```





Le bus I2C : STM32

Principales caractéristiques :

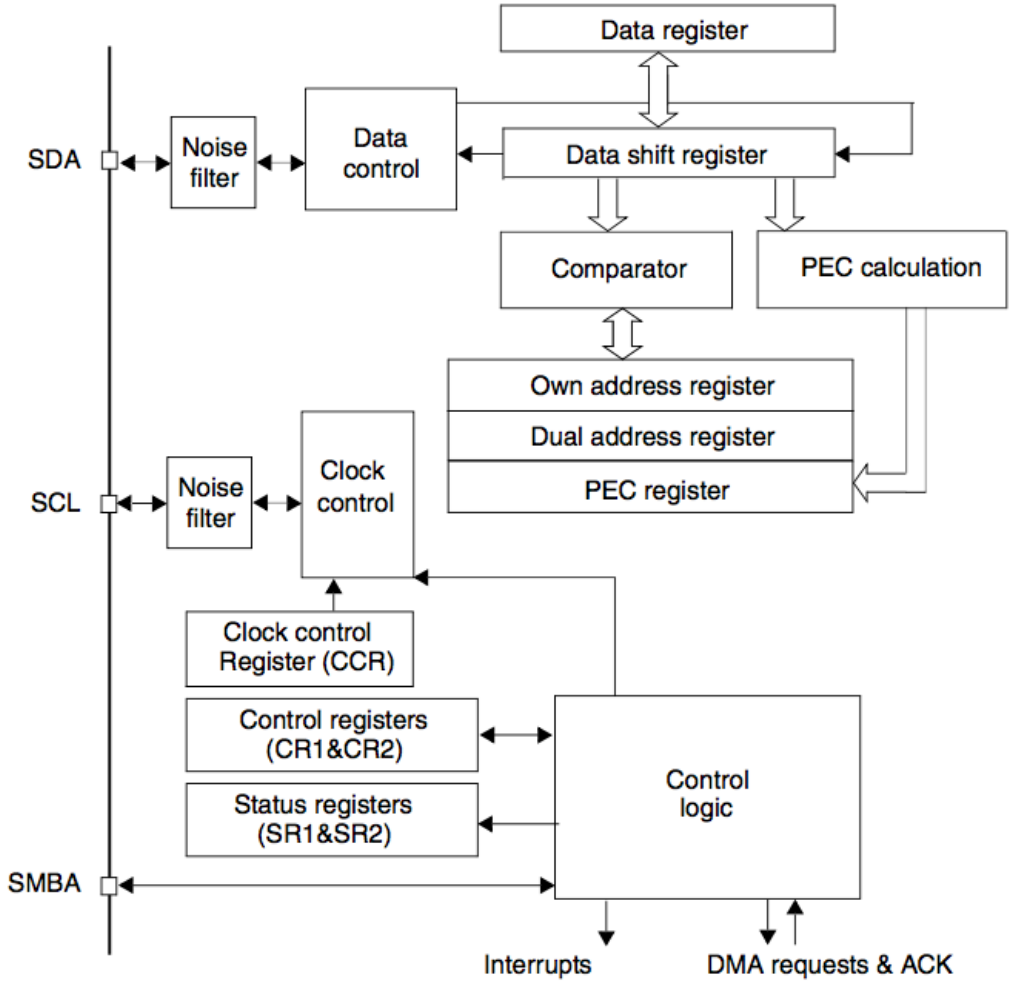
- Multimaster capability: la même interface peut être Maitre (Master) ou Esclave (slave)
- I2C Master – caractéristiques :
  - Génération d'horloge (Clock Generation)
  - Génération de signaux de Start et Stop
- I2C Slave – caracteristiques :
  - Détection d'adresse I2C programmable
  - Capacité d'adressage dual pour acquitter deux adresses esclaves
  - Détection de bit de Stop
- Génération et détection d'adressages 7-bit/10-bit et appel general
- Support de différentes vitesses de communication :
  - Vitesse standard (Standard Speed – jusqu'a 100 kHz)
  - Vitesse rapide (Fast Speed – jusqu'à 400 kHz)
  - La fréquence du bus I2C peut augmenter jusqu'à 1MHz.

Le bus I2C : STM32

Principales caractéristiques :

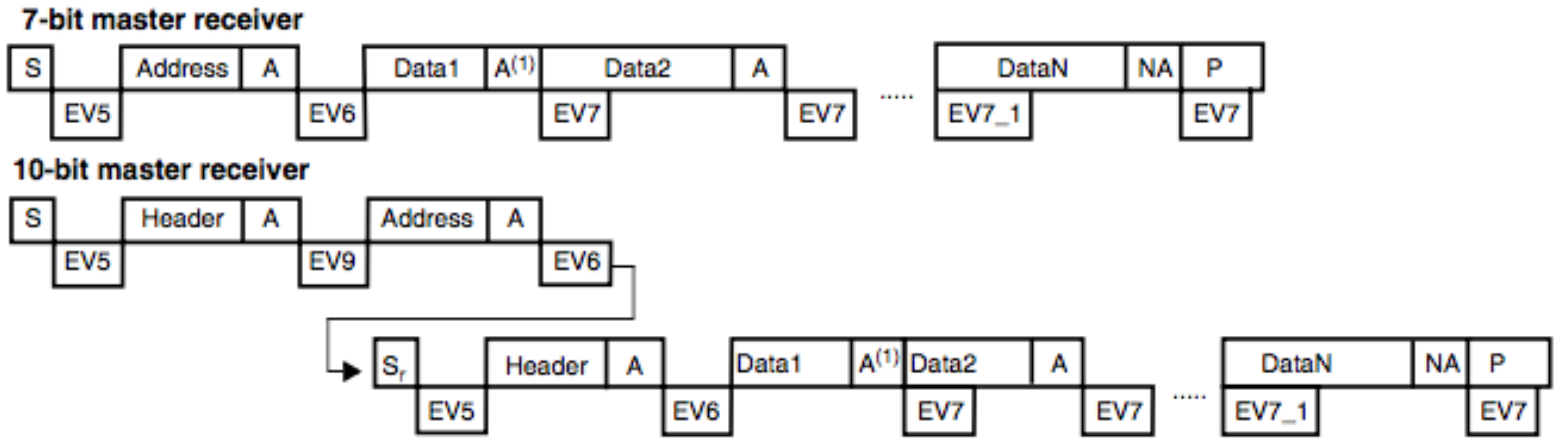
- Status flags:
  - Transmitter/Receiver mode flag
  - End-of-Byte transmission flag
  - I2C busy flag Error flags:
- SMBus 2.0 Compatibility:
  - 25 ms clock low timeout delay
  - 10 ms master cumulative clock low extend time
  - 25 ms slave cumulative clock low extend time
  - Hardware PEC generation/verification with ACK control
  - Address Resolution Protocol (ARP) supported
- PMBus Compatibility
- Interrupt vectors:
  - 1 Interrupt for successful address/ data communication
  - 1 interrupt for error condition

Le bus I2C : STM32



Le bus I2C : STM32

Séquence de transfert en transmission en mode master



Legend: S= Start, S_r = Repeated Start, P= Stop, A= Acknowledge, NA= Non-acknowledge, EVx= Event (with interrupt if ITEVFEN=1)

EV5: SB=1, cleared by reading SR1 register followed by writing DR register.

EV6: ADDR=1, cleared by reading SR1 register followed by reading SR2. In 10-bit master receiver mode, this sequence should be followed by writing CR2 with START = 1.

In case of the reception of 1 byte, the Acknowledge disable must be performed during EV6 event, i.e. before clearing ADDR flag.

EV7: RxNE=1 cleared by reading DR register.

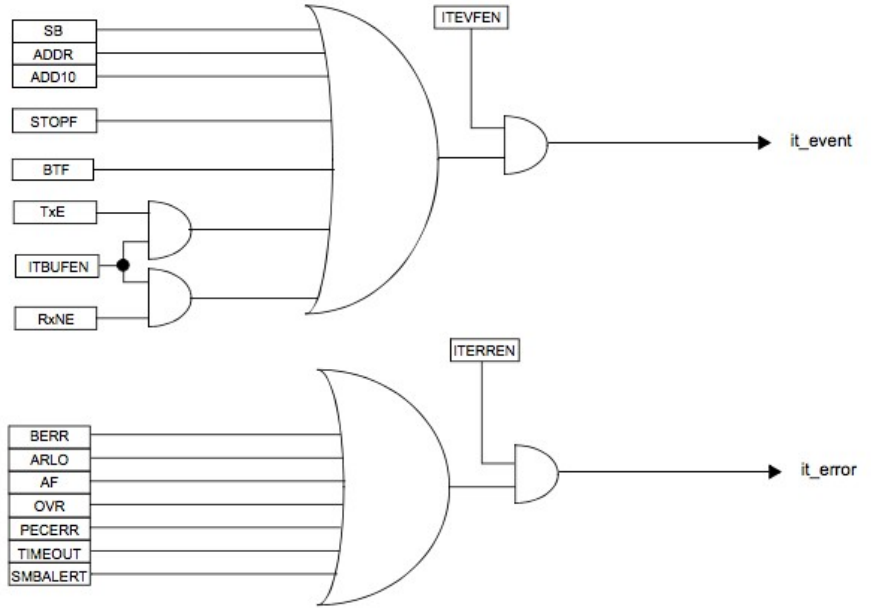
EV7_1: RxNE=1 cleared by reading DR register, program ACK=0 and STOP request

EV9: ADD10=1, cleared by reading SR1 register followed by writing DR register.

Le bus I2C : STM32

Sources d'interruption liées à une communication I2C

Interrupt event	Event flag	Enable control bit
Start bit sent (Master)	SB	ITEVFEN
Address sent (Master) or Address matched (Slave)	ADDR	
10-bit header sent (Master)	ADD10	
Stop received (Slave)	STOPF	
Data byte transfer finished	BTF	
Receive buffer not empty	RxNE	ITEVFEN and ITBUFEN
Transmit buffer empty	TxE	
Bus error	BERR	ITERREN
Arbitration loss (Master)	ARLO	
Acknowledge failure	AF	
Overrun/Underrun	OVR	
PEC error	PECERR	
Timeout/Tlow error	TIMEOUT	
SMBus Alert	SMBALERT	



Le bus I2C : STM32

Les registres de contrôles

- Registres de contrôles
- Registres d'adresse
- Registre de données
- Registres d'état
- Registres de contrôles des horloges

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	I2C_CR1	Reserved																SWRST	Reserved	ALERT	PEC	POS	ACK	STOP	START	NOSTRETCH	ENGC	ENPEC	ENARP	SMBTYPE	Reserved	SMBUS	PE
	Reset value																	0		0	0	0	0	0	0	0	0	0	0	0		0	0
0x04	I2C_CR2	Reserved																			LAST	DMAEN	ITBUFEN	ITEVTEN	ITERREN	Reserved	FREQ[5:0]						
	Reset value																				0	0	0	0	0		0	0	0	0	0	0	
0x08	I2C_OAR1	Reserved																ADDMODE	Reserved					ADD[9:8]		ADD[7:1]							ADD0
	Reset value																	0						0	0	0	0	0	0	0	0		0
0x0C	I2C_OAR2	Reserved																								ADD2[7:1]							ENDUAL
	Reset value																									0	0	0	0	0	0	0	0
0x10	I2C_DR	Reserved																								DR[7:0]							
	Reset value																									0	0	0	0	0	0	0	0
0x14	I2C_SR1	Reserved																SMBALERT	TIMEOUT	Reserved	PECERR	OVR	AF	ARLO	BERR	TxE	RxNE	Reserved	STOPF	ADD10	BTF	ADDR	SB
	Reset value																	0	0		0	0	0	0	0	0	0		0	0	0	0	0
0x18	I2C_SR2	Reserved																PEC[7:0]								DUALF	SMBHOST	SMBDEFAULT	GENCALL	Reserved	TRA	BUSY	MSL
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0
0x1C	I2C_CCR	Reserved																F/S	DUTY	Reserved	CCR[11:0]												
	Reset value																	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0
0x20	I2C_TRISE	Reserved																										TRISE[5:0]					
	Reset value																											0	0	0	0	1	0
0x24	I2C_FLTR	Reserved																											ANOFF	DNF[3:0]			
	Reset value																												0	0	0	0	0

Séance 3

IUT CERGY-PONTOISE

Partie 3

SPI: Serial Peripheral Interface

Le bus SPI

Le bus SPI

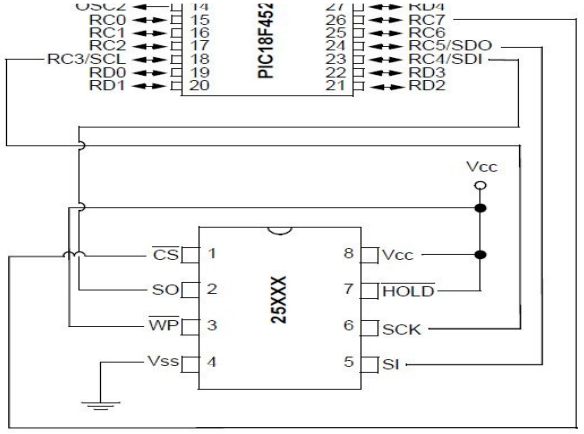
- Inventé par Motorola
- Bus synchrone
- Bus permettant le transfert des informations de 8 bits entre un micro-controller et un nombre de périphérique connecté sur le même bus.
- Le bus est full-duplex : les opérations d'écriture et de lecture se font simultanément
  - Des lignes du bus sont dédiées aux données en écriture
  - Des lignes du bus sont dédiées aux données en écriture
- Pas de partie dans le bus dédiée à l'adressage
  - Utilise la notion de chip selects pour choisir le périphérique M dans les N périphériques connectés
    - Nécessite de la logique externe pour en choisir 1
    - Sans logique, tous les périphériques récupèrent toutes les informations en même temps

Le bus SPI

- Bus sans mécanisme de surcharge logiciel/matérielle sur les conditions de start et de stop
  - Codage des trames : par exemple bus I2C
  - Bits dédiés : UART
- Taux de transmission (débit): 250Kb/s jusqu'à 10Mb/s
- Fonctionnement : maître ou esclave
- Principales utilisations:
  - Communication vers un périphérique dédié :
    - Mémoire : RAM/EEPROM/FLASH
    - Capteur, actionneur
    - Convertisseur CAN et CAN
    - Décodeur/Encodeur MP3
  - Communication entre microprocesseurs

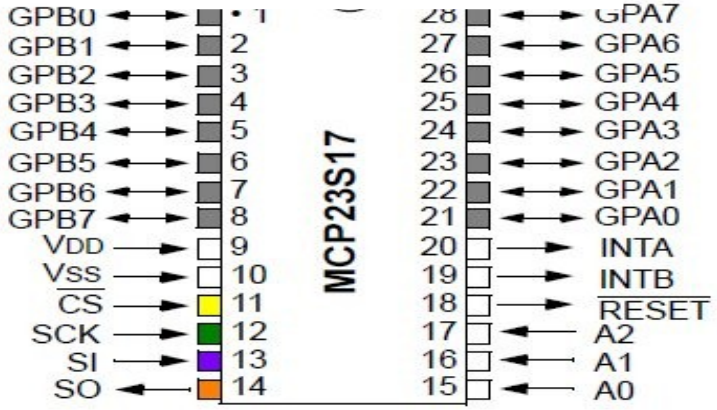
Le bus SPI

Mémoires

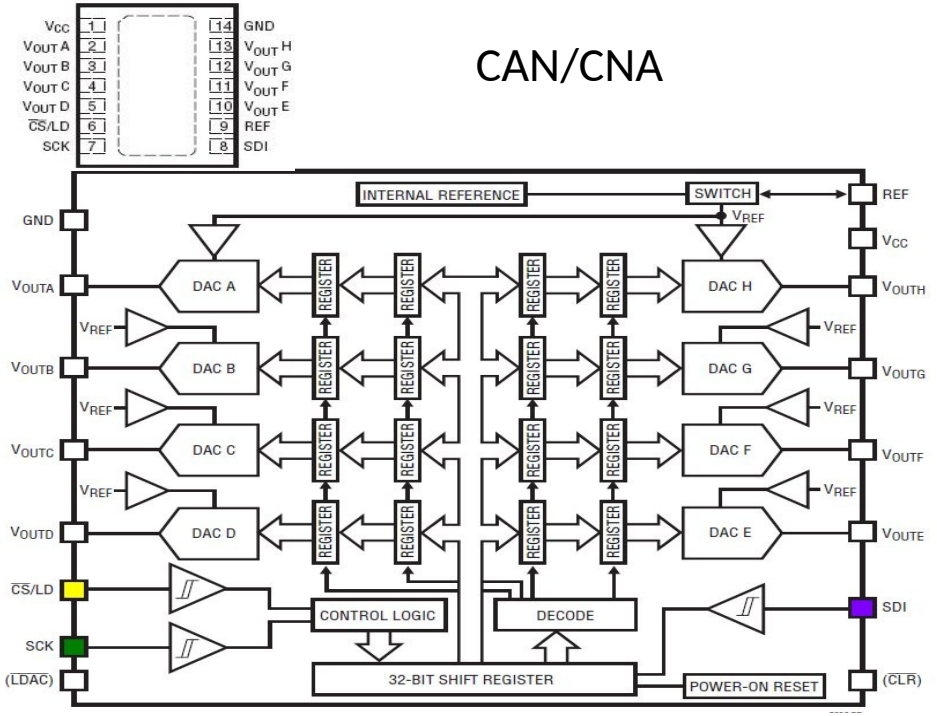


Afficheurs

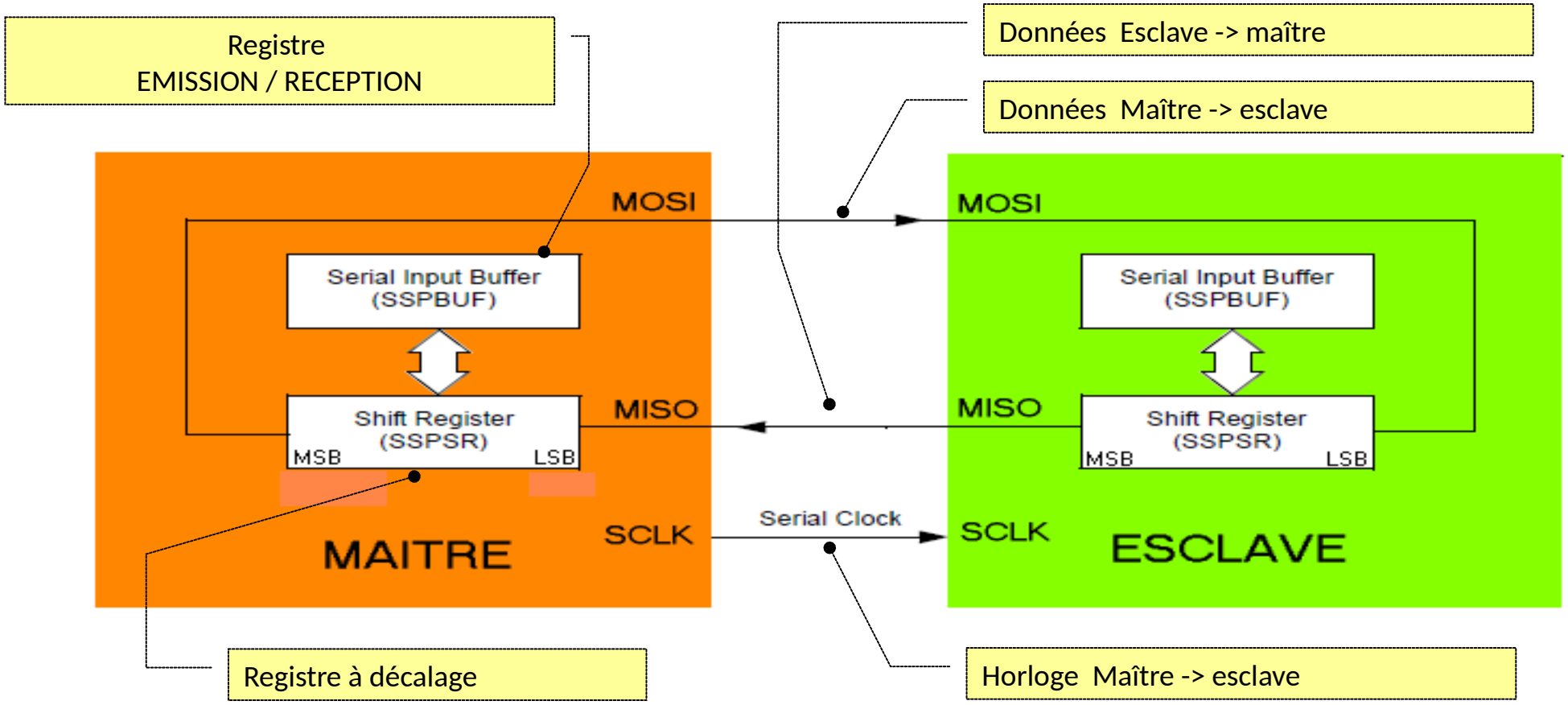
Extension de bus



CAN/CNA



Le bus SPI : principe



Le bus SPI : signaux

- Le bus SPI est un bus parallèle de 4 fils indépendant : 4 signaux //

- SCLK (Serial Clock)
  - Horloge délivrée par le maître

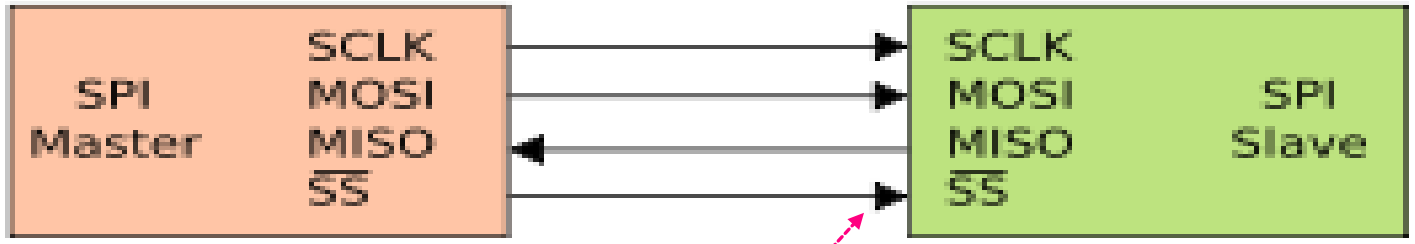
- MOSI (Master Output, Slave Input)
  - Sortie donnée maître, entrée donnée esclave

- MISO (Master Input, Slave Output)
  - Entrée donnée maître, sortie donnée esclave

- SS (Slave Select)
  - Sélection esclave

Le bus SPI : topologie

Un seul esclave 4 signaux

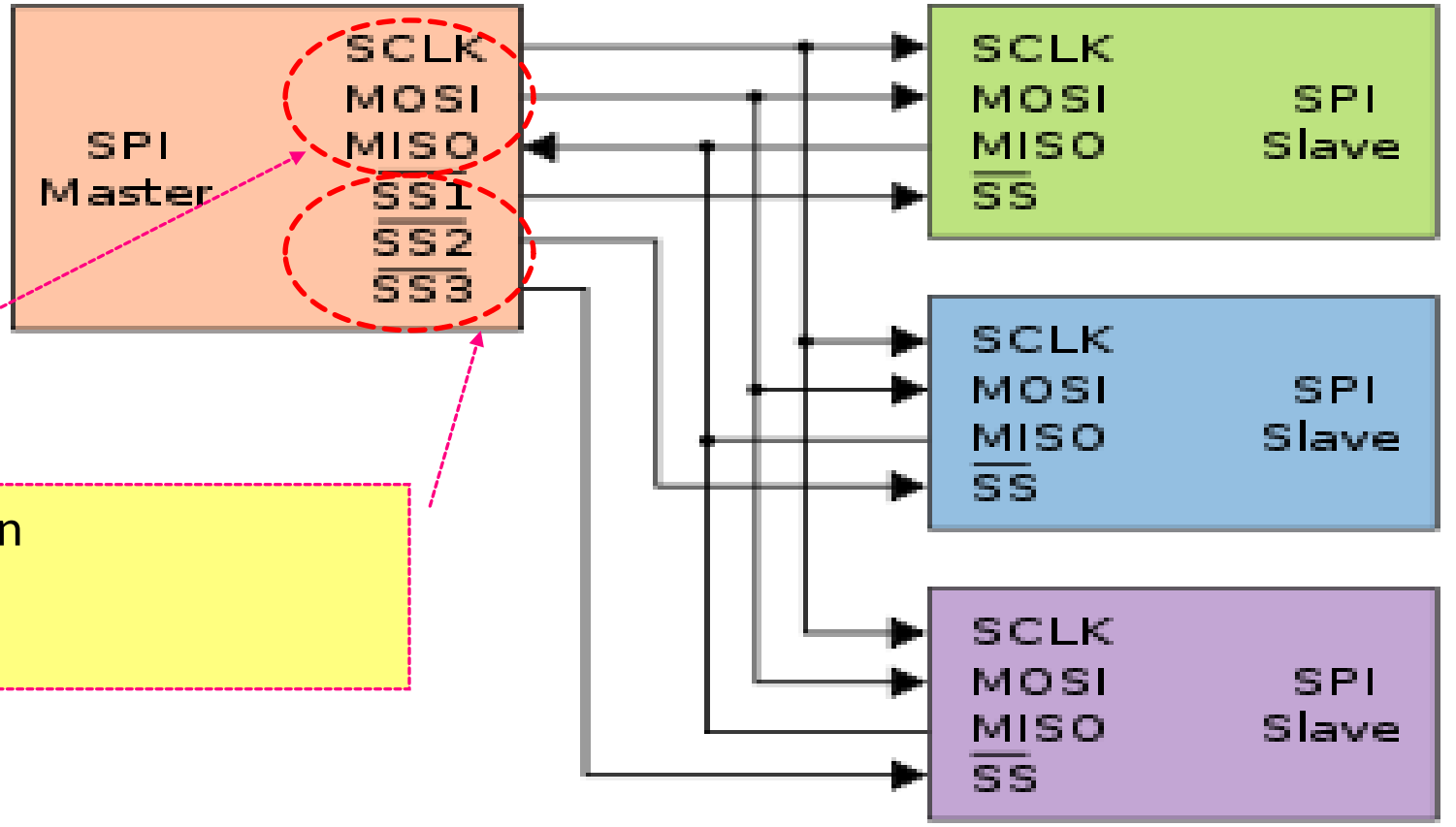


Sélection esclave
Optionnel

Un seul esclave : câblage minimum

Le bus SPI : topologie

Un seul esclave actif à la fois

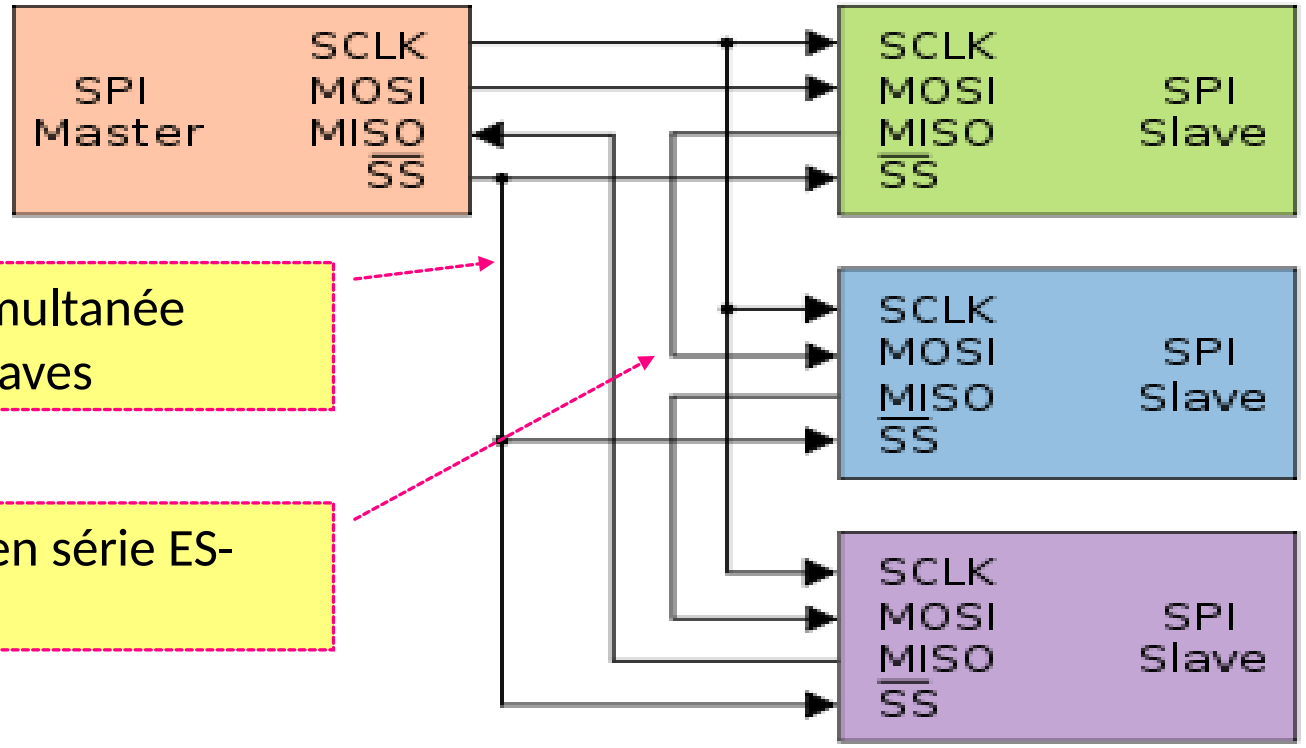


Signaux du bus

3 esclaves = 3 signaux de sélection
⇒ 3GPIO dédié
⇒ Daisy chain

Le bus SPI : topologie

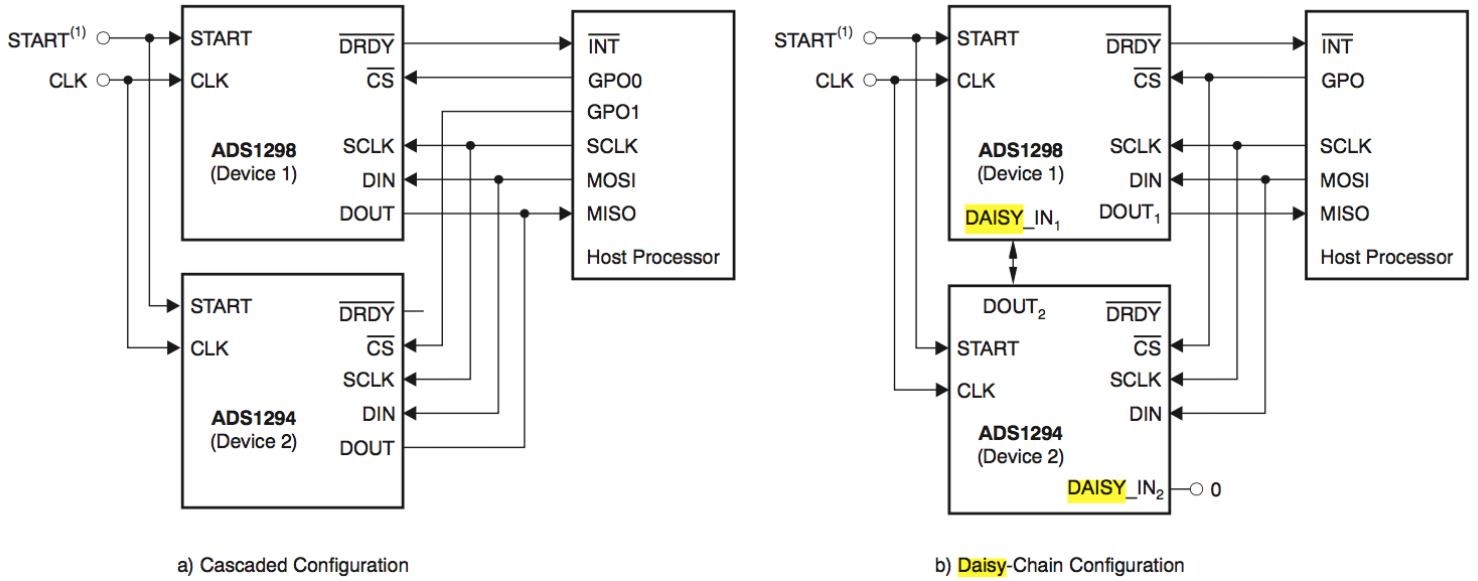
Les esclaves sont « chaînés » ₂ₑ les données envoyées se suivent



Sélection simultanée des esclaves

Les esclaves sont en série ES->ES->ES->..

Le bus SPI : exemple de daisy_chained

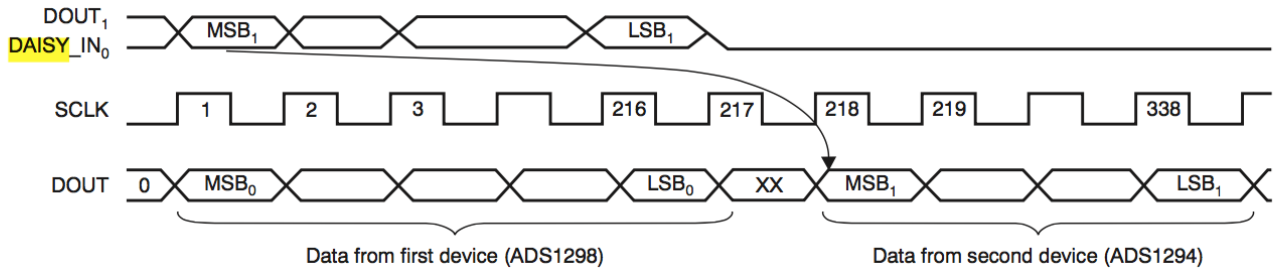


a) Cascaded Configuration

b) Daisy-Chain Configuration

(1) To reduce pin count, set the START pin low and use the START opcode command to synchronize and start conversions.

Figure 66. Multiple Device Configurations



Le bus SPI : Les paramètres de l'horloge

Trois paramètres :

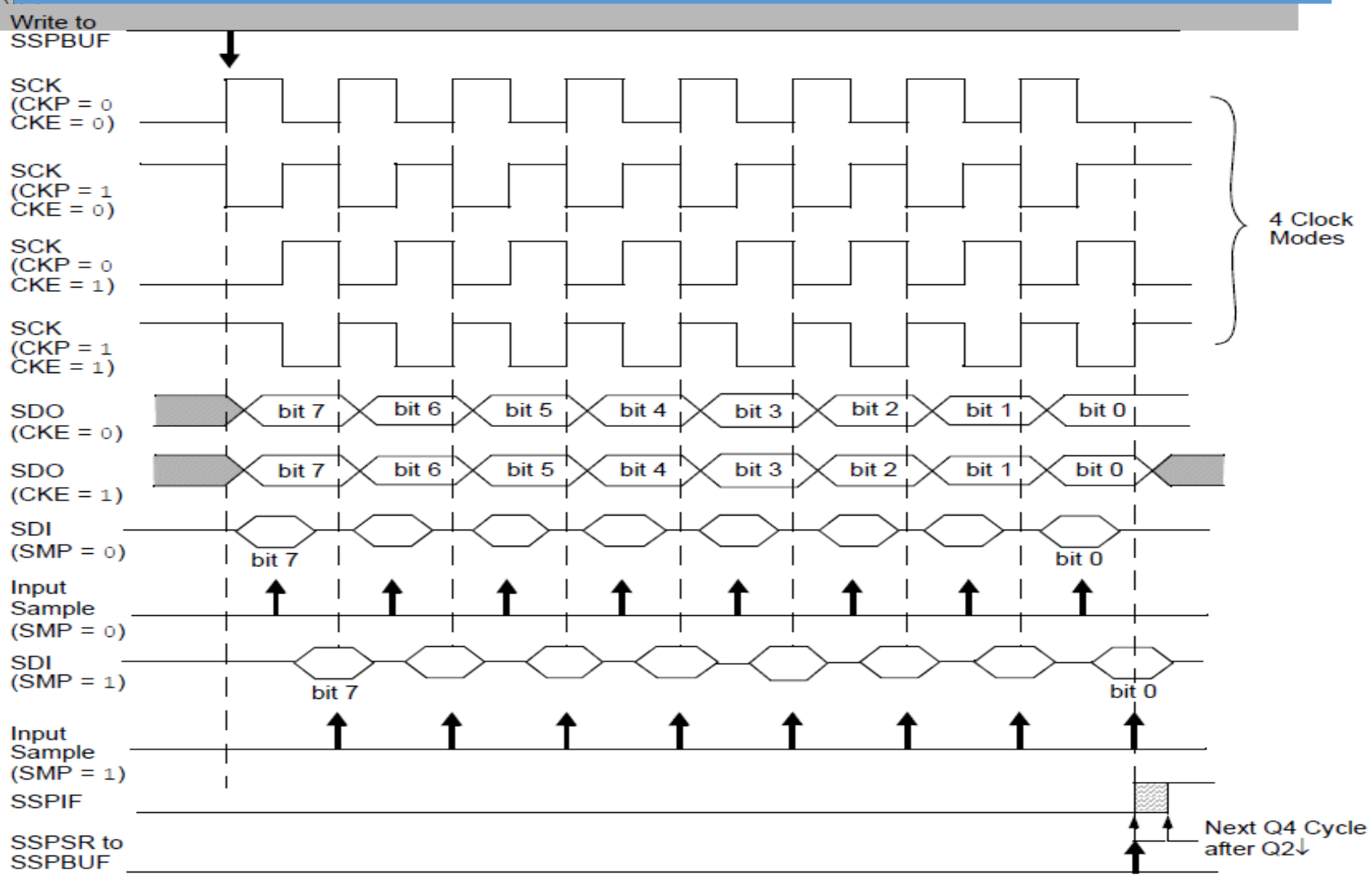
1. La fréquence d'horloge.
2. La polarité de l'horloge, paramètre CPOL (Clock polarity)
3. La phase de l'horloge, paramètre CPHA (Clock phase).

SPI-mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

- CPOL et CPHA ont deux état possible :
  ⇒ 4 possibilités de configuration.
- Les configurations étant incompatibles entre elles :
  ⇒ Maître et esclave doivent avoir les mêmes paramètres.

- La fréquence de l 'horloge est fixée par le maître :
  ⇒ Elle doit tenir compte des possibilités de l 'esclave.
  ⇒ Pas de contrainte sur la précision.

Le bus SPI : Les paramètres de l'horloge



Le bus SPI : Les paramètres de l'horloge

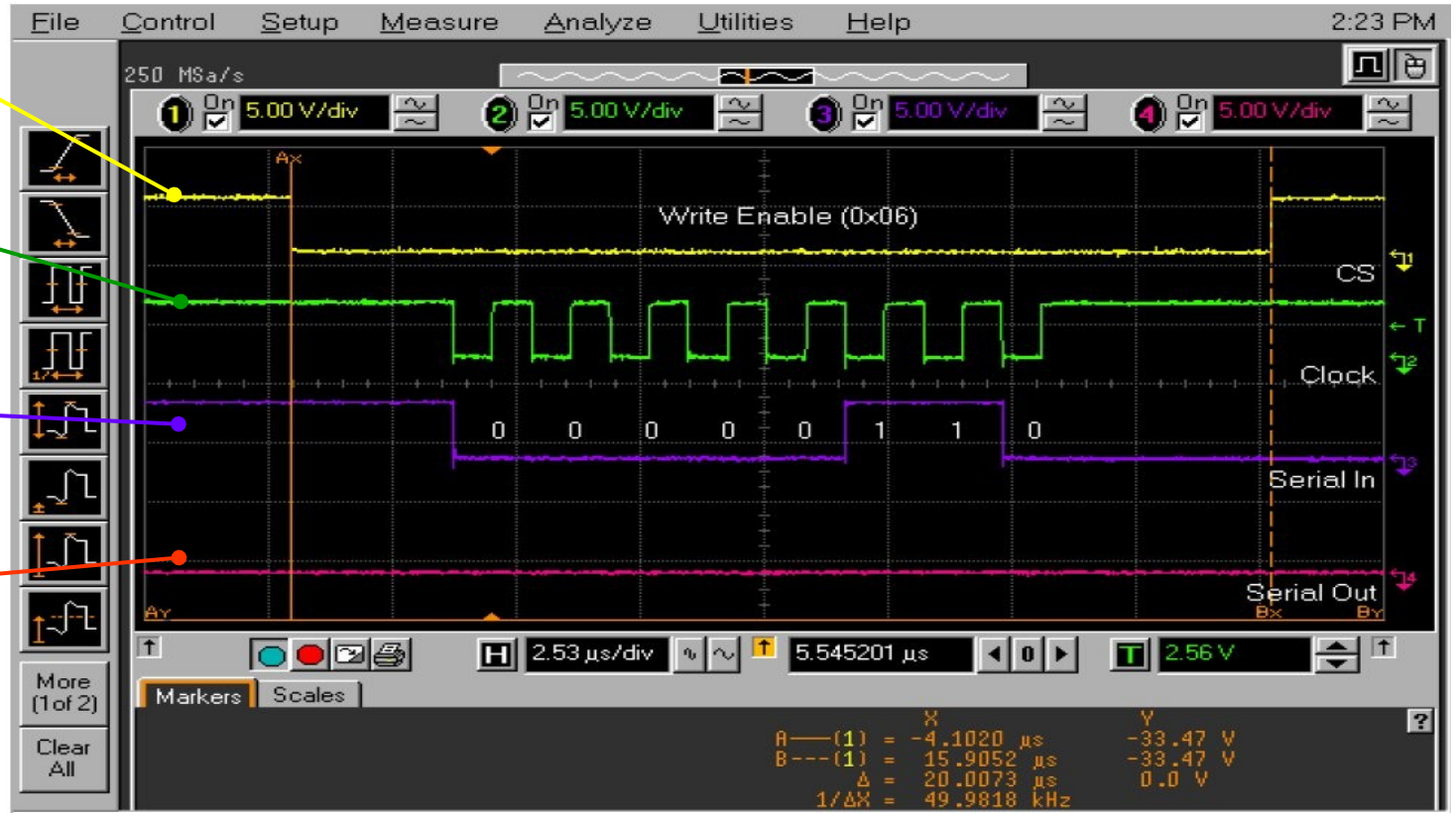
Exemple : autorisation d 'écriture dans une EEPROM

Sélection
-CS = 0

Horloge
-Sur front ⤙

µC -> 00000110
-Le MSB en premier

Sortie EEPROM
-En haute impédance



Le bus SPI : Les paramètres de l'horloge

Exemple : lecture du registre d'état d 'une EEPROM

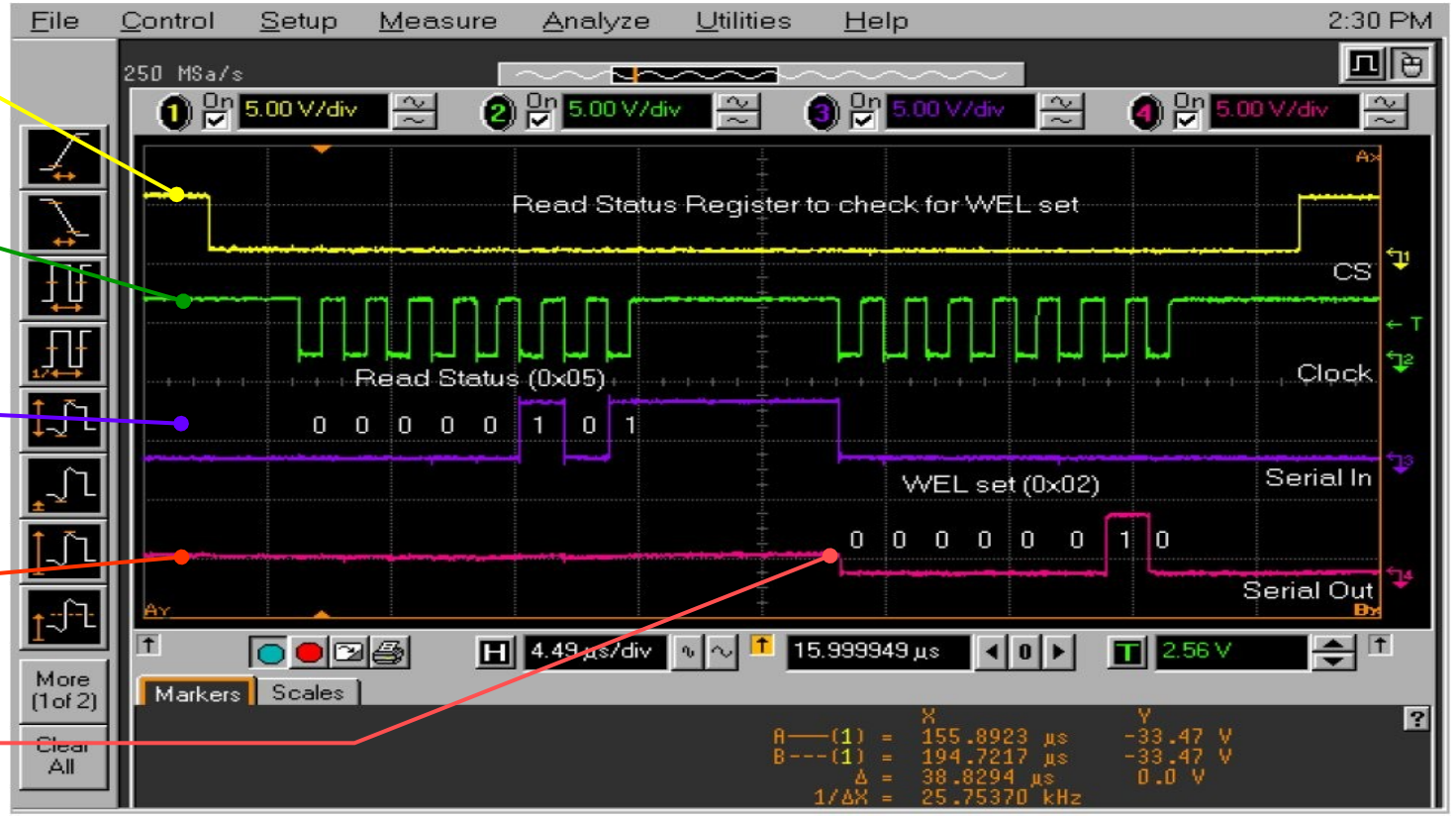
Sélection
-CS = 0

Horloge
-Sur front ⤙

µC -> 00000101
-Le MSB en premier

Sortie EEPROM
-En haute impédance

EEPROM-> 00000010
-Le MSB en premier



SPI – MBED

Exemple sous MBED

```
#include "mbed.h"

SPI device(SPI_MOSI, SPI_MISO, SPI_SCK);

int main() {
    int i = 0;
    while(1) {
        device.write(0x55);
        device.write(i++);
        device.write(0xE0);
        wait_us(50);
    }
}
```

Public Member Functions

	SPI (PinName mosi, PinName miso, PinName sclk, PinName ssel=NC) Create a **SPI** master connected to the specified pins.
void	**format** (int bits, int mode=0) Configure the data transmission format.
void	**frequency** (int hz=1000000) Set the spi bus clock frequency.
virtual int	**write** (int value) Write to the **SPI** Slave and return the response.
virtual void	**lock** (void) Acquire exclusive access to this **SPI** bus.
virtual void	**unlock** (void) Release exclusive access to this **SPI** bus.
template<typename Type > int	**transfer** (const Type *tx_buffer, int tx_length, Type *rx_buffer, int rx_length, const event_callback_t &callback, int event=SPI_EVENT_COMPLETE) Start non-blocking **SPI** transfer using 8bit buffers.
void	**abort_transfer** () Abort the on-going **SPI** transfer, and continue with transfer's in the queue if any.
void	**clear_transfer_buffer** () Clear the transaction buffer.
void	**abort_all_transfers** () Clear the transaction buffer and abort on-going transfer.
int	**set_dma_usage** (DMAUsage usage) Configure DMA usage suggestion for non-blocking transfers.

SPI – MBED

Exemple sous MBED

```
#include "mbed.h"

SPI spi(p5, p6, p7); // mosi, miso, sclk
DigitalOut cs(p8);

int main() {
    // Chip must be deselected
    cs = 1;

    // Setup the spi for 8 bit data, high steady state clock,
    // second edge capture, with a 1MHz clock rate
    spi.format(8,3);
    spi.frequency(1000000);

    // Select the device by seting chip select low
    cs = 0;

    // Send 0x8f, the command to read the WHOAMI register
    spi.write(0x8F);

    // Send a dummy byte to receive the contents of the WHOAMI register
    int whoami = spi.write(0x00);
    printf("WHOAMI register = 0x%X\n", whoami);

    // Deselect the device
    cs = 1;
}
```

SPI – MBED

HAL Mbed (suite)

Protected Member Functions	
void	**irq_handler_asynch** (void) **SPI** IRQ handler.
int	**transfer** (const void *tx_buffer, int tx_length, void *rx_buffer, int rx_length, unsigned char bit_width, const event_callback_t &callback, int event) Common transfer method.
int	**queue_transfer** (const void *tx_buffer, int tx_length, void *rx_buffer, int rx_length, unsigned char bit_width, const event_callback_t &callback, int event)
void	**start_transfer** (const void *tx_buffer, int tx_length, void *rx_buffer, int rx_length, unsigned char bit_width, const event_callback_t &callback, int event) Configures a callback, spi peripheral and initiate a new transfer.
void	**start_transaction** (**transaction_t** *data) Start a new transaction.
void	**dequeue_transaction** () Dequeue a transaction.

SPI – STM32

Spécification du SMT32F

- Full-duplex synchronous transfers on three lines
- Simplex synchronous transfers on two lines with or without a bidirectional data line
- 8- or 16-bit transfer frame format selection
- Master or slave operation
- Multimaster mode capability
- 8 master mode baud rate prescalers (fPCLK/2 max.)
- Slave mode frequency (fPCLK/2 max)

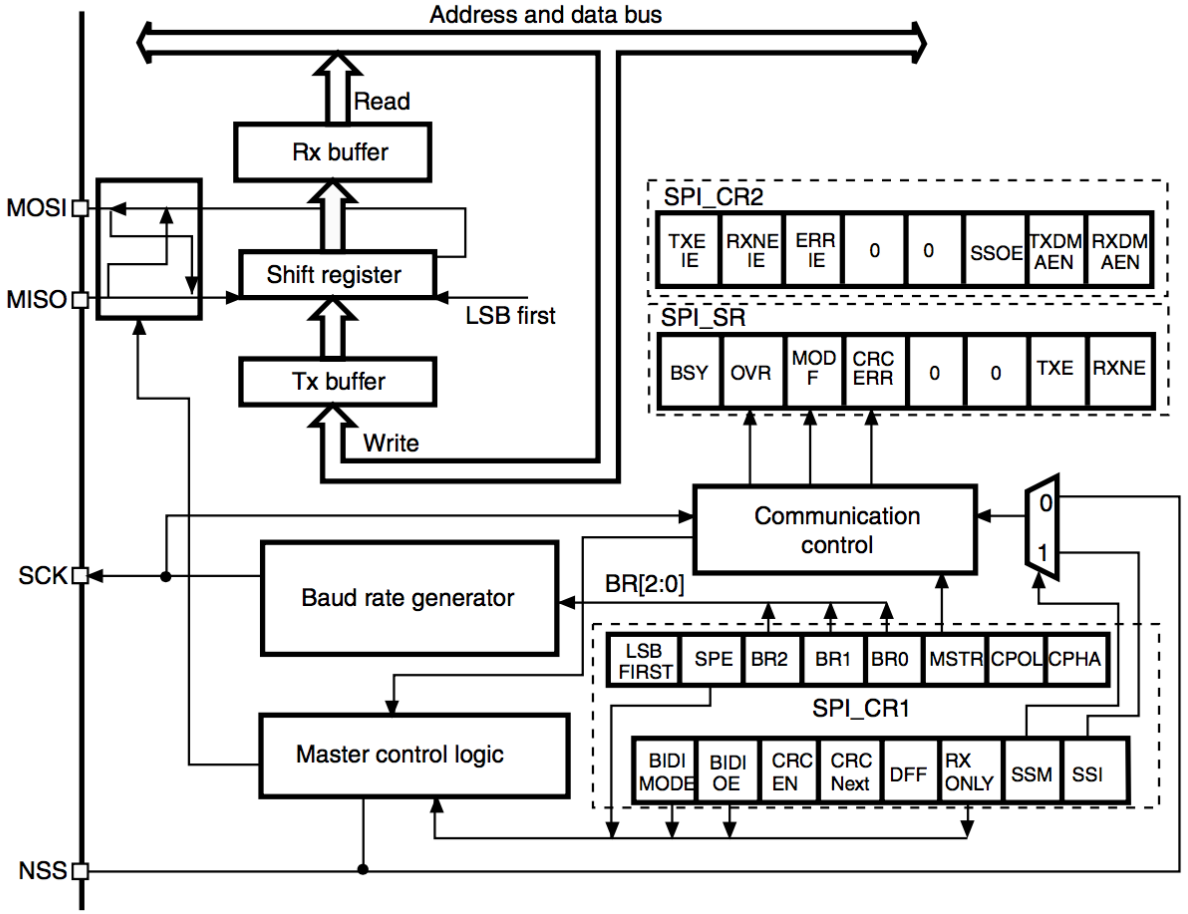
SPI – STM32

Spécification du SMT32F

- Faster communication for both master and slave
- NSS management by hardware or software for both master and slave: dynamic change of master/slave operations
- Programmable clock polarity and phase
- Programmable data order with MSB-first or LSB-first shifting Dedicated transmission and reception flags with interrupt capability SPI bus busy status flag
  SPI TI mode
  Hardware CRC feature for reliable communication:
- CRC value can be transmitted as last byte in Tx mode
- Automatic CRC error checking for last received byte Master mode fault, overrun and CRC error flags with interrupt capability1-byte transmission and reception buffer with DMA capability: Tx and Rx requests

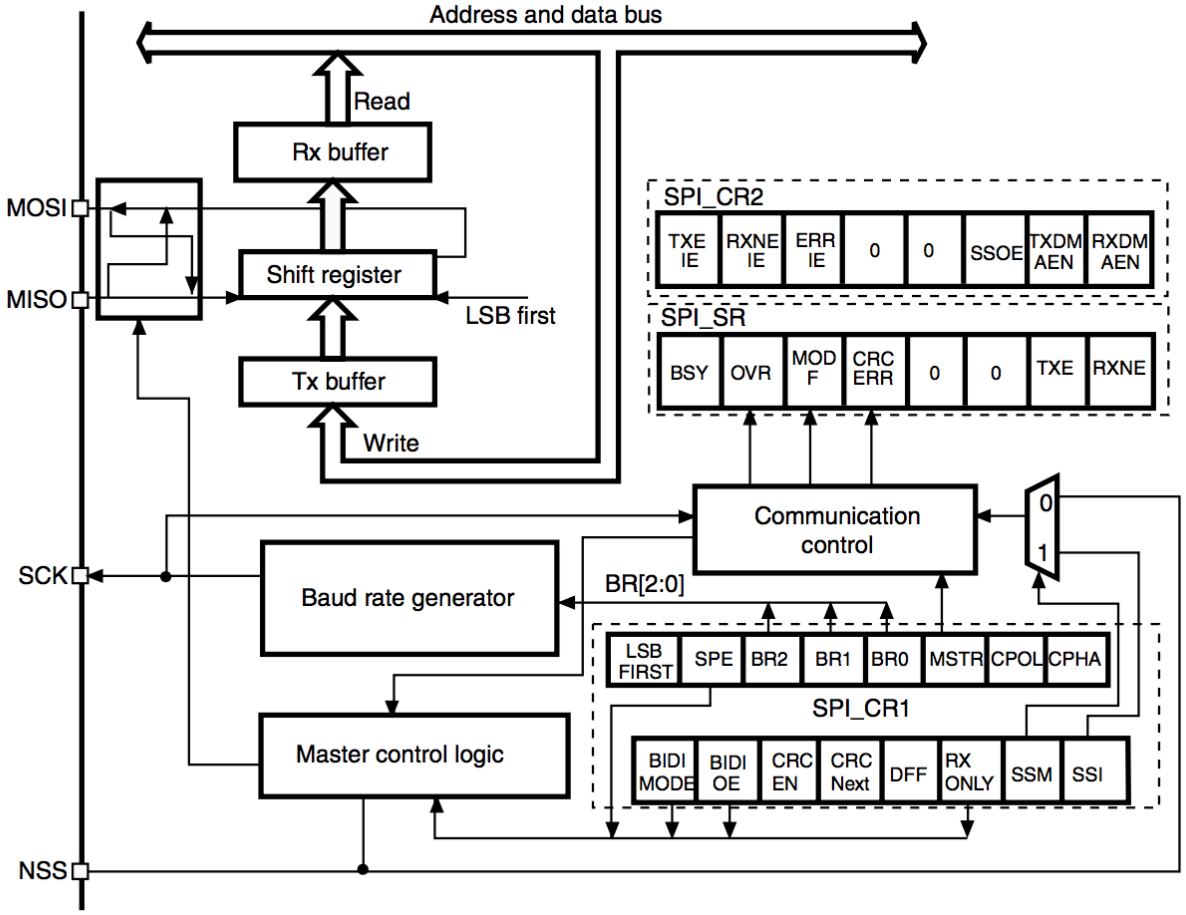
SPI – STM32

Architecture interne du périphérique



SPI – STM32

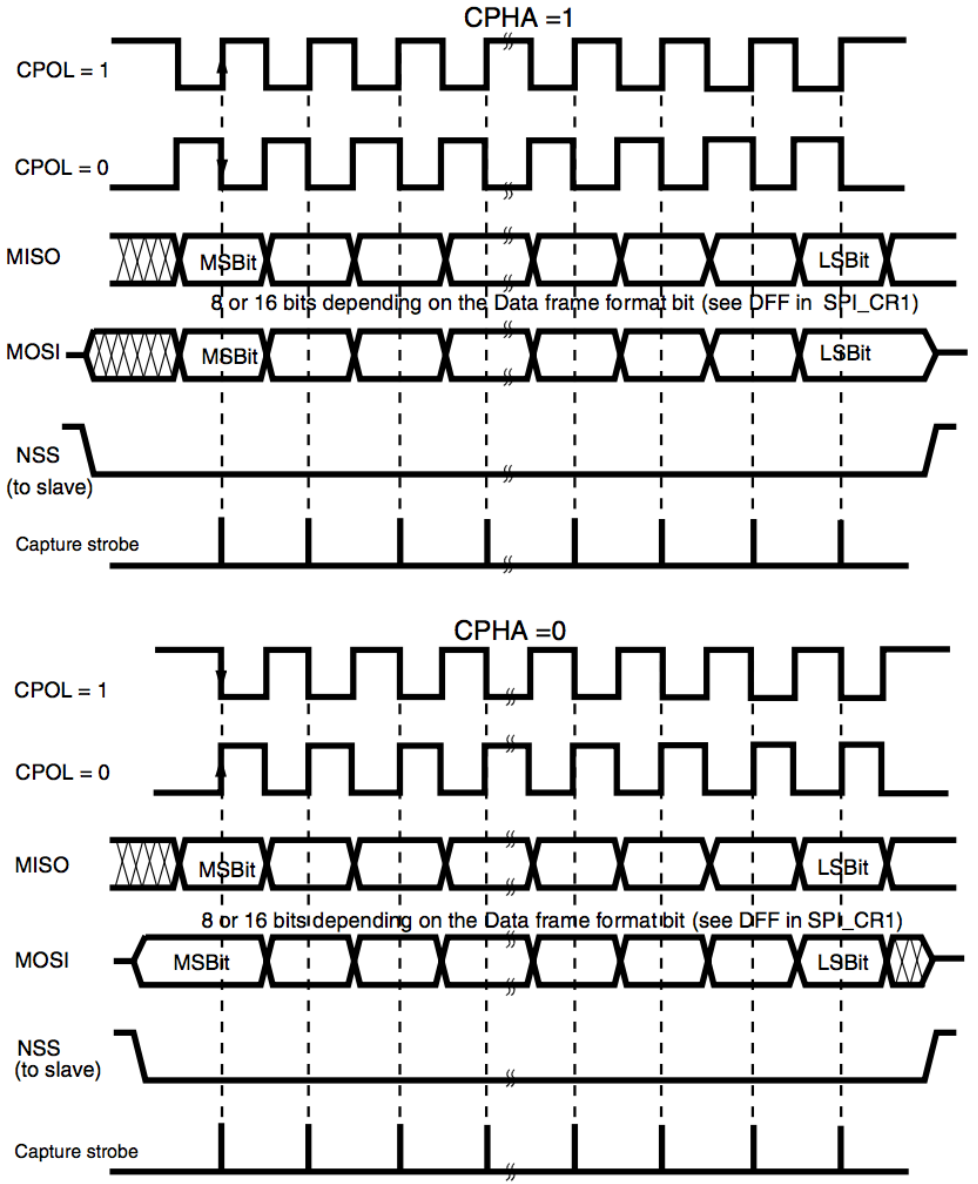
Architecture interne du périphérique



IUT CERGY-PONTOISE

SPI – STM32

Timing



SPI – STM32

Procédure de configuration

- Select the BR[2:0] bits to define the serial clock baud rate (see SPI_CR1 register).
- Select the CPOL and CPHA bits to define one of the four relationships between the data transfer and the serial clock (see Figure 194). This step is not required when the TI mode is selected.
- Set the DFF bit to define 8- or 16-bit data frame format
- Configure the LSBFIRST bit in the SPI_CR1 register to define the frame format. This step is not required when the TI mode is selected.
- If the NSS pin is required in input mode, in hardware mode, connect the NSS pin to a high-level signal during the complete byte transmit sequence. In NSS software mode, set the SSM and SSI bits in the SPI_CR1 register. If the NSS pin is required in output mode, the SSOE bit only should be set. This step is not required when the TI mode is selected.
- Set the FRF bit in SPI_CR2 to select the TI protocol for serial communications.
- The MSTR and SPE bits must be set (they remain set only if the NSS pin is connected to a high-level signal).

SPI – STM32

Registres du périphérique SPI

Registres de contrôles

Registres d'état

Registre de données

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**SPI_CR1**	Reserved																BIDIMODE	BIDIOE	CRCEN	CRCNEXT	DFF	RXONLY	SSM	SSI	*LSBFIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	**SPI_SR**	Reserved																							FRE	BSY	OVR	MODF	CRCERR	UDR	CHSIDE	TXE	RXNE
	Reset value																								0	0	0	0	0	0	0	1	0
0x0C	**SPI_DR**	Reserved																DR[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**SPI_CRCPR**	Reserved																CRCPOLY[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
0x14	**SPI_RXCRCR**	Reserved																RxCRC[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	**SPI_TXCRCR**	Reserved																TxCRC[15:0]															
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1C	**SPI_I2SCFGR**	Reserved																				I2SMOD	I2SE	I2SCFG		PCMSYNC	Reserved	I2SSTD		CKPOL	DATLEN		CHLEN
	Reset value																					0	0	0	0	0		0	0	0	0	0	0
0x20	**SPI_I2SPR**	Reserved																						MCKOE	ODD	I2SDIV							
	Reset value																							0	0	0	0	0	0	0	0	1	0

SPI – STM32

Exemple de configuration

Registre SPI_CR1
Adresse 0x00

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	*LSB FIRST*	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Bit 15 ***BIDIMODE:*** Bidirectional data mode enable
0: 2-line unidirectional data mode selected
1: 1-line bidirectional data mode selected
Note: This bit is not used in I²S mode

Bit 14 **BIDIOE:** Output enable in bidirectional mode
This bit combined with the BIDImode bit selects the direction of transfer in bidirectional mode
0: Output disabled (receive-only mode)
1: Output enabled (transmit-only mode)
Note: This bit is not used in I²S mode.
In master mode, the MOSI pin is used while the MISO pin is used in slave mode.

Bit 13 **CRCEN:** Hardware CRC calculation enable
0: CRC calculation disabled
1: CRC calculation enabled
Note: This bit should be written only when SPI is disabled (SPE = '0') for correct operation.
It is not used in I²S mode.

Bit 12 **CRCNEXT:** CRC transfer next
0: Data phase (no CRC phase)
1: Next transfer is CRC (CRC phase)
Note: When the SPI is configured in full duplex or transmitter only modes, CRCNEXT must be written as soon as the last data is written to the SPI_DR register.
When the SPI is configured in receiver only mode, CRCNEXT must be set after the second last data reception.
This bit should be kept cleared when the transfers are managed by DMA.
It is not used in I²S mode.

Bit 11 **DFF:** Data frame format
0: 8-bit data frame format is selected for transmission/reception
1: 16-bit data frame format is selected for transmission/reception
Note: This bit should be written only when SPI is disabled (SPE = '0') for correct operation.
It is not used in I²S mode.

Bit 10 **RXONLY:** Receive only
This bit combined with the BIDImode bit selects the direction of transfer in 2-line unidirectional mode. This bit is also useful in a multislave system in which this particular slave is not accessed, the output from the accessed slave is not corrupted.
0: Full duplex (Transmit and receive)
1: Output disabled (Receive-only mode)
Note: This bit is not used in I²S mode

Bit 9 **SSM:** Software slave management
When the SSM bit is set, the NSS pin input is replaced with the value from the SSI bit.
0: Software slave management disabled
1: Software slave management enabled
Note: This bit is not used in I²S mode and SPI TI mode

Bit 8 **SSI:** Internal slave select
This bit has an effect only when the SSM bit is set. The value of this bit is forced onto the NSS pin and the IO value of the NSS pin is ignored.
Note: This bit is not used in I²S mode and SPI TI mode

Bit 7 ***LSBFIRST:*** Frame format
0: MSB transmitted first
1: LSB transmitted first
Note: This bit should not be changed when communication is ongoing.
It is not used in I²S mode and SPI TI mode

Bit 6 **SPE:** SPI enable
0: Peripheral disabled
1: Peripheral enabled
Note: This bit is not used in I²S mode.
When disabling the SPI, follow the procedure described in Section 20.3.8: Disabling the SPI.

Bits 5:3 **BR[2:0]:** Baud rate control
000: $f_{PCLK}/2$
001: $f_{PCLK}/4$
010: $f_{PCLK}/8$
011: $f_{PCLK}/16$
100: $f_{PCLK}/32$
101: $f_{PCLK}/64$
110: $f_{PCLK}/128$
111: $f_{PCLK}/256$
Note: These bits should not be changed when communication is ongoing.
They are not used in I²S mode.

Bit 2 **MSTR:** Master selection
0: Slave configuration
1: Master configuration
Note: This bit should not be changed when communication is ongoing.
It is not used in I²S mode.

Bit1 **CPOL:** Clock polarity
0: CK to 0 when idle
1: CK to 1 when idle
Note: This bit should not be changed when communication is ongoing.
It is not used in I²S mode and SPI TI mode.

Bit 0 **CPHA:** Clock phase
0: The first clock transition is the first data capture edge
1: The second clock transition is the first data capture edge
Note: This bit should not be changed when communication is ongoing.
It is not used in I²S mode and SPI TI mode.

SPI – STM32

Exemple d'initialisation sous keil uVision

```
void SPI3_Master_init(void) {
    GPIO_InitTypeDef GPIO_InitStruct;
    SPI_InitTypeDef SPI_InitStruct;

//  enable peripheral clock
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI3, ENABLE);

//  enable clock for used IO pins
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

/* configure pins used by SPI3
 * PC10 = SCK
 * PC11 = MISO
 * PC12 = MOSI
 */
    GPIO_InitStruct.GPIO_Pin   = GPIO_Pin_12 | GPIO_Pin_11
                               | GPIO_Pin_10;
    GPIO_InitStruct.GPIO_Mode  = GPIO_Mode_AF;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStruct.GPIO_PuPd  = GPIO_PuPd_NOPULL;
    GPIO_Init(GPIOA, &GPIO_InitStruct);
```

```
//  connect SPI3 pins to SPI alternate function
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource10, GPIO_AF_SPI3);
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource11, GPIO_AF_SPI3);
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource12, GPIO_AF_SPI3);

//  enable clock for used IO pins
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

  GPIO_PinAFConfig(GPIOA, GPIO_PinSource15,GPIO_AF_SPI3);
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_15;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
  GPIO_Init( GPIOA, &GPIO_InitStructure );
```

SPI – STM32

Exemple d'initialisation sous keil uVision

```
  SPI_InitStruct.SPI_Direction = SPI_Direction_2Lines_FullDuplex;    // set to full duplex
  SPI_InitStruct.SPI_Mode = SPI_Mode_Master;                  // transmit in master mode, NSS pin has
  SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b;              // one packet of data is 8 bits wide
  SPI_InitStruct.SPI_CPOL = SPI_CPOL_Low;                     // clock is low when idle
  SPI_InitStruct.SPI_CPHA = SPI_CPHA_1Edge;                   // data sampled at first edge
  SPI_InitStruct.SPI_NSS = SPI_NSS_Hard;                  // set the NSS HARD
  SPI_InitStruct.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256;  // SPI frequency is
  SPI_InitStruct.SPI_FirstBit = SPI_FirstBit_MSB;             // data is transmitted MSB first
  SPI_Init(SPI1, &SPI_InitStruct);

 SPI_SSOutputCmd(SID_MASTER_SPI,ENABLE);                 // Set SSOE bit in SPI_CR1 register
 SPI_Cmd(SPI3, ENABLE);                          // enable SPI3
}
```