

MINF
Programmation des PIC32MX

Chapitre 7

Gestion de la communication série



Théorie PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version 1.92 septembre 2022

CONTENU DU CHAPITRE 7

7. Gestion de la communication sériele	7-1
7.1. Principe de la liaison sériele	7-2
7.1.1. Principe	7-2
7.1.2. Communication série asynchrone	7-3
7.1.2.1. Exemple de transmission asynchrone	7-4
7.1.2.2. Fonctionnement de la synchronisation	7-4
7.1.3. Communication série synchrone	7-5
7.2. La liaison RS232	7-5
7.2.1. Normes Electriques des communications série	7-5
7.2.1.1. Normes EIA (Electronic Industries Association)	7-6
7.2.1.2. Niveau logique norme V28	7-6
7.2.1.3. Niveau logique norme V11	7-6
7.2.2. Les signaux et la connectique	7-7
7.2.2.1. Connecteur 9 pôles du PC AT	7-7
7.2.2.2. Rôle des signaux	7-7
7.2.3. Câble Null Modem	7-8
7.2.4. Contrôle de flux hardware	7-9
7.2.5. Contrôle de flux software	7-9
7.2.6. Absence de contrôle de flux	7-9
7.3. Les USART du PIC32MX	7-10
7.3.1. Liste des UART à disposition (PIC32MX795F512L)	7-10
7.3.2. Schéma bloc de principe de l'UART	7-10
7.3.3. Schéma détaillé de la transmission	7-10
7.3.4. Schéma détaillé de la réception	7-12
7.3.5. Réalisation de la liaison RS232 sur le Kit	7-13
7.3.5.1. Câblage sur le PIC32MX795F512L	7-13
7.3.5.2. Adaptation pour les signaux RS232	7-13
7.3.6. Les registres des USART	7-14
7.3.6.1. Le registre UxMODE (UARTx Mode register)	7-14
7.3.6.2. Le registre UxSTA (UARTx Status and Control Register)	7-16
7.3.6.3. Le registre UxTXREG (UARTx Transmit Register)	7-18
7.3.6.4. Le registre UxRXREG (UARTx Receive Register)	7-18
7.3.6.5. Le registre UxBRG (UARTx Baudrate Register)	7-18
7.3.7. Le générateur de baudrate	7-19
7.3.7.1. Exemple calcul du baudrate	7-20
7.4. Configuration de l'USART avec la PLIB_USART	7-21
7.4.1. Le type énuméré USART_MODULE_ID	7-21
7.4.2. La fonction PLIB_USART_BaudRateSet	7-21
7.4.3. La fonction PLIB_USART_HandshakeModeSelect	7-21
7.4.4. La fonction PLIB_USART_OperationModeSelect	7-22
7.4.5. La fonction PLIB_USART_LineControlModeSelect	7-22
7.4.5.1. Le type énuméré USART_LINECONTROL_MODE	7-23
7.4.5.2. USART_LINECONTROL_MODE, exemple	7-23
7.4.6. La fonction PLIB_USART_TransmitterEnable	7-23

7.4.7.	PLIB_USART_TransmitterInterruptModeSelect	7-23
7.4.7.1.	Le type énuméré USART_TRANSMIT_INTR_MODE	7-24
7.4.7.2.	PLIB_USART_TransmitterInterruptModeSelect, exemple	7-24
7.4.8.	La fonction PLIB_USART_ReceiverEnable	7-24
7.4.9.	PLIB_USART_ReceiverInterruptModeSelect	7-24
7.4.9.1.	Le type énuméré USART_RECEIVE_INTR_MODE	7-24
7.4.9.2.	PLIB_USART_ReceiverInterruptModeSelect, exemple	7-24
7.4.10.	Exemple de configuration	7-25
7.5.	Fonctions de l'émetteur	7-26
7.5.1.	La fonction PLIB_USART_TransmitterByteSend	7-26
7.5.2.	La fonction PLIB_USART_TransmitterBufferIsFull	7-26
7.5.3.	Exemple d'émission avec TransmitterByteSend	7-26
7.6.	Fonctions du récepteur	7-27
7.6.1.	La fonction PLIB_USART_ReceiverByteReceive	7-27
7.6.2.	La fonction PLIB_USART_ReceiverDataIsAvailable	7-27
7.6.2.1.	PLIB_USART_ReceiverDataIsAvailable, exemple	7-27
7.6.3.	Fonction PLIB_USART_ReceiverOverrunErrorClear	7-27
7.7.	Fonctions générales de l'USART	7-28
7.7.1.	La fonction PLIB_USART_Disable	7-28
7.7.2.	La fonction PLIB_USART_Enable	7-28
7.7.3.	La fonction PLIB_USART_ErrorGet	7-29
7.7.4.	Exemple gestion des erreurs	7-29
7.8.	Emission et réception sans interruption	7-30
7.8.1.	Configuration USART sans interruption	7-30
7.8.2.	Emission en polling	7-30
7.8.3.	Utilisation de la fonction d'émission et observations	7-31
7.8.3.1.	Comportement de l'émission	7-31
7.8.3.2.	Résultat émission avec Terminal	7-31
7.8.4.	Réception en polling	7-33
7.8.4.1.	Fonction de remplissage du FIFO	7-33
7.8.4.2.	Appel de la fonction dans la réponse à l'interruption	7-33
7.8.4.3.	La fonction APP_GetMessAD	7-34
7.8.4.4.	Utilisation dans l'application	7-35
7.8.4.5.	Exemple de résultat avec l'application	7-36
7.8.4.6.	Comportement de la réception	7-36
7.8.4.7.	Conclusion sur réception en polling	7-36
7.9.	Nécessité du recours à un FIFO	7-37
7.9.1.	Rôles du FIFO	7-37
7.9.2.	Situation communication avec des FIFO	7-37
7.10.	Principe du FIFO	7-38
7.10.1.	Séparation des contextes	7-38
7.10.2.	Principe de l'écriture dans le FIFO	7-38
7.10.3.	Principe de la lecture dans le FIFO	7-40
7.11.	Analyse du fonctionnement d'un FIFO	7-41
7.11.1.	Situation de départ	7-41
7.11.2.	Situation après écriture de 2 caractères	7-41
7.11.3.	Situation après lecture des 2 caractères	7-41

7.11.4.	Situation de reboucllement	7-42
7.11.5.	Situation FIFO plein	7-42
7.11.6.	FIFO plein conclusion	7-42
7.12.	Réalisation FIFO avec des pointeurs	7-43
7.12.1.	Structure décrivant un FIFO	7-43
7.12.2.	Déclaration d'un FIFO	7-43
7.12.3.	Initialisation d'un FIFO	7-43
7.12.4.	La fonction GetWriteSpace	7-44
7.12.5.	La fonction GetReadSize	7-44
7.12.6.	La fonction PutCharInFifo	7-45
7.12.7.	La fonction GetCharFromFifo	7-46
7.12.8.	Gestion des FIFO, librairie à disposition	7-46
7.13.	Exemple utilisation des FIFOs et des interruptions	7-47
7.13.1.	Configuration de l'USART	7-47
7.13.1.1.	Configuration d'un USART Driver avec Harmony	7-47
7.13.1.2.	Contenu de la fonction DRV_USART0_Initialize	7-48
7.13.2.	Contexte de réception	7-49
7.13.3.	Détails configuration interruption réception	7-49
7.13.3.1.	Choix d'un mode de déclenchement de Int RX	7-50
7.13.4.	Section réception de l'interruption de l'USART	7-51
7.13.5.	Principe traitement interruption de réception	7-52
7.13.6.	Comportement interruption RX avec HALF_FULL	7-53
7.13.7.	Comportement interruption RX avec 3B4FULL	7-54
7.13.8.	Comportement interruption RX avec ONE_CHAR	7-55
7.13.9.	Principe du traitement de la réception (GetMessage)	7-56
7.13.9.1.	Test disponibilité du message dans RxFifo	7-56
7.13.9.2.	Traitement du Contrôle de flux en réception	7-57
7.13.10.	Vérification contrôle de flux de réception	7-58
7.13.10.1.	Vue d'ensemble contrôle flux réception	7-58
7.13.10.1.	Vue de détail contrôle flux réception	7-58
7.13.11.	Contexte d'émission	7-59
7.13.12.	Détail de la configuration de l'interruption d'émission	7-59
7.13.13.	Réponse à l'interruption d'émission	7-61
7.13.14.	Principe de l'interruption d'émission	7-62
7.13.14.1.	Gestion du controle de flux dans interruption d'émission	7-63
7.13.15.	Principe du traitement de l'émission (SendMessage)	7-63
7.13.15.1.	Code partiel du mécanisme d'émission	7-64
7.13.16.	Observation de l'émission	7-64
7.13.16.1.	Vue d'ensemble	7-64
7.13.16.2.	Vue de détail	7-65
7.13.16.3.	Check des 6 caractères	7-65
7.13.17.	Vérification du contrôle de flux à l'émission	7-66
7.13.17.1.	Vue sans action du contrôle flux	7-66
7.13.17.2.	Vue d'ensemble contrôle flux d'émission	7-67
7.13.17.1.	Vue de détail du contrôle flux d'émission	7-67
7.13.17.2.	Vue de la reprise de l'émission	7-68
7.14.	Structure d'un message	7-69
7.14.1.	Données binaires ou ASCII	7-69
7.14.2.	Calcul d'un CRC 16 bits	7-69

7.15. Principe du calcul d'un CRC	7-70
7.15.1. Les polynômes utilisés pour le calcul du CRC	7-70
7.15.2. Variété d'algorithmes	7-71
7.15.3. Exemple de schéma logique	7-71
7.15.4. Exemple d'algorithme	7-72
7.15.4.1. Calcul du CRC bit à bit sur 1 byte	7-72
7.15.4.2. Calcul du CRC d'un message	7-72
7.15.4.3. Exemple de résultat	7-73
7.15.5. Calcul du CRC au moyen d'une table	7-74
7.15.5.1. Formule	7-74
7.15.5.2. Table calcul du CRC	7-74
7.15.5.3. Fonction pour calcul du CRC avec la table	7-75
7.15.5.1. Calcul du CRC d'un message	7-76
7.15.5.2. Exemple de résultat	7-76
7.15.6. Calcul du CRC au moyen de deux tables	7-76
7.15.6.1. Les 2 Tables de calcul du CRC	7-77
7.15.6.2. Sous-fonctions et fonctions	7-77
7.15.6.3. Calcul du CRC d'un message	7-78
7.15.6.4. Exemple de résultat	7-78
7.15.7. Exemple d'évolution du CRC sur 2 trames	7-79
7.15.8. Vérification du CRC à la réception	7-80
7.15.8.1. Algorithme de calcul du CRC lors de l'émission	7-80
7.15.8.2. Contrôle du CRC à la réception	7-80
7.15.8.3. Vérification du résultat	7-81
7.15.9. Calcul Crc16, librairie à disposition	7-81
7.16. Exemple complet avec émission et réception	7-82
7.16.1. Structure du message de l'exemple	7-82
7.16.2. Cycles de traitements	7-82
7.16.3. Définitions et éléments globaux	7-83
7.16.4. La fonction SendMessage	7-84
7.16.5. La fonction GetMessage	7-85
7.16.6. Réaction du système à des messages corrompus	7-87
7.16.6.1. Vue générale réaction à mauvais message	7-87
7.16.6.2. Vue réaction à mauvais message	7-87
7.16.6.3. Observation attente STX	7-88
7.16.6.1. Observation attente STX détail	7-88
7.16.6.2. Conclusion sur le traitement de mauvais message	7-88
7.16.7. Conclusion sur l'exemple	7-89
7.17. Conclusion générale	7-89
7.18. Historique des Versions	7-90
7.18.1. Version 1.5 janvier 2015	7-90
7.18.2. Version 1.6 mars 2015	7-90
7.18.3. Version 1.7 mars 2016	7-90
7.18.4. Version 1.7_1 mars 2016	7-90
7.18.5. Version 1.8 février 2017	7-90
7.18.1. Version 1.9 janvier 2018	7-90
7.18.1. Version 1.91 janvier 2019	7-90
7.18.2. Version 1.92 septembre 2022	7-90

7. GESTION DE LA COMMUNICATION SÉRIELLE

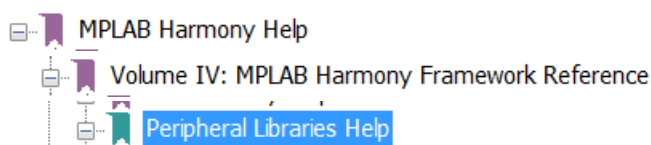
Ce chapitre traite de la communication série asynchrone en vue de réaliser une liaison RS232 avec une PIC32MX. La problématique de l'usage d'un FIFO est aussi abordée, ainsi que la sécurisation des trames par un CRC.

Les points suivants sont abordés :

- Principe de la liaison série
- La liaison RS232
- Les USART du PIC32MX
- Les fonctions pour la gestion de la communication
- Concept de FIFO
- Principe de la réception sous interruption
- Exemple d'application
- Validation des trames par CRC

Les documents de référence sont :

- La documentation "PIC32 Family Reference Manual" :
Section 21 : UART
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" :
Section 20 : Universal Asynchronous Receiver Transmitter (UART)
- La documentation d'Harmony, qui se trouve dans <Répertoire Harmony>\v<n>\doc :
Section MPLAB Harmony Framework Reference > Peripheral Libraries Help,
sous-section USART Peripheral Library



Ce document a été établi sur la base de Harmony v1.08, sauf contre-indication.

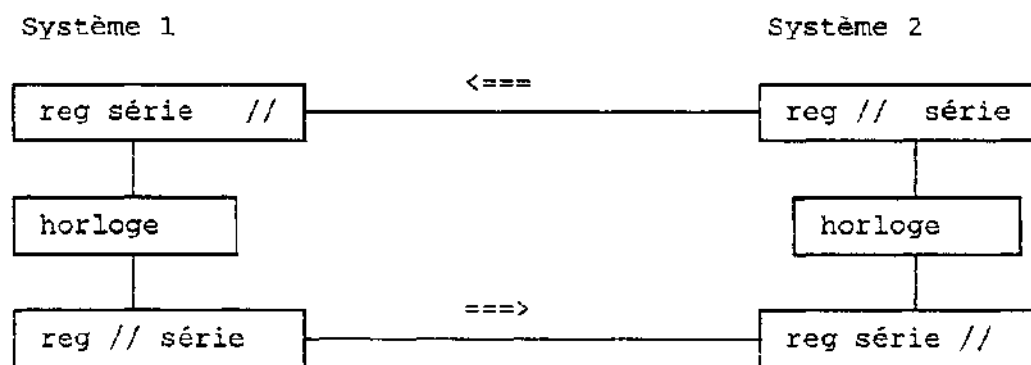
7.1. PRINCIPE DE LA LIAISON SÉRIELLE

Voici quelques rappels sur les concepts de la communication série.

7.1.1. PRINCIPE

Plutôt que de véhiculer l'information en parallèle, on transforme celle-ci en une information série à l'aide d'un registre à décalage. On effectue ainsi un multiplexage dans le temps de l'information à transmettre.

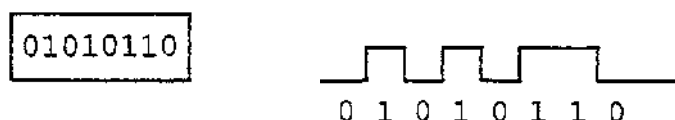
Cette méthode présente l'avantage de ne véhiculer des signaux électriques que sur une ligne par rapport aux 8 lignes d'un bus de 8 bits. Afin de pouvoir communiquer de façon bidirectionnelle entre deux systèmes, il est nécessaire d'utiliser deux fils : un fil sera utilisé pour communiquer du système 1 vers le système 2, l'autre fil sera utilisé pour communiquer du système 2 vers le système 1.



Le byte à transmettre est écrit dans le registre à décalage par le processeur.

Le registre à décalage, piloté par une horloge, sort les 8 bits du byte à transmettre les uns à la suite des autres à la fréquence d'un bit par période d'horloge.

Par exemple l'envoi de 0x56 correspond à (en commençant par le bit de poids fort) :



On constate que :

- A la réception, le registre à décalage ne sait pas quand le premier bit arrive sur la ligne.
- Si plusieurs bits identiques se suivent sur la ligne, le registre à décalage de la réception ne connaît pas ce nombre de bits identiques.
- Les bits sont émis sur la ligne à une certaine vitesse. Cette vitesse de transmission doit être identique dans les deux systèmes. Elle est définie en nombre de bits transmis par seconde, et est exprimée en bauds.

Il faut donc ajouter à ce schéma bloc un élément de synchronisation entre le récepteur et l'émetteur.

Cette synchronisation peut se faire de deux méthodes différentes.

7.1.2. COMMUNICATION SÉRIE ASYNCHRONE

Dans ce mode de transmission, on ajoute à la donnée devant être transmise une enveloppe de synchronisation, ce qui donne la structure suivante :

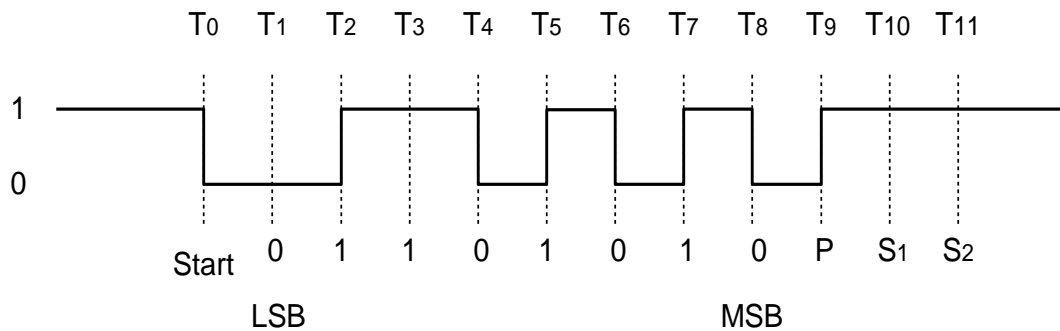
- Un bit de start de polarité inverse de l'état de repos de la ligne. Ce bit est utilisé pour synchroniser et pour indiquer au registre à décalage de réception qu'une nouvelle donnée est transmise sur la ligne.
- La donnée à transmettre, formée de 5, 6, 7 ou 8 bits.
- D'un bit de parité paire ou impaire (facultatif) utilisé pour contrôler la transmission.
- D'un ou de plusieurs bits de stop indiquant au récepteur la fin de la transmission de la donnée, ce qui permet à l'ordinateur de pouvoir lire la donnée.

7.1.2.1. EXEMPLE DE TRANSMISSION ASYNCHRONE

Paramètres de la communication :

- Vitesse 9'600 bauds.
- Nombre de bits de la donnée : 8.
- Parité : paire.
- Nombre de bits de stop : 2.

Transmission de 0x56 soit 0101'0110



- Avant le temps T_0 , la ligne est à l'état de repos (niveau 1).
- A la suite du bit de start, on a l'envoi des bits D0 à D7 (temps T_1 à T_8).
- Au temps T_9 , on trouve le bit de parité paire, calculé sur les 8 bits de la donnée.
- Au temps T_{10} , le premier bit de stop, et à T_{11} le 2ème bit de stop puis fin de transmission de la donnée.
- Après le temps T_{11} , la ligne peut soit rester au niveau de repos, soit émettre le bit de start de la prochaine donnée.

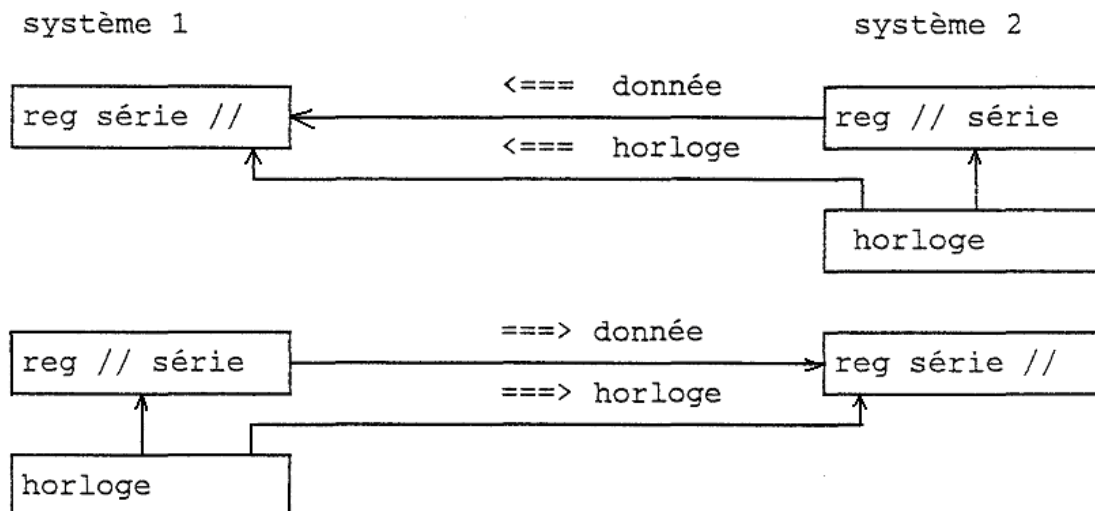
On constate que pour transmettre une donnée de 8 bits, il a fallu émettre 12 bits sur la ligne (start + 8 datas + parité + 2 stop). Le minimum est 10 bits sans parité et 1 seul stop.

7.1.2.2. FONCTIONNEMENT DE LA SYNCHRONISATION

Le récepteur possède une horloge qui pilote le registre à décalage. Cette horloge est resynchronisée par l'arrivée du bit de start. Le bit est mémorisé dans le registre à décalage à l'instant qui correspond à la moitié de la durée de ce bit.

7.1.3. COMMUNICATION SÉRIE SYNCHRONE

Dans ce mode de transmission, on ajoute une ligne supplémentaire transmettant une horloge dont la fréquence correspond à la vitesse de transmission. De ce fait, il n'est plus nécessaire de rajouter les bits de start et de stop utilisés en mode asynchrone. Le schéma bloc devient :

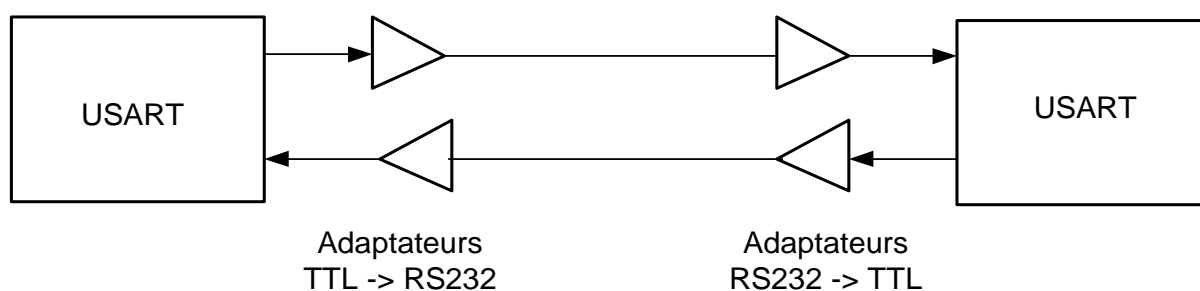


7.2. LA LIAISON RS232

Cette liaison série correspond à la norme EIA232 ou V28.

7.2.1. NORMES ELECTRIQUES DES COMMUNICATIONS SÉRIE

Le but des lignes de communication série étant de pouvoir communiquer entre des systèmes informatiques distants, il est nécessaire de travailler avec des signaux électriques différents des signaux logiques de l'informatique, qui ne se prêtent pas à des liaisons sur de grandes distances.



Remarque : Au niveau de la ligne RS232, le '0' est représenté par le +12V et le '1' par le -12V.

De plus, il est nécessaire de normaliser ces lignes électriques afin que l'on puisse y brancher des systèmes de différents fournisseurs.

Il existe des normes fixant :

- Les niveaux électriques.
- La connectique.
- Le protocole de communication.

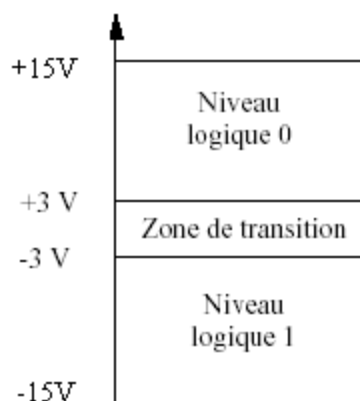
7.2.1.1. NORMES EIA (ELECTRONIC INDUSTRIES ASSOCIATION)

Les normes EIA définissent notamment les normes suivantes dont voici quelques caractéristiques :

Norme	RS232	RS423	RS422	RS485
Mode d'opération	Simple	Simple	Différentiel	Différentiel
Nombre d'émetteurs et de récepteurs	1 émetteur 1 récepteur	1 émetteur 10 récepteurs	1 émetteur 10 récepteurs	32 émetteurs 32 récepteurs
Longueur max (m)	15	1'200	1'200	1'200
Vitesse max (Bits/sec)	20k	100k	10M	10M
Signal de sortie min	$\pm 5V$	± 3	$\pm 2V$	$\pm 1,5V$
Signal de sortie max	$\pm 15V$	$\pm 6V$	$\pm 6V$	$\pm 6V$

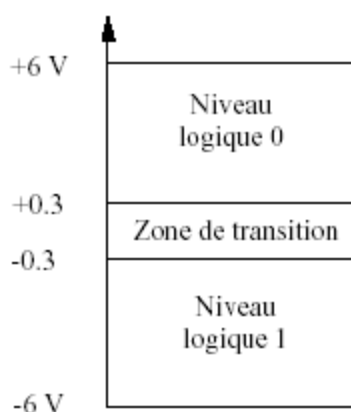
7.2.1.2. NIVEAU LOGIQUE NORME V28

La norme V28 correspond à la norme RS232 :



7.2.1.3. NIVEAU LOGIQUE NORME V11

La norme V11 correspond aux normes RS422 et RS485 :



7.2.2. LES SIGNAUX ET LA CONNECTIQUE

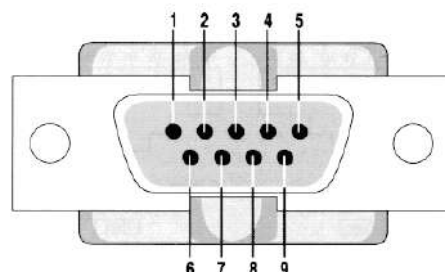
7.2.2.1. CONNECTEUR 9 PÔLES DU PC AT

Voici la représentation du connecteur série 9 pôles (port COM pour les OS Microsoft) comme on peut la voir sur un PC. Il s'agit d'un connecteur mâle de type SUB-D.

Connecteur SUB-D 9 pôles mâle



Numérotation (vue de devant)



7.2.2.2. RÔLE DES SIGNAUX

Broche	Signal	Description	E/S
1	DCD	Détection de porteuse	Entrée
2	RX	Réception de données	Entrée
3	TX	Emission de données	Sortie
4	DTR	Terminal de données prêt	Sortie
5	GND	Masse de signal	-
6	DSR	Données prêtes	Entrée
7	RTS	Requête d'émission	Sortie
8	CTS	Prêt pour l'émission	Entrée
9	RI	Indicateur d'appel	Entrée

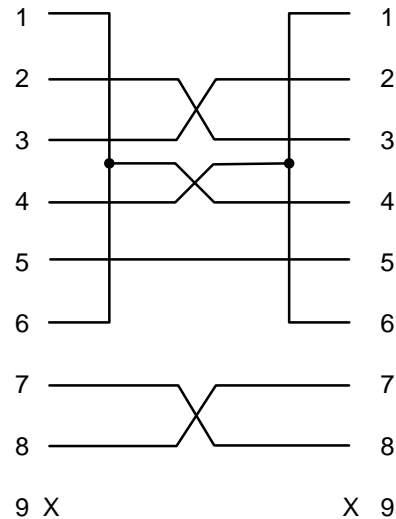
- **DCD** (Data Carrier Detect) : cette ligne est une entrée (active = logic '0', tension positive). Elle signale à l'ordinateur qu'une liaison a été établie avec un correspondant.
- **RX** (Receive Data) : cette ligne est une entrée. C'est ici que transitent les informations du correspondant vers l'ordinateur.
- **TX** (Transmit Data) : cette ligne est une sortie. Les données de l'ordinateur vers le correspondant sont véhiculées par son intermédiaire.
- **DTR** (Data Terminal Ready) : cette ligne est une sortie (active = logic '0', tension positive). Elle permet à l'ordinateur de signaler au correspondant que le port série a été libéré et qu'il peut être utilisé s'il le souhaite.
- **GND** (GrouND) : c'est la masse.
- **DSR** (Data Set Ready) : cette ligne est une entrée (active = logic '0', tension positive). Elle permet au correspondant de signaler qu'une donnée est prête.
- **RTS** (Request To Send) : cette ligne est une sortie (active = logic '0', tension positive). Elle indique au correspondant que l'ordinateur veut lui transmettre des données.
- **CTS** (Clear To Send) : cette ligne est une entrée (active = logic '0', tension positive). Elle indique à l'ordinateur que le correspondant est prêt à recevoir des données.

- **RI** (Ring Indicator) : cette ligne est une entrée (active = logic '0', tension positive). Elle permet à l'ordinateur de savoir qu'un correspondant veut initier une communication avec lui.

7.2.3. CÂBLE NULL MODEM

Lorsque l'on souhaite relier deux processeurs pour transmettre des données de l'un à l'autre, il est nécessaire d'utiliser un câble effectuant un certain nombre de croisements des lignes pour permettre le fonctionnement. Un type de câble communément utilisé est le câble dit "null modem" :

Câble destiné à relier directement 2 PC IBM AT par l'interface série, le câble agissant comme modem zéro

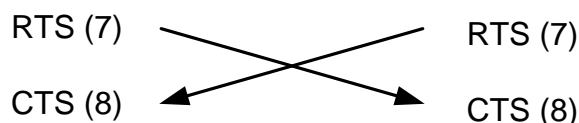


Il existe différents types de câbles croisés, tous permettant le transfert de données simple (lignes RX, TX et GND) sans contrôle de flux matériel. Cependant, plusieurs principes de contrôle de flux hardware existent, d'où des différences de câblages entre les lignes restantes. Le câble null modem ci-dessus est un type de câble simple et répandu.

7.2.4. CONTRÔLE DE FLUX HARDWARE

Une partie des signaux servent au contrôle du flux de la communication.

Un des mécanismes de base est le couple des signaux RTS et CTS :



Dans ce type de contrôle de flux, chaque correspondant à sa sortie RTS câblée sur l'entrée CTS de l'autre et peut ainsi signaler s'il est prêt à recevoir des données.

En réception, si le système n'est plus capable de recevoir des données (tampon plein) alors la ligne RTS sera mise au niveau logique 1 (tension négative) pour demander à l'émetteur de stopper l'émission. Dès que le tampon a de nouveau une place suffisante, la ligne RTS est remise au niveau logique 0.

En émission, il est nécessaire de tester le niveau du CTS. Si CTS = 0 il est possible d'émettre; si CTS = 1, il faut stopper l'émission.

7.2.5. CONTRÔLE DE FLUX SOFTWARE

Deux caractères ASCII particuliers XON et XOFF (respectivement caractères ASCII n°17 et 19) sont utilisés pour un contrôle de flux par logiciel.

Lorsqu'un système reçoit le caractère XOFF il doit cesser d'émettre jusqu'à la réception du caractère XON.

C'est en général lorsque le tampon de réception est plein à 75% que le système envoie le caractère XOFF. Lorsque le tampon de réception n'est plus utilisé qu'à 25% alors le caractère XON est envoyé.

7.2.6. ABSENCE DE CONTRÔLE DE FLUX

Lorsqu'un système (par exemple une carte à microcontrôleur) n'est pas équipé des lignes pour le contrôle de flux hardware, il est important sur l'autre système, par exemple un PC, de configurer la communication sans gestion du contrôle de flux hardware. Ceci évite d'avoir un blocage de la communication.

7.3. LES USART DU PIC32MX

Les PIC32MX795F512L possèdent 6 USART. Ils peuvent être configurés pour une transmission asynchrone Full-Duplex (bidirectionnelle) ou pour une transmission synchrone en master ou en slave. En plus il est possible d'obtenir une gestion par le processeur des lignes CTS et RTS.

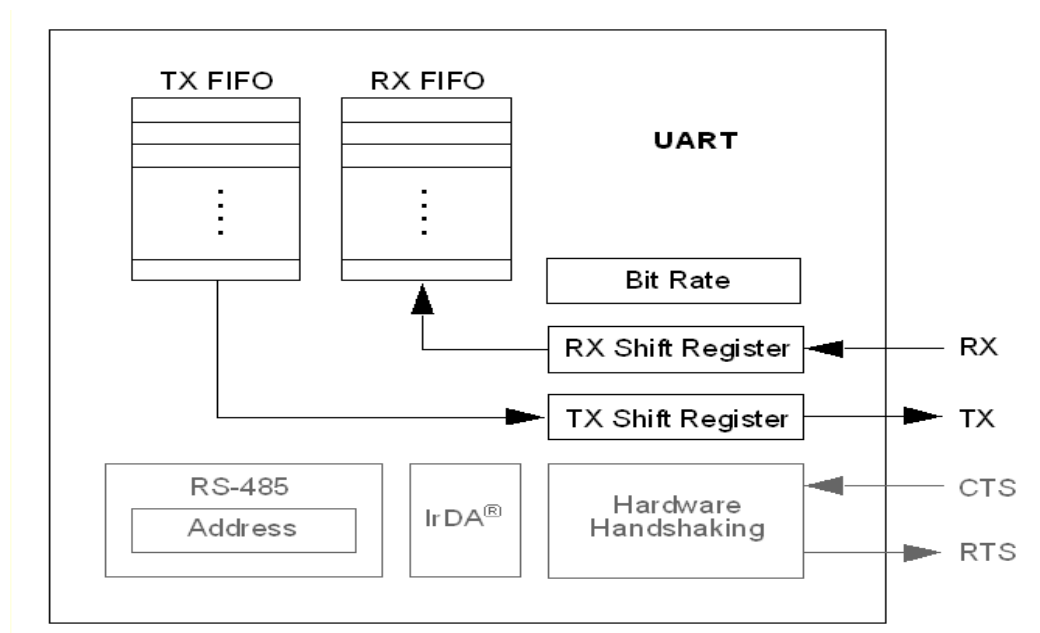
👉 Seule la transmission asynchrone nous intéresse dans le cadre de ce chapitre.

7.3.1. LISTE DES UART À DISPOSITION (PIC32MX795F512L)

On dispose de 6 UARTs. Les UARTS 1, 2 et 3 disposent de CTS et RTS.

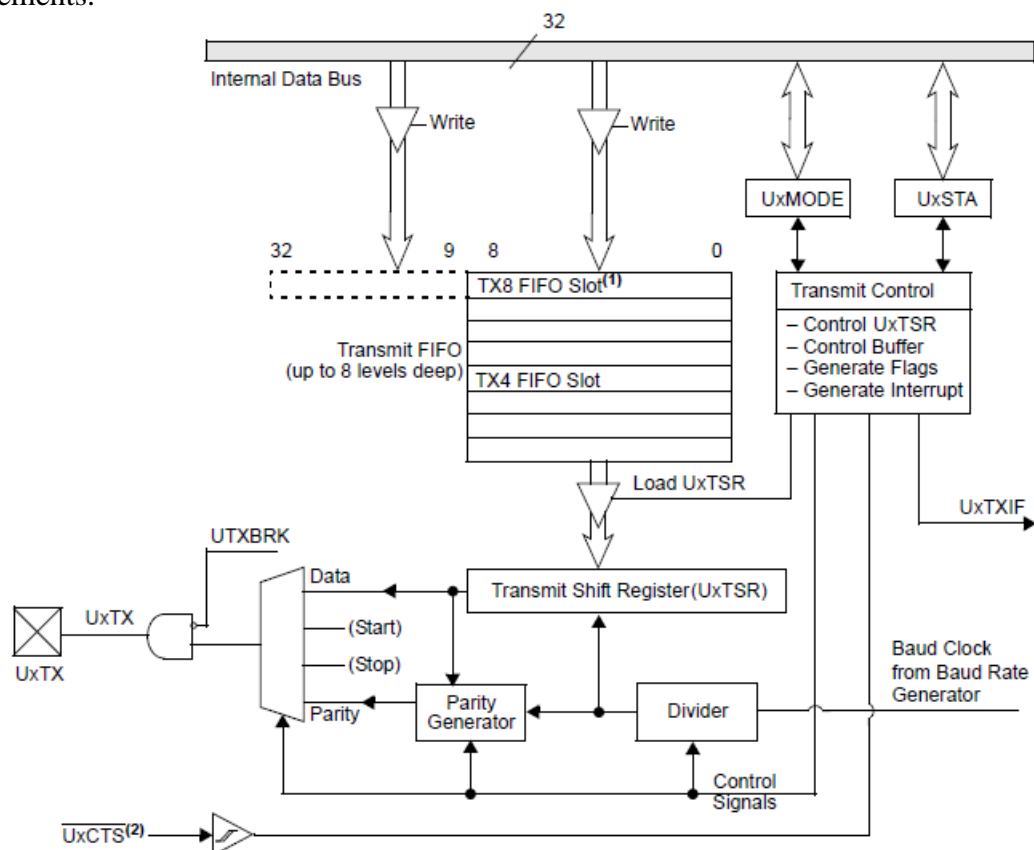
Pin Name	Pin Number ⁽¹⁾			Pin Type	Buffer Type	Description
	64-Pin QFN/TQFP	100-Pin TQFP	121-Pin XBGA			
U1CTS	43	47	L9	I	ST	UART1 clear to send
U1RTS	49	48	K9	O	—	UART1 ready to send
U1RX	50	52	K11	I	ST	UART1 receive
U1TX	51	53	J10	O	—	UART1 transmit
U3CTS	8	14	F3	I	ST	UART3 clear to send
U3RTS	4	10	E3	O	—	UART3 ready to send
U3RX	5	11	F4	I	ST	UART3 receive
U3TX	6	12	F2	O	—	UART3 transmit
U2CTS	21	40	K6	I	ST	UART2 clear to send
U2RTS	29	39	L6	O	—	UART2 ready to send
U2RX	31	49	L10	I	ST	UART2 receive
U2TX	32	50	L11	O	—	UART2 transmit
U4RX	43	47	L9	I	ST	UART4 receive
U4TX	49	48	K9	O	—	UART4 transmit
U6RX	8	14	F3	I	ST	UART6 receive
U6TX	4	10	E3	O	—	UART6 transmit
U5RX	21	40	K6	I	ST	UART5 receive
U5TX	29	39	L6	O	—	UART5 transmit

7.3.2. SCHÉMA BLOC DE PRINCIPE DE L'UART



7.3.3. SCHÉMA DÉTAILLÉ DE LA TRANSMISSION

Voici le schéma détaillé de la transmission. Le PIC32MX795F512L possède un FIFO de 8 éléments.



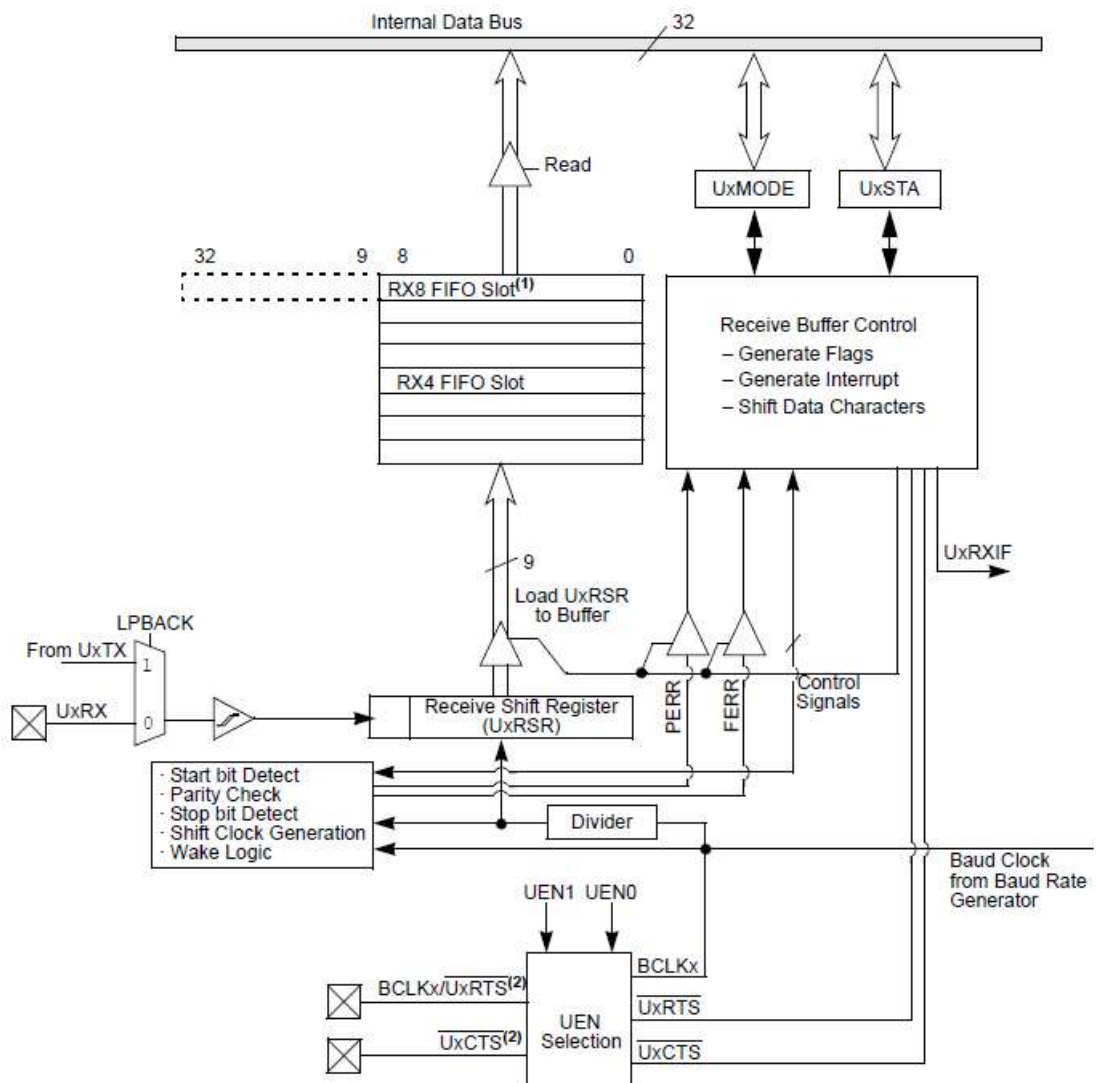
Note 1: Refer to the specific device data sheet for availability of 8-level-deep FIFO.

Note 2: Refer to the specific device data sheet for availability of UxCTS pin.

On constate la présence de la broche `_UxCTS` qui permet l'autorisation/blocage de l'émission par hardware.

7.3.4. SCHÉMA DÉTAILLÉ DE LA RÉCEPTION

Voici le schéma détaillé de la réception. Le PIC32MX795F512L possède un FIFO de 8 éléments.



Note 1: Refer to the specific device data sheet for availability of 8-level-deep FIFO.

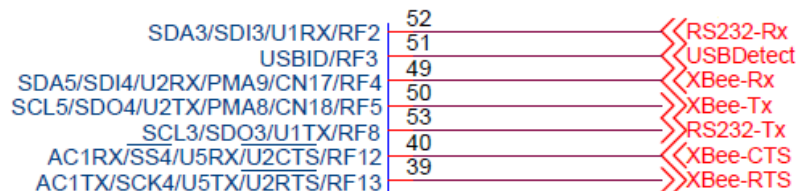
2: Refer to the specific device data sheet for availability of $\overline{\text{UxRTS}}$ and $\overline{\text{UxCTS}}$ pins.

On constate la présence des broches _UxRTS et _UxCTS qui permettent le handshaking hardware.

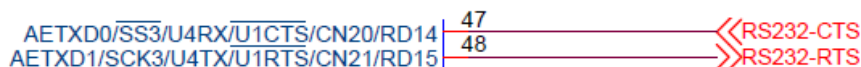
7.3.5. RÉALISATION DE LA LIAISON RS232 SUR LE KIT

7.3.5.1. CÂBLAGE SUR LE PIC32MX795F512L

Comme on peut l'observer sur l'extrait du câblage du PIC32MX795F512L, c'est l'UART 1 qui est utilisé.

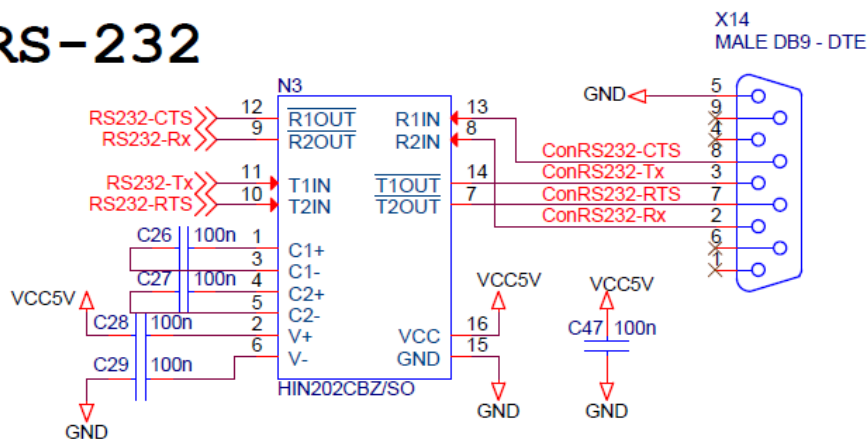


Les signaux CTS et RTS sont câblés sur les broches permettant un traitement automatique (par le PIC) du handshaking.



7.3.5.2. ADAPTATION POUR LES SIGNAUX RS232

RS-232



7.3.6. LES REGISTRES DES USART

Voici la vue d'ensemble des registres SFR en relation avec les USART :

Table 21-1: UART SFRs Summary

Name	Bit	Bit	Bit	Bit	Bit	Bit	Bit	Bit
	31/23/15/7	30/22/14/6	29/21/13/5	28/20/12/4	27/19/11/3	26/18/10/2	25/17/9/1	24/16/8/0
UxMODE ^(1,2,3)	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	ON	FRZ	SIDL	IREN	RTSMD ⁽⁴⁾	UEN<1:0> ⁽⁴⁾	
	7:0	WAKE	LPBACK	ABAUD	RXINV	BRGH	PDSEL<1:0>	STSEL
UxSTA ^(1,2,3)	31:24	—	—	—	—	—	—	ADM_EN
	23:16	ADDR<7:0>						
	15:8	UTXISEL<1:0>		UTXINV	URXEN	UTXBRK	UTXEN	UTXBF
	7:0	URXISEL<1:0>		ADDEN	RIDLE	PERR	FERR	OERR
UxTXREG	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	—	—	—	—	—	—	UTX<8>
	7:0	UTX<7:0>						
UxRXREG	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	—	—	—	—	—	—	RX<8>
	7:0	RX<7:0>						
UxBRG ^(1,2,3)	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	BRG<15:8>						
	7:0	BRG<7:0>						

7.3.6.1. LE REGISTRE UxMODE (UARTx MODE REGISTER)

Ces registres permettent la configuration des USART.

Register 21-1: UxMODE: UARTx Mode Register^(1,2,3)

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31				bit 24			

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23				bit 16			

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ON	FRZ	SIDL	IREN	RTSMD ⁽⁴⁾	—	UEN<1:0> ⁽⁴⁾	
bit 15				bit 8			

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WAKE	LPBACK	ABAUD	RXINV	BRGH	PDSEL<1:0>		STSEL
bit 7				bit 0			

Legend:

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

7.3.6.1.1. UxMode, rôle des bits

bit 31-16	Unimplemented: Read as '0'
bit 15	ON: UARTx Enable bit 1 = UARTx is enabled. UARTx pins are controlled by UARTx as defined by UEN<1:0> and UTXEN control bits 0 = UARTx is disabled. All UARTx pins are controlled by corresponding bits in the PORTx, TRISx and LATx registers; UARTx power consumption is minimal Note: When using 1:1 PBCLK divisor, the user software should not read/write the peripheral SFRs in the SYSCLK cycle immediately following the instruction that clears the module's ON bit.
bit 14	FRZ: Freeze in Debug Exception Mode bit 1 = Freeze operation when CPU is in Debug Exception mode 0 = Continue operation when CPU is in Debug Exception mode Note: FRZ is writable in Debug Exception mode only, it is forced to '0' in Normal mode.
bit 13	SIDL: Stop in Idle Mode bit 1 = Discontinue operation when device enters Idle mode 0 = Continue operation in Idle mode
Register 21-1: UxMODE: UARTx Mode Register^(1,2,3) (Continued)	
bit 12	IREN: IrDA Encoder and Decoder Enable bit 1 = IrDA is enabled 0 = IrDA is disabled
bit 11	RTSMD: Mode Selection for <u>UxRTS</u> Pin bit ⁽⁴⁾ 1 = <u>UxRTS</u> pin is in Simplex mode 0 = <u>UxRTS</u> pin is in Flow Control mode
bit 10	Unimplemented: Read as '0'
bit 9-8	UEN<1:0>: UARTx Enable bits ⁽⁴⁾ 11 = UxTX, UxRX and UxBCLK pins are enabled and used; <u>UxCTS</u> pin is controlled by corresponding bits in the PORTx register 10 = UxTX, UxRX, <u>UxCTS</u> and <u>UxRTS</u> pins are enabled and used 01 = UxTX, UxRX and <u>UxRTS</u> pins are enabled and used; <u>UxCTS</u> pin is controlled by corresponding bits in the PORTx register 00 = UxTX and UxRX pins are enabled and used; <u>UxCTS</u> and <u>UxRTS</u> /UxBCLK pins are controlled by corresponding bits in the PORTx register
bit 7	WAKE: Enable Wake-up on Start bit Detect During Sleep Mode bit 1 = Wake-up enabled 0 = Wake-up disabled
bit 6	LPBACK: UARTx Loopback Mode Select bit 1 = Loopback mode is enabled 0 = Loopback mode is disabled
bit 5	ABAUD: Auto-Baud Enable bit 1 = Enable baud rate measurement on the next character – requires reception of Sync character (0x55); cleared by hardware upon completion 0 = Baud rate measurement disabled or completed
bit 4	RXINV: Receive Polarity Inversion bit 1 = UxRX Idle state is '0' 0 = UxRX Idle state is '1'
bit 3	BRGH: High Baud Rate Enable bit 1 = High-Speed mode – 4x baud clock enabled 0 = Standard Speed mode – 16x baud clock enabled
bit 2-1	PDSEL<1:0>: Parity and Data Selection bits 11 = 9-bit data, no parity 10 = 8-bit data, odd parity 01 = 8-bit data, even parity 00 = 8-bit data, no parity
bit 0	STSEL: Stop Selection bit 1 = 2 Stop bits 0 = 1 Stop bit

7.3.6.2. LE REGISTRE UxSTA (UARTx STATUS AND CONTROL REGISTER)

Ces registres permettent la configuration des USART ainsi que l'obtention d'informations sur la situation.

Register 21-2: UxSTA: UARTx Status and Control Register^(1,2,3)

U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
—	—	—	—	—	—	—	ADM_EN
bit 31							bit 24
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADDR<7:0>							
bit 23							bit 16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-1
UTXISEL<1:0> ⁽⁴⁾		UTXINV	URXEN	UTXBRK	UTXEN	UTXBF	TRMT
bit 15							bit 8
R/W-0	R/W-0	R/W-0	R-1	R-0	R-0	R/W-0	R-0
URXISEL<1:0> ⁽⁴⁾		ADDEN	RIDLE	PERR	FERR	OERR	URXDA
bit 7							bit 0

Legend:

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

7.3.6.2.1. UxSTA, rôle des bits

bit 31-25

Unimplemented: Read as '0'

bit 24

ADM_EN: Automatic Address Detect Mode Enable bit

1 = Automatic Address Detect mode is enabled

0 = Automatic Address Detect mode is disabled

bit 23-16

ADDR<7:0>: Automatic Address Mask bits

When the ADM_EN bit is '1', this value defines the address character to use for automatic address detection.

bit 15-14

UTXISEL<1:0>: TX Interrupt Mode Selection bits⁽⁴⁾

For 4-level deep FIFO UART modules:

11 = Reserved, do not use

10 = Interrupt is generated when the transmit buffer becomes empty

01 = Interrupt is generated when all characters have been transmitted

00 = Interrupt is generated when the transmit buffer contains at least one empty space

For 8-level deep FIFO UART modules:

11 = Reserved, do not use

10 = Interrupt is generated and asserted while the transmit buffer is empty

01 = Interrupt is generated and asserted when all characters have been transmitted

00 = Interrupt is generated and asserted while the transmit buffer contains at least one empty space

Register 21-2: UxSTA: UARTx Status and Control Register^(1,2,3) (Continued)

bit 13

UTXINV: Transmit Polarity Inversion bit

If IrDA mode is disabled (i.e., IREN (UxMODE<12>) is '0'):

1 = UxTX Idle state is '0'

0 = UxTX Idle state is '1'

If IrDA mode is enabled (i.e., IREN (UxMODE<12>) is '1'):

1 = IrDA encoded UxTX Idle state is '1'

0 = IrDA encoded UxTX Idle state is '0'

bit 12

URXEN: Receiver Enable bit

1 = UARTx receiver is enabled. UxRX pin is controlled by UARTx (if ON = 1)

0 = UARTx receiver is disabled. UxRX pin is ignored by the UARTx module. UxRX pin is controlled by port.

bit 11	UTXBRK: Transmit Break bit 1 = Send Break on next transmission. Start bit followed by twelve '0' bits, followed by Stop bit; cleared by hardware upon completion 0 = Break transmission is disabled or completed
bit 10	UTXEN: Transmit Enable bit 1 = UARTx transmitter is enabled. UxTX pin is controlled by UARTx (if ON = 1) 0 = UARTx transmitter is disabled. Any pending transmission is aborted and buffer is reset. UxTX pin is controlled by port.
bit 9	UTXBF: Transmit Buffer Full Status bit (read-only) 1 = Transmit buffer is full 0 = Transmit buffer is not full, at least one more character can be written
bit 8	TRMT: Transmit Shift Register is Empty bit (read-only) 1 = Transmit shift register is empty and transmit buffer is empty (the last transmission has completed) 0 = Transmit shift register is not empty, a transmission is in progress or queued in the transmit buffer
bit 7-6	URXISEL<1:0>: Receive Interrupt Mode Selection bit ⁽⁴⁾ For 4-level deep FIFO UART modules: 11 = Interrupt flag bit is set when receive buffer becomes full (i.e., has 4 data characters) 10 = Interrupt flag bit is set when receive buffer becomes 3/4 full (i.e., has 3 data characters) 0x = Interrupt flag bit is set when a character is received For 8-level deep FIFO UART modules: 11 = Reserved; do not use 10 = Interrupt flag bit is asserted while receive buffer is 3/4 or more full (i.e., has 6 or more data characters) 01 = Interrupt flag bit is asserted while receive buffer is 1/2 or more full (i.e., has 4 or more data characters) 00 = Interrupt flag bit is asserted while receive buffer is not empty (i.e., has at least 1 data character)
bit 5	ADDEN: Address Character Detect bit (bit 8 of received data = 1) 1 = Address Detect mode is enabled. If 9-bit mode is not selected, this control bit has no effect. 0 = Address Detect mode is disabled
Register 21-2: UxSTA: UARTx Status and Control Register^(1,2,3) (Continued)	
bit 4	RIDLE: Receiver Idle bit (read-only) 1 = Receiver is Idle 0 = Data is being received
bit 3	PERR: Parity Error Status bit (read-only) 1 = Parity error has been detected for the current character 0 = Parity error has not been detected
bit 2	FERR: Framing Error Status bit (read-only) 1 = Framing error has been detected for the current character 0 = Framing error has not been detected
bit 1	OERR: Receive Buffer Overrun Error Status bit. This bit is set in hardware and can only be cleared (= 0) in software. Clearing a previously set OERR bit resets the receiver buffer and RSR to empty state. 1 = Receive buffer has overflowed 0 = Receive buffer has not overflowed
bit 0	URXDA: Receive Buffer Data Available bit (read-only) 1 = Receive buffer has data, at least one more character can be read 0 = Receive buffer is empty

7.3.6.3. LE REGISTRE UxTXREG (UARTx TRANSMIT REGISTER)

Ce registre permet la transmission. A noter les 8 bits utiles sur les 32.

Register 21-3: UxTXREG: UARTx Transmit Register

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31				bit 24			
U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23				bit 16			
U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
—	—	—	—	—	—	—	TX<8>
bit 15				bit 8			
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
TX<7:0>							
bit 7				bit 0			

Legend:

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

bit 31-9 **Unimplemented:** Read as '0'

bit 8-0 **TX<8:0>:** Data bits 8-0 of the character to be transmitted

7.3.6.4. LE REGISTRE UxRXREG (UARTx RECEIVE REGISTER)

Ce registre permet la réception. A noter les 8 bits utiles sur les 32.

Register 21-4: UxRXREG: UARTx Receive Register

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31				bit 24			
U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23				bit 16			
U-0	U-0	U-0	U-0	U-0	U-0	U-0	R-0
—	—	—	—	—	—	—	RX<8>
bit 15				bit 8			
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
RX<7:0>							
bit 7				bit 0			

Legend:

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

bit 31-9 **Unimplemented:** Read as '0'

bit 8-0 **RX<8:0>:** Data bits 8-0 of the received character

7.3.6.5. LE REGISTRE UxBRG (UARTx BAUDRATE REGISTER)

Ce registre permet la configuration du générateur de baudrate. Plus précisément il s'agit d'établir la valeur du diviseur du PB_CLOCK pour obtenir le baudrate voulu.

Register 21-5: UxBRG: UARTx Baud Rate Register^(1,2,3)

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31				bit 24			

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23				bit 16			

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BRG<15:8>							
bit 15				bit 8			

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BRG<7:0>							
bit 7				bit 0			

Legend:

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

bit 31-16

Unimplemented: Read as '0'

bit 15-0

BRG<15:0>: Baud Rate Divisor bits

7.3.7. LE GÉNÉRATEUR DE BAUDRATE

Chaque USART possède un générateur de baudrate indépendant qui lui est associé. Il s'agit d'établir la valeur de UxBRG selon la formule ci-dessous.

Equation 21-1: UART Baud Rate with BRGH = 0

$$\text{Baud Rate} = \frac{F_{PB}}{16 \cdot (UxBRG + 1)}$$

$$UxBRG = \frac{F_{PB}}{16 \cdot \text{Baud Rate}} - 1$$

Note: F_{PB} denotes the PBCLK frequency.

En fonction du bit 3 du registre UxMODE on dispose d'une pré-division par 4 ou 16.

bit 3

BRGH: High Baud Rate Enable bit

1 = High-Speed mode – 4x baud clock enabled

0 = Standard Speed mode – 16x baud clock enabled

Si BRGH=1, l'équation ci-dessus est modifiée avec un facteur 4 au lieu de 16. Cela résulte en une meilleure finesse de réglage pour les baudrates élevés.

7.3.7.1. EXEMPLE CALCUL DU BAUDRATE

Avec BRGH = 0 :

$$\text{Baudrate} = \text{PB_FREQ} / (16 * (X+1))$$

où X représente UxBRG.

$$\text{D'où : } X = ((\text{PB_FREQ} / (16 * \text{Baudrate})) - 1)$$

Par exemple pour 57'600 bauds et PB_FREQ = 80 MHz

$$X = ((80'000'000 / (57'600 * 16)) - 1) = 86.805 - 1 = 85.8 \rightarrow \text{choix } 86$$

Il est nécessaire de calculer l'erreur :

$$\text{Baudrate réel} = \text{PB_FREQ} / (16(X+1)) = 80'000'000 / (16(86+1)) = 80'000'000 / 1'392 = 57'471.2$$

L'erreur est de $(57'471.2 - 57'600) / 57'600 = -0.0022$ soit - 0.22 %

L'erreur ne doit pas dépasser 2%, sinon il y a des risques d'erreur de lecture. On peut améliorer la précision pour les baudrate élevés en utilisant le pré-diviseur par 4.

7.3.7.1.1. Tableau des écarts pour 40 MHz

Le fabricant fournit un tableau des écarts pour PB_FREQ = 40 MHz et BRGH = 0

Target Baud Rate	Peripheral Bus Clock: 40 MHz		
	Actual Baud Rate	% Error	BRG Value (decimal)
110	110.0	0.00	22726.0
300	300.0	0.00	8332.0
1200	1200.2	0.02	2082.0
2400	2399.2	-0.03	1041.0
9600	9615.4	0.16	259.0
19.2 K	19230.8	0.16	129.0
38.4 K	38461.5	0.16	64.0
56 K	55555.6	-0.79	44.0
115 K	113636.4	-1.19	21.0
250 K	250000.0	0.00	9.0
300 K			
500 K	500000.0	0.00	4.0
Min. Rate	38.1	0.0	65535
Max. Rate	2500000	0.0	0

7.4. CONFIGURATION DE L'USART AVEC LA PLIB_USART

La configuration de l'USART est réalisée par les étapes suivantes :

- Sélection du baudrate (**BaudRateSet**)
- Sélection du mode de handshake (**HandshakeModeSelect**)
- Sélection du mode d'opération (**OperationModeSelect**)
- Activation du transmetteur (**TransmitterEnable**)
- Sélection du mode d'interruption du transmetteur (**TransmitterInterruptModeSelect**)
- Activation du récepteur (**ReceiverEnable**)
- Sélection du mode d'interruption du récepteur (**ReceiverInterruptModeSelect**)

7.4.1. LE TYPE ÉNUMÉRÉ USART_MODULE_ID

Le type énuméré USART_MODULE_ID liste les USART à disposition.

```
typedef enum {  
    USART_ID_1 = 0,  
    USART_ID_3,  
    USART_ID_2,  
    USART_ID_4,  
    USART_ID_6,  
    USART_ID_5,  
    USART_NUMBER_OF_MODULES  
} USART_MODULE_ID;
```

7.4.2. LA FONCTION PLIB_USART_BAUDRATESET

La fonction **PLIB_USART_BaudRateSet**, permet d'établir le baudrate.

```
void PLIB_USART_BaudRateSet(USART_MODULE_ID index, uint32_t clockFrequency, uint32_t baudRate);
```

Exemple :

```
PLIB_USART_BaudRateSet(USART_ID_1,  
                        SYS_DEVCON_SYSTEM_CLOCK, 57600);
```

7.4.3. LA FONCTION PLIB_USART_HANDSHAKEMODESELECT

La fonction **PLIB_USART_HandshakeModeSelect**, permet d'établir comment est géré le handshaking.

```
void PLIB_USART_HandshakeModeSelect(USART_MODULE_ID index, USART_HANDSHAKE_MODE handshakeConfig);
```

Pour le handshakeConfig, on dispose de 2 possibilités :

Members	Description
USART_HANDSHAKE_MODE_FLOW_CONTROL	Enables flow control
USART_HANDSHAKE_MODE_SIMPLEX	Enables simplex mode communication, no flow control

Le mode FLOW_CONTROL établit le mode de gestion par le hardware. Le mode SIMPLEX permet une gestion par logiciel.

Exemple :

```
PLIB_USART_HandshakeModeSelect(USART_ID_1,
                                USART_HANDSHAKE_MODE_SIMPLEX);
```

7.4.4. LA FONCTION **PLIB_USART_OPERATIONMODESELECT**

La fonction **PLIB_USART_OperationModeSelect**, permet d'établir comment sont gérées les broches TX, RX, CTS et RTS.

```
void PLIB_USART_OperationModeSelect(USART_MODULE_ID index, USART_OPERATION_MODE operationmode);
```

Pour l'operation mode, on dispose de 4 possibilités :

Members	Description
USART_ENABLE_TX_RX_BCLK_USED	TX, RX and BCLK pins are used by USART module
USART_ENABLE_TX_RX_CTS_RTS_USED	TX, RX, CTS and RTS pins are used by USART module
USART_ENABLE_TX_RX_RTS_USED	TX, RX and RTS pins are used by USART module
USART_ENABLE_TX_RX_USED	TX and RX pins are used by USART module

Pour être cohérent avec le mode SIMPLEX, il faut utiliser le dernier élément pour utiliser uniquement les broches TX et RX de l'USART.

Exemple :

```
PLIB_USART_OperationModeSelect(USART_ID_1,
                                USART_ENABLE_TX_RX_USED);
```

7.4.5. LA FONCTION **PLIB_USART_LINECONTROLMODESELECT**

La fonction **PLIB_USART_LineControlModeSelect**, permet d'établir comment sont traitées les données.

```
void PLIB_USART_LineControlModeSelect(USART_MODULE_ID index, USART_LINECONTROL_MODE dataFlowConfig);
```

On dispose des possibilités suivantes pour le dataFlowConfig :

Members	Description
USART_8N1	8 Data Bits, No Parity, one Stop Bit
USART_8E1	8 Data Bits, Even Parity, 1 stop bit
USART_8O1	8 Data Bits, odd Parity, 1 stop bit
USART_8N2	8 Data Bits, No Parity, two Stop Bits
USART_8E2	8 Data Bits, Even Parity, 2 stop bits
USART_8O2	8 Data Bits, odd Parity, 2 stop bits
USART_9N1	9 Data Bits, No Parity, 1 stop bit
USART_9N2	9 Data Bits, No Parity, 2 stop bits

7.4.5.1. LE TYPE ÉNUMÉRÉ USART_LINECONTROL_MODE

Le type énuméré USART_LINECONTROL_MODE est défini ainsi:

```
typedef enum {
    USART_8N1 = 0x00,
    USART_8E1 = 0x01,
    USART_8O1 = 0x02,
    USART_9N1 = 0x03,
    USART_8N2 = 0x04,
    USART_8E2 = 0x05,
    USART_8O2 = 0x06,
    USART_9N2 = 0x07
} USART_LINECONTROL_MODE;
```

7.4.5.2. USART_LINECONTROL_MODE, EXEMPLE

Etablit le classique data bits = 8, parity = none et stop bits = 1

```
PLIB_USART_LineControlModeSelect(USART_ID_1,
    USART_8N1);
```

7.4.6. LA FONCTION PLIB_USART_TRANSMITTERENABLE

La fonction **PLIB_USART_TransmitterEnable**, permet d'activer le transmetteur.

Exemple :

```
PLIB_USART_TransmitterEnable(USART_ID_1);
```

7.4.7. PLIB_USART_TRANSMITTERINTERRUPTMODESELECT

La fonction **PLIB_USART_TransmitterInterruptModeSelect**, permet d'établir comment doit réagir l'interruption de transmission.

```
void PLIB_USART_TransmitterInterruptModeSelect(USART_MODULE_ID index, USART_TRANSMIT_INTR_MODE
    fifolevel);
```

Les choix sont les suivants :

Members	Description
USART_TRANSMIT_FIFO_EMPTY	Interrupt when the transmit buffer becomes empty
USART_TRANSMIT_FIFO_IDLE	Interrupt when all characters are transmitted
USART_TRANSMIT_FIFO_NOT_FULL	Interrupt when at least one location is empty in the transmit buffer

7.4.7.1. LE TYPE ÉNUMÉRÉ USART_TRANSMIT_INTR_MODE

Le type énuméré USART_TRANSMIT_INTR_MODE est défini ainsi :

```
typedef enum {
    USART_TRANSMIT_FIFO_NOT_FULL = 0x00,
    USART_TRANSMIT_FIFO_IDLE = 0x01,
    USART_TRANSMIT_FIFO_EMPTY = 0x02
} USART_TRANSMIT_INTR_MODE;
```

7.4.7.2. PLIB_USART_TRANSMITTERINTERRUPTMODESELECT, EXEMPLE

Dans cet exemple, l'interruption de transmission est configurée pour se produire lorsque le tampon de transmission est vide.

```
PLIB_USART_TransmitterInterruptModeSelect(USART_ID_1,
                                           USART_TRANSMIT_FIFO_EMPTY);
```

7.4.8. LA FONCTION PLIB_USART_RECEIVERENABLE

La fonction **PLIB_USART_ReceiverEnable**, permet d'activer le récepteur.

Exemple :

```
PLIB_USART_ReceiverEnable(USART_ID_1);
```

7.4.9. PLIB_USART_RECEIVERINTERRUPTMODESELECT

La fonction **PLIB_USART_ReceiverInterruptModeSelect**, permet d'établir comment doit réagir l'interruption de réception.

```
void PLIB_USART_ReceiverInterruptModeSelect(USART_MODULE_ID index, USART_RECEIVE_INTR_MODE
interruptMode);
```

Les choix sont les suivants :

Members	Description
USART_RECEIVE_FIFO_HALF_FULL	Interrupt when receive buffer is half full
USART_RECEIVE_FIFO_3B4FULL	Interrupt when receive buffer is 3/4 full
USART_RECEIVE_FIFO_ONE_CHAR	Interrupt when a character is received

7.4.9.1. LE TYPE ÉNUMÉRÉ USART_RECEIVE_INTR_MODE

Le type énuméré USART_TRANSMIT_INTR_MODE est défini ainsi :

```
typedef enum {
    USART_RECEIVE_FIFO_3B4FULL = 0x02,
    USART_RECEIVE_FIFO_HALF_FULL = 0x01,
    USART_RECEIVE_FIFO_ONE_CHAR = 0x00
} USART_RECEIVE_INTR_MODE;
```

7.4.9.2. PLIB_USART_RECEIVERINTERRUPTMODESELECT, EXEMPLE

Dans cet exemple, l'USART est configuré pour que l'interruption de réception se produise lorsque le tampon est aux ¾ plein.

```
PLIB_USART_ReceiverInterruptModeSelect(USART_ID_1,
                                         USART_RECEIVE_FIFO_3B4FULL);
```

7.4.10. EXEMPLE DE CONFIGURATION

Voici un exemple complet de configuration avec aussi la configuration des interruptions. La configuration établit 57'600 bauds et data bits = 8, parity = none et stop bits = 1.

```
PLIB_USART_Disable(USART_ID_1);
PLIB_USART_BaudRateSet(USART_ID_1, SYS_DEVCON_SYSTEM_CLOCK,
                        57600);
PLIB_USART_HandshakeModeSelect(USART_ID_1,
                                USART_HANDSHAKE_MODE_SIMPLEX);
PLIB_USART_OperationModeSelect(USART_ID_1,
                                USART_ENABLE_TX_RX_USED);
PLIB_USART_LineControlModeSelect(USART_ID_1, USART_8N1);
PLIB_USART_TransmitterEnable(USART_ID_1);
PLIB_USART_TransmitterInterruptModeSelect(USART_ID_1,
                                            USART_TRANSMIT_FIFO_EMPTY);
PLIB_USART_ReceiverEnable(USART_ID_1);
PLIB_USART_ReceiverInterruptModeSelect(USART_ID_1,
                                        USART_RECEIVE_FIFO_3B4FULL);

/* Initialize interrupts */
PLIB_INT_SourceEnable(INT_ID_0,
                      INT_SOURCE_USART_1_TRANSMIT);
PLIB_INT_SourceEnable(INT_ID_0,
                      INT_SOURCE_USART_1_RECEIVE);
PLIB_INT_SourceEnable(INT_ID_0, INT_SOURCE_USART_1_ERROR);
PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_UART1,
                           INT_PRIORITY_LEVEL5);
PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_UART1,
                              INT_SUBPRIORITY_LEVEL0);

// activation en tout dernier
PLIB_USART_Enable(USART_ID_1);
```


7.5. FONCTIONS DE L'ÉMETTEUR

On dispose d'un certain nombre de fonctions liées à l'émission.

Name	Description
<code>PLIB_USART_Transmitter9BitsSend</code>	Data to be transmitted in the byte mode with the 9th bit.
<code>PLIB_USART_TransmitterBreakSend</code>	Transmits the break character.
<code>PLIB_USART_TransmitterBreakSendIsComplete</code>	Returns the status of the break transmission
<code>PLIB_USART_TransmitterBufferIsFull</code>	Gets the transmit buffer full status.
<code>PLIB_USART_TransmitterByteSend</code>	Data to be transmitted in the Byte mode.
<code>PLIB_USART_TransmitterDisable</code>	Disables the specific USART module transmitter.
<code>PLIB_USART_TransmitterEnable</code>	Enables the specific USART module transmitter.
<code>PLIB_USART_TransmitterIdleIsLowDisable</code>	Disables the Transmit Idle Low state.
<code>PLIB_USART_TransmitterIdleIsLowEnable</code>	Enables the Transmit Idle Low state.
<code>PLIB_USART_TransmitterInterruptModeSelect</code>	Set the USART transmitter interrupt mode.
<code>PLIB_USART_TransmitterIsEmpty</code>	Gets the transmit shift register empty status.
<code>PLIB_USART_TransmitterIsIdle</code>	Gets the transmit buffer full status.
<code>PLIB_USART_TransmitterAddressGet</code>	Returns the address of the USART Tx register

Voici la description des fonctions les plus utiles.

7.5.1. LA FONCTION `PLIB_USART_TRANSMITTERBYTESEND`

La fonction `PLIB_USART_TransmitterByteSend` permet de transmettre un octet.

```
void PLIB_USART_TransmitterByteSend(USART_MODULE_ID index, int8_t data);
```

Il est nécessaire de déterminer s'il y a de la place dans le tampon d'émission avant d'appeler la fonction.

7.5.2. LA FONCTION `PLIB_USART_TRANSMITTERBUFFERISFULL`

La fonction `PLIB_USART_TransmitterBufferIsFull` permet de connaître la situation du tampon d'émission.

```
bool PLIB_USART_TransmitterBufferIsFull(USART_MODULE_ID index);
```

Si true, le buffer est plein. Avec false, le buffer n'est pas plein, il y a au moins la place pour un caractère.

7.5.3. EXEMPLE D'ÉMISSION AVEC `TRANSMITTERBYTESEND`

Voici un exemple d'émission d'une chaîne de caractères (terminée par un caractère 0x00). Ce code va attendre tant que le tampon d'émission est plein avant chaque byte.

```
while (Buffer[i] != 0) {
    // Attente si buffer full
    while (PLIB_USART_TransmitterBufferIsFull
            (USART_ID_1)) {
    }
    PLIB_USART_TransmitterByteSend(USART_ID_1, Buffer[i]);
    i++;
}
```


7.6. FONCTIONS DU RÉCEPTEUR

On dispose d'un certain nombre de fonctions liées à la réception.

Name	Description
PLIB_USART_ReceiverAddressAutoDetectDisable	Disables the automatic Address Detect mode.
PLIB_USART_ReceiverAddressAutoDetectEnable	Setup the automatic Address Detect mode.
PLIB_USART_ReceiverAddressDetectDisable	Enables the Address Detect mode.
PLIB_USART_ReceiverAddressDetectEnable	Enables the Address Detect mode.
PLIB_USART_ReceiverAddressIsReceived	Checks and return if the data received is an address.
PLIB_USART_ReceiverByteReceive	Data to be received in the Byte mode.
PLIB_USART_ReceiverDataIsAvailable	Identifies if the receive data is available for the specified USART module.
PLIB_USART_ReceiverDisable	Disables the USART receiver.
PLIB_USART_ReceiverEnable	Enables the USART receiver.
PLIB_USART_ReceiverFramingErrorHasOccurred	Gets the framing error status.
PLIB_USART_ReceiverIdleStateLowDisable	Disables receive polarity inversion.
PLIB_USART_ReceiverIdleStateLowEnable	Enables receive polarity inversion.
PLIB_USART_ReceiverInterruptModeSelect	Sets the USART receiver FIFO level.
PLIB_USART_ReceiverIsIdle	Identifies if the receiver is idle.
PLIB_USART_ReceiverModeSelect	Disables Single Receive mode.
PLIB_USART_ReceiverOverrunErrorClear	Clears a USART Overrun error.
PLIB_USART_ReceiverOverrunHasOccurred	Identifies if there was a receiver overrun error.
PLIB_USART_ReceiverParityErrorHasOccurred	Gets the parity error status.
PLIB_USART_ReceiverAddressGet	Returns the address of the USART Rx register

Voici la description des fonctions les plus utiles.

7.6.1. LA FONCTION **PLIB_USART_RECEIVERBYTERECEIVE**

La fonction **PLIB_USART_ReceiverByteReceive** permet de lire un byte des données reçues par l'USART et stockées dans le tampon de réception.

```
int8_t PLIB_USART_ReceiverByteReceive(USART_MODULE_ID index);
```

👉 Il faut s'assurer qu'un byte est disponible.

7.6.2. LA FONCTION **PLIB_USART_RECEIVERDATAISAVAILABLE**

La fonction **PLIB_USART_ReceiverDataIsAvailable** permet de savoir si une donnée est disponible.

```
bool PLIB_USART_ReceiverDataIsAvailable(USART_MODULE_ID index);
```

7.6.2.1. **PLIB_USART_RECEIVERDATAISAVAILABLE, EXEMPLE**

Cet exemple repris de la réponse à l'interruption de réception montre comment exploiter le tampon de réception. Avec int8_t c;

```
while (PLIB_USART_ReceiverDataIsAvailable(USART_ID_1))
{
    c = PLIB_USART_ReceiverByteReceive(USART_ID_1);
    PutCharInFifo (&descrFifoRX, c);
}
```

7.6.3. FONCTION **PLIB_USART_RECEIVEROVERRUNERRORCLEAR**

La fonction **PLIB_USART_ReceiverOverrunErrorClear** permet de supprimer une erreur d'overrun (débordement du buffer de réception). L'appel de cette fonction efface les données reçues.

```
void PLIB_USART_ReceiverOverrunErrorClear (USART_MODULE_ID index);
```

7.7. FONCTIONS GÉNÉRALES DE L'USART

Voici les nombreuses fonctions générales qui ne s'appliquent pas toutes pour le PIC32MX.

Name	Description
PLIB_USART_AlternateIODisable	Disables the use of alternate I/O pins for the receiver and transmitter.
PLIB_USART_AlternateIOEnable	Enables the use of alternate I/O pins for the receiver and transmitter.
PLIB_USART_Disable	Disables the specific USART module
PLIB_USART_Enable	Enables the specific USART module.
PLIB_USART_HandshakeModeSelect	Sets the data flow configuration.
PLIB_USART_IrDADisable	Disables the IrDA encoder and decoder.
PLIB_USART_IrDAEnable	Enables the IrDA encoder and decoder.
PLIB_USART_LineControlModeSelect	Sets the data flow configuration.
PLIB_USART_LoopbackDisable	Disables Loopback mode.
PLIB_USART_LoopbackEnable	Enables Loopback mode.
PLIB_USART_OperationModeSelect	Configures the operation mode of the USART module.
PLIB_USART_StopInIdleDisable	Disables the Stop in Idle mode (the USART module continues operation when the device is in Idle mode).
PLIB_USART_StopInIdleEnable	Discontinues operation when the device enters Idle mode.
PLIB_USART_SyncClockPolarityIdleHighDisable	Sets the idle state of the clock to low.
PLIB_USART_SyncClockPolarityIdleHighEnable	Sets the idle state of the clock to high.
PLIB_USART_SyncClockSourceSelect	Sets the clock source.
PLIB_USART_SyncModeSelect	Selects the USART mode to be asynchronous.
PLIB_USART_WakeOnStartDisable	Disables the wake-up on start bit detection feature during Sleep mode.
PLIB_USART_WakeOnStartEnable	Enables the wake-up on start bit detection feature during Sleep mode.
PLIB_USART_WakeOnStartIsEnabled	Gets the state of the sync break event completion.
PLIB_USART_ErrorsGet	Return the status of all errors in the specified USART module.

Voici la description des fonctions les plus utiles :

7.7.1. LA FONCTION **PLIB_USART_DISABLE**

La fonction **PLIB_USART_Disable** permet de désactiver l'USART. Cela a pour effet de vider les tampons d'émission et de réception.

```
void PLIB_USART_Disable (USART_MODULE_ID index);
```

7.7.2. LA FONCTION **PLIB_USART_ENABLE**

La fonction **PLIB_USART_Enable** permet d'activer l'USART. Cela a pour effet que les broches RX et TX sont gérées par l'USART.

```
void PLIB_USART_Enable (USART_MODULE_ID index);
```

7.7.3. LA FONCTION **PLIB_USART_ERRORGET**

La fonction **PLIB_USART_ErrorGet** permet d'obtenir la situation des bits d'erreurs.

```
USART_ERROR PLIB_USART_ErrorsGet(USART_MODULE_ID index);
```

Le type USART_ERROR est défini ainsi :

```
typedef enum {  
    USART_ERROR_NONE = 0x00,  
    USART_ERROR_RECEIVER_OVERRUN = 0x01,  
    USART_ERROR_FRAMING = 0x02,  
    USART_ERROR_PARITY = 0x04  
} USART_ERROR;
```

7.7.4. EXEMPLE GESTION DES ERREURS

L'exemple repris de la réponse à l'interruption de réception montre le traitement des erreurs.

```
USART_ERROR UsartStatus = PLIB_USART_ErrorsGet(USART_ID_1);  
  
if ( (UsartStatus & (USART_ERROR_PARITY |  
                    USART_ERROR_FRAMING |  
                    USART_ERROR_RECEIVER_OVERRUN)) == 0) {  
    // OK traitement de réception possible  
} else {  
    // Suppression des erreurs  
    // La lecture des erreurs les efface sauf pour overrun  
    if ( (UsartStatus & USART_ERROR_RECEIVER_OVERRUN) ==  
         USART_ERROR_RECEIVER_OVERRUN) {  
        PLIB_USART_ReceiverOverrunErrorClear(USART_ID_1);  
    }  
}
```

7.8. EMISSION ET RÉCEPTION SANS INTERRUPTION

Cette section a pour but de montrer l'utilisation des fonctions de base pour réaliser une transmission et une réception.

7.8.1. CONFIGURATION USART SANS INTERRUPTION

Voici la configuration de l'USART sans interruption, pour un baudrate de 19'200.

```
void DRV_USART0_Initialize(void)
{
    /* Initialize USART */
    PLIB_USART_BaudRateSet(USART_ID_1,
                           SYS_DEVCON_SYSTEM_CLOCK, 19200);
    PLIB_USART_HandshakeModeSelect(USART_ID_1,
                                    USART_HANDSHAKE_MODE_SIMPLEX);
    PLIB_USART_OperationModeSelect(USART_ID_1,
                                    USART_ENABLE_TX_RX_USED);
    PLIB_USART_LineControlModeSelect(USART_ID_1,
                                     USART_8N1);
    PLIB_USART_TransmitterEnable(USART_ID_1);
    PLIB_USART_TransmitterInterruptModeSelect(USART_ID_1,
                                              USART_TRANSMIT_FIFO_EMPTY);
    PLIB_USART_ReceiverEnable(USART_ID_1);
    PLIB_USART_ReceiverInterruptModeSelect(USART_ID_1,
                                           USART_RECEIVE_FIFO_ONE_CHAR);
    PLIB_USART_Enable(USART_ID_1); // nécessaire
}
```

Dans ce code, on configure tout de même le comportement des interruptions de l'USART, même si on ne les utilise pas.

7.8.2. EMISSION EN POLLING

Voici une fonction qui reçoit la valeur de deux entrées analogiques pour en fabriquer un message complet et le transmettre :

```
void APP_SendMessAD ( S_ADCResults AdcRes )
{
    char Buffer[40];
    int i;

    sprintf(Buffer, "Canal0 %4d  Canal1 %4d \n",
            AdcRes.Chan0, AdcRes.Chan1);

    while (Buffer[i] != 0) {
        // Attente si buffer full
        while (PLIB_USART_TransmitterBufferIsFull(USART_ID_1)) {
            BSP_LEDToggle(BSP_LED_2);
        }
        PLIB_USART_TransmitterByteSend(USART_ID_1, Buffer[i]);
        i++;
    }
}
```

Cette fonction attend tant que le tampon d'émission est plein en utilisant la fonction `PLIB_USART_TransmitterBufferIsFull`. On insère dans la boucle d'attente l'inversion de la `LED_2` pour observer la durée des attentes.

Lorsque l'on sort de la boucle d'attente, on appelle la fonction `PLIB_USART_TransmitterByteSend` qui dépose le caractère à émettre dans le tampon de transmission qui a alors au moins une place de disponible.

7.8.3. UTILISATION DE LA FONCTION D'ÉMISSION ET OBSERVATIONS

Grace à une interruption, on active l'application toutes les 20 ms et on place l'appel de la fonction dans le case `APP_STATE_SERVICE_TASKS`.

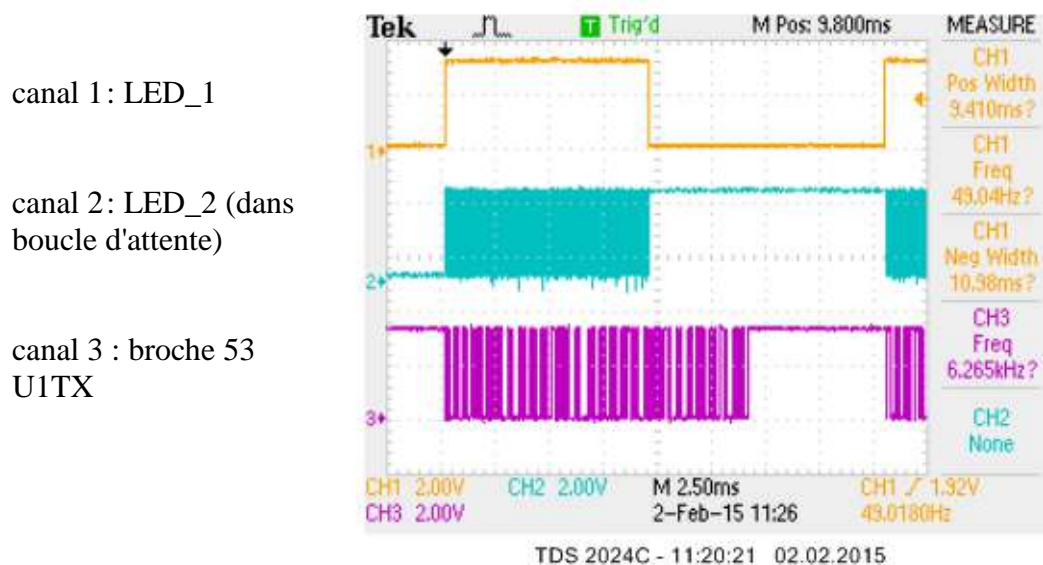
```
case APP_STATE_SERVICE_TASKS:
    // Lecture des 2 pots
    AdcRes = BSP_ReadAllADC();
    // affichage local des valeurs de l'AD
    lcd_gotoxy(1,3);
    printf_lcd("Ad Chan0 %4d      ", AdcRes.Chan0);
    lcd_gotoxy(1,4);
    printf_lcd("Ad Chan1 %4d      ", AdcRes.Chan1);

    BSP_LEDOff(BSP_LED_1);
    // Emet sur la comm serie
    APP_SendMessAD(AdcRes);
    BSP_LEDOn(BSP_LED_1);

    appData.state = APP_STATE_WAIT;
break;
```

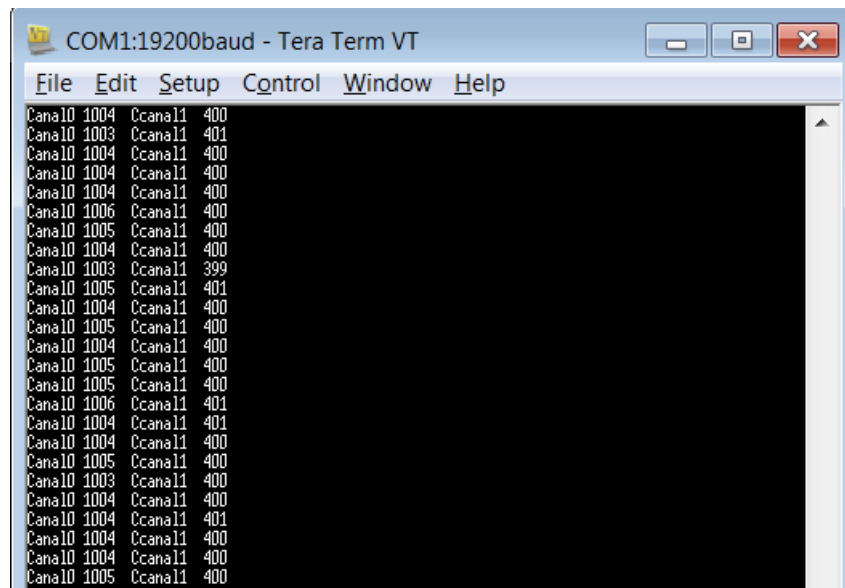
7.8.3.1. COMPORTEMENT DE L'ÉMISSION

De l'observation ci-dessous, on déduit que le temps d'exécution de la fonction `APP_SendMessAD` est de 9.4 ms. On observe encore que la transmission continue après la fin de la fonction car on a rempli le tampon d'émission.



7.8.3.2. RÉSULTAT ÉMISSION AVEC TERMINAL

On obtient :



7.8.4. RÉCEPTION EN POLLING

Pour tester la réception, nous utiliserons une application qui envoie un message toutes les 25 ms. Il s'agit d'un message de type texte reprenant le principe du message émis, mais auquel nous ajoutons un caractère de début et un caractère de fin. Voici un exemple :

```
!Ch0 0976 Ch1 0478#
```

Cela nous fait un message de 19 caractères. A 19'200 bauds, avec 10 bits par caractère, le temps pour un bit est de $1/19'200 = 0.052$ ms, donc 0.52 ms pour un caractère et 9.88 ms pour la transmission du message complet. Comme l'application a un cycle de 20 ms, cela convient pour le traitement du message. Par contre, comme le tampon de réception hardware de l'USART ne contient que 8 caractères, il est indispensable de traiter beaucoup plus rapidement la réception des caractères. Nous disposons d'une interruption à 400 us qui convient bien même pour une réception caractère par caractère. Nous avons encore besoin d'un élément de stockage pour l'ensemble du message avant son traitement par l'application, pour cela nous utiliserons un FIFO software.

7.8.4.1. FONCTION DE REMPLISSAGE DU FIFO

Cette fonction est appelée cycliquement (période de 400 us). Elle prend les caractères reçus dans le tampon de réception et les copie dans le FIFO.

```
void APP_FillRxFifo(void) {
    USART_ERROR  UsartStatus;
    int8_t c;

    // Test si erreur parité ou overrun
    UsartStatus = PLIB_USART_ErrorsGet(USART_ID_1);

    if ( (UsartStatus & (USART_ERROR_PARITY |
                        USART_ERROR_FRAMING |
                        USART_ERROR_RECEIVER_OVERRUN)) == 0) {
        // transfert dans le FIFO de tous les char reçus
        while (PLIB_USART_ReceiverDataIsAvailable
                (USART_ID_1))
        {
            c = PLIB_USART_ReceiverByteReceive
                (USART_ID_1);
            PutCharInFifo ( &descrFifoRX, c);
            BSP_LEDToggle(BSP_LED_3);
        }
    } else {
        // La lecture des erreurs les efface
        // sauf pour overrun
        if ( (UsartStatus & USART_ERROR_RECEIVER_OVERRUN)
            == USART_ERROR_RECEIVER_OVERRUN) {
            PLIB_USART_ReceiverOverrunErrorClear
                (USART_ID_1);
        }
    }
}
```

7.8.4.2. APPEL DE LA FONCTION DANS LA RÉPONSE À L'INTERRUPTION

Dans la réponse à l'interruption du Timer1, on appelle à chaque cycle la fonction **APP_FillRxFifo()**. Tous les 50 cycles (20 ms) on active l'application.

```
// ISR Timer1 période 400 us
void __ISR(_TIMER_1_VECTOR, ipl4)
    _IntHandlerDrvTmrInstance0(void)
{
    static int count = 0;
    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_1);
    BSP_LEDToggle(BSP_LED_0);    // pour contrôle au scope
    // Appel fonction de réception
    APP_FillRxFifo();

    count++;
    if ( count > 50 ) { // 20 ms
        count = 0;
        APP_UpdateState(APP_STATE_SERVICE_TASKS);
    }
}
```

7.8.4.3. LA FONCTION APP_GETMESSAD

Cette fonction traite le message en fonction du nombre de caractères disponibles. Elle se synchronise sur le caractère de début '!' et termine le traitement lorsqu'elle trouve le caractère de fin '#'. La fonction retourne vrai lorsque le message est disponible.

```
bool APP_GetMessAD(char *pMess) {
    bool stat = false;
    bool EndMess = false;
    uint8_t NbCharToRead = 0;
    static uint8_t RxC;
    static int GetSituation = WaitSTX;
    static int MessIdx = 0;

    // Détermine le nombre de caractères à lire
    NbCharToRead = GetReadSize ( &descrFifoRX);

    EndMess = false;
    while ( (NbCharToRead >= 1) &&
            ( EndMess == false) ) {
```



```
// Lis un caractère sans gestion du status
GetCharFromFifo ( &descrFifoRX, &RxC );
NbCharToRead--;

switch (GetSituation) {
    case WaitSTX :
        if (RxC == '!') {
            MessIdx = 0;
            pMess[MessIdx] = RxC;
            MessIdx++;
            GetSituation = WaitETX;
            BSP_LEDToggle(BSP_LED_4);
        }
        break;

    case WaitETX :
        pMess[MessIdx] = RxC;
        MessIdx++;
        if (RxC == '#') {
            pMess[MessIdx] = 0; // nul
            MessIdx = 0;
            EndMess = true;
            stat = true;
            GetSituation = WaitSTX;
            BSP_LEDToggle(BSP_LED_4);
        }
        break;
} // end switch
} // end while
return stat;
}
```

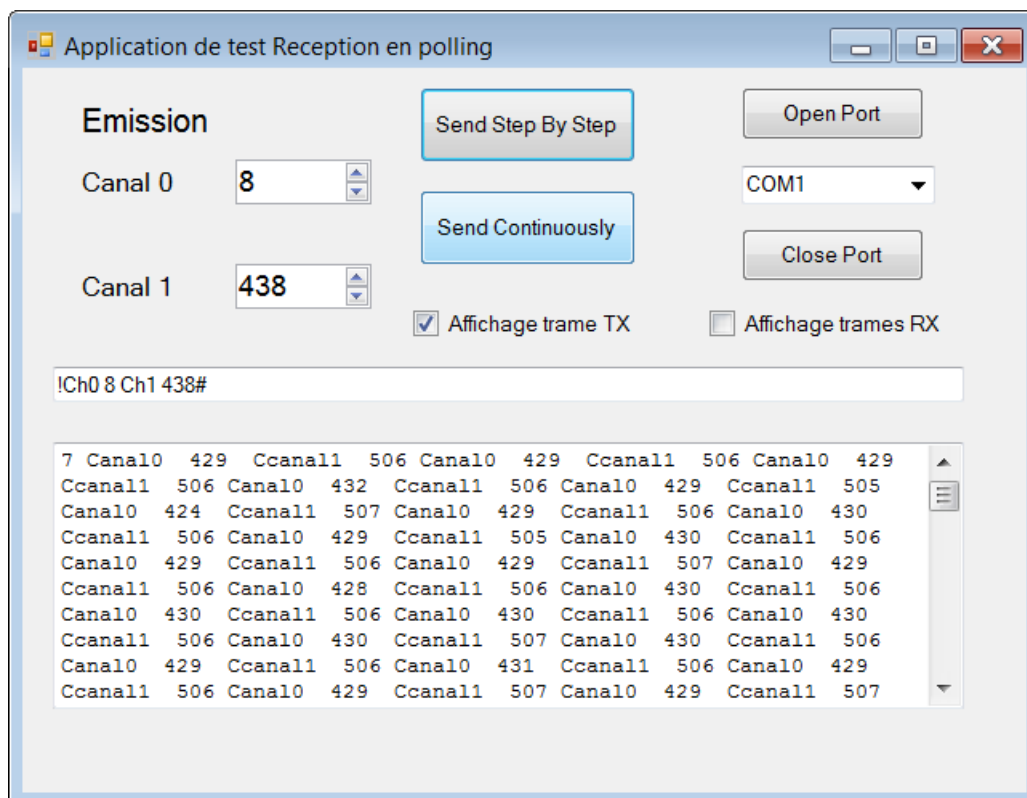
7.8.4.4. UTILISATION DANS L'APPLICATION

Appel de la fonction dans le case APP_STATE_SERVICE_TASKS avec affichage si StatMess est vrai.

```
StatMess = APP_GetMessAD(RxMess);
if (StatMess == true) {
    lcd_gotoxy(1,4);
    printf_lcd("%s", RxMess);
    StatMess == false;    // consommation du message
}
```

7.8.4.5. EXEMPLE DE RÉSULTAT AVEC L'APPLICATION

Sur l'affichage du KIT, on retrouve bien le message émis :

**7.8.4.6. COMPORTEMENT DE LA RÉCEPTION**

canal 1 : LED_1

canal 2 : LED_3
(inversion à chaque caractère obtenu du tampon HW de réception)

canal 3 : broche 52
U1RX



Le signal sur la LED_3 montre que l'on n'obtient pas un caractère à chaque interruption, donc que le tampon de réception ne stocke pas.

7.8.4.7. CONCLUSION SUR RÉCEPTION EN POLLING

Cette solution est fonctionnelle et convient si l'on dispose d'une interruption cyclique. Un FIFO software est par contre indispensable.

7.9. NÉCESSITÉ DU RECOURS À UN FIFO

Un tampon (*Buffer*) est très souvent utilisé en communication pour gérer le classique problème du producteur et du consommateur. En général, le producteur (réception des caractères) travaille caractère par caractère (en général sous interruption). Le consommateur (traitement du message reçu) travaille plutôt par à-coup en lisant message après message.

Au niveau émission, le principe s'inverse : le producteur dépose un message et ce message est émis caractère par caractère en fonction des possibilités de l'USART.

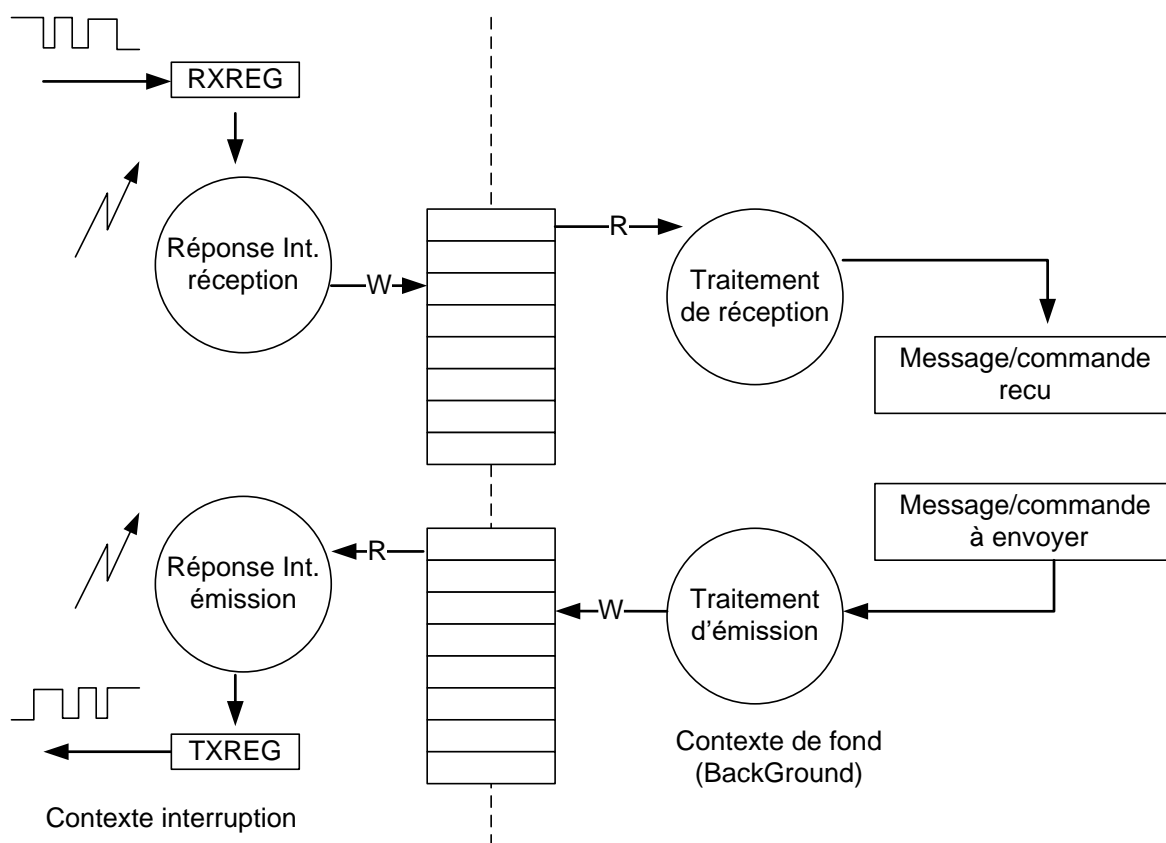
7.9.1. RÔLES DU FIFO

Le FIFO a 2 fonctions importantes :

- Absorber temporairement les à-coups production/consommation.
- Séparer 2 contextes (par exemple, routine de réponse à l'interruption et programme principal).

7.9.2. SITUATION COMMUNICATION AVEC DES FIFO

Le diagramme ci-dessous illustre une situation de communication avec un FIFO de réception et un FIFO d'émission. Il illustre aussi la situation des tampons hardware de réception et d'émission.



7.10. PRINCIPE DU FIFO

L'abréviation FIFO provient de l'anglais *First In First Out* (à ne pas confondre avec une pile, LIFO, *Last In First Out*).

Un FIFO est donc un élément de stockage temporaire, comme par exemple une série de billes enfilées dans un tube.

Dans la pratique, on ne va pas déplacer les données dans le FIFO mais faire évoluer le pointeur d'écriture et de lecture pour « simuler » cette situation de déplacement des données.

Par FIFO, on sous-entend FIFO circulaire, en ce sens que lorsque l'on atteint la fin du FIFO on recommence au début.

Au niveau de la gestion du FIFO il y a l'aspect production (écriture ou remplissage) et l'aspect consommation (lecture ou vidage).

7.10.1. SÉPARATION DES CONTEXTES

Lorsqu'un FIFO est utilisé pour séparer un contexte d'interruption du contexte du programme principal, il faut soigner les fonctions d'écriture et de lecture de telle manière que des erreurs sur la situation du FIFO ne puissent se produire.

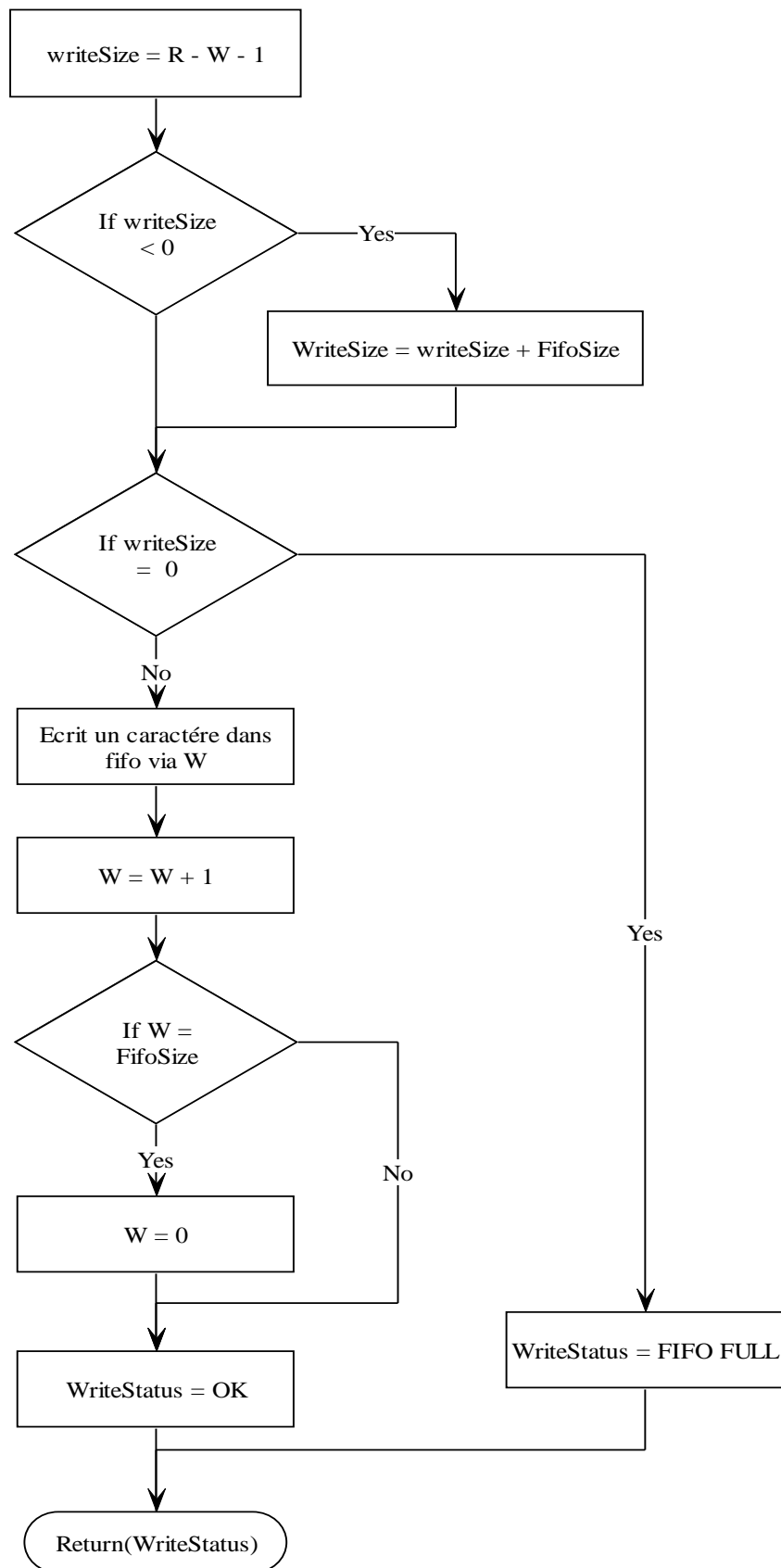
Par exemple lors de la détermination de la situation en lecture, il ne faut lire qu'une seule fois le pointeur d'écriture et le mémoriser dans une variable locale (car l'interruption peut survenir et ajouter un caractère, donc modifier le pointeur pendant le traitement).

7.10.2. PRINCIPE DE L'ÉCRITURE DANS LE FIFO

La contrainte lors de l'écriture est de déterminer s'il y a encore de la place dans le FIFO. Si le FIFO est plein (FIFO full), il est interdit d'écrire car on écrase une donnée non encore lue. Le problème est souvent : que faire lorsque le FIFO est plein ?

Remarque : seule la fonction d'écriture a le droit de modifier le pointeur d'écriture, le pointeur de lecture peut être lu pour déterminer la situation.

Voici un organigramme décrivant le principe d'écriture. Dans l'organigramme, **W** représente le pointeur ou l'indice d'écriture, **R** celui de lecture.



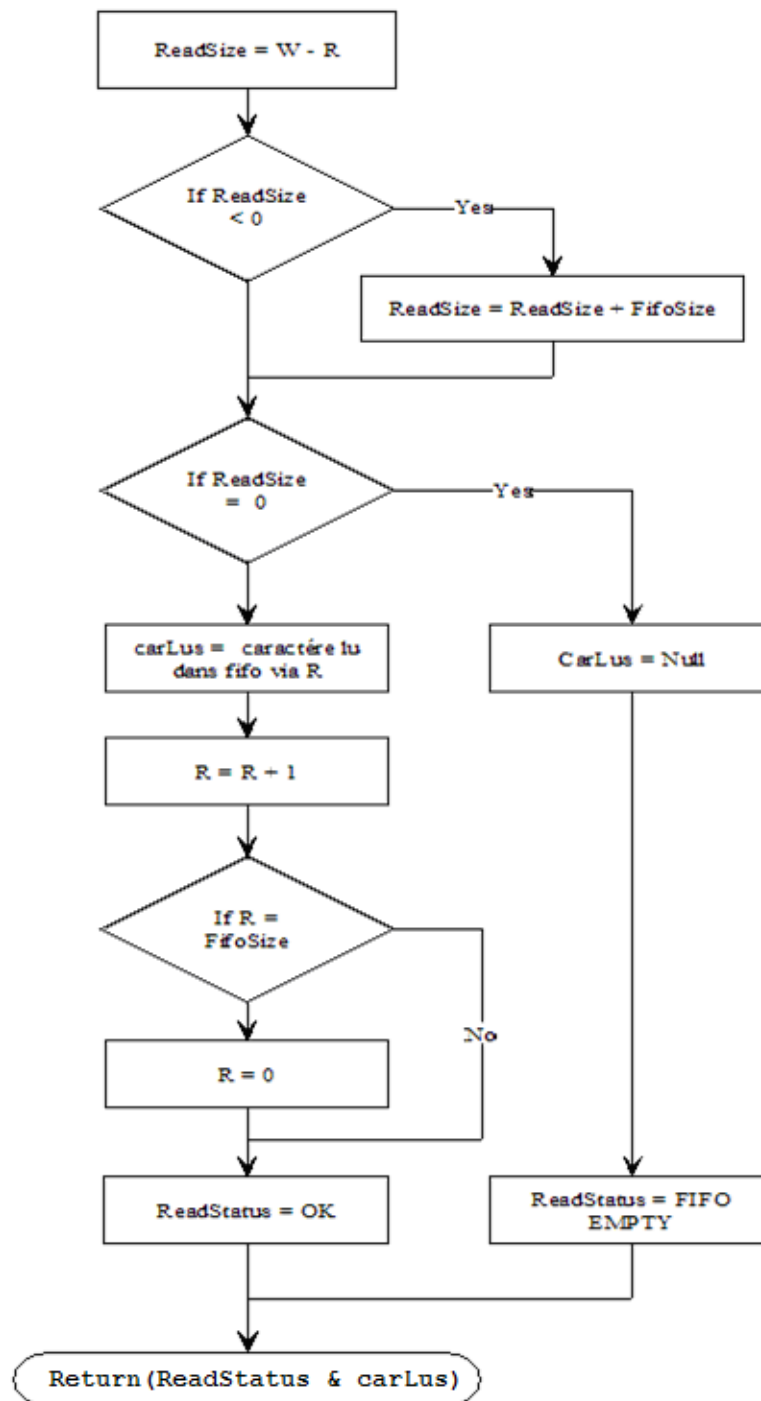
Ecriture dans le FIFO

7.10.3. PRINCIPE DE LA LECTURE DANS LE FIFO

La lecture commence toujours par le test de la présence de données dans le FIFO. S'il n'y a pas de donnée, le FIFO est vide (FIFO empty).

Remarque : seule la fonction de lecture a le droit de modifier le pointeur de lecture, le pointeur d'écriture peut être lu pour déterminer la situation.

Voici un organigramme décrivant le principe de lecture dans le FIFO. Dans l'organigramme, R représente le pointeur ou l'indice de lecture, W celui d'écriture.



Lecture dans le FIFO

7.11. ANALYSE DU FONCTIONNEMENT D'UN FIFO

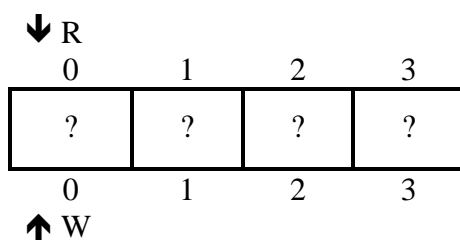
Pour bien comprendre comment fonctionne un FIFO circulaire, nous allons étudier quelques situations sur la base d'un FIFO de 4 caractères. Cela permettra de mieux comprendre les organigrammes fournis précédemment.

↓ R représente l'index ou le pointeur de lecture.

↑ W représente l'index ou le pointeur d'écriture.

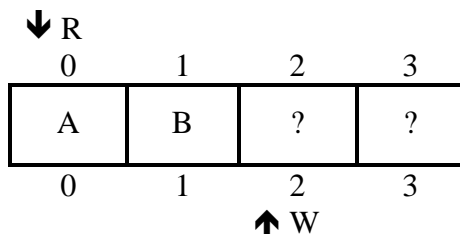
7.11.1. SITUATION DE DÉPART

Situation après l'initialisation. Les deux pointeurs sont au début du FIFO.



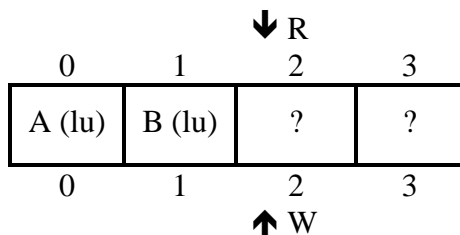
7.11.2. SITUATION APRÈS ÉCRITURE DE 2 CARACTÈRES

Situation après l'écriture de 2 caractères. Le nombre de caractères disponibles pour la lecture correspond à $W - R$ ($2 - 0 = 2$). Si la différence est nulle, cela signifie qu'il n'y a pas de caractères de disponible. Notez que le pointeur d'écriture est géré de la manière suivante : écriture du caractère, puis incrément du pointeur. Il pointe donc sur la prochaine place disponible pour écriture.



7.11.3. SITUATION APRÈS LECTURE DES 2 CARACTÈRES

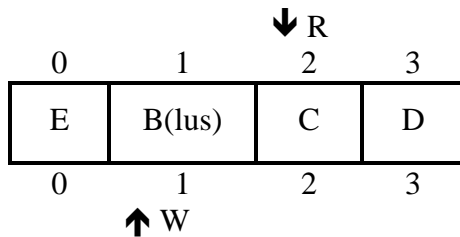
Situation après la lecture des 2 caractères. Le nombre de caractères disponibles pour la lecture correspond à $W - R$ ($2 - 2 = 0$). Dans ce cas, la différence est nulle, cela signifie qu'il n'y a pas de caractères de disponible. Notez que le pointeur de lecture est géré de la manière suivante : lecture du caractère, puis incrément du pointeur. Il pointe donc sur le prochain caractère disponible pour lecture.



7.11.4. SITUATION DE REBOUCLEMENT

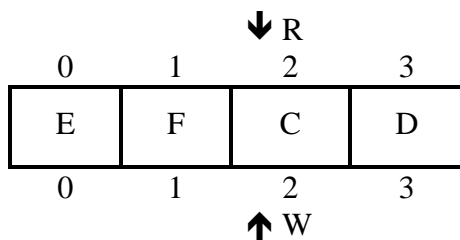
Situation après écriture de 3 caractères supplémentaires (C, D, E). Le nombre de caractères disponibles pour la lecture correspond $W - R$ ($1 - 2 = -1$). Si la différence est négative, il faut la corriger en ajoutant la taille du FIFO, d'où $-1 + 4 = 3$, soit 3 caractères à lire.

En ce qui concerne l'écriture, il faut vérifier la place disponible par la différence des pointeurs, soit $R - W$. Dans notre situation $2 - 1 = 1$, il est théoriquement encore possible d'écrire 1 caractère.



7.11.5. SITUATION FIFO PLEIN

La situation précédente est déjà une situation de FIFO plein. Car si l'on écrit encore un caractère (F) sans modifier la situation de la lecture, on se trouve dans une situation qui n'est pas admissible : $R=W$, ce qui donne une différence nulle donc un FIFO vide côté lecture, alors qu'il y a des caractères à lire.



Situation interdite !

7.11.6. FIFO PLEIN CONCLUSION

Donc, de cette situation interdite, on conclut les règles suivantes :

- Dans un FIFO de taille **n**, il est possible d'écrire **n-1** caractères.
- Le FIFO est plein si $R - W - 1 = 0$.
- Le nombre de caractères que l'on peut écrire est donné par la formule $R - W - 1$, lorsque $R = W = 0$, on obtient $0 - 0 - 1 = -1 + 4 = 3$.

7.12. RÉALISATION FIFO AVEC DES POINTEURS

Dans cet exemple, en plus de l'usage de pointeurs, nous allons introduire une structure décrivant un FIFO dans le but d'obtenir des fonctions qui ne soient plus spécifiques à un FIFO, mais au contraire que l'on puisse utiliser pour gérer plusieurs FIFOs.

7.12.1. STRUCTURE DÉCRIVANT UN FIFO

```
typedef struct fifo {
    int32_t fifoSize;    // taille du fifo
    int8_t *pDebFifo;    // pointeur sur début du fifo
    int8_t *pFinFifo;    // pointeur sur fin du fifo
    int8_t *pWrite;      // pointeur d'écriture
    int8_t *pRead;       // pointeur de lecture
} S_fifo;
```

Le pointeur de début est nécessaire lorsqu'on détecte le reboucllement du FIFO pour réinitialiser le pointeur courant sur le début. Le pointeur de fin permet de détecter le reboucllement. La taille du FIFO est utile pour les calculs de nombres d'éléments disponibles.

7.12.2. DÉCLARATION D'UN FIFO

Voici la déclaration d'un FIFO, il s'agit de variables globales.

```
// Déclaration du FIFO pour la réception
#define FIFO_RX_SIZE ( (2*8) + 1) // 2 messages

int8_t fifoRX[FIFO_RX_SIZE];
// Déclaration du descripteur du FIFO de réception
S_fifo descrFifoRX;
```

7.12.3. INITIALISATION D'UN FIFO

Voici l'initialisation d'un descripteur de FIFO. Cette initialisation est effectuée dans le programme principal. Pour faciliter l'initialisation, une fonction est mise à disposition. Il est nécessaire de connaître l'adresse du descripteur du FIFO, l'adresse du début du FIFO et la taille du FIFO.

```
/*-----*/
/* InitFifo */
/*=====*/
// Initialisation du descripteur de FIFO
// avec possibilité de fournir une valeur de remplissage
void InitFifo ( S_fifo *pDescrFifo, int32_t FifoSize,
int8_t *pDebFifo, int8_t InitVal )
{
    int32_t i;
    int8_t *pFif;
    pDescrFifo->fifoSize =    FifoSize;
    pDescrFifo->pDebFifo =    pDebFifo; // début du fifo
    // fin du fifo
    pDescrFifo->pFinFifo =    pDebFifo + (FifoSize - 1);
    pDescrFifo->pWrite =      pDebFifo; // début du fifo
```

```

    pDescrFifo->pRead      =   pDebFifo; // début du fifo
    pFif = pDebFifo;
    for (i=0; i < FifoSize; i++) {
        *pFif = InitVal;
        pFif++;
    }
} /* InitFifo */

```

Appel de la fonction d'initialisation :

```
InitFifo ( &descrFifoRX, FIFO_RX_SIZE, fifoRX, 0 );
```

7.12.4. LA FONCTION GETWRITESPACE

Cette fonction permet de déterminer la place disponible en écriture dans le FIFO. Elle est utilisée dans la fonction d'écriture dans le FIFO et aussi comme test indépendant en vue de la gestion du contrôle de flux.

```

/*-----*/
/* GetWriteSpace */
/*=====*/
// Retourne la place disponible en écriture
int32_t GetWriteSpace ( S_fifo *pDescrFifo)
{
    int32_t writeSize;

    // Détermine le nb de car. que l'on peut déposer
    writeSize = pDescrFifo->pRead - pDescrFifo->pWrite -1;
    if (writeSize < 0) {
        writeSize = writeSize + pDescrFifo->fifoSize;
    }
    return (writeSize);
} /* GetWriteSpace */

```

La détermination du nombre de caractères que l'on peut écrire est réalisée par la différence entre les pointeurs.

7.12.5. LA FONCTION GETREADSIZE

Cette fonction permet de déterminer le nombre de caractères présents dans le FIFO disponibles pour la lecture. Elle est utilisée dans la fonction de lecture du FIFO et aussi comme test indépendant en vue de la gestion du contrôle de flux.

```

/*-----*/
/* GetReadSize */
/*=====*/
// Retourne le nombre de caractères à lire
int32_t GetReadSize ( S_fifo *pDescrFifo)
{
    int32_t readSize;

    readSize = pDescrFifo->pWrite - pDescrFifo->pRead;
}

```

```
    if (readSize < 0) {
        readSize = readSize + pDescrFifo->fifoSize;
    }
    return (readSize);
} /* GetReadSize */
```

La détermination du nombre de caractères que l'on peut lire est réalisée par la différence entre les pointeurs.

7.12.6. LA FONCTION PUTCHARINFIFO

Cette fonction permet de déposer un caractère dans le FIFO dont on fournit le descripteur. Cette fonction n'est plus spécifique à un FIFO.

```
/*-----*/
/* PutCharInFifo */
/*=====*/
// Dépose un caractère dans le FIFO
// Retourne 0 si OK, 1 si FIFO full
uint8_t PutCharInFifo ( S_fifo *pDescrFifo,
                        int8_t charToPut )
{
    uint8_t writeStatus;

    // test si fifo est FULL
    if (GetWriteSpace(pDescrFifo) == 0) {
        writeStatus = 1; // fifo FULL
    }
    else {
        // écrit le caractère dans le FIFO
        *(pDescrFifo->pWrite) = charToPut;

        // incrément le pointeur d'écriture
        pDescrFifo->pWrite++;
        // gestion du reboucllement
        if (pDescrFifo->pWrite > pDescrFifo->pFinFifo) {
            pDescrFifo->pWrite = pDescrFifo->pDebFifo;
        }

        writeStatus = 0; // OK
    }
    return (writeStatus);
} // PutCharInFifo
```

La détection du reboucllement se fait par test du dépassement par pWrite de la fin du FIFO.

7.12.7. LA FONCTION GETCHARFROMFIFO

Cette fonction permet d'obtenir un caractère du FIFO dont on fournit le descripteur. Cette fonction n'est plus spécifique à un FIFO.

```

/*-----*/
/* GetCharFromFifo */
/*=====*/
// Obtient (lecture) un caractère du fifo
// retourne 0 si OK, 1 si empty
// le caractère lu est retourné par référence
uint8_t GetCharFromFifo (S_fifo *pDescrFifo, int8_t *carLu)
{
    int32_t readSize;
    uint8_t readStatus;

    // détermine le nb de car. que l'on peut lire
    readSize = GetReadSize(pDescrFifo);

    // test si fifo est vide
    if (readSize == 0) {
        readStatus = 1; // fifo EMPTY
        *carLu = 0;      // carLu = NULL
    }
    else {
        // lis le caractère dans le FIFO
        *carLu = *(pDescrFifo->pRead);

        // incrément du pointeur de lecture
        pDescrFifo->pRead++;
        // gestion du reboucllement
        if (pDescrFifo->pRead > pDescrFifo->pFinFifo) {
            pDescrFifo->pRead = pDescrFifo->pDebFifo;
        }
        readStatus = 0; // OK
    }
    return (readStatus);
} // GetCharFromFifo

```

La détection du reboucllement se fait par test du dépassement par pRead de la fin du FIFO.

7.12.8. GESTION DES FIFO, LIBRAIRIE À DISPOSITION

L'ensemble des fonctions de gestion du FIFO correspondant à ce chapitre est regroupé sur le réseau dans les fichiers **GesFifoTh32.h** et **GesFifoTh32.c** sous:

...\\Maitres-Eleves\\SLO\\Modules\\SL229_MINF\\PIC32MX_Utilitaires(PlibHarmony)\\
Fifo&Antirebond.

7.13. EXEMPLE UTILISATION DES FIFOs ET DES INTERRUPTIONS

Cette application utilise la réception et la transmission sous interruption en utilisant un FIFO pour chacune, ainsi que la gestion du contrôle de flux.

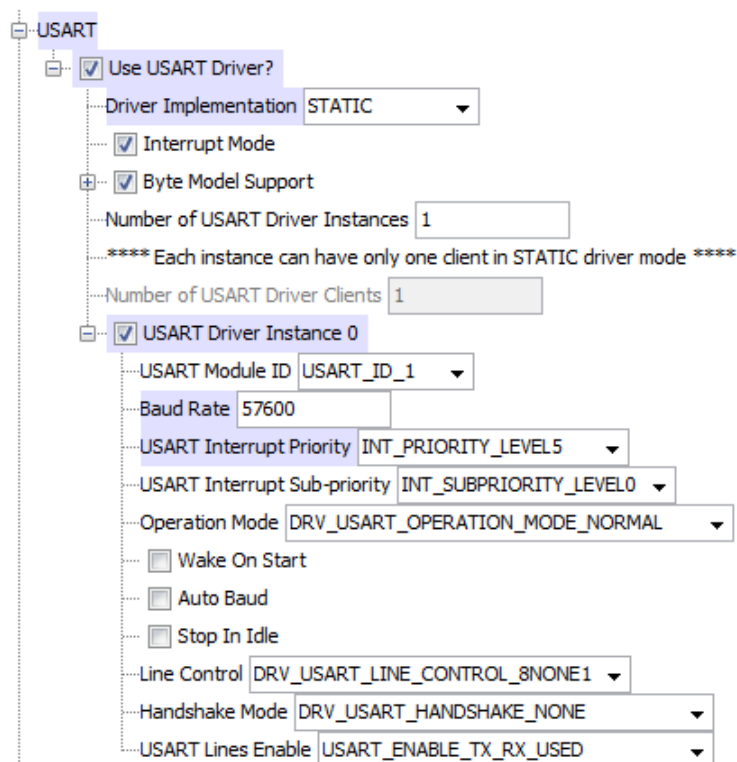
L'objectif de cet exemple est de montrer le comportement de l'UART du PIC32MX qui possède un tampon de réception et un tampon d'émission, comportement en relation avec les FIFOs software et la problématique de la gestion du contrôle de flux par le programme.

Dans cet exemple, nous utilisons un message d'une taille de 6 caractères et nous nous focalisons uniquement sur les mécanismes de communication sans nous préoccuper de comment sont formatés et reconstitués les messages (cet autre aspect fait l'objet d'un exemple complet à la fin de ce chapitre).

7.13.1. CONFIGURATION DE L'USART

Voici la configuration de base d'un USART driver au niveau de Harmony 2.05. Des modifications seront réalisées au niveau du code généré pour vérifier l'effet de certains paramètres.

7.13.1.1. CONFIGURATION D'UN USART DRIVER AVEC HARMONY



7.13.1.2. CONTENU DE LA FONCTION DRV_USART0_INITIALIZE

L'ensemble des actions de configuration de l'USART sont regroupées dans la fonction DRV_USART0_Initialize, dont voici le contenu.

```
SYS_MODULE_OBJ DRV_USART0_Initialize(void)
{
    uint32_t clockSource;

    /* Disable the USART module to configure it*/
    PLIB_USART_Disable (USART_ID_1);

    /* Initialize the USART based on configuration
       settings */
    PLIB_USART_InitializeModeGeneral(USART_ID_1,
                                     false, /*Auto baud*/
                                     false, /*LoopBack mode*/
                                     false, /*Auto wakeup on start*/
                                     false, /*IRDA mode*/
                                     false); /*Stop In Idle mode*/

    /* Set the line control mode */
    PLIB_USART_LineControlModeSelect(USART_ID_1,
                                     DRV_USART_LINE_CONTROL_8NONE1);

    /* We set the receive interrupt mode to receive an
       interrupt whenever FIFO is not empty */
    PLIB_USART_InitializeOperation(USART_ID_1,
                                   USART_RECEIVE_FIFO_ONE_CHAR,
                                   USART_TRANSMIT_FIFO_IDLE,
                                   USART_ENABLE_TX_RX_USED);

    /* Get the USART clock source value*/
    clockSource = SYS_CLK_PeripheralFrequencyGet (
        CLK_BUS_PERIPHERAL_1 );

    /* Set the baud rate and enable the USART */
    PLIB_USART_BaudSetAndEnable(USART_ID_1,
                                clockSource, 57600); /*Desired Baud rate value*/

    /* Clear the interrupts to be on the safer side*/
    SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_TRANSMIT);
    SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_RECEIVE);
    SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_ERROR);

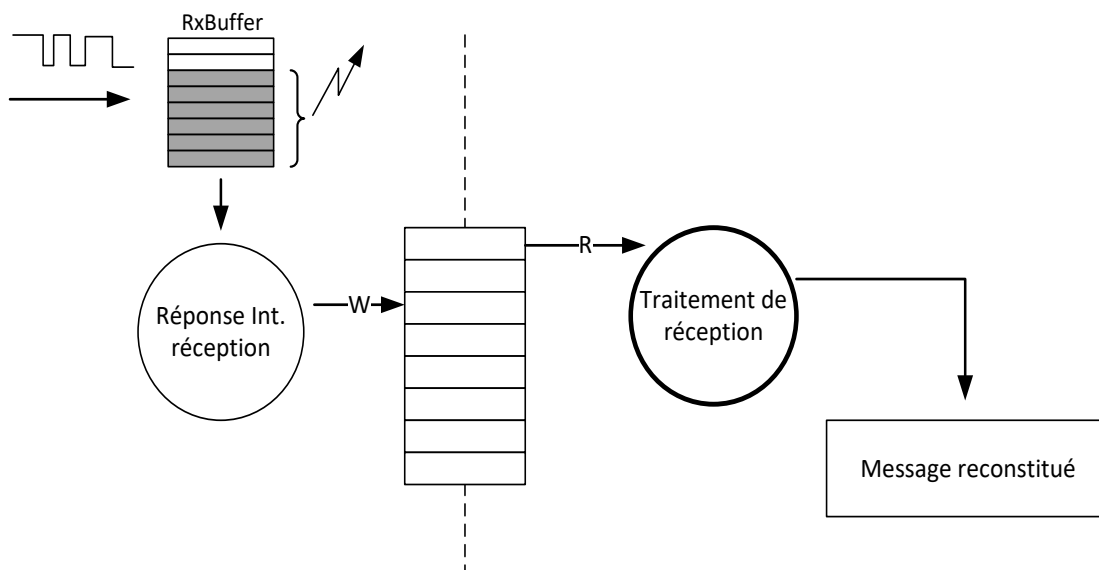
    /* Enable the error interrupt source */
    SYS_INT_SourceEnable(INT_SOURCE_USART_1_ERROR);

    /* Enable the Receive interrupt source */
    SYS_INT_SourceEnable(INT_SOURCE_USART_1_RECEIVE);

    /* Return the driver instance value*/
    return (SYS_MODULE_OBJ) DRV_USART_INDEX_0;
}
```

7.13.2. CONTEXTE DE RÉCEPTION

Voici un diagramme montrant le contexte de réception.



7.13.3. DÉTAILS CONFIGURATION INTERRUPTION RÉCEPTION

Voici une partie de la configuration de l'USART, en particulier la section qui configure le comportement de l'interruption de réception :

```

PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_HALF_FULL,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
...
SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_RECEIVE);
...
SYS_INT_SourceEnable(INT_SOURCE_USART_1_RECEIVE);
    
```

Et après l'appel de la fonction DRV_USART0_Initialize, on trouve :

```

SYS_INT_VectorPrioritySet(INT_VECTOR_UART1,
    INT_PRIORITY_LEVEL5);
SYS_INT_VectorSubprioritySet(INT_VECTOR_UART1,
    INT_SUBPRIORITY_LEVEL0);
    
```

7.13.3.1. CHOIX D'UN MODE DE DÉCLENCHEMENT DE INT RX

Il y a 3 possibilités pour le déclenchement de l'interruption RX, suivant le niveau de remplissage du buffer de réception :

Members	Description
USART_RECEIVE_FIFO_HALF_FULL	Interrupt when receive buffer is half full
USART_RECEIVE_FIFO_3B4FULL	Interrupt when receive buffer is 3/4 full
USART_RECEIVE_FIFO_ONE_CHAR	Interrupt when a character is received

La taille du tampon hardware de réception est de 8 caractères, donc avec HALF_FULL on déclenche l'interruption dès la réception de 4 caractères. Pour 3B4FULL c'est lorsque l'on a reçu les $\frac{3}{4}$ soit 6 caractères. Avec ONE_CHAR l'interruption est déclenchée à chaque caractère.

Le choix de déclencher l'interruption de réception lorsque le tampon de réception est à moitié plein (donc 4 caractères) aura un comportement particulier avec les paquets de 6 caractères de nos messages.

A l'origine, le code généré contenait le mode USART_RECEIVE_FIFO_ONE_CHAR. Le configurateur graphique ne permettant pas de modifier ce choix, il faut le faire manuellement dans le code.

7.13.4. SECTION RÉCEPTION DE L'INTERRUPTION DE L'USART

Voici la section réception de l'interruption de l'USART :

```

if ( PLIB_INT_SourceFlagGet(INT_ID_0,
                            INT_SOURCE_USART_1_RECEIVE) &&
    PLIB_INT_SourceIsEnabled(INT_ID_0,
                            INT_SOURCE_USART_1_RECEIVE) ) {
    BSP_LEDToggle(BSP_LED_3); // marque int RX
    // Oui, test si erreur parité ou overrun
    UsartStatus = PLIB_USART_ErrorsGet(USART_ID_1);

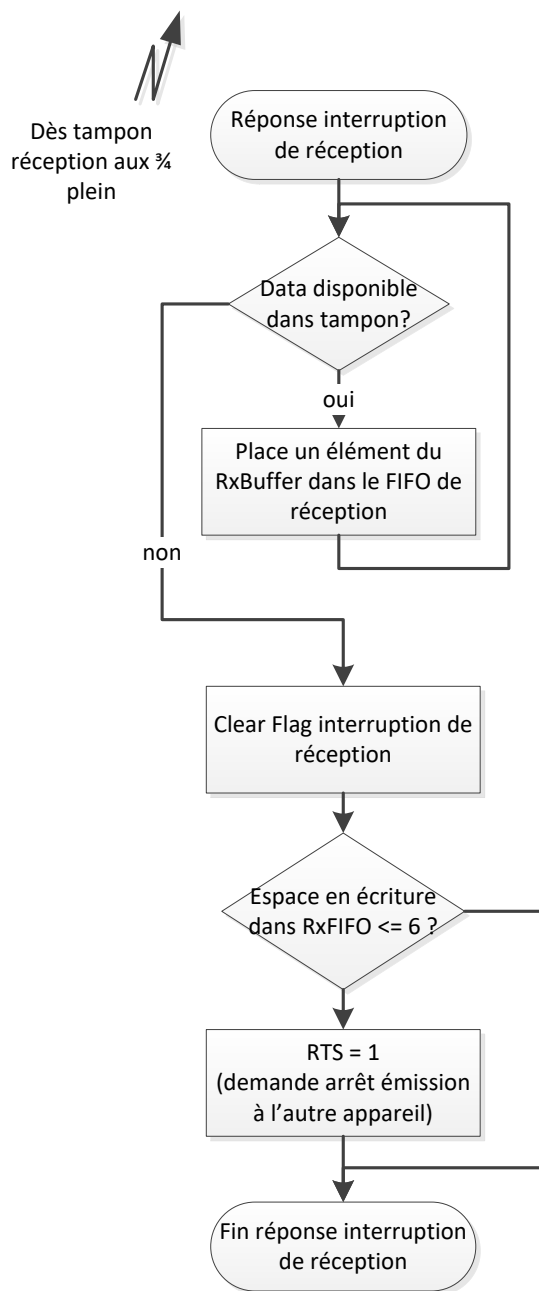
    if ( (UsartStatus & (USART_ERROR_PARITY |
                        USART_ERROR_FRAMING |
                        USART_ERROR_RECEIVER_OVERRUN)) == 0) {
        // Transfert dans le FIFO de tous les chars reçus
        // 1 Si ONE_CHAR, 4 si HALF_FULL et 6 3B4FULL
        while (PLIB_USART_ReceiverDataIsAvailable
                (USART_ID_1))
        {
            c = PLIB_USART_ReceiverByteReceive(USART_ID_1);
            PutCharInFifo (&descrFifoRX, c);
            BSP_LEDToggle(BSP_LED_5); // pour comptage
        }
        // buffer is empty, clear interrupt flag
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                INT_SOURCE_USART_1_RECEIVE);
    } else {
        // La lecture des erreurs les efface
        // sauf pour overrun
        if ((UsartStatus & USART_ERROR_RECEIVER_OVERRUN)
            == USART_ERROR_RECEIVER_OVERRUN) {
            PLIB_USART_ReceiverOverrunErrorClear(USART_ID_1);
        }
    }

    // Traitement du contrôle de flux
    freeSize = GetWriteSpace (&descrFifoRX);
    if (freeSize <= 6) // a cause d'un int pour 6 char
    {
        // Demande de ne plus émettre
        RS232_RTS = 1;

        if (freeSize == 0) {
            ErrFifoFull = 1; // pour debugging
        }
    }
} // end if RX
    
```

7.13.5. PRINCIPE TRAITEMENT INTERRUPTION DE RÉCEPTION

Le principe du traitement de l'interruption est le suivant :



Comme la gestion du contrôle de flux est supposée garantir que le FIFO de réception permette l'enregistrement de 6 caractères, il est possible de réaliser une boucle de copie du tampon de réception HW dans le RxFifo software sans test de la situation du FIFO soft.

La situation du FIFO en écriture est testée après. Si la place disponible est \leq à 6 alors il faut activer RTS à high pour demander l'arrêt de l'émission.

Cet organigramme et les quelques suivants incluent une gestion du contrôle de flux par software. Une gestion automatique par hardware est également possible.

Le traitement du contrôle de flux dans la réponse à l'interruption, qui consiste à imposer $RTS = 1$ pour demander à l'autre appareil de stopper l'émission, implique que le RTS doit être remis à 0 **hors de l'interruption**, lorsqu'il y a à nouveau suffisamment de place dans le FIFO software.

👉 La valeur de 6 est liée au réglage $\frac{3}{4}$ full de l'interruption de réception, cela correspond aussi à la taille d'un message.

7.13.6. COMPORTEMENT INTERRUPTION RX AVEC HALF_FULL

Voici l'observation du comportement de l'interruption de réception avec le réglage HALF_FULL. Obtenu par :

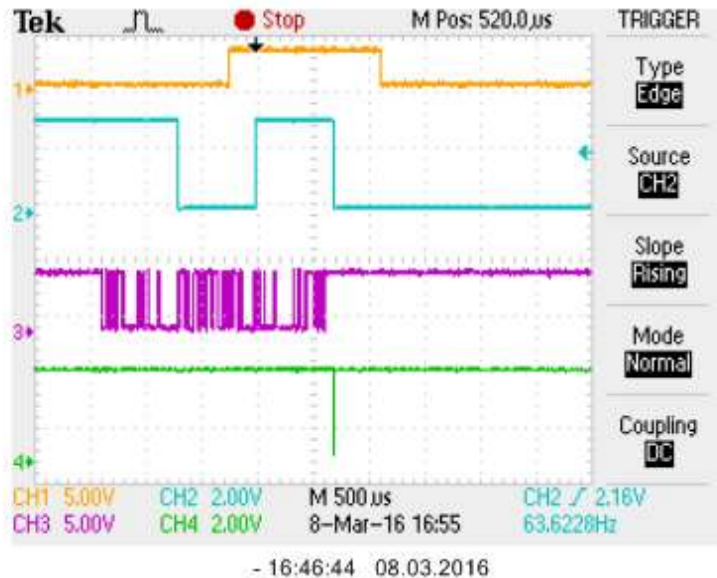
```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_HALF_FULL,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
```

canal 1: LED_0
(marque traitement
dans l'application)

canal 2: LED_3
(inversion à chaque
int RX)

canal 3 : broche 52
U1RX

canal 4: LED_5
(inversion dans
boucle copie)



On constate 3 interruptions pour le même paquet de 12 caractères fournis par l'application Windows.

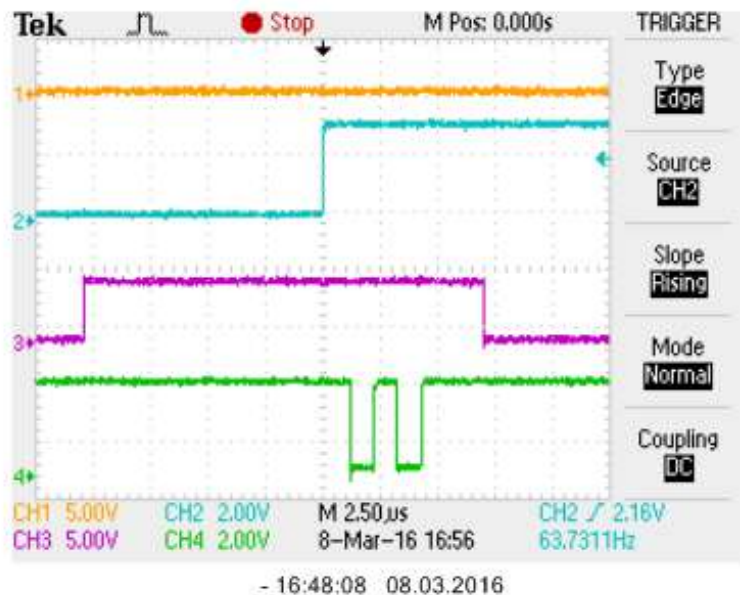
Et avec l'observation détaillée :

canal 1: LED_0
(marque traitement
dans l'application)

canal 2: LED_3
(inversion à chaque
int RX)

canal 3 : broche 52
U1RX

canal 4: LED_5
(inversion dans
boucle copie)



Et on observe bien les 4 transitions indiquant l'extraction de 4 caractères du tampon de réception de l'USART.

7.13.7. COMPORTEMENT INTERRUPTION RX AVEC 3B4FULL

Voici l'observation du comportement de l'interruption de réception avec le réglage 3B4_FULL. Obtenu par :

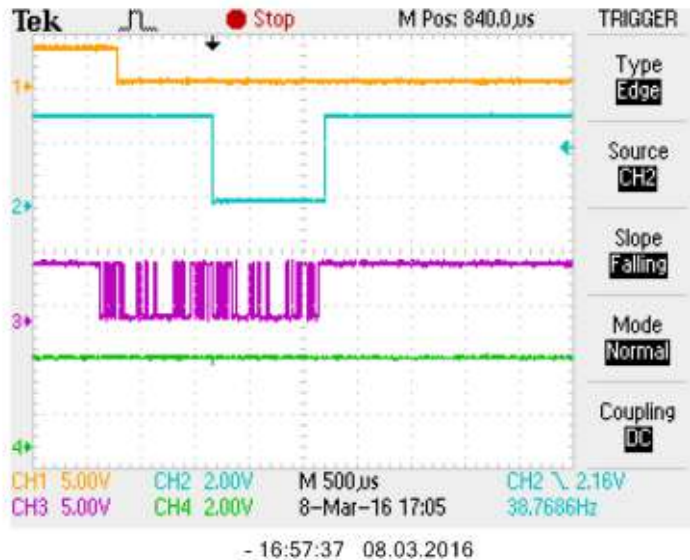
```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_3B4FULL,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
```

canal 1 : LED_0
(marque traitement
dans l'application)

canal 2 : LED_3
(inversion à chaque
int. RX)

canal 3 : broche 52
U1RX

canal 4 : LED_5
(inversion dans
boucle copie)



On constate 2 interruptions pour le même paquet de 12 caractères fournis par l'application Windows.

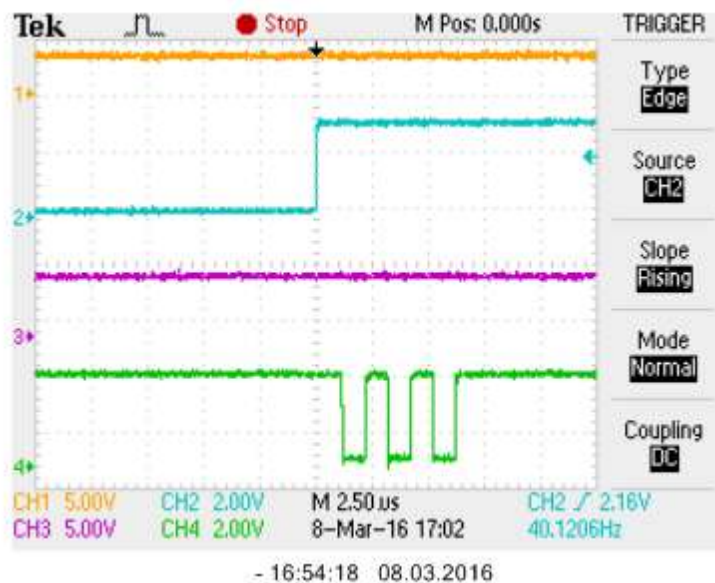
Et avec l'observation détaillée :

canal 1 : LED_0
(marque traitement
dans l'application)

canal 2 : LED_3
(inversion à chaque
int RX)

canal 3 : broche 52
U1RX

canal 4 : LED_5
(inversion dans
boucle copie)



Et on observe bien les 6 transitions indiquant l'extraction de 6 caractères du tampon de réception de l'USART.

7.13.8. COMPORTEMENT INTERRUPTION RX AVEC ONE_CHAR

Voici l'observation du comportement de l'interruption de réception avec le réglage ONE_CHAR. Obtenu par :

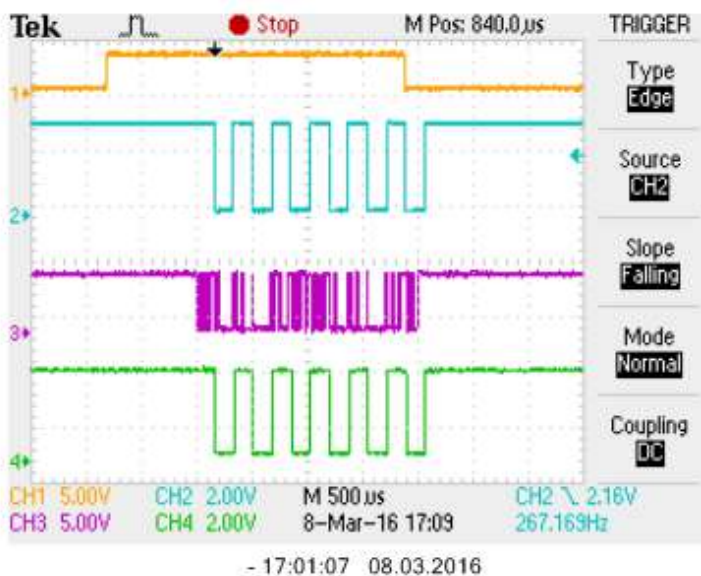
```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_ONE_CHAR,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
```

canal 1: LED_0
(marque traitement
dans l'application)

canal 2: LED_3
(inversion à chaque
int RX)

canal 3 : broche 52
U1RX

canal 4: LED_5
(inversion dans
boucle copie)



On constate 12 interruptions pour le même paquet de 12 caractères fournis par l'application Windows.

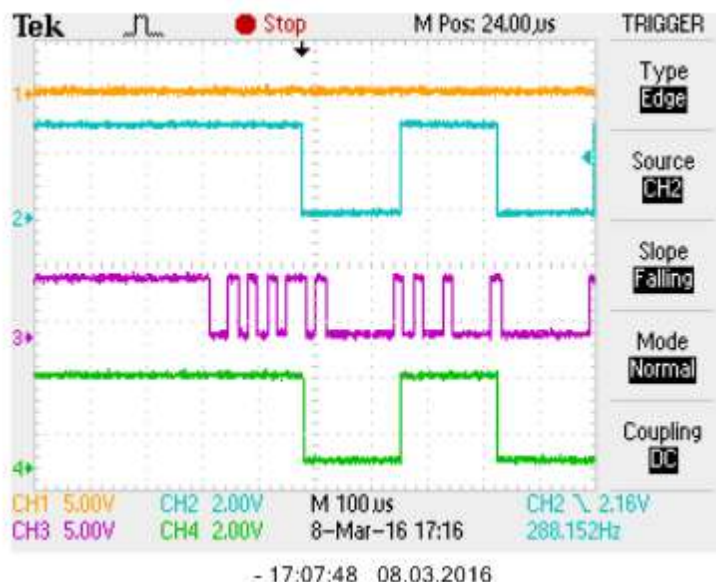
Et avec l'observation détaillée :

canal 1: LED_0
(marque traitement
dans l'application)

canal 2: LED_3
(inversion à chaque
int RX)

canal 3 : broche 52
U1RX

canal 4: LED_5
(inversion dans
boucle copie)



Et on observe qu'après la reception d'un seul caractère, on obtient une interruption et une action de copie du tampon de réception de l'USART.

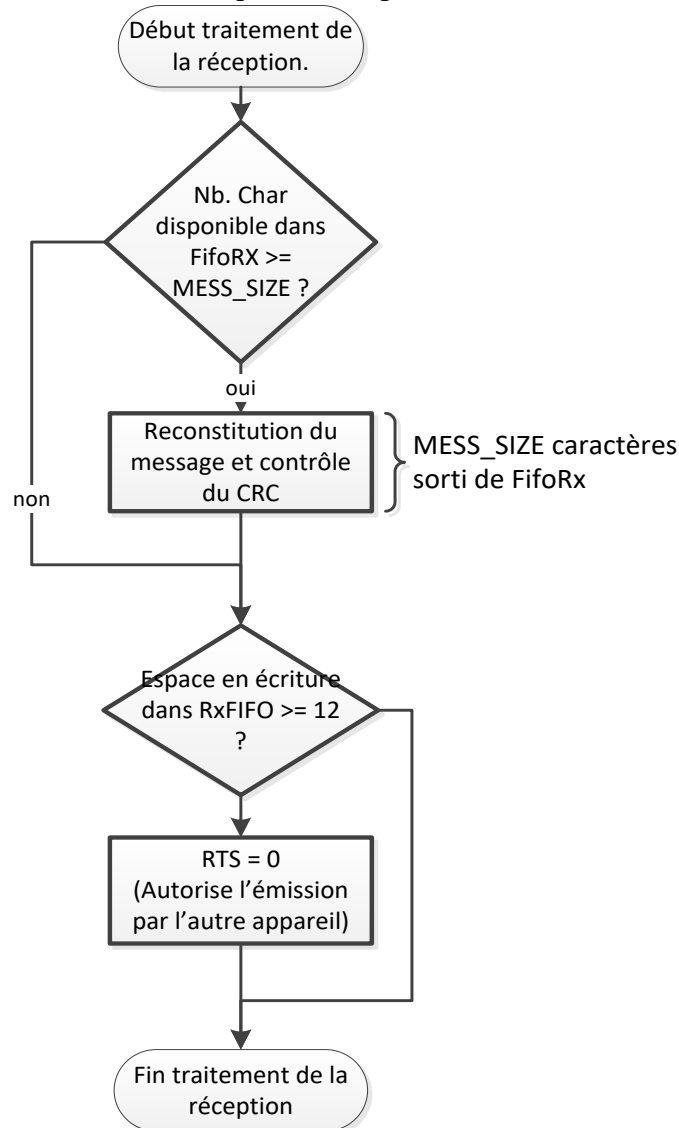
☹ Ce mode est peu performant; il génère inutilement des interruptions. Mais il peut s'avérer nécessaire suivant le cas de figure de transmission.

7.13.9. PRINCIPE DU TRAITEMENT DE LA RÉCEPTION (GETMESSAGE)

Dans le traitement de la réception au niveau de la partie application (fonction GetMessage), il y a deux aspects :

- La gestion du FIFO software (disponibilité d'un message et à l'opposé place disponible pour les nouveaux messages, ceci en relation avec la gestion du control de flux).
- La reconstitution de message et le contrôle de son CRC.

Dans ce qui suit, nous ne traiterons que le 1^{er} aspect.



7.13.9.1. TEST DISPONIBILITÉ DU MESSAGE DANS RxFIFO

Pour faciliter la reconstitution du message, on teste la situation de remplissage du Rx FIFO. Si le FIFO contient au minimum le nb de caractères correspondant à la taille du message, alors on essaie de traiter le message. Cela permet une lecture du FIFO en boucle sans nécessité de tester la situation du FIFO à chaque appel de **GetCharFromFifo**.

Voici le code correspondant :

```
// Détermine le nombre de caractères à lire
NbCharToRead = GetReadSize ( &descrFifoRX);

// Si >= taille message alors traite
if (NbCharToRead >= MESS_SIZE) {
    EndMess = 0;
    while ( (NbCharToRead >= 1) && ( EndMess == 0) ) {
        // Lis un caractère sans gestion du status
        GetCharFromFifo ( &descrFifoRX, &RxC );
        NbCharToRead--;
        // Traitement selon situation du message
        switch (GetSituation) {
```

7.13.9.2. TRAITEMENT DU CONTRÔLE DE FLUX EN RÉCEPTION

Il faut se souvenir que dans la réponse à l'interruption, on bloque l'émission par l'autre appareil via RTS = 1.

Si on n'agit pas pour rétablir l'émission (RTS = 0) l'autre appareil n'émet plus. Par contre, il est nécessaire de rétablir l'émission lorsque l'on dispose d'un peu de marge d'où le $12 = 2 * 6$. Pour autoriser l'émission lorsqu'on a la place pour 2 paquets d'une taille de 6 octets chacun.

👉 La valeur de 6 est liée au réglage $\frac{3}{4}$ full de l'interruption de réception qui correspond par hasard à la taille du message.

Voici le code correspondant:

```
// Test si place en écriture pour 2 paquets de 6 char
if(GetWriteSpace ( &descrFifoRX) >= 12) {
    // autorise émission par l'autre
    RS232_RTS = 0;
}
```


7.13.10. VÉRIFICATION CONTRÔLE DE FLUX DE RÉCEPTION

Pour vérifier la gestion de la ligne RTS, nous devons créer une situation où le débit des messages émis dépasse celui de réception, idéalement par à-coup. Obtenu en modifiant l'application pour un envoi un cycle sur 3 de 4 messages.

Nous observons la ligne RTS en relation avec l'interruption de réception ainsi que la broche RX. Nous conservons l'observation de la LED_0 pour le cycle d'exécution de l'application.

👉 Observation en 3B4FULL pour l'interruption de réception.

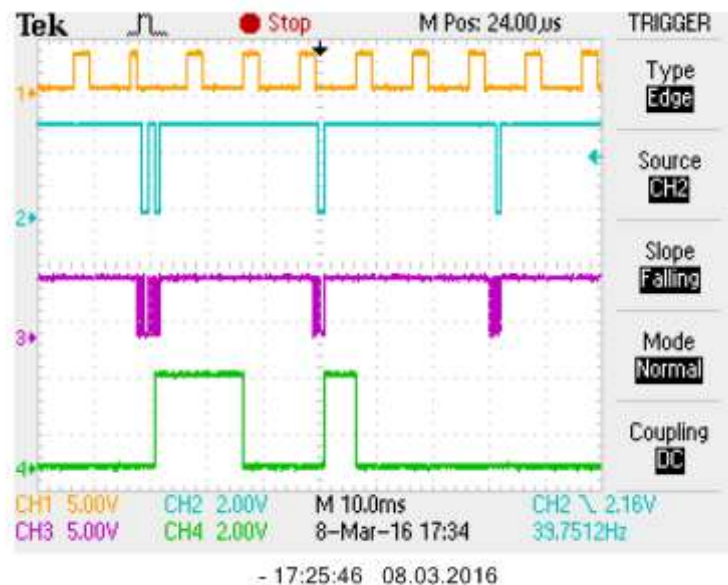
7.13.10.1. VUE D'ENSEMBLE CONTRÔLE FLUX RÉCEPTION

canal 1 : LED_0
(marque traitement dans l'application)

canal 2 : LED_3
(inversion à chaque int RX)

canal 3 : broche 52
U1RX

canal 4:
RS232_RTS
(broche 48)



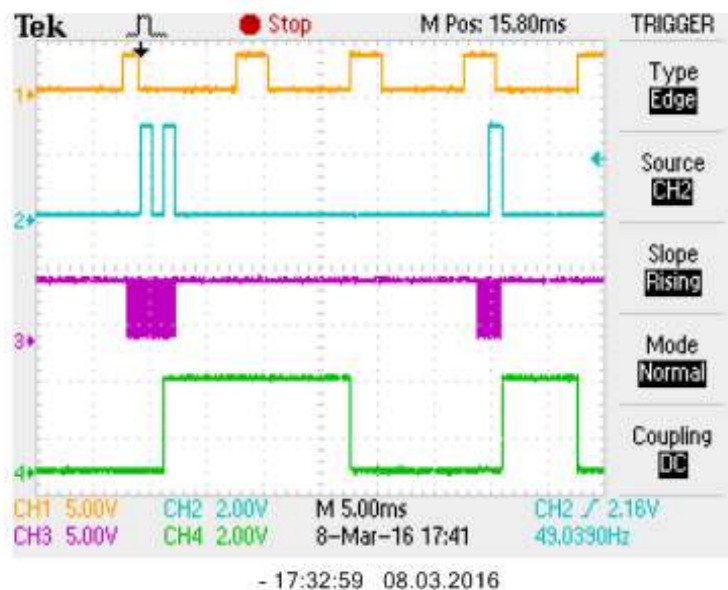
7.13.10.1. VUE DE DÉTAIL CONTRÔLE FLUX RÉCEPTION

canal 1 : LED_0
(marque traitement dans l'application)

canal 2 : LED_3
(inversion à chaque int RX)

canal 3 : broche 52
U1RX

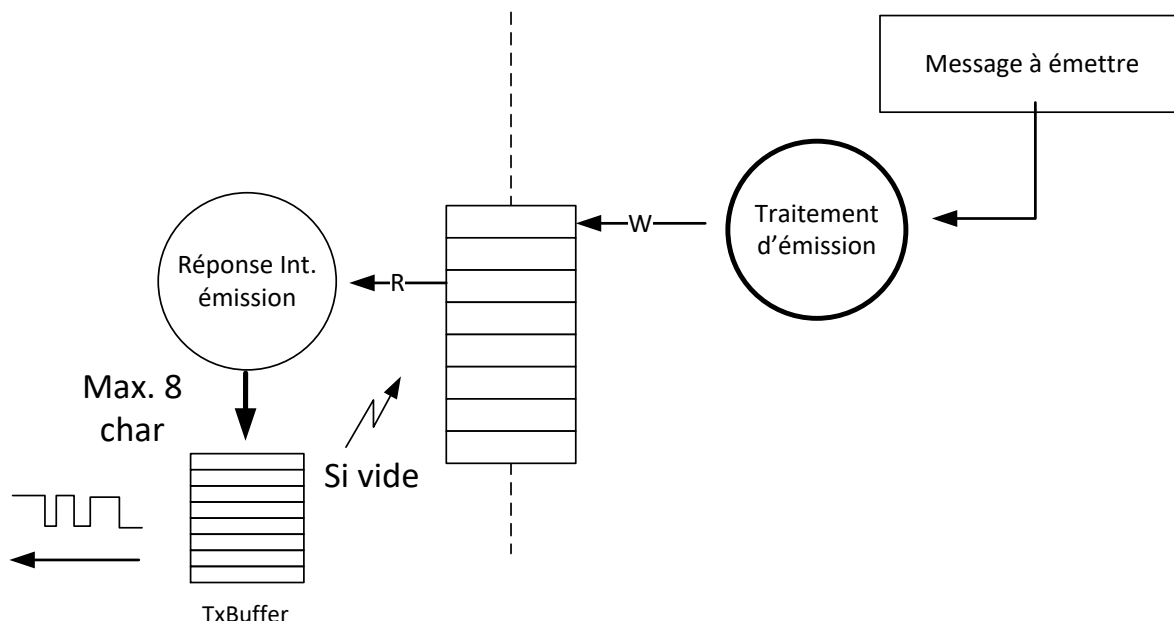
canal 4:
RS232_RTS
(broche 48)



On peut observer que suite à l'arrivée du bloc de 4 messages, la ligne RTS est mise à 1 et qu'elle n'est remise à 0 qu'après 2 cycles de traitement de l'application.

7.13.11. CONTEXTE D'ÉMISSION

Le diagramme ci-dessous décrit le contexte d'émission.



En émission, nous obtenons une interruption d'émission lorsque le tampon d'émission (TxBuffer) est vide. Il est possible de le remplir jusqu'à ce qu'il soit plein, soit un maximum de 8 caractères.

7.13.12. DÉTAIL DE LA CONFIGURATION DE L'INTERRUPTION D'ÉMISSION

Voici les éléments de la configuration de l'USART qui concernent l'interruption d'émission :

```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_HALF_FULL,
    USART_TRANSMIT_FIFO_EMPTY,
    USART_ENABLE_TX_RX_USED);
...
SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_TRANSMIT);
```

Et après l'appel de la fonction DRV_USART0_Initialize, on trouve :

```
SYS_INT_VectorPrioritySet(INT_VECTOR_UART1,
    INT_PRIORITY_LEVEL5);
SYS_INT_VectorSubprioritySet(INT_VECTOR_UART1,
    INT_SUBPRIORITY_LEVEL0);
```

Le code généré n'active pas l'interruption d'émission, ce qui est logique. Cela devra être fait par l'application lorsqu'une émission sera souhaitée.

Concernant le comportement de l'interruption d'émission selon le niveau de remplissage du buffer, les choix disponibles sont :

Members	Description
USART_TRANSMIT_FIFO_EMPTY	Interrupt when the transmit buffer becomes empty
USART_TRANSMIT_FIFO_IDLE	Interrupt when all characters are transmitted
USART_TRANSMIT_FIFO_NOT_FULL	Interrupt when at least one location is empty in the transmit buffer

Tout comme pour l'interruption de réception, le configurateur graphique ne permet pas ce choix. Il faut le faire manuellement dans le code. A l'origine, le code généré contenait le mode USART_TRANSMIT_FIFO_IDLE.

7.13.13. RÉPONSE À L'INTERRUPTION D'ÉMISSION

L'interruption d'émission a été configurée pour se produire lorsque le tampon d'émission est vide. Dans cette situation il est possible de réaliser une boucle de transfert entre le FIFO d'émission software et le tampon hardware d'émission.

```
// Is this an TX interrupt ?
if ( PLIB_INT_SourceFlagGet(INT_ID_0,
                             INT_SOURCE_USART_1_TRANSMIT) &&
     PLIB_INT_SourceIsEnabled(INT_ID_0,
                              INT_SOURCE_USART_1_TRANSMIT) ) {

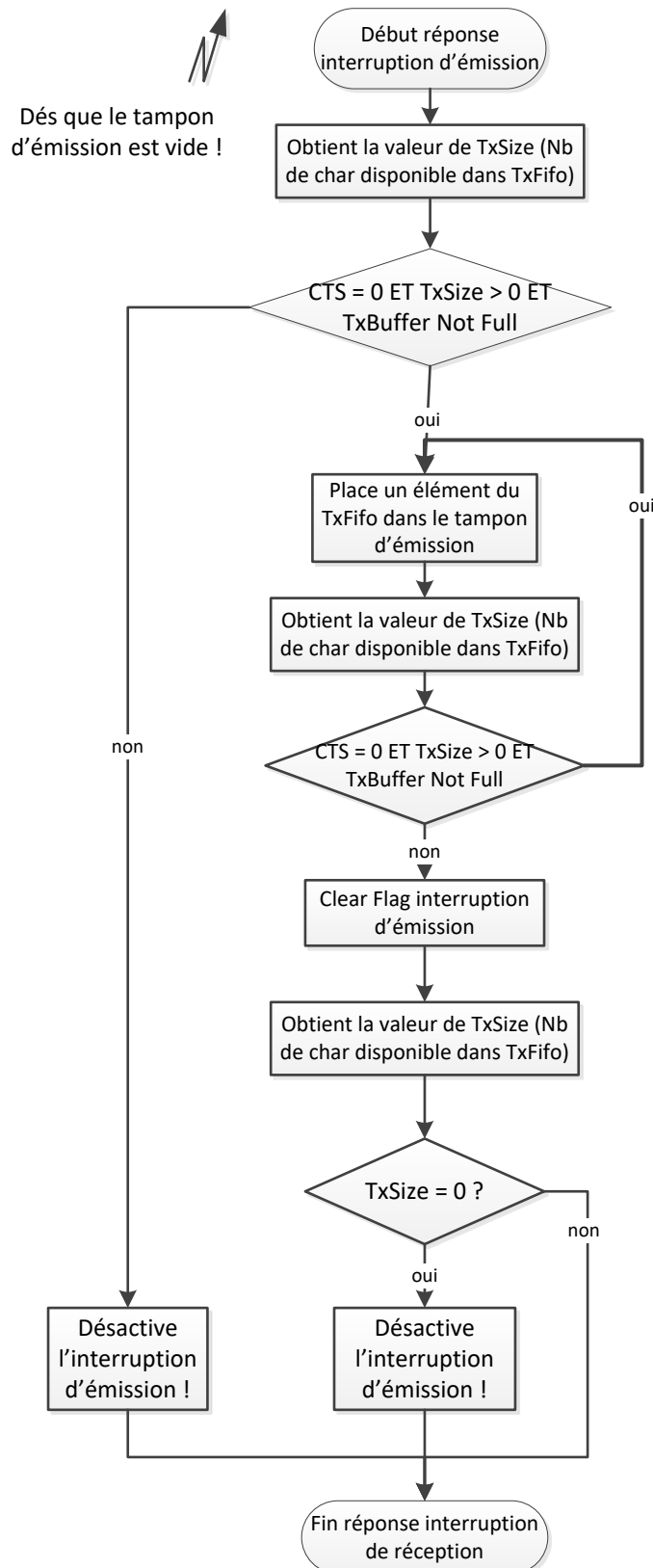
    BSP_LEDToggle(BSP_LED_4); // marque int TX

    // On vérifie 3 conditions :
    //     Si CTS = 0 (autorisation d'émettre)
    //     Si il y a des caractères à émettre
    //     Si le TxBuffer n'est pas plein
    i_cts = RS232_CTS;
    TXsize = GetReadSize (&descrFifoTX);
    TxBuffFull =
        PLIB_USART_TransmitterBufferIsFull(USART_ID_1);
    if ( (i_cts == 0) && ( TXsize > 0 ) &&
        TxBuffFull == false ) {
        do {
            GetCharFromFifo(&descrFifoTX, &c);
            PLIB_USART_TransmitterByteSend(USART_ID_1, c);
            BSP_LEDToggle(BSP_LED_6); // pour comptage
            i_cts = RS232_CTS;
            TXsize = GetReadSize (&descrFifoTX);
            TxBuffFull =
                PLIB_USART_TransmitterBufferIsFull(USART_ID_1);
        } while ( (i_cts == 0) && ( TXsize > 0 ) &&
                 TxBuffFull == false );

        // Clear the TX interrupt Flag
        // (Seulement après TX)
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                 INT_SOURCE_USART_1_TRANSMIT);
        TXsize = GetReadSize (&descrFifoTX);
        if (TXsize == 0 ) {
            // pour éviter une interruption inutile
            PLIB_INT_SourceDisable(INT_ID_0,
                                   INT_SOURCE_USART_1_TRANSMIT);
        }
    } else {
        // disable TX interrupt
        PLIB_INT_SourceDisable(INT_ID_0,
                               INT_SOURCE_USART_1_TRANSMIT);
    }
}
```

7.13.14. PRINCIPE DE L'INTERRUPTION D'ÉMISSION

Voici un diagramme qui décrit le principe de l'interruption d'émission.



Pour émettre un caractère, il faut que CTS=0 (autorisation d'émettre) et qu'il y ait des caractères dans le FIFO. Il faut également que le tampon d'émission ne soit pas plein.

Il est possible de réaliser une boucle prenant un caractère du FIFO d'émission (TxFifo) et le déposant dans le tampon d'émission (TxBuffer).

Dès qu'une des conditions n'est plus vraie (TxBuffer full ou plus de caractère dans le TxFifo), on quitte la boucle.

Le flag d'interruption ne doit être mis à 0 que lorsque que l'on a rempli le TxBuffer.

Pour éviter une interruption inutile, il faut désactiver l'interruption d'émission si le FIFO d'émission est vide.

Si les conditions ne permettent pas de placer des caractères dans le TxBuffer, il faut impérativement désactiver l'interruption. Ceci évite d'avoir en permanence l'interruption active alors que l'on n'a rien à émettre.

7.13.14.1. GESTION DU CONTROLE DE FLUX DANS INTERRUPTION D'ÉMISSION

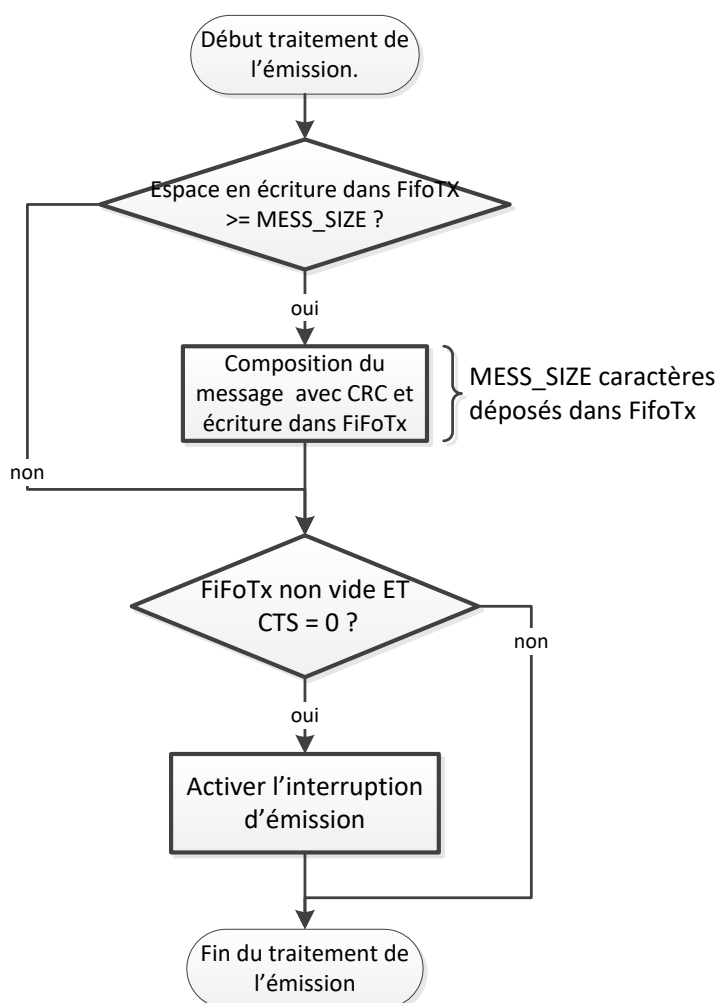
Le traitement du contrôle de flux dans la réponse à l'interruption, qui consiste à inhiber l'interruption d'émission lorsque CTS n'est pas égal à 0, implique que l'interruption d'émission doit être autorisée à nouveau **hors de l'interruption**, lorsqu'il y a des caractères dans le FIFO d'émission et que la ligne CTS vaut 0.

7.13.15. PRINCIPE DU TRAITEMENT DE L'ÉMISSION (SENDMESSAGE)

Dans le traitement de l'émission, au niveau de la partie application (fonction SendMessage), il y a deux aspects :

- La gestion du FIFO software d'émission (place pour déposer un message et à l'opposé contenu à émettre, ceci en relation avec la gestion du contrôle de flux).
- La composition du message et le calcul de son CRC.

Dans ce qui suit, nous ne traiterons que le 1^{er} aspect.



S'il y a assez de place dans le FIFO d'émission, on y dépose un message complet.

Après la tentative d'écriture dans le FIFO, si le nombre de caractères placé dans le FIFO est > 0 ET que CTS = 0, il y a autorisation de l'interruption d'émission, ce qui permettra l'émission des caractères.

7.13.15.1. CODE PARTIEL DU MÉCANISME D'ÉMISSION

Voici le code partiel du mécanisme d'émission au niveau application (SendMessage) :

```
// Test si place pour écrire 1 message
FreeSize = GetWriteSpace ( &descrFifoTX);
if (FreeSize >= (MESS_SIZE) ) {

    // Compose le message
    // Ajoute le CRC au message
    // Dépose le message dans le FIFO d'émission
}
// Gestion du control de flux
// Si on a un caractère à envoyer et que CTS = 0
FreeSize = GetReadSize(&descrFifoTX);
if ((RS232_CTS == 0) && (FreeSize > 0))
{
    // Autorise int émission
    PLIB_INT_SourceEnable(INT_ID_0,
                          INT_SOURCE_USART_1_TRANSMIT);
}
}
```

7.13.16. OBSERVATION DE L'ÉMISSION**7.13.16.1. VUE D'ENSEMBLE**

La situation est la suivante :

canal 1 : LED_0
(marque traitement
dans l'application)

canal 2 : LED_4
(inversion dans
réponse à
l'interruption TX)

canal 3 : broche 53
UITX



On peut observer qu'à la fin du traitement dans l'application, on voit apparaître une interruption de transmission et que sur la broche TX on voit apparaître la transmission des caractères.

Le message émis étant de 6 caractères et le tampon d'émission ayant une taille de 8, nous avons une seule interruption à l'occasion de laquelle il est possible de placer les 6 caractères dans le tampon.

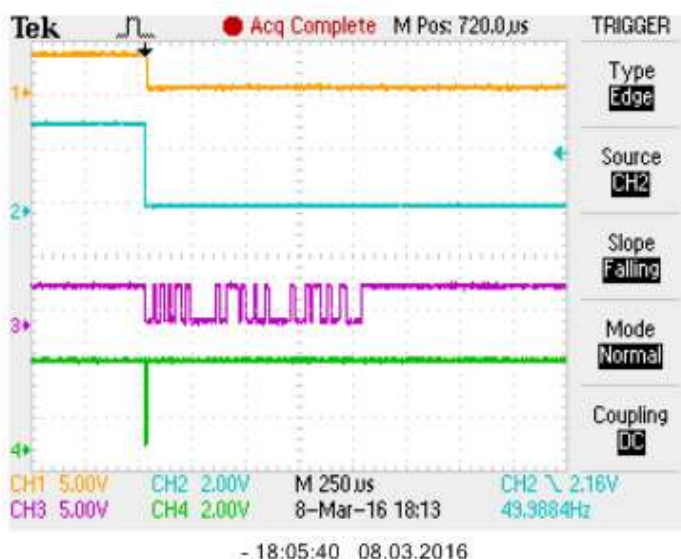
7.13.16.2. VUE DE DÉTAIL

canal 1 : LED_0
(marque traitement dans l'application)

canal 2 : LED_4
(inversion dans réponse à l'interruption TX)

canal 3 : broche 53 U1TX

canal 4 : LED_6
(inversion dans réponse à l'interruption, dans la boucle de dépôt dans le tampon d'émission)



7.13.16.3. CHECK DES 6 CARACTÈRES

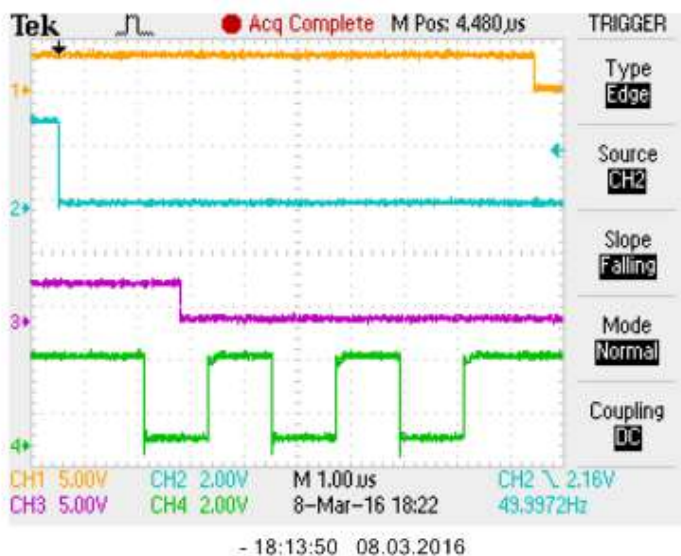
On vérifie que l'on a bien transféré 6 caractères dans le tampon hardware.

canal 1 : LED_0
(marque traitement dans l'application)

canal 2 : LED_4
(inversion dans réponse à l'interruption)

canal 3 : broche 53 U1TX

canal 4 : LED_6
(inversion dans réponse à l'interruption, dans la boucle de dépôt dans le tampon d'émission)



7.13.17. VÉRIFICATION DU CONTRÔLE DE FLUX À L'ÉMISSION

Nous allons vérifier si notre système cesse d'émettre lorsque l'entrée CTS n'est plus au niveau bas. Pour vérifier cela, nous devons créer une situation avec une émission par salves (par exemple en appelant 2 fois SendMessage en plus tous les 3 cycles).

Nous observons la ligne CTS en relation avec l'interruption d'émission ainsi que la broche TX. Nous conservons l'observation de la LED_0 pour le cycle d'exécution de l'application.

7.13.17.1. VUE SANS ACTION DU CONTRÔLE FLUX

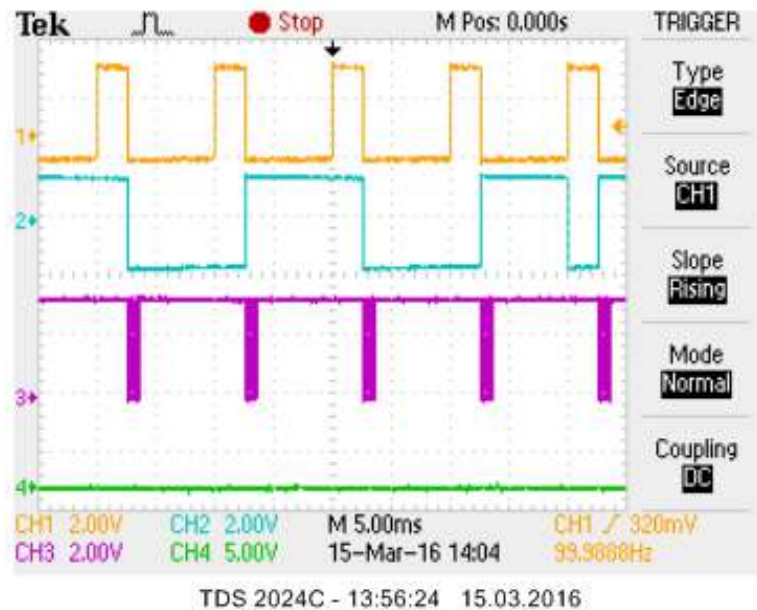
Situation sans l'envoi de trame supplémentaire.

canal 1 : LED_0
(marque
traitement dans
l'application)

canal 2 : LED_4
(inversion dans
réponse à
l'interruption TX)

canal 3 : broche
53 U1TX

canal 4:
RS232_CTS
(broche 47)



A chaque cycle application (10 ms), envoi d'un message. Le mécanisme de réception supporte ce débit.

7.13.17.2. VUE D'ENSEMBLE CONTRÔLE FLUX D'ÉMISSION

Avec l'ajout de trames supplémentaires, on aboutit à une situation d'accumulation des messages à envoyer. On constate l'envoi d'un message par cycle application mais avec 2 interruptions et d'un cycle sans envoi à cause du CTS à 1.

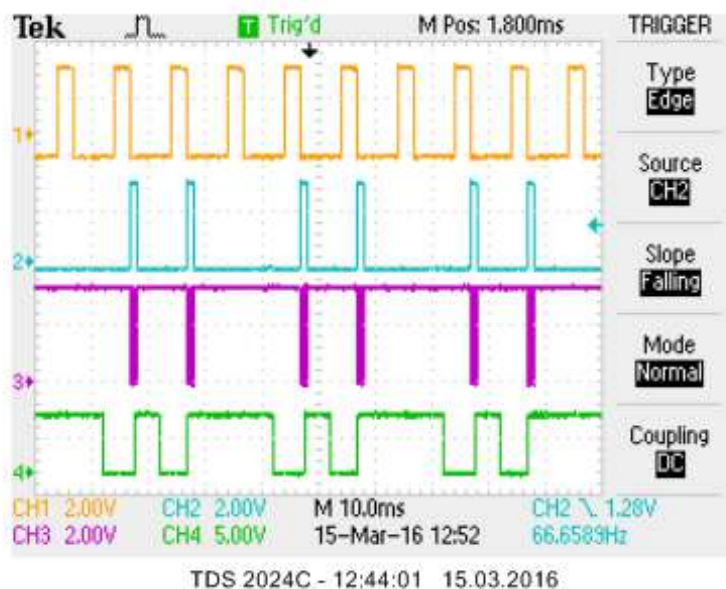
Il est possible d'observer la situation du signal CTS, donc la réaction du système récepteur.

canal 1: LED_0
(marque traitement dans l'application)

canal 2: LED_4
(inversion dans réponse à l'interruption TX)

canal 3 : broche 53
U1TX

canal 4:
RS232_CTS
(broche 47)



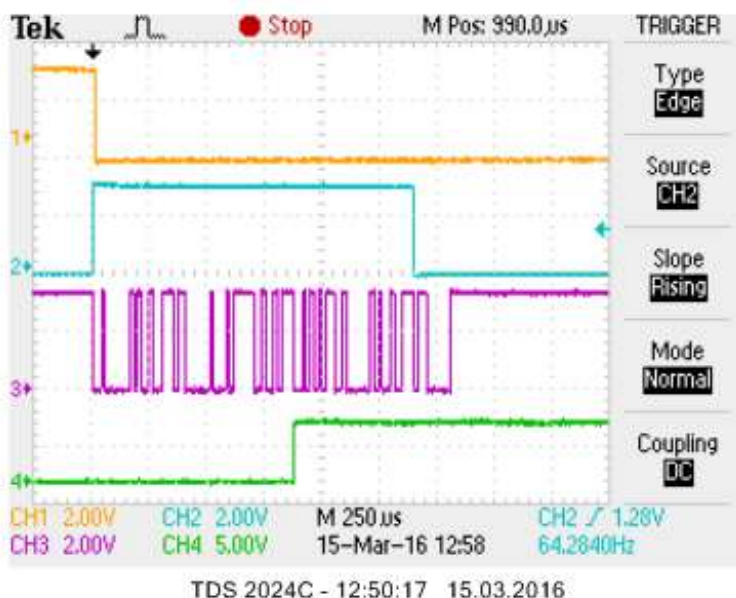
7.13.17.1. VUE DE DÉTAIL DU CONTRÔLE FLUX D'ÉMISSION

canal 1: LED_0
(marque traitement dans l'application)

canal 2: LED_4
(inversion dans réponse à l'interruption TX)

canal 3 : broche 53
U1TX

canal 4:
RS232_CTS
(broche 47)



On peut observer que dès que le signal CTS passe à 1, la 2^{ème} interruption d'émission ne produit plus de transmission de caractères.

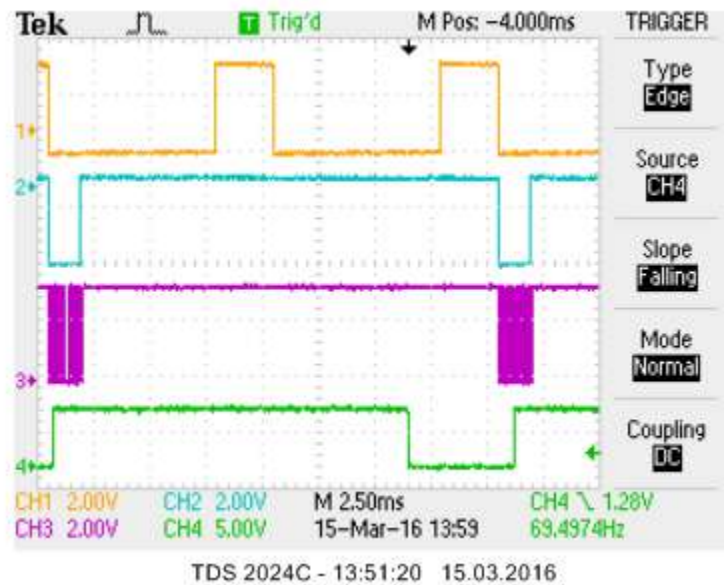
7.13.17.2. VUE DE LA REPRISE DE L'ÉMISSION

canal 1: LED_0
(marque traitement
dans l'application)

canal 2: LED_4
(inversion dans
réponse à
l'interruption TX)

canal 3 : broche 53
UITX

canal 4:
RS232_CTS
(broche 47)



On observe une situation avec 1 cycle application ne produisant pas de transmission car CTS = 1, puis la réapparition de l'émission qui est à nouveau stoppée.

7.14. STRUCTURE D'UN MESSAGE

La structure d'un message n'est pas forcément de taille fixe. Bien souvent, on est amené à créer des messages de taille variable. Et si la fiabilité de la liaison n'est pas bonne (par exemple transmission sans fil), on ajoutera au message une somme de contrôle (Checksum) ou un CRC.

Pour faciliter le l'analyse du message (appelé *parsing* en anglais), on ajoute un caractère de début du message STX et un caractère de fin du message ETX. Un champ du message est chargé d'indiquer la longueur des données utiles.

Exemple de structure :

STX	LEN	DATA	CRC	ETX
-----	-----	------	-----	-----

Lors de la réception d'un tel message, il est nécessaire d'attendre le STX, puis le byte de longueur (LEN). Ensuite il est nécessaire de lire les N bytes de DATA, N étant donné par la valeur du champ LEN.

Ensuite, s'il s'agit d'une valeur de CRC sur 16 bits, il y aura lecture des deux caractères donnant la valeur du CRC. En général le CRC est calculé au fur et à mesure de la lecture. La valeur calculée est comparée à la valeur fournie dans le message ou si on a pris certaines précautions, la valeur calculée y compris le CRC doit valoir zéro. Il est important de définir si la valeur de CRC correspond seulement à la zone DATA ou si on calcule le CRC en incluant STX + LEN + DATA

Si la comparaison échoue, on en déduit que les données reçues ont été altérées, et le message ne sera pas pris en compte.

Pour les messages au contenu binaire, il n'est pas très utile de disposer d'un caractère de fin de message qu'il serait nécessaire de lire après le traitement du CRC.

7.14.1. DONNÉES BINAIRES OU ASCII

Pour obtenir un message plus court ou transmettre directement la valeur d'une variable (8, 16 ou 32 bits), il est fréquent de transmettre la valeur binaire. Un des inconvénients est que chaque octet peut prendre n'importe quelle valeur, d'où parfois des risques de confondre un byte de donnée avec le début du message. Ce risque est important si un message a été mal reçu ou fragmenté.

L'autre option est de transmettre les valeurs des variables sous forme de caractères ASCII. Une variable 16 bits dont la valeur hexadécimale est 0x1234 demandera l'envoi de 4 caractères (1, 2, 3 et 4). Par contre ce type de message est facile à observer et évite la confusion entre un caractère de donnée et un caractère spécial comme STX ou ETX.

7.14.2. CALCUL D'UN CRC 16 BITS

Le calcul d'un CRC est un calcul successif, chaque caractère du message est utilisé pour cumuler la valeur du CRC. Il est possible d'utiliser une table pour obtenir rapidement la valeur à cumuler.

Avant de passer à un exemple pratique, il est nécessaire de revoir le principe du calcul d'un CRC.

7.15. PRINCIPE DU CALCUL D'UN CRC

Le CRC (Cyclic Redundancy Codes) est une évolution du principe d'une simple somme de contrôle (Checksum). Une somme de contrôle consiste à ajouter successivement la valeur des caractères (8 bits) d'un message à une variable 16 ou 32 bits en ne se préoccupant pas des reports si la somme dépasse la taille prévue.

Si on effectue une simple sommation sur 8 bits, pour les 2 messages ci-dessous :

- Pour la chaîne "Bonjour Papa" : La somme est 0x81.
- Pour la chaîne "Banjour Papo" : La somme est 0x81.

Le message n'est pas le même, mais le calcul de la somme donne la même valeur. Dans cet exemple, le deuxième message contient exactement les mêmes caractères, mais dans un ordre différent, ce qui produit la même somme. C'est cette faiblesse qui conduit à augmenter la complexité du calcul.

L'idée fondamentale des algorithmes de CRC est simplement de traiter le message comme une grande valeur numérique, de le diviser par une valeur fixe, de prendre le reste de la division en tant que somme de contrôle.

La valeur numérique du message est bien supérieure à la valeur maximum d'un mot de 32 bits de large. Nous allons donc faire la division Byte par Byte mais cette fois en binaire. Le calcul bit à bit se fait par un "ou exclusif" (encore appelé XOR).

Ce principe peut être simplifié et adapté pour aboutir à l'usage d'une table utilisée pour calculer chaque étape du CRC.

La table doit être construite en fonction du polynôme utilisé pour le calcul du CRC.

Sans entrer dans les détails, il paraît important de mentionner à quoi correspondent les polynômes utilisés pour le calcul des CRC.

7.15.1. LES POLYNÔMES UTILISÉS POUR LE CALCUL DU CRC

Les polynômes utilisés pour le calcul du CRC sont des polynômes binaires. En voici quelques-uns couramment utilisés :

- Le polynôme CRC16-CCITT correspond à $x^{16} + x^{12} + x^5 + x^0$.
Si on considère $x=2$, on peut calculer la valeur du polynôme soit $65536 + 4096 + 32 + 1 = 69665$, soit 0x11021. La valeur utilisée en pratique est 0x1021, donc sans le x^{16} .
- Le polynôme CRC16 correspond à $x^{16} + x^{15} + x^2 + x^0$.
Si on considère $x=2$, on peut calculer la valeur du polynôme soit $65536 + 32768 + 4 + 1 = 98309$, soit 0x18005. La valeur utilisée en pratique est 0x8005, donc sans le x^{16} .
- Le polynôme CRC32 pour les trames Ethernet correspond à :
 $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0$.

7.15.2. VARIÉTÉ D'ALGORITHMES

De par ces différents polynômes, il existe donc différents algorithmes de calcul de CRC, qui donneront chacun un CRC différent pour des mêmes données. Mais il peut y avoir encore d'autres différences entre les algorithmes, qui peuvent provenir :

- du polynôme utilisé,
- de la taille du polynôme et donc du CRC résultant (typiquement 8, 16 ou 32 bits),
- de la valeur d'initialisation avant le calcul,
- de la manière d'utiliser le polynôme dans les calculs.

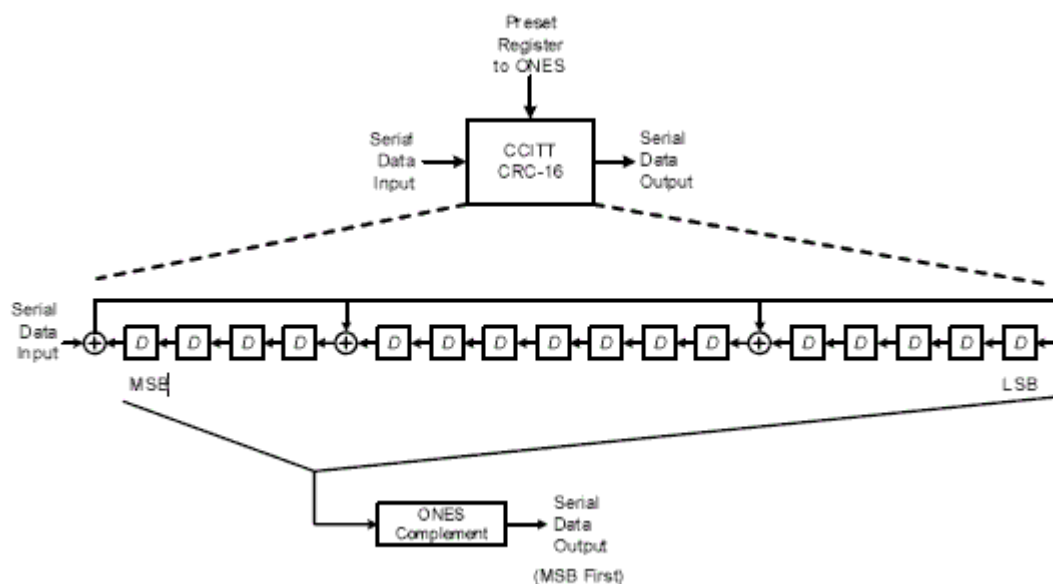
A ce sujet, différentes sources se contredisent, et on rencontre plusieurs versions donnant des résultats différents pour un polynôme et une valeur d'initialisation identiques.

👉 Il est donc important d'utiliser des algorithmes de génération et vérification du CRC qui soient compatibles.

La version de CRC présentée dans ce qui suit est un CRC 16 bits basé sur le polynôme CRC16-CCITT, avec valeur d'initialisation 0xFFFF.

7.15.3. EXEMPLE DE SCHÉMA LOGIQUE

Voici un schéma d'exemple hardware servant à calculer un CRC, basé sur le polynôme CRC16-CCITT :



Source : UWB PLCP header generator, Keysight,

http://literature.cdn.keysight.com/litweb/pdf/ads2008/uwb/ads2008/UWB_PLCP_Header.html

On remarque la simplicité des composants utilisés : des bascules D ainsi que des portes XOR. A l'initialisation, on doit preseter les bascules à 1, puis on fait circuler le message bit à bit. A la fin, le CRC est contenu dans les bascules.

On notera la correspondance entre les connexions des bascules et le polynôme CRC16-CCITT (0x1021).

7.15.4. EXEMPLE D'ALGORITHME

7.15.4.1. CALCUL DU CRC BIT À BIT SUR 1 BYTE

Voici la fonction qui met à jour le CRC en fonction d'un byte :

```
// Fonction pour calcul du CRC byte à byte avec
// algorithme bit à bit
unsigned short updateCRC16(unsigned short crc,
                          unsigned char  data)
{
    unsigned short i, xor_flag, ch;
    #define polyDirect  0x1021

    ch = data << 8;
    for(i = 0; i < 8; i++)
    {
        if ((crc ^ ch) & 0x8000)    // get MSBit
        {
            crc <<= 1;
            crc ^= polyDirect;
        }
        else
        {
            crc <<= 1;
        }
        ch <<= 1;
    }
    return crc;
}
```

On peut voir dans cet algorithme le traitement successif de chaque bit du nouvel octet reçu avec mise à jour du CRC en fonction.

7.15.4.2. CALCUL DU CRC D'UN MESSAGE

Pour calculer le CRC16 d'un message, il suffit d'initialiser la valeur du CRC à 0xFFFF, puis d'appeler la fonction updateCrc16 pour chaque byte du message.

```
unsigned short crc16

// initialisation selon CCITT
crc16 = 0xFFFF;
//--- Boucle de lecture de la chaine
for (i = 0; i < strlen(Mess); i++) {
    crc16 = updateCRC16(crc16, Mess[i]);
}
printf("CRC16 = %04X  \n", crc16);
```

7.15.4.3. EXEMPLE DE RÉSULTAT

Voici les résultats obtenus avec le message "123456789"

```

D:\EtCoursHW\SL229_MINF\CoursPIC32\ProjCours\Chap7RS232\crc16bitaBit...
Calcul CRC16 bit a bit
E pour executer, Q pour quitter
E
Entrez un message
123456789
123456789
CRC16 = 29B1
E pour executer, Q pour quitter
    
```

On obtient 0x29B1, ce qui correspond à la valeur d'un CRC16 CCITT :

"123456789"	
1 byte checksum	221
CRC-16	0xBB3D
CRC-16 (Modbus)	0x4B37
CRC-16 (Sick)	0x56A6
CRC-CCITT (XModem)	0x31C3
CRC-CCITT (0xFFFF)	0x29B1
CRC-CCITT (0x1D0F)	0xE5CC
CRC-CCITT (Kermit)	0x8921
CRC-DNP	0x82EA
CRC-32	0xCBF43926

Source : <https://www.lammertbies.nl/comm/info/crc-calculation.html>

Relevons par ailleurs la multitude de versions de CRC proposée ici.

7.15.5. CALCUL DU CRC AU MOYEN D'UNE TABLE

Pour un traitement plus rapide, la succession de décalages pour un byte peut être remplacée par un calcul utilisant une valeur pré-calculée dans une table et une combinaison de OU-exclusif, comme ci-dessous.

Le choix est alors laissé au programmeur entre consommation mémoire (pour la table de 256 valeurs), ou le temps d'exécution (succession d'opérations-décalages).

7.15.5.1. FORMULE

```
crc = CRC16_table[((crc >> 8) & 0xFF) ^ data] ^ (crc << 8);
```

Afin de pouvoir vérifier le fonctionnement du calcul du CRC avec une table, voici l'adaptation en projet Visual C++, de l'exemple fourni par Michael Neumann.

7.15.5.2. TABLE CALCUL DU CRC

```
// Table calcul CRC16 (Polynome 0x1021)
const unsigned short CRC16_table[256] = {
0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
0xdbfd, 0xcdbdc, 0xfdbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};
```

La fonction de calcul du CRC pour 1 byte devient très simple.

7.15.5.3. FONCTION POUR CALCUL DU CRC AVEC LA TABLE

```
unsigned short updateCRC16(unsigned short crc,  
                           unsigned char data)  
{  
    // retourne la nouvelle valeur du crc  
    return (CRC16_table[((crc >> 8) & 0xFF) ^ data ] ^ (crc << 8));  
}
```

7.15.5.1. CALCUL DU CRC D'UN MESSAGE

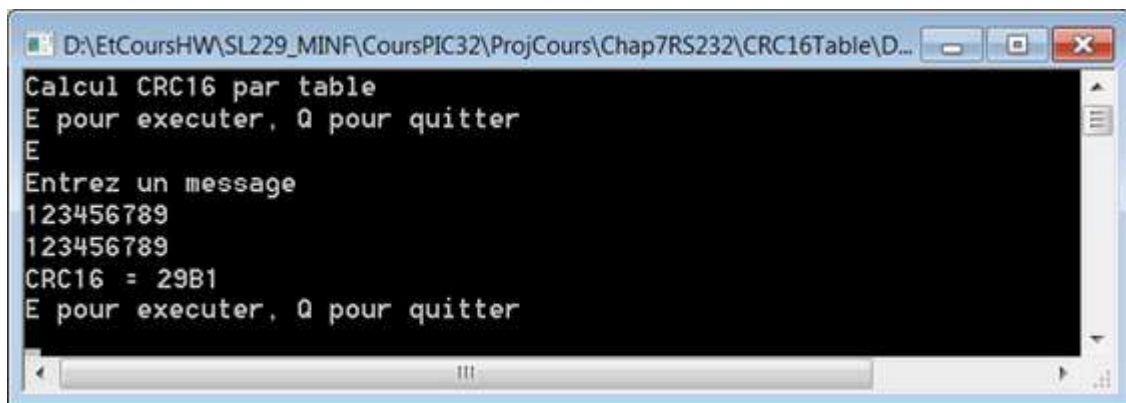
Le principe du calcul du CRC16 d'un message reste le même, il suffit d'appeler la fonction `updateCRC16` pour chaque byte du message.

```
unsigned short crc16
// initialisation selon CCITT
crc16 = 0xFFFF;

//--- Boucle de lecture de la chaine
for (i = 0; i < strlen(Mess); i++) {
    crc16 = updateCRC16(crc16, Mess[i]);
}
printf("CRC16 = %04X  \n", crc16);
```

7.15.5.2. EXEMPLE DE RÉSULTAT

Voici les résultats obtenus avec le message "123456789"



On obtient bien 0x29B1.

7.15.6. CALCUL DU CRC AU MOYEN DE DEUX TABLES

Un compromis pour réduire l'encombrement de la table de 256 valeurs, est possible au moyen de table plus petites.

Cet algorithme provient de Ashley Roll, Digital Nemesis Pty Ltd, www.digitalnemesis.com.

7.15.6.1. LES 2 TABLES DE CALCUL DU CRC

```
// CRC16 Lookup tables (High and Low Byte) for 4 bits per iteration.
// Polynome CCITT 0x1021
unsigned char CRC16_LookupHigh[16] = {
    0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70,
    0x81, 0x91, 0xA1, 0xB1, 0xC1, 0xD1, 0xE1, 0xF1
};

unsigned char CRC16_LookupLow[16] = {
    0x00, 0x21, 0x42, 0x63, 0x84, 0xA5, 0xC6, 0xE7,
    0x08, 0x29, 0x4A, 0x6B, 0x8C, 0xAD, 0xCE, 0xEF
};
```

7.15.6.2. SOUS-FONCTIONS ET FONCTIONS

```
// Process 4 bits of the message to update the CRC Value.
// Note that the data must be in the low nibble of val.
void CRC16_Update4Bits( unsigned char val )
{
    unsigned char    t;

    // Extract the Most significant 4 bits of the CRC register
    t = (CRC16_High >> 4) & 0x0F;

    // XOR in the Message Data into the extracted bits
    t = t ^ val;

    // Shift the CRC Register left 4 bits
    CRC16_High = ((CRC16_High << 4) & 0xF0) | ((CRC16_Low >> 4)
                                                & 0x0F);
    CRC16_Low = CRC16_Low << 4;

    // Do the table lookups and XOR the result into the CRC Tables
    CRC16_High = CRC16_High ^ CRC16_LookupHigh[t];
    CRC16_Low  = CRC16_Low  ^ CRC16_LookupLow[t];
}

// Process one Message Byte to update the current CRC Value
void CRC16_Update( unsigned char val )
{
    CRC16_Update4Bits( (val >> 4) & 0x0F );    // High nibble first
    CRC16_Update4Bits( val & 0x0F );           // Low nibble
}
```

7.15.6.3. CALCUL DU CRC D'UN MESSAGE

Le principe du calcul du CRC16 d'un message reste le même, il suffit d'appeler la fonction `updateCRC16` pour chaque byte du message.

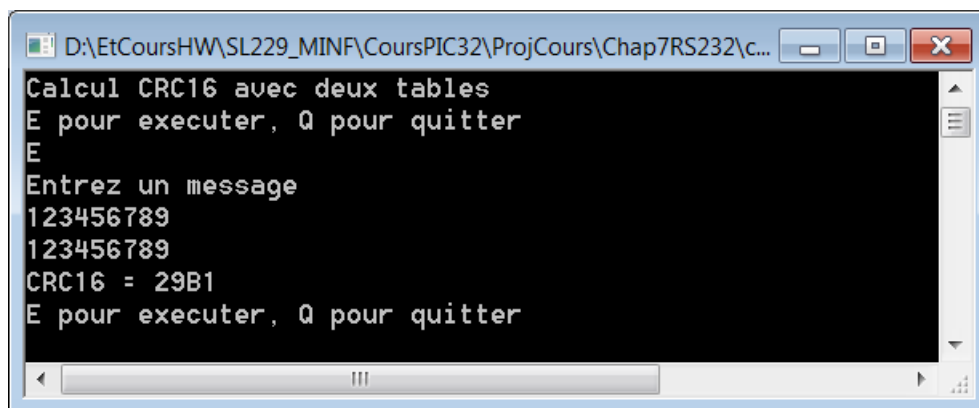
```
unsigned char CRC16_High, CRC16_Low;

// Initialise the CRC to 0xFFFF for the CCITT specification
CRC16_High = 0xFF;
CRC16_Low  = 0xFF;

//--- Boucle de lecture de la chaine
for (i = 0; i < strlen(Mess); i++) {
    CRC16_Update(Mess[i]);
    Res1 = (CRC16_High << 8) | CRC16_Low;
}
Res1 = (CRC16_High << 8) | CRC16_Low;
printf("CRC16 = %04X \n", Res1);
```

7.15.6.4. EXEMPLE DE RÉSULTAT

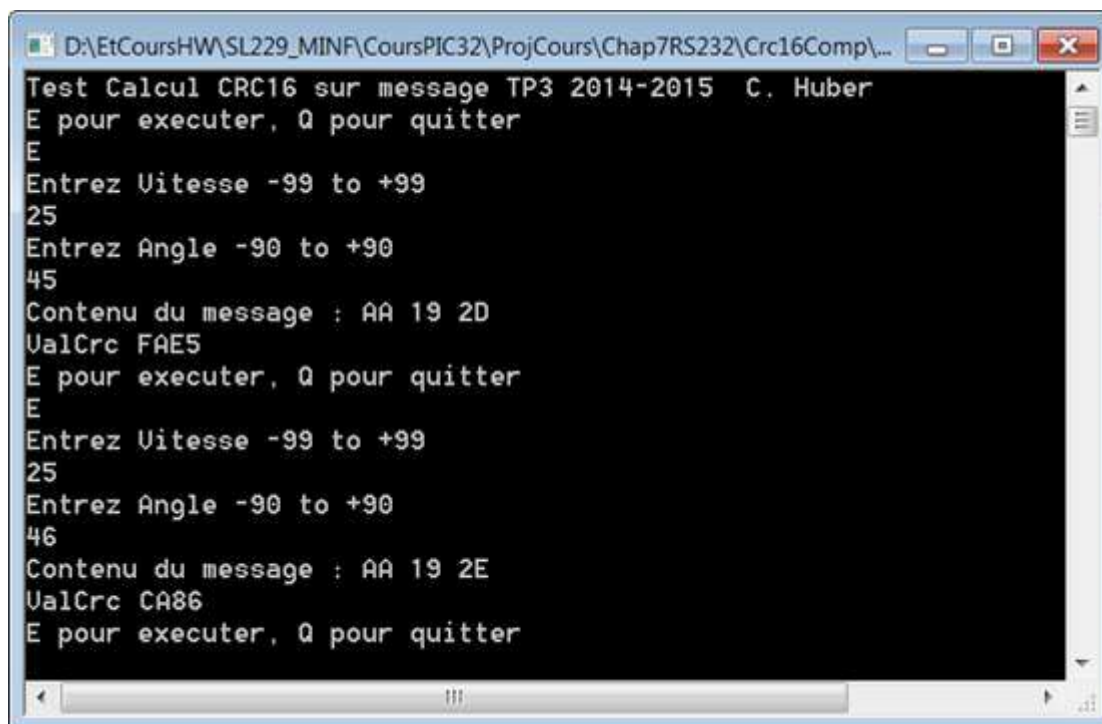
Voici les résultats obtenus avec le message "123456789" :



On obtient le même résultat que précédemment et la valeur 0x29B1 correspond à la valeur indiquée par l'auteur pour le message "123456789".

7.15.7. EXEMPLE D'ÉVOLUTION DU CRC SUR 2 TRAMES

Voici la comparaison des valeurs de CRC obtenues sur 2 messages de 3 octets qui diffèrent peu :



```
D:\EtCoursHW\SL229_MINF\CoursPIC32\ProjCours\Chap7RS232\Crc16Comp\...
Test Calcul CRC16 sur message TP3 2014-2015 C. Huber
E pour executer, Q pour quitter
E
Entrez Vitesse -99 to +99
25
Entrez Angle -90 to +90
45
Contenu du message : AA 19 2D
ValCrc FAE5
E pour executer, Q pour quitter
E
Entrez Vitesse -99 to +99
25
Entrez Angle -90 to +90
46
Contenu du message : AA 19 2E
ValCrc CA86
E pour executer, Q pour quitter
```

Evolution de la valeur de CRC pour une modification de la valeur du 3^{ème} octet du message.

- Message 1 : AA 19 2D, CRC = FAE5
- Message 2 : AA 19 2E, CRC = CA86

On constate que la valeur du CRC change complètement, ce qui est une force de d'un algorithme de type CRC par rapport aux autres types de vérification d'erreur (par exemple parité ou checksum).

7.15.8. VÉRIFICATION DU CRC À LA RÉCEPTION

En organisant un message binaire comme ci-dessous, lors du calcul du CRC à la réception sur l'entier du message nous devons obtenir 0, ce qui facilite le test.

Start [0]	Vitesse [1]	Angle [2]	[3] CRC16 [4]
0xAA	1 byte	1 byte	MSB CRC LSB CRC

⚠ Attention : ceci est uniquement valable si on place le CRC dans le message avec MSB puis LSB. Comme précédemment, la méthode de calcul du CRC à l'émission dépend de l'algorithme utilisé.

7.15.8.1. ALGORITHME DE CALCUL DU CRC LORS DE L'ÉMISSION

Voici la préparation du message, le calcul du CRC et sa mise en place dans le message.

```
// Préparation du message (start + payload)
TxBuffer2[0] = 0xAA;
TxBuffer2[1] = Vitesse;
TxBuffer2[2] = Angle;
crc16 = 0xFFFF;
// Calcul du CRC sur les 3 1er octets
for (i=0 ; i < 3 ; i++) {
    crc16 = updateCRC16_new(crc16, TxBuffer2[i]);
}
// mise en place du CRC dans le message
uTemp.val = crc16;
TxBuffer2[3] = uTemp.sh1.msb;
TxBuffer2[4] = uTemp.sh1.lsb;
```

7.15.8.2. CONTRÔLE DU CRC À LA RÉCEPTION

Calcul du CRC sur l'entier du message, y compris les 2 octets du CRC.

```

crc16 = 0xFFFF;
printf("CRC calcule sur 5 octets \n");
printf("Contenu message : \n");
for (i = 0; i < MESS_SIZE; i++) {
    printf ("%02X ", TxBuffer2[i]);
    crc16 = updateCRC16_new(crc16, TxBuffer2[i]);
}
printf(" CRC obtenu en reception = %04X \n", crc16);
if (crc16 == 0) {
    printf("CRC OK \n");
} else {
    printf("CRC pas OK : val = %04X \n", crc16);
}
```

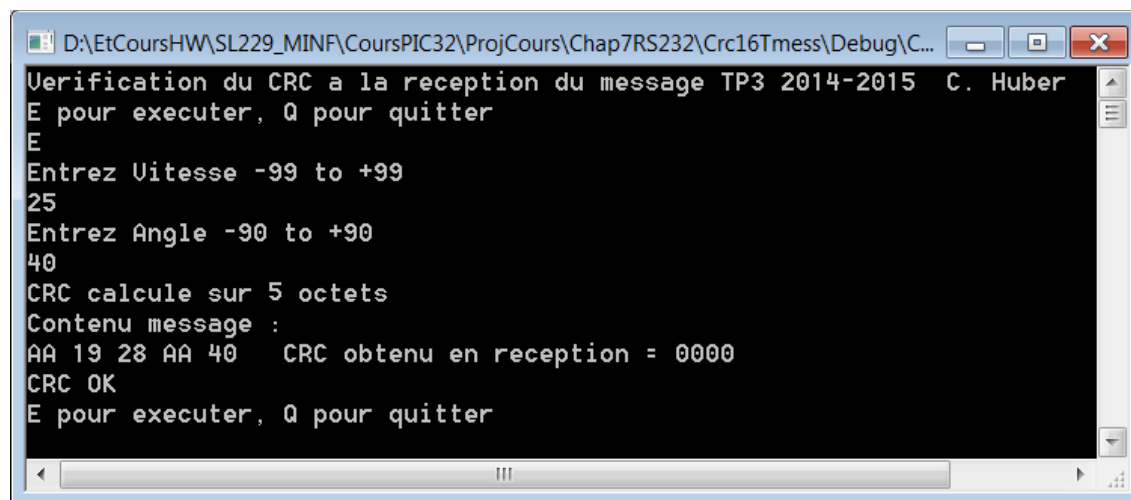
7.15.8.2.1. Définition de l'union

Voici la définition de l'union utilisée pour manipuler une valeur 16 bits.

```
typedef union {  
    unsigned short val;  
    struct {unsigned char lsb;  
           unsigned char msb;} sh1;  
} U_manip16;
```

7.15.8.3. VÉRIFICATION DU RÉSULTAT

Comme le montre la copie d'écran ci-dessous, on obtient bien 0 comme résultat du CRC calculé sur l'entier du message :



7.15.9. CALCUL CRC16, LIBRAIRIE À DISPOSITION

La fonction de calcul du CRC16-CCITT et la table des 256 valeurs sont fournies sur le réseau dans les fichiers **Mc32CalCrc16.h** et **Mc32CalCrc16.c** sous:

...\Maitres-Eleves\SLO\Modules\SL229_MINF\PIC32MX_Utilitaires(PlibHarmony)\
Fifo&Antirebond.

7.16. EXEMPLE COMPLET AVEC ÉMISSION ET RÉCEPTION

Dans cet exemple, nous allons montrer l'émission d'un message sécurisé par CRC ainsi que le traitement de la réception, avec la reconstitution du message qui est la partie la plus délicate.

Nous allons prendre le cas d'un message binaire. Dans cet exemple nous nous focalisons sur les traitements hors interruption, c'est-à-dire les fonctions SendMessage et GetMessage.

7.16.1. STRUCTURE DU MESSAGE DE L'EXEMPLE

	No	DATA		CRC16	
0xAA	1 ou 2	Lsb	Msb	Msb	Lsb

En émission, le CRC16 est calculé sur 0xAA + No Canal + DATA.

Pour le CRC16 on utilisera le CRC16-CCITT. Calcul par table de 256 valeurs.

En réception, le calcul du CRC est réalisé sur l'entier du message et s'il est valable la valeur obtenue doit être de zéro.

Le data est une valeur 16 bits qui permet de transporter la valeur lue correspondant au convertisseur A/D des PIC32MX. Le No indique le numéro du canal de l'A/D 1 ou 2.

👉 Le data est transmis avec le Lsb en premier ce qui permet la manipulation aisée par une union. Comme nous l'avons vu précédemment le CRC doit être transmis avec le Msb en premier pour permettre le contrôle du CRC sur l'entier du message.

7.16.2. CYCLES DE TRAITEMENTS

L'application est activée toutes les 10 ms. La fonction GetMessage est appelée à chaque activation de l'application. La fonction SendMessage est appelée au même rythme de manière à envoyer un seul message en alternant le numéro du canal.

Le débit maximum de la transmission avec une taille de message de 6 octets et un baudrate de 57'600 sans parité et avec 1 seul bit de stop, ce qui fait 10 bits pour un octet donc 60 bits par message. D'où $57'600 / 60 = 960$ messages par seconde.

Le système mis en place est capable de recevoir 100 messages par seconde et il en émet 100 par seconde.

7.16.3. DÉFINITIONS ET ÉLÉMENTS GLOBAUX

Voici les différents éléments permettant la gestion des messages. Le choix s'est porté sur une structure pour décrire le message, ce qui donne une meilleure description mais ne permet pas de traitement en boucle.

On trouve aussi la déclaration des FIFO avec le choix de leur taille pour accueillir 4 messages.

Voici encore une union pour la manipulation des éléments 16 bits :

```
typedef union {
    uint16_t val;
    struct {uint8_t lsb;
            uint8_t msb;} shl;
} U_manip16;
// Définitions pour les messages
#define MESS_SIZE 6
#define STX_code (0xAA)

// Structure décrivant le message
typedef struct {
    uint8_t Start;
    uint8_t NoCh;
    uint16_t Data;
    uint8_t MsbCrc;
    uint8_t LsbCrc;
} StruMess;

// Type énuméré pour les étapes de traitement en réception
typedef enum { WaitSTX = 1, WaitNoCh, WaitLsbData, WaitMsbData ,
               WaitMsbCrc, WaitLsbCrc
} E_MessSituation;

// Structure pour émission des messages
StruMess TxMess;
// Structure pour réception des messages
StruMess RxMess;

// Déclaration des FIFO pour réception et émission
#define FIFO_RX_SIZE ((4*MESS_SIZE) + 1) // 4 messages
#define FIFO_TX_SIZE ((4*MESS_SIZE) + 1) // 4 messages

int8_t fifoRX[FIFO_RX_SIZE];
// Déclaration du descripteur du FIFO de réception
S_fifo descrFifoRX;

int8_t fifoTX[FIFO_TX_SIZE];
// Déclaration du descripteur du FIFO d'émission
S_fifo descrFifoTX;
```

7.16.4. LA FONCTION SENDMESSAGE

La fonction **SendMessage** détermine s'il y a la place pour déposer un message complet. Si c'est le cas, le message est composé avec le calcul du CRC16, puis le message est placé dans le FifoTX. A la fin de la fonction, on trouve la gestion du contrôle de flux.

```
void SendMessage(int8_t NoCh, int16_t Data)
{
    uint8_t FreeSize;
    uint16_t ValCrc16 = 0xFFFF;
    U_manip16 tmpCrc, TmpData;
    // Test si place Pour écrire 1 message
    FreeSize = GetWriteSpace ( &descrFifoTX);
    if (FreeSize >= (MESS_SIZE) ) {
        // Compose le message
        TxMess.Start = 0xAA; // start
        ValCrc16 = updateCRC16(ValCrc16, TxMess.Start);

        TxMess.NoCh = NoCh;
        ValCrc16 = updateCRC16(ValCrc16, TxMess.NoCh);

        // Traitement data
        TxMess.Data = Data;
        TmpData.val = Data;
        ValCrc16 = updateCRC16(ValCrc16, TmpData.shl.lsb);
        ValCrc16 = updateCRC16(ValCrc16, TmpData.shl.msb);

        // Traitement CRC
        tmpCrc.val = ValCrc16;
        TxMess.MsbCrc = tmpCrc.shl.msb;
        TxMess.LsbCrc = tmpCrc.shl.lsb;

        // Dépose le message dans le fifo
        PutCharInFifo ( &descrFifoTX, TxMess.Start);
        PutCharInFifo ( &descrFifoTX, TxMess.NoCh);
        PutCharInFifo ( &descrFifoTX, TmpData.shl.lsb);
        PutCharInFifo ( &descrFifoTX, TmpData.shl.msb);
        PutCharInFifo ( &descrFifoTX, TxMess.MsbCrc);
        PutCharInFifo ( &descrFifoTX, TxMess.LsbCrc);
    }

    // Gestion du control de flux
    // Si on a un caractère à envoyer et que CTS = 0
    FreeSize = GetReadSize(&descrFifoTX);
    if ((RS232_CTS == 0) && (FreeSize > 0))
    {
        // Autorise l'interruption d'émission
        PLIB_INT_SourceEnable(INT_ID_0,
                               INT_SOURCE_USART_1_TRANSMIT);
    }
} // End SendMessage
```

7.16.5. LA FONCTION GETMESSAGE

A chaque appel de la fonction GetMessage, on détermine si on peut traiter un message complet. Le traitement est réalisé par une boucle qui décompte le nombre de caractères obtenus du FIFO tout en surveillant la fin du message. Le mécanisme doit supporter un abandon de la boucle sans obtenir un message complet, ceci dans une situation de message incomplet. Pour savoir quel caractère est traité, on utilise une variable d'état dont la valeur définie par un type énuméré permet de progresser dans un Switch.

```
bool GetMessage(uint16_t *Ch1, uint16_t *Ch2)
{
    static E_MessSituation GetSituation = WaitSTX ;

    static uint16_t ValCrc;           // en cas d'abandon en cours
    static U_manip16 TmpData;         // en cas d'abandon en cours

    uint8_t EndMess;
    uint8_t RxC;
    uint8_t NbCharToRead = 0;
    bool MessReady = false;           // indique validité message reçu

    // Détermine le nombre de caractères à lire
    NbCharToRead = GetReadSize ( &descrFifoRX);

    // Si >= taille message alors traite
    if (NbCharToRead >= MESS_SIZE) {
        EndMess = 0;
        while ( (NbCharToRead >= 1) && ( EndMess == 0) ) {
            // Lis un caractère sans gestion du status
            GetCharFromFifo ( &descrFifoRX, &RxC );
            NbCharToRead--;

            // Traitement selon état du message
            switch (GetSituation) {
                case WaitSTX :
                    if ( RxC == STX_code) {
                        RxMess.Start = RxC;
                        ValCrc = 0xFFFF;
                        ValCrc = updateCRC16(ValCrc, RxC);
                        GetSituation = WaitNoCh;
                    } else {
                        BSP_LEDToggle(BSP_LED_1); // pour test
                    }
                    break;
                case WaitNoCh :
                    RxMess.NoCh = RxC;
                    ValCrc = updateCRC16(ValCrc, RxC);
                    GetSituation = WaitLsbData;
                    break;
                case WaitLsbData :
                    ValCrc = updateCRC16(ValCrc, RxC);
                    TmpData.sh1.lsb = RxC;
            }
        }
    }
}
```

```

        GetSituation = WaitMsbData;
    break;

    case WaitMsbData :
        ValCrc = updateCRC16(ValCrc, RxC);
        TmpData.shl.msb = RxC;
        RxMess.Data = TmpData.val;
        GetSituation = WaitMsbCrc;
    break;

    case WaitMsbCrc :
        RxMess.MsbCrc = RxC;
        ValCrc = updateCRC16(ValCrc, RxC);
        GetSituation = WaitLsbCrc;
    break;

    case WaitLsbCrc :
        RxMess.LsbCrc = RxC;
        ValCrc = updateCRC16(ValCrc, RxC);

        if (ValCrc == 0 ) {
            // Contenu OK
            if (RxMess.NoCh == 1) {
                *Ch1 = RxMess.Data;
            }
            if (RxMess.NoCh == 2) {
                *Ch2 = RxMess.Data;
            }
            MessReady = true;
        } else {
            BSP_LEDToggle(BSP_LED_7); // pour test
        }
        GetSituation = WaitSTX;
        EndMess = 1; // pour ne traiter qu'un seul
                    // message à la fois

    break;
} // end switch
} // end while
}

// Gestion control de flux de la réception
// 12 correspond à 2 paquets de 6 (3/4 RxBuffer de 8)
if(GetWriteSpace ( &descrFifoRX) >= 12) {
    // Autorise émission par l'autre
    RS232_RTS = 0;
}
return MessReady;
} // End GetMessage

```

7.16.6. RÉACTION DU SYSTÈME À DES MESSAGES CORROMPUS

Pour vérifier si le système de réception supporte des messages corrompus, il est nécessaire d'introduire depuis l'application la possibilité d'injecter, par exemple une fois sur 10, un message incomplet. Nous choisissons un message pour lequel les 2 octets de CRC manquent. Sa taille sera donc de 4 octets au lieu de 6.

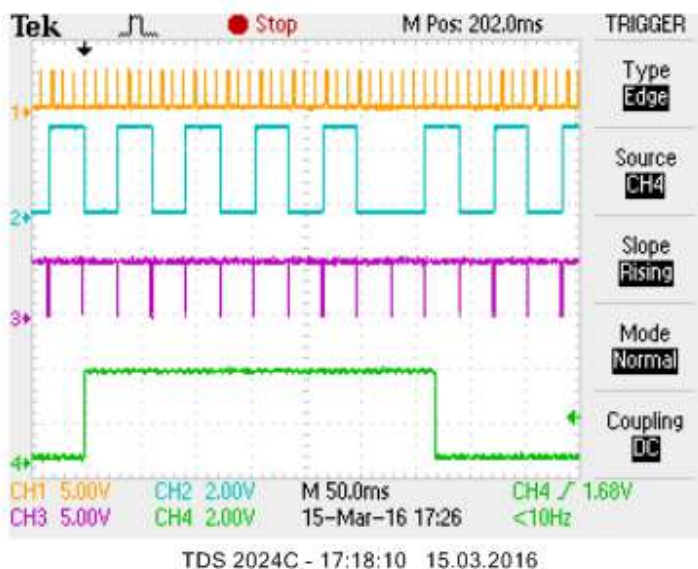
7.16.6.1. VUE GÉNÉRALE RÉACTION À MAUVAIS MESSAGE

canal 1 : LED_0
(marque traitement dans l'application)

canal 2 : LED_3
(inversion à chaque int RX)

canal 3 : broche 52
U1RX

canal 4 : LED_7
(inversion si erreur CRC)



On remarque l'erreur CRC tous les 10 messages.

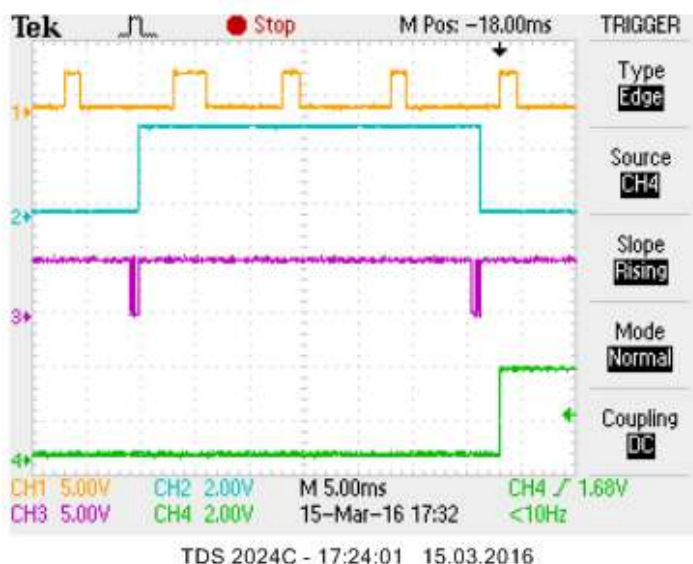
7.16.6.2. VUE RÉACTION À MAUVAIS MESSAGE

canal 1 : LED_0
(marque traitement dans l'application)

canal 2 : LED_3
(inversion à chaque int RX)

canal 3 : broche 52
U1RX

canal 4 : LED_7
(inversion si erreur CRC)



Il est difficile de repérer le message plus court.

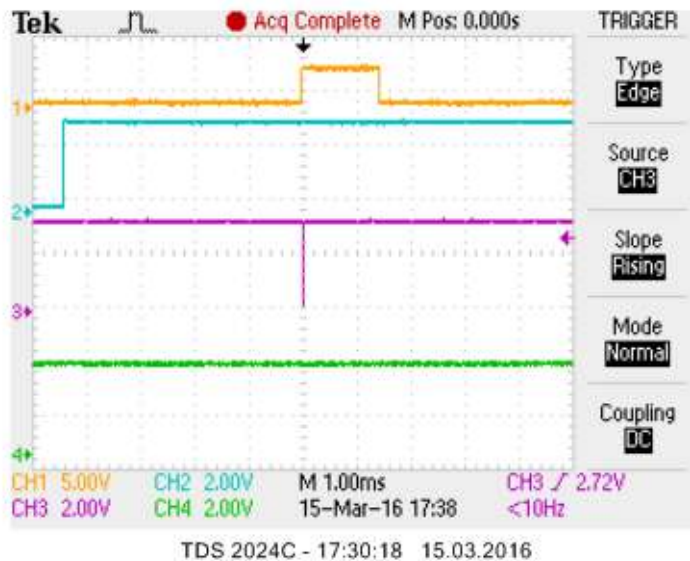
7.16.6.3. OBSERVATION ATTENTE STX

canal 1 : LED_0
(marque traitement
dans l'application)

canal 2 : LED_3
(inversion à chaque
int RX)

canal 3 : LED_1
inversion dans attente
STX

canal 4: LED_7
(inversion si erreur
CRC)

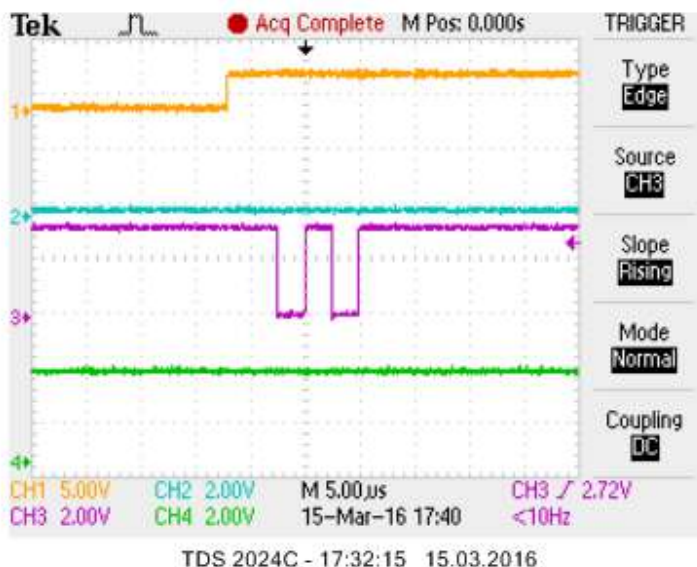
**7.16.6.1. OBSERVATION ATTENTE STX DÉTAIL**

canal 1 : LED_0
(marque traitement
dans l'application)

canal 2 : LED_3
(inversion à chaque
int RX)

canal 3 : LED_1
inversion dans attente
STX

canal 4: LED_7
(inversion si erreur
CRC)



On peut observer les 4 flancs qui nous indiquent la consommation de 4 caractères n'étant pas le STX.

7.16.6.2. CONCLUSION SUR LE TRAITEMENT DE MAUVAIS MESSAGE

Le système se comporte correctement grâce à l'attente du STX, ce qui a pour effet de consommer une partie de message pour se recaler sur un bon message.

7.16.7. CONCLUSION SUR L'EXEMPLE

Cet exemple a montré comment composer un message avec le calcul et la mise en place d'un CRC. Le principe est facilement adaptable à d'autres structures, il est aussi possible de remplacer la structure décrivant le message par un tableau.

En ce qui concerne la fonction GetMessage, la solution proposée combine une structure avec une avance étape par étape dans le switch. Cette solution un peu lourde présente l'avantage de pouvoir être facilement adaptée à d'autres organisations du message. Il est aussi possible d'utiliser un tableau à la place de la structure, ce qui permettrait de réduire le switch, voire de le supprimer puisqu'il suffit d'attendre le début du message, puis de compter les caractères traités, le critère de fin étant la bonne taille.

7.17. CONCLUSION GÉNÉRALE

Ce chapitre a présenté différents aspects de la communication série asynchrone, dans le but de permettre à l'étudiant de disposer d'une méthode pour réaliser une telle communication adaptée aux contraintes d'un projet spécifique.

Il est important d'introduire des moyens pour vérifier le bon comportement de la communication au niveau des interruptions et du contrôle de flux, car des changements de taille de message, de FIFO ou autre peuvent modifier le comportement.

La combinaison de FIFO software avec les interruptions de réception et d'émission est le meilleur moyen d'obtenir des communications sérieelles efficaces en découplant les traitements. Les queues hardware de l'USART, bien utilisées, permettent de réduire la fréquence des interruptions, ce qui réduit la charge du processeur.

Suivant le type de transmission, le contrôle de l'intégrité des données transmises peut s'avérer nécessaire.

Il faut garder à l'esprit que, sans être complexe, une communication sérieelle demande un certain soin dans sa réalisation et sa mise au point.

7.18. HISTORIQUE DES VERSIONS

7.18.1. VERSION 1.5 JANVIER 2015

Création de ce document par transformation du chapitre 12 du PIC18F. Version 1.5 pour indiquer l'utilisation des PLIB de Harmony. Mise à jour du traitement par CRC. Cette version n'est pas entièrement adaptée au PIC32MX.

7.18.2. VERSION 1.6 MARS 2015

Introduction émission et réception sans interruptions. Adaptation des mécanismes d'émission et réception avec interruptions et FIFO au PIC32MX. Remplacement de l'exemple de réception par un exemple complet avec les fonctions SendMessage et GetMessage.

7.18.3. VERSION 1.7 MARS 2016

Reprise des exemples et complément pour les modes du FIFO de réception. Ajout observation situation erreur CRC.

7.18.4. VERSION 1.7_1 MARS 2016

Retouche 7.15.3.3 page 75 et retouche exemple et résultat calcul CRC par table.

7.18.5. VERSION 1.8 FÉVRIER 2017

Reprise par SCA. Relecture générale. Elucidation et nettoyage câbles (link et null-modem) p. 7 et 12. Enlevé version de code d'émission non amélioré (avec interruption parasite après une transmission). Simplification partie CRC (gardé que le nouvel algorithme, qui semble être le bon).

7.18.1. VERSION 1.9 JANVIER 2018

Ajouts documents de référence. Corrections mineures.

7.18.1. VERSION 1.91 JANVIER 2019

Adaptation exemple §7.13 "Exemple utilisation des FIFOs et des interruptions" à Harmony 2.05 (configurateur graphique et code init. généré ont changé). Ajout §7.15.3 exemple de schéma logique pour génération hardware d'un CRC.

7.18.2. VERSION 1.92 SEPTEMBRE 2022

Mise à jour du code de la librairie GesFifoTh32.