Fiche explicative : Liaisons série UART, SPI

Periph'Team - INSA de Toulouse

1 Introduction

Avec *l'UART* et le bus *SPI*, nous entrons dans le domaine de la communication entre deux ou plusieurs entités, c'est à dire une transmission de données. Parmi les autres bus célèbres, on peut citer le bus *I2C*, le bus *CAN*, l'*Ethernet*. Il en existe bien d'autres.

Les transmissions mises en place au travers de ces bus sont dits transmission en **bande de base**. Cela signifie qu'il n'y pas de modulation (contrairement à des transmissions *PSK*, *QAM*, *ADSL*...).

Une des caractéristiques essentielles des transmissions en bande de base est le débit de symboles R, qui s'exprime en **Bauds (Bd)**. Un symbole est un motif (une forme de tension) **stable** pendant une durée donnée, T_S.

Le débit de symbole ou vitesse de transmission est définie par :

$$R = \frac{1}{T_s}$$

Si un symbole est défini par 2 niveaux (le cas pour les *UART, SPI, I2C, CAN, Ethernet*), alors un symbole correspond à un bit. Le débit binaire (*bit/s*) est donc égal au débit de symboles (*Bd*).

On distingue les bus de transmission selon que les données sont accompagnées ou non d'une horloge. En l'**absence d'horloge**, on parle de **transmission asynchrone**, s'il existe une horloge, la transmission est dite synchrone.

Pour terminer cette introduction, précisons que les interfaces *UART* et *SPI* font partie des périphériques de communication les plus élémentaires. Il n'y a aucun protocole réseau associé. La seule sécurité implémentée est le contrôle de parité et le calcul de *CRC (**C**yclic **R**eduncy **C**heck)*. Il n'y a pas de notion d'adressage.

Le bus *I2C* est un bus synchrone qui est associé à un protocole. Avec ce bus, on commence à rentrer dans une logique de réseau, mais qui reste à un niveau très bas (pas de couche *OSI*). L'*I2C* utilise des adresses 7 bits.

Avec les bus *CAN* et *Ethernet*, on entre dans la catégorie des réseaux respectant les couches *OSI (Open System Interconnexion)* .

2 L'UART

2.1 Généralités

L'UART (Universal Asynchronous Receiver Transmitter) est un périphérique de micro-contrôleur qui permet d'émettre et de recevoir en même temps (full duplex) des octets de manière *asynchrone*.

On peut citer quelques exemples d'échange entre deux dispositifs *UART* : un PC et un *STM32F103*, ou bien encore un *STM32F103* et une carte *Raspberry* π. Deux *UARTs* qui communiquent doivent avoir des configurations communes. Parmi les éléments de configurations, et par ordre d'importance, on trouve :

- **le débit** : typiquement 9600 Bd, 19200 Bd, 38400 Bd,..., 115200 Bd
- **le nombre de bit de stop** : 1, 1.5 ou 2.
- **la parité** : c'est un $9^{ième}$ bit qui vaut $'1'$ selon que le nombre de bit à $'1'$ est pair ou impair (on parle de parité paire et impaire).
- **le contrôle de flux** : bien souvent désactivé, le contrôle de flux met en œuvre des lignes de contrôles supplémentaires pour gérer les situations d'un buffer plein à la réception.

Une *UART* (figure 1) émet une donnée par sa ligne *Tx*. Elle reçoit une donnée sur la ligne *Rx*. Pour signaler une émission, *l'UART* insère un bit spécifique, le bit de *start*, reconnaissable par *l'UART* de réception. Chaque bit est alors transmis, l'un après l'autre selon la vitesse de transmission fixée. Le bit de poids faible (*lsb : Least Significant Bit*) est transmis en premier. Le dernier bit est immédiatement suivi par un (ou plusieurs) *bits de stop*.

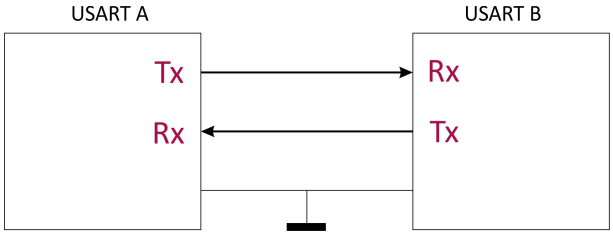

Figure 1: *Interconnexion de deux USART*

La figure 2 représente une transmission asynchrone. Dans l'exemple le caractère *0x5D* est émis (caractère ASCII ']').

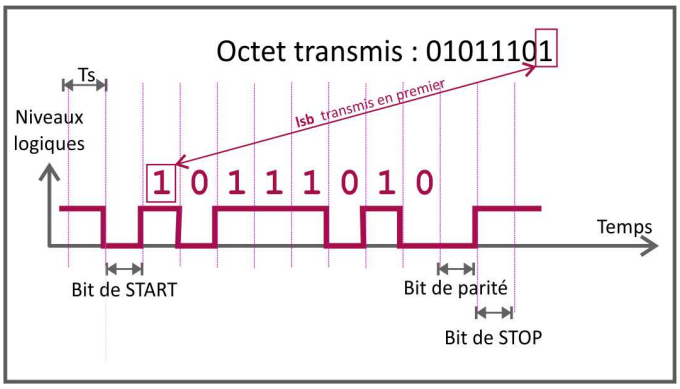

Figure 2: *Exemple de trame asynchrone*

On remarque que le signal est au niveau $'1'$ au repos. Le bit de *start* vaut $'0'$ tandis que le bit de stop vaut $'1'$. Si cette trame est émise par *l'UART A* , *l'UART B* qui est en écoute, va détecter le front descendant du bit de *start*. Connaissant la vitesse de transmission, *l'UART B* va donc pouvoir fixer les instants de décision qui vont lui permettre d'échantillonner chacun des bits, et donc de recomposer l'octet (transformation série / parallèle). Passé le bit de stop, *l'UART B* repasse en attente du prochain bit de *start*.

2.2 Le protocole RS232

La figure 1 montre une interconnexion entre deux *UART* avec des niveaux de tension typiques des circuits numériques ($5V$ ou $3,3V$). Ces niveaux sont parfois insuffisants pour pouvoir transmettre sur de longue distance ou / et dans un environnement bruité. La norme *RS232* propose des niveaux de tensions largement plus élevés pour garantir une plus grande robustesse.

La figure 3(a) montre la correspondance de niveaux entre la tension issue du contrôleur et la tension sur la ligne. On note également une inversion logique. Enfin, la norme précise une tension entre $+3V$ et $+25V$ pour un niveau $'0'$ et entre $-3V$ et $-25V$ pour un niveau $'1'$.

Il existe des circuits spécialisés pour opérer l'adaptation des tensions. Typiquement le *MAX232* (fig 3(b)). Avec seulement quelques condensateurs autour du composant, le *MAX232* est capable de générer les tensions $+$ et $-12V$.

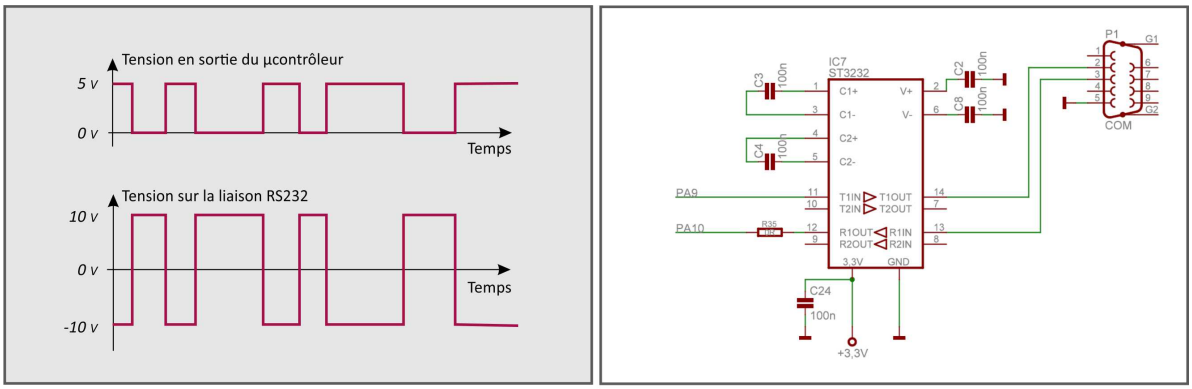

Figure 3: *La norme RS232*

2.3 Détails structurels de *l'UART* du *STM32F103*

Le périphérique *UART* du *STM32F103* dispose d'un registre de donnée, *DR* dans lequel on peut écrire ou lire, mais avec des fonctions asymétriques comme le montre la figure 4. En réalité, *DR* n'existe pas. En écriture, c'est *TDR* (*Transmit DR*) qui est écrit. En lecture, c'est *RDR* (*Received Data Register*) qui est lu.

En réception, un flag qui fait partie du *SR* (*Status Register*) permet de savoir si une donnée est présente dans *RDR* (qui vient d'être reçue par *l'UART*). Il s'agit du bit *RXNE* (*Read Data Register Not Empty*). On peut lui associer une interruption.

En émission, il existe également un flag indiquant que le pseudo registre *DR* (*TDR* en fait) est vide : la donnée vient d'être sérialisée. Il s'agit du bit *TXE* (*Transmit Data Register Empty*). Une interruption peut également lui être associée.

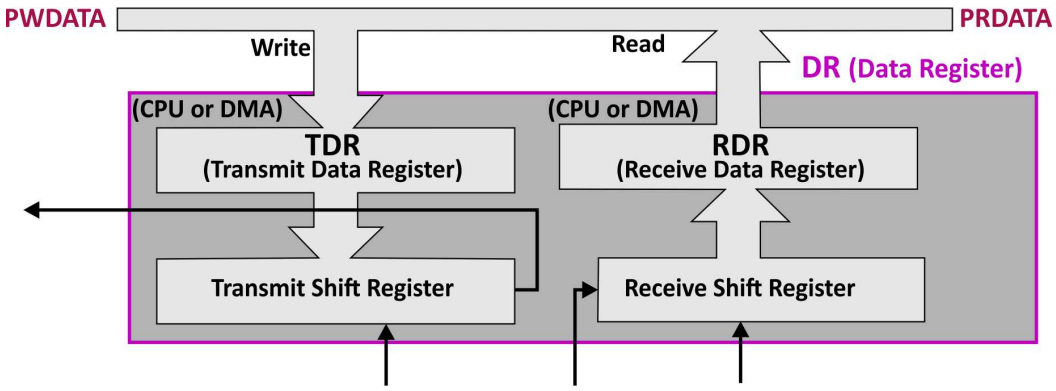

Figure 4: *Partie de l'USART, R/W dans DR*

Remarque : Sur certains micro-contrôleurs dont le *STM32*, la fonctionnalité UART peut être implantée dans un périphérique appelé *USART* (*Universal Synchronous Asynchronous Receiver Transmitter*) qui supporte des options supplémentaires telles qu'un mode synchrone (peu utilisées).

2.4 Logique de programmation de l'*UART*

Il existe plusieurs approches pour gérer émission et réception de données, de la plus efficace en terme de charge *CPU* (tout interruptif) mais complexe, au moins efficace (scrutation *polling* des flags) mais simple en terme de code. Nous proposons une méthode intermédiaire.

2.4.1 Emission par l'*UART*

Une émission comme un *putchar* ou un *printf* par la liaison série, est entièrement contrôlée par le μC puisqu'il « sait » quand il faut envoyer quelque chose. La méthode par scrutation est alors envisageable et consiste à :

- écrire une donnée dans *DR*
- attendre (par *polling* sur *TXE*) que la donnée soit transférée dans le *Shift Register*.

Cette approche simple impose que l'**émission** soit placée dans une partie logicielle facilement interruptible : typiquement en **tâche de fond**. De cette manière, l'émission sera non bloquante pour toute autre interruption. Le fait d'entrecouper l'émission d'une chaîne de caractères est sans conséquences dans la plupart des cas.

Comment procéder si l'on souhaite émettre une chaîne de caractères dans une interruption ?

$\rightsquigarrow$ Il suffit de déclarer un **flag visible** par l'interruption et par **la tâche de fond**. L'interruption met simplement le *flag* à $'1'$ lorsqu'une émission est souhaitée : l'opération est non bloquante. Au niveau de la tâche de fond, l'état du *flag* est détecté, ce qui déclenche l'émission. Le *flag* est ensuite rabaissé.

Cette approche a l'avantage de :

- la simplicité (pas d'interruption),
- la possibilité de donner l'ordre d'émettre depuis une interruption sans allonger son temps d'exécution.

2.4.2 Réception par l'UART

Par définition, la réception de données est asynchrone dans le sens où le contrôleur "ne sait pas" quand une ou plusieurs données vont arriver. On ne peut que rarement se permettre d'attendre que le *flag RXNE* se lève et bloquer toute autre activité dans ce laps de temps. La scrutation est donc à proscrire. Dans le cas d'une réception, il est largement préférable d'utiliser une interruption sur activation du *flag RXNE*. On peut alors programmer l'interruption pour qu'elle range dans un tableau le caractère en cours, jusqu'à détection du *CR (Carriage Return)* qui délimite la fin d'une chaîne. Ainsi, une chaîne de caractères peut se former progressivement de manière automatique sans perte de tps *CPU*.

3 Le bus SPI

3.1 Présentation

Le bus *SPI (Serial Peripheral Interface)* est le plus simple des bus synchrones. Il est basé sur l'utilisation de registres à décalage (bascules D en série, voir figure 5). Il dispose de deux lignes de données (l'une en émission, l'autre en réception). Une troisième ligne véhicule l'horloge.

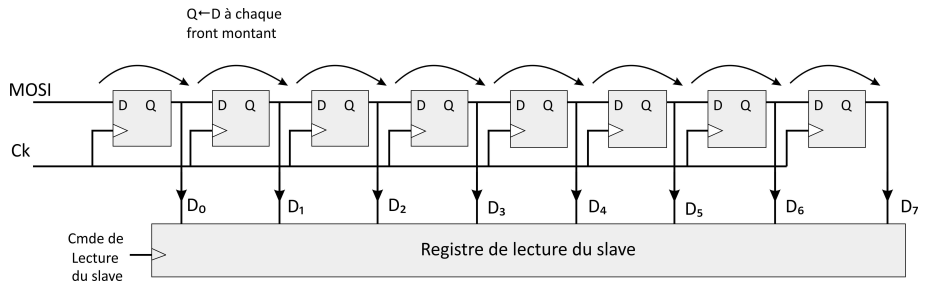

Figure 5: *Registre à décalage et registre de lecture d'une unité* SPI

La configuration la plus simple met en œuvre deux unités *SPI*. Celle qui contrôle l'horloge est le **maître (*master*)**. L'autre est **l'esclave (*slave*)**. Ces statut étant fixé, on peut définir les lignes de données :

- Pour le maître :
  - la ligne *MOSI (Master Out Slave In)* est la ligne par laquelle sortent les données, puisque l'unité est maître,
  - la ligne *MISO (Master In Slave Out)* est la ligne par laquelle entrent les données pour la même raison.
- Pour l'esclave :
  - la ligne *MOSI* est la ligne par laquelle entrent les données, puisque l'unité est l'esclave,
  - la ligne *MISO* est la ligne par laquelle sortent les données, pour la même raison.

Le maître peut utiliser une ou plusieurs autres lignes */CS1, /CS2 ...* *(Chip Select)* permettant d'avoir une topologie avec un maître et plusieurs esclaves avec qui communiquer.

Figure 6: *Configuration SPI : 1 master ⟷ 1 slave*

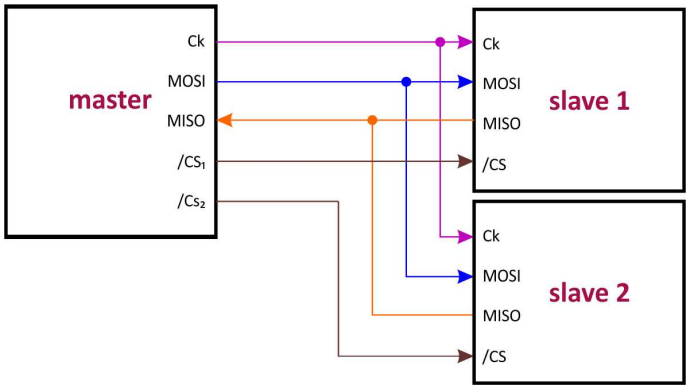

Figure 7: *Configuration SPI : 1 master⟷ 2 slaves*

Remarque : sur la figure 7, les lignes *MISO* du *Slave 1* et *Slave 2* sont reliées. Or elles sont toutes les deux susceptibles d'émettre des données, donc sont des "générateurs" pouvant entrer en conflit (l'un imposant $'1'$, l'autre $'0'$). Le rôle des entrées */CS* est donc de :

- déconnecter (grâce à un interrupteur commandé interne au *Slave*) la ligne *MISO* si */CS* $= '1'$ (*Slave* non sélectionné),
- connecter le ligne *MISO* si */CS* $= '0'$ (*Slave* sélectionné).

De cette manière, un seul slave se retrouve connecté au *master* et on retrouve la configuration de la figure 6.

3.2 Ecriture et lecture en mode Master sur *STM32F10x*

> *Remarque : Nous ne considérons ici uniquement la mode à 4 fils (Fig 7). Il est cependant possible d'utiliser également le* SPI *en mode à 3 fils.*

Afin de mieux comprendre la liaison SPI, nous allons nous appuyer sur la figure 8 qui correspond à l'architecture du *STM32F103* largement simplifiée. Le schéma est inspiré de la figure 208 (p 590) du *RM008*. Elle montre l'essentiel pour le propos qui va suivre :

- Le μcontrôleur en mode *Master* est relié à un *Device* quelconque *SPI* en mode 4 fils (donc fonctionnement en *full duplex*);
- en rouge (MISO) et en vert (MISO), on voit le trajet des bits en série sur les lignes de transfert de données;
- en violet les registres essentiels concernant les données du périphérique *SPI*. Le *shift register* joue un rôle central côté *STM32 Master*. Il est constitué de 8 bascules D reliées en série. Il peut être forcé en chargement parallèle via *Tx buff* et il peut être lu au travers de *Rx Buffer*. Le *shift Reg* est **unique**. Il sert à la fois en émission ET en réception.

> Tx Buffer *et* RxBuffer *sont en réalité un seul et même registre , le registre* DR. *Une écriture dans* DR *se fait dans* TxBuffer *alors qu'une lecture de DR se fait au travers de* RxBuff. *Ainsi, lire DR ne donnera pas ce que le programme vient d'écrire dans DR ! Lire DR c'est récupérer la donnée reçue par le SPI*

- Côté *Device slave*, dans un soucis de généralité, nous n'avons représenté que deux registres série, l'un en entrée (écriture μcontrôleur) l'autre en sortie (lecture μcontrôleur), chacun cadencé par la même horloge *SCK*.

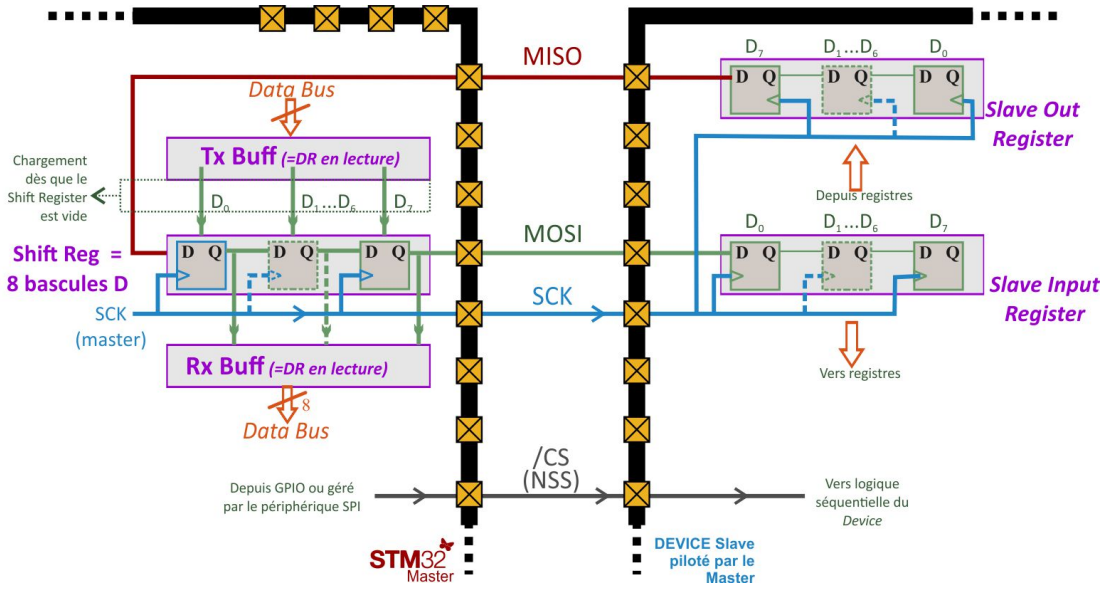

Figure 8: *Architecture simplifiée d'un fonctionnement Master/Slave*

Précisons enfin que, côté *STM32F103*, la configuration SPI correspond à :

- MSB first;
- 8 bits de transmission (en mode 16 bits, les registres TxBuff /RxBuff sont étendus à 16 bits);
- configuration 4 fils ful duplex;
- mode Master.

Il faut noter que l'horloge de synchronisation *SCK* pilote à la fois le registre série du côté du *STM32* et les registres séries du du côté du *Device*.

Lorsque le CPU écrit dans *DR*, il écrit dans *TxBuff*. Dès que possible, le contenu est transféré dans le *shift register*. Automatiquement, 8 fronts d'horloge sont envoyés à une fréquence donnée (via le *baud rate generator*). Les 8 bits sont alors complètement recopiés dans le registre *Slave Input Reg* du *Slave*.

Mais attention, en même temps le registre *shift register* du *STM32* se remplit avec ce que le *Device* présente sur la ligne *MISO* (bits de sortie du registre *Slave Out Reg*).

Le séquencement de la figure décrit les opérations à chaque front de l'horloge synchrone.

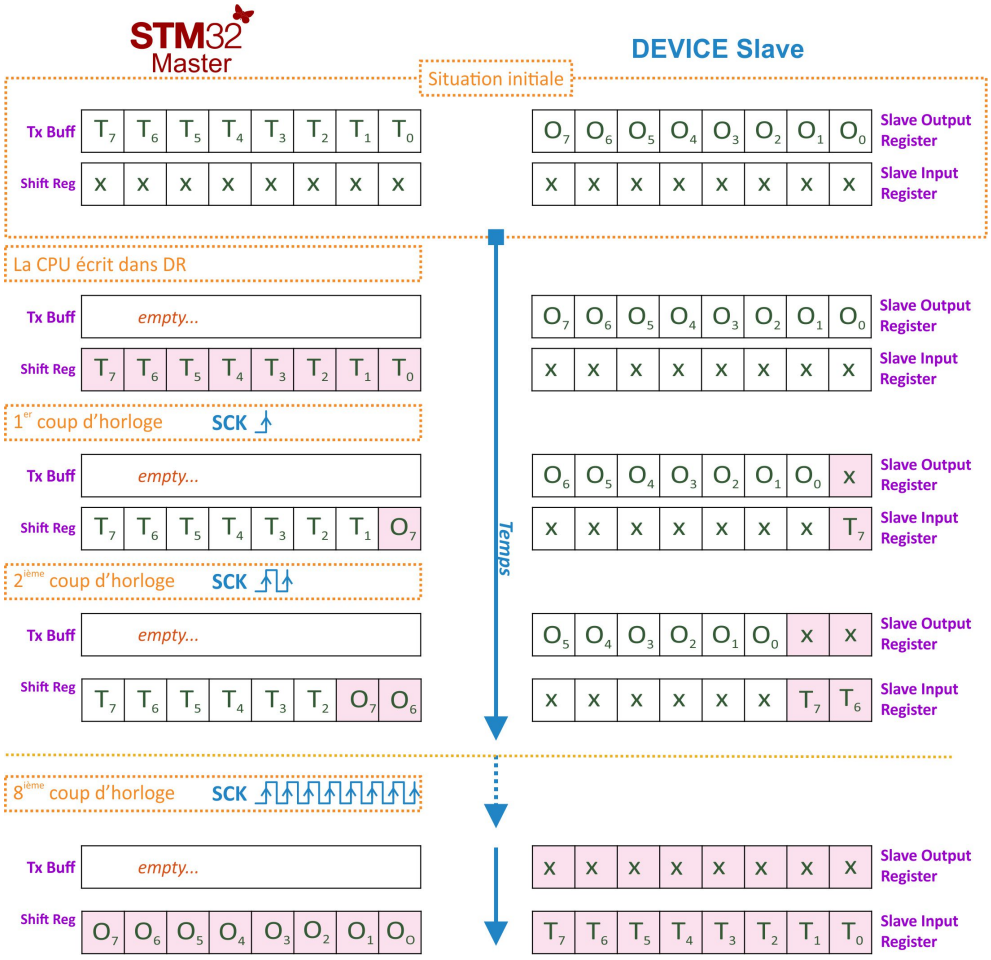

Figure 9: *Séquencement temporelle d'un fonctionnement Master/Slave*

La séquence temporelle précédente décrit précisément le transfert série pour chaque front d'horloge. A la fin du processus, les bits à transmettre $T_7...T_0$ sont bien recopiés dans le registre d'entrée du slave, et les bits de sortie du slave $O_7...O_0$ sont quant à eux recopiés dans le *shift register* du *Master*.

Signalons enfin que le **flag *RxNE (Rx Not Empty)*** indique que 8 nouveaux bits sont arrivés dans le *shift register* qui demande à être lu via *DR*.

Dans l'autre sens, le **flag *TxE (Tx Empty)*** précise quant à lui que le registre *TxBuff* est libre pour recevoir un nouvel octet. La CPU peut alors écrire une donnée dans *DR* (si une transmission série est en cours, le périphérique *SPI* va attendre la fin du transfert pour copier la donnée dans le *shift register*).

En résumé : Pour écrire un octet MaData*, il faut :*

- *attendre que le* DR *soit vide (*TxE*)*
- *placer* MadData *dans* DR *(dès lors les bits de* MaData *partent pendant que des bits inutiles sont lus.*
- *attendre la réception complète du byte inutile (*RxNE*)*
- *lire* DR *dans une variable "poubelle"*

Pour lire un octet, il faut

- *attendre que le* DR *soit vide (*TxE*);*
- *envoyer un octet bidon, typiquement* 0×00 *pour démarrer la* SCK *pendant que des bits utiles sont lus;*
- *attendre la réception complète du byte souhaité (*RxNE*);*
- *lire* DR *dans une variable et renvoyer sa valeur.*

Dernier commentaire au sujet des différences entre mode synchrone et mode asynchrone.

Du point de vue électronicien, la différence entre les deux, en approche globale, est la présence d'une horloge pour indiquer l'état stable de chaque bit. L'échantillonnage de chacun d'eux est alors trivial. Dans une transmission asynchrone, il faut anticiper les moments d'échantillonnage, en connaissant précisément la vitesse de transmission (le *baud rate*), c'est le cas d'une *UART*.

Du point de vue système : la notion d'horloge importe peu. La question est : "*est-ce que je maîtrise le moment d'arrivée d'un message ?*". Si oui, alors la transmission est dite synchrone. C'est le cas du *SPI* en mode *Master*. En mode *Slave*, "*je ne sais pas quand le device Master va m'envoyer un message*" . On est alors en mode asynchrone au sens système, tout comme pour une *UART*.

La distinction du point de vue système est importante. En effet, si la transmission est asynchrone, on utilisera systématiquement une interruption qui signale l'arrivée d'une nouvel octet (*UART* ou *SPI* mode *Slave*). Par contre en mode synchrone (*SPI* mode *Master*) on peut utiliser les interruptions mais ce n'est pas une obligation critique.

4 Chronogrammes

Il existe 4 types de configuration de l'horloge suivant que :

- son état de repos est $'0'$ ou $'1'$,
- son front actif (échantillonnage du bit correspondant) est montant ou descendant. La distinction se fait parfois non pas en terme de front montant ou descendant, mais en terme de premier front ou de second front actif.

Sur les figures 10 et 11 (extrait du manuel de référence RM008), on voit les 4 configurations possibles dépendant des bits *CPOL* et *CPHA*.

4.1 L'utilisation de */NSS* dans le *SPI*

Lorsqu'on communique avec un seul esclave, on peut être tenté de ne pas utiliser la sortie */NSS* [1] connectée au */CS* du slave.

Ce serait une erreur.

En effet, sans l'information */CS*, le slave ne peut décider de prélever la donnée 8 bits du registre à décalage qu'en comptant le nombre de fronts. Après le 8ème front, l'octet est supposé entièrement mémorisé et peut donc être exploité (lu dans le registre à décalage). Donc, si un front d'horloge est perdu, alors **tous** les caractères qui suivent sont erronés.

En utilisant le */CS*, le slave peut adopter une autre logique : au front descendant de */CS*, son compteur de fronts est remis à 0. Ainsi, si un front d'horloge est perdu, seul l'octet en cours est erroné. Les suivants seront recalés (compteur de fronts remis à 0 au prochain front descendant de */CS*).

Conseil : toujours utiliser */NSS* (hard ou soft) relié au */CS* du slave, même si un seul slave est utilisé.

[1] */NSS* est une seconde notation de */CS* pour le Master

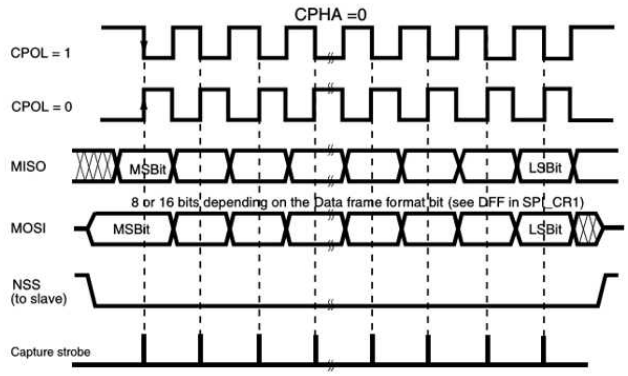

Figure 10: *SPI avec verrouillage au premier front*

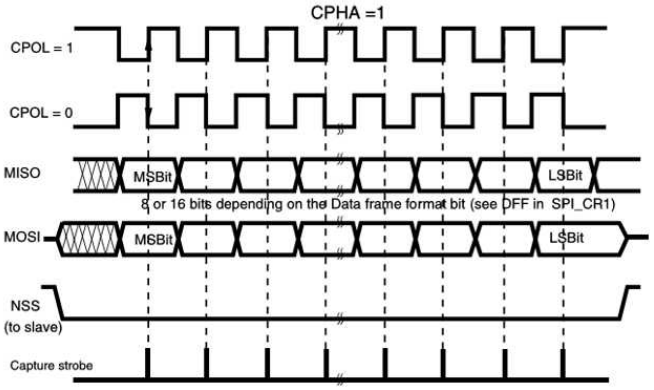

Figure 11: *SPI avec verrouillage au second front*

5 Annexe : Table ASCII

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 12: *LA table ASCII*