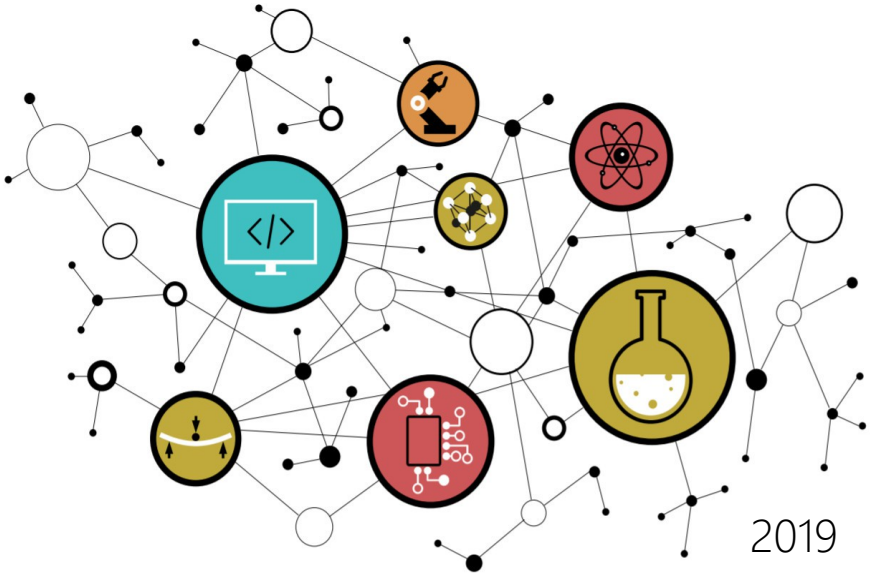
TRAVAUX PRATIQUES



2019

SOMMAIRE

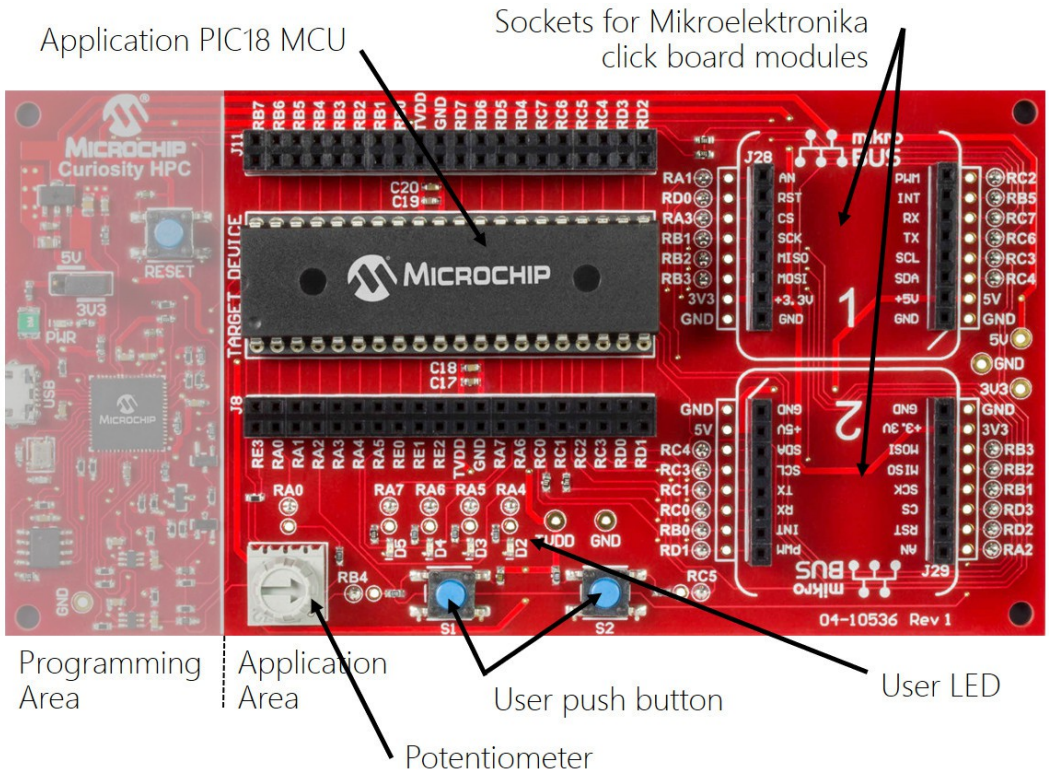
1. PREAMBULE

1. PREAMBULE

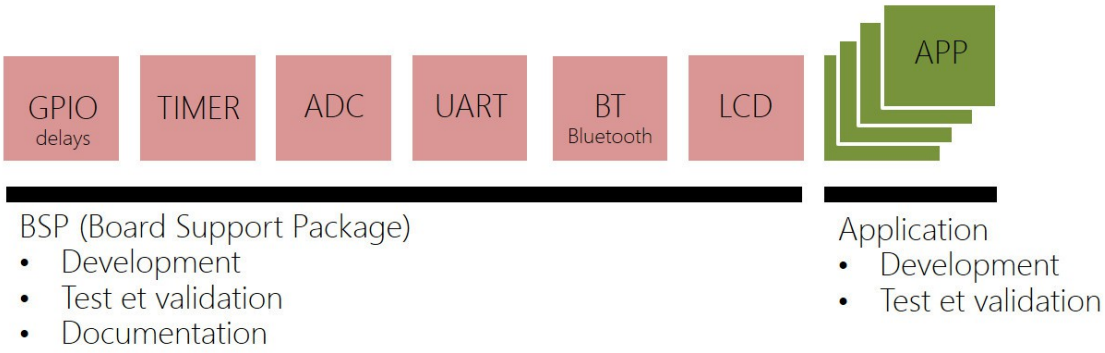
La totalité de la trame de Travaux Pratiques sera réalisée sur solution Microchip PIC18, société et technologie leader du marché des MCU 8bits. Les outils de développement logiciel (IDE MPLABX et toolchains XC8/32) sont librement téléchargeables et installables depuis le site web officiel de Microchip. De même, n'hésitez pas à parcourir le document d'annexes fourni (*mcu-tp-annexes.pdf*) afin de vous aiguiller dans vos premières phases de développement. S'appuyer sur la plateforme *moodle* afin d'installer les outils (section outils de développement) :

https://foad.ensicaen.fr/course/view.php?id=116

Les développements seront réalisés sur PIC18F27K40, MCU 8bits 28 broches intégrant un CPU PIC18 et offrant de larges ressources mémoires Flash et SRAM (pour un MCU 8bits). La carte de développement sélectionnée est le starter kit Curiosity HPC de Microchip. Elle intègre nativement une sonde de programmation JTAG et deux connecteurs permettant d'ajouter des modules click board proposés par la société Mikroelektronika (https://www.mikroe.com/click). Des centaines de modules sont disponibles en catalogue (Bluetooth, audio, WIFI, contrôle moteur, etc).

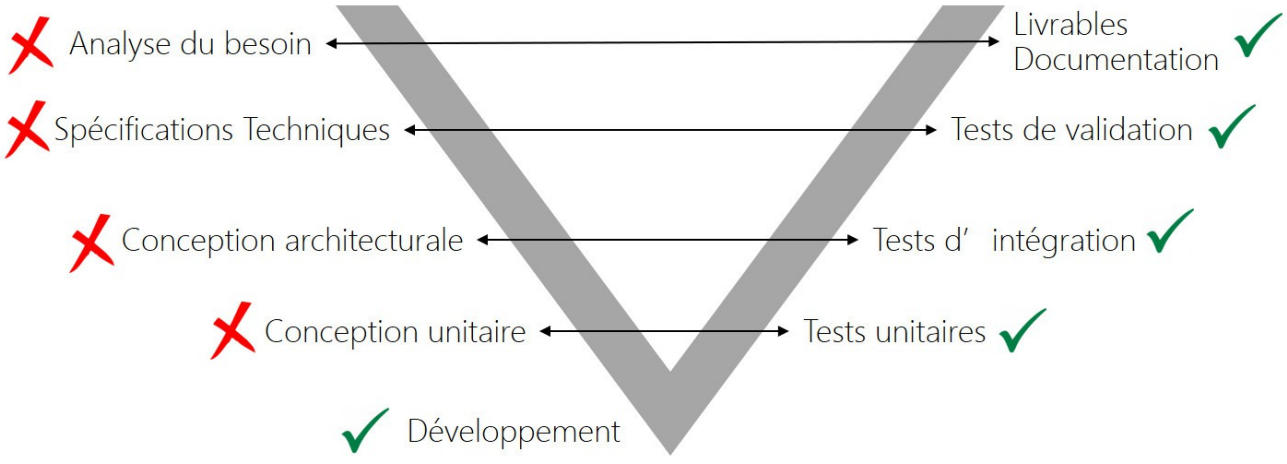


Objectifs et réalisations



Les objectifs de cet enseignement sont multiples. Pour une grande partie des réalisations, nous aurons à développer un BSP (Board Support Package) from scratch (en partant de rien) et en travaillant à l'étage registre du processeur (plus bas niveau de développement sur machine). En résumé, nous allons tout faire de A à Z. Dans notre cas, le BSP doit être vu comme une collection de fonctions logicielles pilotes (drivers) assurant le contrôle des fonctions matérielles périphériques internes voire externes. Notre BSP sera dédié à notre processeur et notre carte de développement. Une migration de technologie ou de solution (processeur et/ou carte) nécessiterait un redéveloppement. Une fois le BSP développé, testé, validé, documenté et la bibliothèque statique générée, nous développerons une application audio Bluetooth l'utilisant. L'application implémentera un *scheduler offline*, ce point sera vu plus en détail dans la suite du document. Nous ne pourrons alors qu'imaginer l'infini potentiel s'ouvrant alors devant nos yeux !

La chronologie de la trame de TP est construite autour d'un workflow typique rencontré en industrie autour de ce type de développements. Avant de développer tout applicatif, il faut valider unitairement toutes les fonctionnalités (matérielles et logicielles) de l'application avant de se focaliser sur les phases d'intégration successives. Même si des méthodologies agiles de gestion de projet seront utilisées lors de projets en équipe en 2ième et 3ième année en majeures IA et SATE, celui présenté ici présente un cycle en V (V et agilité peuvent très bien cohabiter). Il ne serait pas raisonnable ni pertinent de demander à un élève ingénieur en formation 1ière année de réaliser la conception architecturale du projet. Les spécifications techniques et la conception ont donc été réalisées, vous aurez donc à vous focaliser sur les phases de développement, test, validation et documentation.



2. ASSEMBLEUR
GPIO ET DELAI LOGICIEL

2. ASSEMBLEUR, GPIO ET DELAY LOGICIEL

2.1. Travail préparatoire

- Installer les outils de développement sur votre ordinateur (IDE MPLABX et toolchain C 8bits XC8 dernières versions). Suivre les instructions présentes sur la plateforme *moodle* d'enseignement (section *outils de développemen*t) :

  https://foad.ensicaen.fr/course/view.php?id=116

- Lire attentivement le *préambule* et le chapitre *GPIO* du document support aux TP *mcu-tp-periphériques.pdf*. Il s'agit d'un support de cours spécialement réalisé pour appréhender sereinement les premières séances de TP. Ne pas hésiter à s'aider également d'internet. Pour travailler efficacement, utiliser également la documentation technique ou datasheet du processeur de TP (lien officiel vers les ressources techniques de notre MCU PIC18F27K40 sur le site de Microchip) :

  https://www.microchip.com/wwwproducts/en/PIC18F27K40

- (1pt) Qu'est-ce qu'une GPIO ? Proposer des exemples d'utilisation dans des applications autour de vous

- (1pt) A quoi servent les registres TRISx ? Pourquoi se nomment-ils TRISx ?

- (1pt) Proposer une configuration en assembleur permettant de configurer la broche RD7 du port D en entrée, et les broches RC0 à RC2 du port C en sortie. Appliquer un niveau logique haut sur ces 3 même broches. *Tester votre programme en mode simulation et valider son fonctionnement en environnement de debug (s'aider de tutoriel internet pour réaliser ces opérations voire des annexes)*

- (1pt) Réitérer le travail précédent mais en langage C. *S'aider d'internet pour répondre à cette question*

- (1pt) Qu'est-ce qu'un registre et de quoi est-il constitué ? Pour quelles raisons tous les registres des PIC18 font 8bits ?

- (1pt) Ouvrir la documentation utilisateur de la carte de développement utilisée en TP (Curiosity HPC de Microchip). En regardant le schéma électrique, préciser sur quelles broches sont connectées les LED D2/3/4/5 et les boutons poussoirs S1/2.

  https://www.microchip.com/developmenttools/ProductDetails/dm164136

- (1pt) Sur quelle broche est connecté le bouton poussoir du reset ?

2. ASSEMBLEUR, GPIO ET DELAY LOGICIEL

2.2. Travaux en séances

Ce premier exercice a pour objectif la prise en main des outils de développement, matériels et logiciels, et de réaliser des analyses bas niveau assembleur des programmes développés. Nous profiterons de ce premier TP d'analyse pour réaliser également nos premiers développements de *drivers* (fonctions pilotes de périphériques) en découvrant les mécanismes de gestion des GPIO.

- Si ce n'est pas déjà fait, lire le *préambule* de ce document
- Télécharger l'archive de travail *mcu.zip* sur la plateforme moodle. L'extraire à la racine de votre lecteur réseau école. Tous les développements se feront dans le répertoire *mcu/tp/disco* (discovery)
- Créer un projet MPLABX nommé *gpio_test* (ou autre nom) dans le répertoire *disco/bsp/gpio/test/pjct*. Inclure le fichier *bsp/gpio/test/main.c* et s'assurer de la bonne compilation du projet. S'aider de l'annexe 1.

Analyse assembleur

- Compiler puis charger le programme en mode *debug*. S'aider de l'annexe 3.
- Ouvrir et déplacer dans l'environnement graphique de l'IDE une fenêtre permettant d'observer le code binaire désassemblé : Window → PIC/target Memory Views → Program Memory
- Parcourir la totalité de la mémoire flash et retranscrire le code binaire et assembleur générés ci-dessous en respectant le *mapping* ou modèle mémoire choisi par le *linker*

Address	Opcode	Label	Disassembly

- Comprendre et expliquer le code généré

- Ajouter une instruction assembleur *nop* (no operation) dans la boucle *while* du main. Compiler puis charger le nouveau programme en mode *debug*. Analyser le programme binaire généré. Mettre un point d'arrêt sur le *nop* (double clic dans la fenêtre d'édition au numéro de ligne désiré). Exécuter le programme pas à pas (Step Into ou F7)

```
asm(''nop'');
```

- En analysant la console de sortie de l'IDE (Build, Load,...), préciser dans quel répertoire a été rangé le firmware de sortie après édition des liens

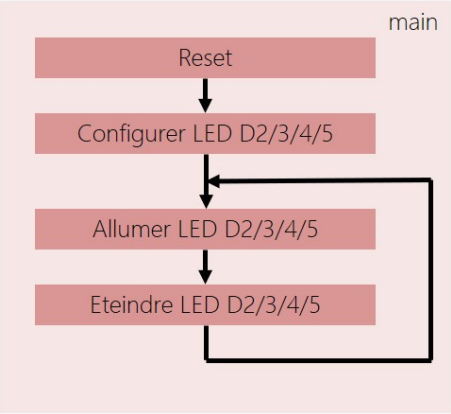
- Retirer l'instruction *nop* pour la suite des développements

Gestion des LED

- Sur quelles broches du MCU sont physiquement connectées les LED D2/3/4/5 ? *S'aider de la documentation utilisateur de la carte Curiosity HPC de Microchip (depuis internet ou dans le répertoire disco/bsp/doc)*

- Créer un répertoire logique *gpiolib* dans le répertoire Sources Files du projet sous l'IDE (clic droit *New Logical Folder*). Ajouter le fichier *src/led_init.c* au projet et vérifier la bonne compilation de celui-ci. Modifier ce fichier pour configurer les broches souhaitées en sortie. Modifier également le fichier d'en-tête *include/gpio.h* pour l'activation ou la désactivation des LED (développement de macro fonction C).

- Développer une application de test implémentant le logigramme suivant :



- Analyser et expliquer le programme binaire généré

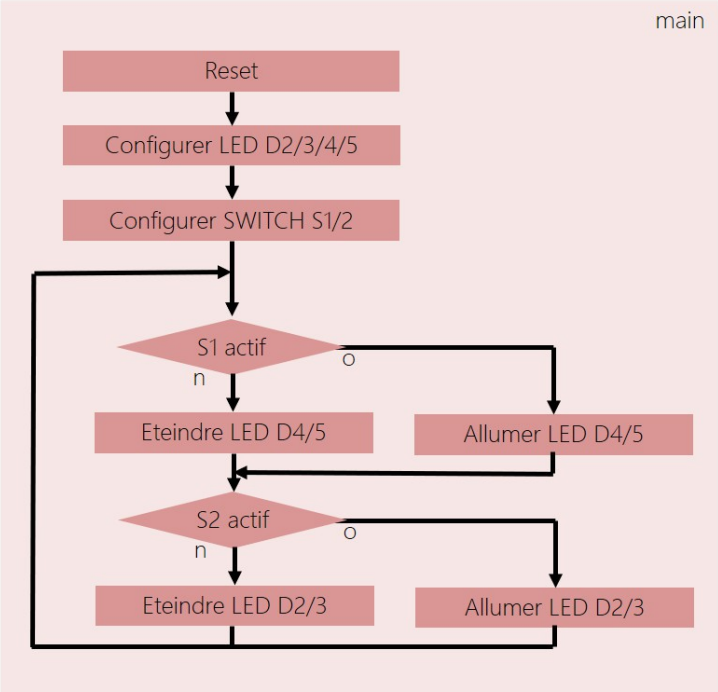
- Observer à l'oscilloscope le signal en sortie du MCU envoyé à la LED D2. Représenter ci-dessous le chronogramme et expliquer les *timing* et la forme d'onde.

Gestion des boutons poussoirs

- Sur quelles broches du MCU sont physiquement connectés les boutons poussoirs S1 et S2 ? *S'aider de la documentation utilisateur de la carte Curiosity HPC de Microchip (depuis internet ou dans le répertoire disco/bsp/doc)*

- Ajouter le fichier assembleur *src/switch_init.as* au projet et vérifier la bonne compilation de celui-ci. Modifier ce fichier en assembleur pour configurer les broches souhaitées en entrée. Modifier également le fichier d'en-tête *include/gpio.h* pour la lecture des états logiques sur les GPIO précédemment configurées (développement de macro fonction C).

- Développer une application de test implémentant le logigramme suivant :



- Valider le bon fonctionnement de votre application de test

- Commenter le code précédemment développé pour la suite de l'exercice

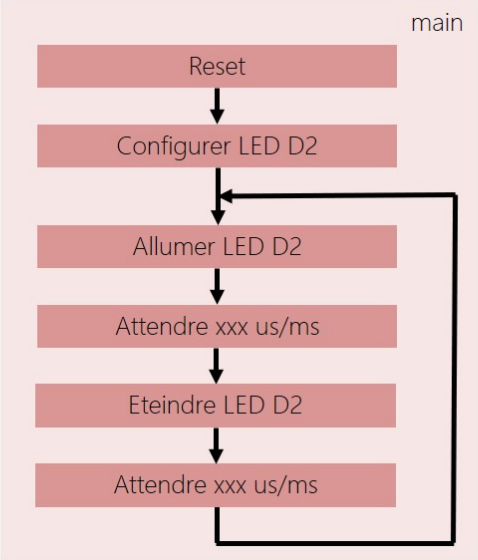
Délai logiciel

Dans une application, nous cherchons au maximum à éviter les temporisations logicielles. Il s'agit de boucles de décrémentation exécutées par le CPU. Nous préférons dédier le CPU à la supervision du système (application) ou aux opérations de calcul. Le reste du temps, le système doit être en veille pour des considérations énergétiques. Nous verrons la mise en veille dans le prochain TP. Néanmoins, dans certains cas, comme les phases d'initialisation des périphériques, ces fonctions peuvent être utiles. Nous en verrons des applications par la suite.

- Quelle est la fréquence de travail du CPU ? Quel fichier du projet devrait-on modifier pour jouer sur ce paramètre ?

- Créer un répertoire logique *delaylib*. Ajouter les fichiers *bsp/common/*delay_*ixxxms.as* au projet et vérifier la bonne compilation de celui-ci. Modifier le fichier *delay_10us.as* afin de réaliser une temporisation logicielle en assembleur de 10us. *S'aider de l'annexe 6 et de la documentation du processeur. Regarder la documentation de l'instruction decfsz.*

- Développer une application de test implémentant le logigramme suivant :



- Valider les durées de temporisation à l'oscilloscope

- Réitérer le travail pour chaque fonction d'attente *delay_1ms, delay_40ms* et *delay_200ms*

- Quelle est le pourcentage de charge CPU (durée de travail du CPU par unité de temps) ?

3. TIMER ET INTERRUPTION

3. TIMER ET INTERRUPTION

3.1. Travail préparatoire

- Lire attentivement les chapitres *INTERRUPTION* et *TIMER* du document support aux TP *mcu-tp-periphériques.pdf*. Ne pas hésiter à s'aider également d'internet. Pour travailler efficacement, utiliser également la documentation technique ou datasheet du MCU PIC18F27K40

- (1pt) Qu'est-ce qu'une IRQ ? *Proposer un schéma.Ne pas répondre uniquement Interrupt Request*

- (1pt) Qu'est-ce qu'une ISR ? *Proposer un schéma.Ne pas répondre uniquement Interrupt Service Routine*

- (1pt) Qu'est-ce qu'un vecteur d'interruption et expliquer son rôle ? *Proposer un schéma.*

- (1pt) Proposer une configuration en assembleur permettant de configurer l'interruption de priorité haute pour le Timer0 avec démasquage global d'interruption. Réitérer ce même travail pour le Timer1.

- (1pt) Proposer une configuration en assembleur pour le Timer0. Nous souhaitons lever une interruption toutes les 20ms. Nous travaillerons avec une référence d'horloge interne (CPU clock 64MHz)

3. TIMER ET INTERRUPTION

3.2. Travaux en séances

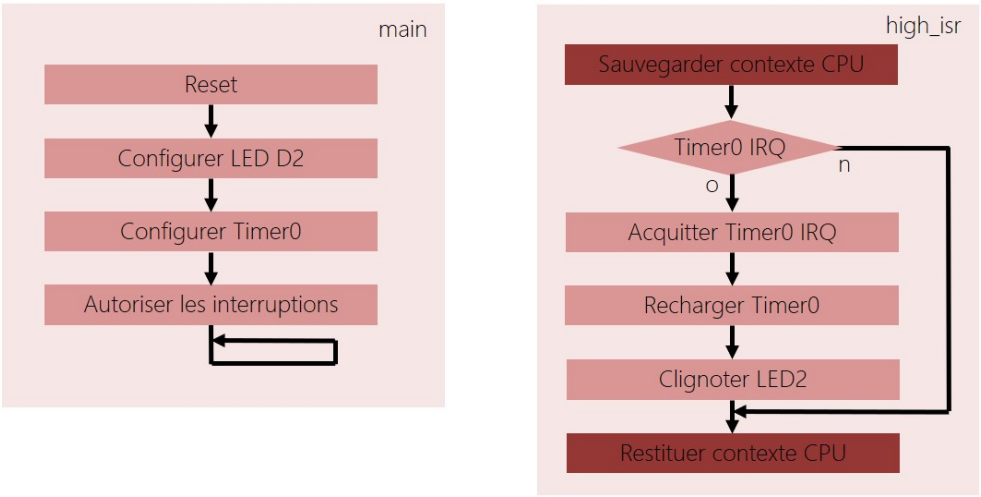
Cet exercice vise à mettre en pratique un concept essentiel lié au développement sur processeur numérique, celui d'interruption. Afin d'appréhender ce nouveau mécanisme, nous travaillerons avec un timer, un périphérique de comptage standard sur MCU. Nous allons nous efforcer de gérer une base de temps en soulageant au maximum le CPU. Objectif, ne travailler qu'en cas d'absolue nécessité, et mettre le cœur en veille le reste du temps.

- Créer un projet MPLABX nommé *timer0_test* (ou autre nom) dans le répertoire *disco/bsp/timer0/test/pjct*. Inclure les fichiers *bsp/timer0/test/main.c, bsp/timer0/src/timer0_initc, bsp/timer0/src/timer0_reload.c* et s'assurer de la bonne compilation du projet. S'aider de l'annexe 1.

Timer0 et interruption

- Modifier les sources du projet afin de configurer le Timer0 pour qu'il puisse lever une interruption toutes les 20ms (horloge CPU à 64MHz). S'aider du fichier d'en-tête *bsp/timer0/include/timer0.h* et de la documentation (cartouche doxygen) de la fonction d'initialisation dans ce même header. Prendre l'habitude de parcourir les fichiers d'en-tête durant la trame de TP, et garder cet habitude pour le reste de votre vie dans le monde du logiciel. Les headers servent à générer les documentation de projets.

- Développer une application de test implémentant le logigramme suivant :



- Valider la base de temps à l'oscilloscope (période 40ms, 20ms LED allumée et 20ms LED éteinte)

- Quelle est la charge CPU (durée de travail du CPU par unité de temps) ?

Analyse assembleur

- Compiler puis charger le programme en mode *debug*
- Ouvrir et déplacer dans l'environnement graphique de l'IDE une fenêtre permettant d'observer le code binaire désassemblé : Window → PIC Memory Views → Program Memory
- Parcourir la totalité de la mémoire flash, identifier l'emplacement des fonctions (*main, timer0_init, timer0_reload et l'ISR*) et retranscrire le code assembleur de l'ISR (seulement) ci-dessous

Label	Disassembly

- Comment se nomment les traitements observés en en-tête et pied d'ISR ? Quels sont leurs rôles ?

- Déclarer l'ISR comme une fonction C classique (sans qualificateur de fonction *interrupt*). Réitérer l'exercice d'analyse et observer le code binaire généré. Verdict ?

Mise en veille

- Appeler l'instruction assembleur *sleep* dans la boucle infinie de la fonction principale. Vérifier le bon fonctionnement de l'application

```
asm(''sleep'');
```

- Expliquer le fonctionnement de l'application ?

- Estimer par le calcul la charge CPU (durée de travail du CPU par unité de temps) ? Verdict ?

4. ADC ET CONVERSION

4. ADC ET CONVERSION

4.1. Travail préparatoire

- (2pts) L'ADC (Analog to Digital Converter) intégré dans le PIC18F27K40 de TP génère après conversion un résultat sur 10bits binaire par approximations successives. A l'aide d'un schéma commenté, des cours connexes à l'école et d'internet, expliquer le fonctionnement d'un ADC à approximation successive.

- (1pt) Sachant que la plage analogique de notre ADC sera configurée entre la masse de référence et la tension d'alimentation Vcc (plage analogique 0V-3,3V) et que nous utiliserons notre ADC en mode 8bits (plage numérique 0x00-0xFF). Représenter graphiquement la caractéristique de transfert analogique/numérique de l'ADC ainsi configuré.

- (1pt) Pour la configuration précédemment présentée, quelle est la résolution de l'ADC ?

- (1pt) Qu'est-ce qu'un échantillonneur bloqueur (s'aider des cours connexes à l'école et d'internet) ? En vous aidant de la documentation technique du MCU (section relative à l'ADC, (ADC2) Analog-to-Digital Converter with Computation Module), représenter le schéma de l'échantillonneur bloqueur intégré dans le MCU (spécifier les valeurs des composants passifs responsables des phases de précharge et d'acquisition).

- (2pts) Pour un ADC à approximation successive, par quels facteurs sera contraint le choix de la période d'échantillonnage ? *Attention, la réponse est subtile (analogique externe et interne au MCU, structure de l'ADC, etc), pas de réponse hâtive.*

- (1pt) En analysant le schéma électrique de la carte Curiosity HPC utilisée en TP, dessiner le schéma de connexion du potentiomètre présent sur la carte avec le MCU 28 broches PIC18F27K40. Quelle broche du MCU est utilisée par défaut comme entrée analogique ?

- (1pt) Proposer une configuration en assembleur des registres ADC0N1 (configuration), ADREF (référence analogique) et ADPCH (sélection du canal) assurant l'initialisation de l'ADC suivante :

    - Mode : *8bits*
    - Plage analogique : *0V-3,3V*
    - Broche : *RA0/AN0*
    - Source d'horloge : *interne FRC (Fixed Reference Clock)*
    - Divers : *mode discontinu, pas d'interruption*

4. ADC ET CONVERSION

4.2. Travaux en séances

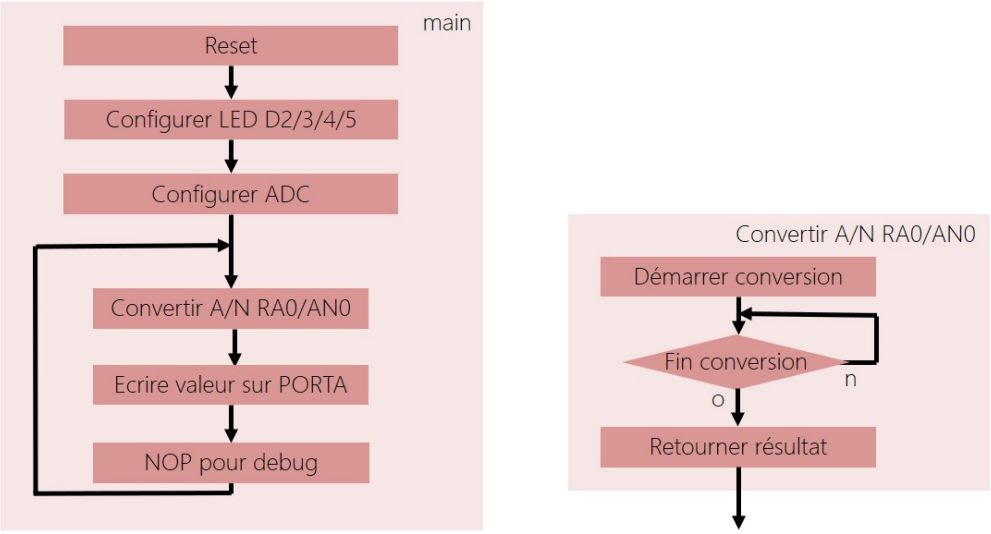
Cet exercice vise à lier et illustrer deux grands pans du large domaine de l'électronique, celui de l'analogique et du numérique. Nous allons découvrir une problématique de conversion sur technologie à approximation successive, solution la plus couramment rencontrée sur MCU pour sa faible empreinte sur silicium.

- Créer un projet MPLABX nommé *adc_test* (ou autre nom) dans le répertoire *disco/bsp/adc/test/pjct*. Inclure les fichiers *bsp/adc/test/main.c, bsp/adc/src/adc_initc, bsp/adc/src/adc_read.c* et s'assurer de la bonne compilation du projet. S'aider de l'annexe 1.

Conversion Analogique/Numérique 8bits

- Modifier les sources du projet afin de configurer l'ADC pour qu'il puisse réaliser une conversion analogique/numérique sur la broche RA0/AN0 (broche connectée au potentiomètre sur la carte Curiosity HPC). S'aider du fichier d'en-tête *bsp/adc/include/adc.h* et de la documentation (cartouche doxygen) de la fonction d'initialisation dans ce même header.

- Développer une application de test implémentant le logigramme suivant :



- Valider le fonctionnement de l'application par observation des états logiques appliqués sur les LED 2/3/4/5 connectées au port A.

- En mode debug, mettre un point d'arrêt sur l'instruction *nop*. Observer les valeurs numériques convertis présentes dans les registres de sortie de l'ADC 10bits (ADRESH et ADRESL) lorsque le potentiomètre est en butée (gauche et droite) : PIC/Target Memory Views → SFRs. Pourquoi la plage numérique convertie n'est-elle pas comprise entre 0x000 et 0x3FF (0V-5V) ?

5. UART
COMMUNICATION ORDINATEUR

5. UART ET COMMUNICATION ORDINATEUR

5.1. Travail préparatoire

- Lire attentivement le chapitre *UART – Universal Asynchronous Receiver Transmitter* du document support aux TP *mcu-tp-peripheriques.pdf*. Ne pas hésiter à s'aider également d'internet. Pour travailler efficacement, utiliser également la documentation technique ou datasheet du MCU PIC18F27K40

- (1pt) Qu'est-ce qu'un périphérique UART ? *Proposer un schéma commenté*

- (1pt) Présenter le protocole de communication d'une liaison série asynchrone. *Proposer un chronogramme commenté*

- (1pt) Proposer un chronogramme présentant l'envoi successif sans temps d'attente des caractères 'D', 'U', 'B' (8bits de donnée, 1 bit de stop et pas de bit de parité).

- (1pt) Qu'est-ce que la norme RS232 et préciser les principales spécifications techniques.

- (1pt) FTDI est un fabricant de composants électroniques. En vous rendant sur leur site web, préciser en quoi ils sont spécialisés sur le marché du silicium.

- (1pt) Quels sont les registres à configurer pour utiliser l'UART1 du MCU PIC18F27K40 utilisé en TP.

- (1pt) Proposer un configuration en assembleur de l'UART1 respectant les contraintes suivantes :

    - Protocole : *payload 8bits,1bit de stop, pas de détection d'erreur*
    - Débit : *9600KBd/s, 16bits mode, high speed mode*

5. UART ET COMMUNICATION ORDINATEUR

5.2. Travaux en séances

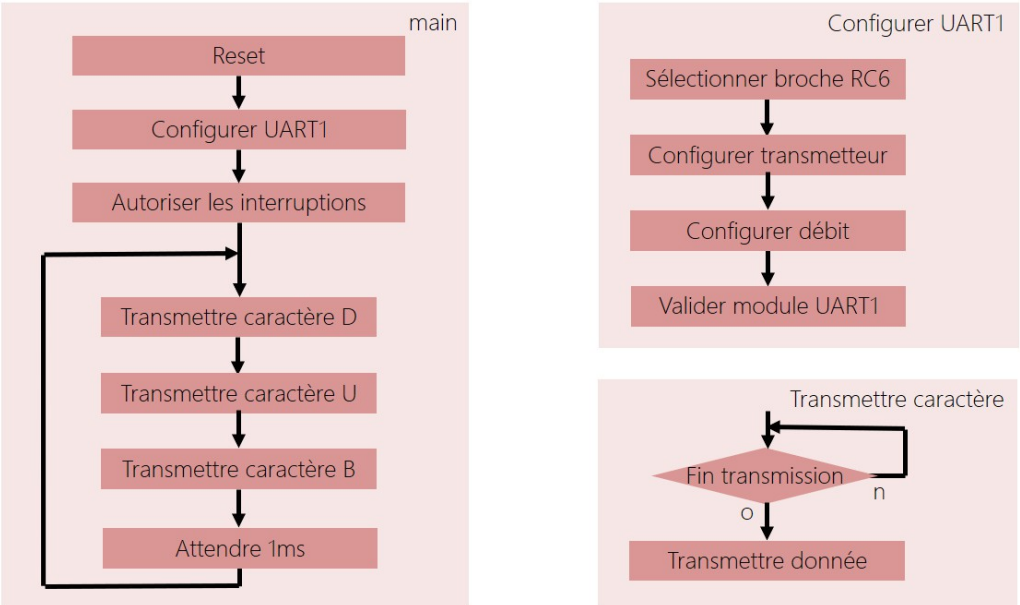
Les exercices précédents avaient pour objectifs la prise en main des outils de développement (logiciel et matériel), comprendre le principe de fonctionnement d'un processeur à CPU illustré sur MCU, s'initier aux mécanismes de communication par interruption des périphériques avec le CPU et découvrir les problématiques de conversion analogique/numérique. Nous allons maintenant découvrir un point central des systèmes électroniques, leur faculté à communiquer entre eux. Cette première découverte se fera à travers une communication série asynchrone entre notre processeur de TP et un ordinateur équipé d'un terminal dédié (TeraTerm, PuTTY, minicom, etc).

- Créer un projet MPLABX nommé *uart1_test* (ou autre nom) dans le répertoire *disco/bsp/uart1/test/pjct*. Inclure les fichiers *bsp/uart1/test/main.c, bsp/common/delay_1ms.c* et *uart1_init.c, uart1_putc.c, uart1_puts.c, uart1_getc.c, uart1_gets.c* présents dans le répertoire *bsp/uart1/src/*. S'assurer de la bonne compilation du projet. S'aider de l'annexe 1.

UART1 en transmission

- Modifier les sources *uart1_init.c* et *uart1_putc.c* afin de configurer l'UART1 pour que le périphérique puisse implémenter une communication avec un débit de 9600Bd/s et respectant la configuration spécifiée dans le fichier d'en-tête *bsp/uart1/include/uart1.h.* S'aider de la documentation (cartouche doxygen) de la fonction d'initialisation dans ce même header. Dans un premier temps, nous ne configurerons que le transmetteur sans gestion d'interruption et le générateur de débit.

- Développer une application de test implémentant le logigramme suivant :



- Observer à l'oscilloscope puis représenter ci-dessous les trames transmises par l'UART1 (broche RC6). Valider le travail préparatoire.

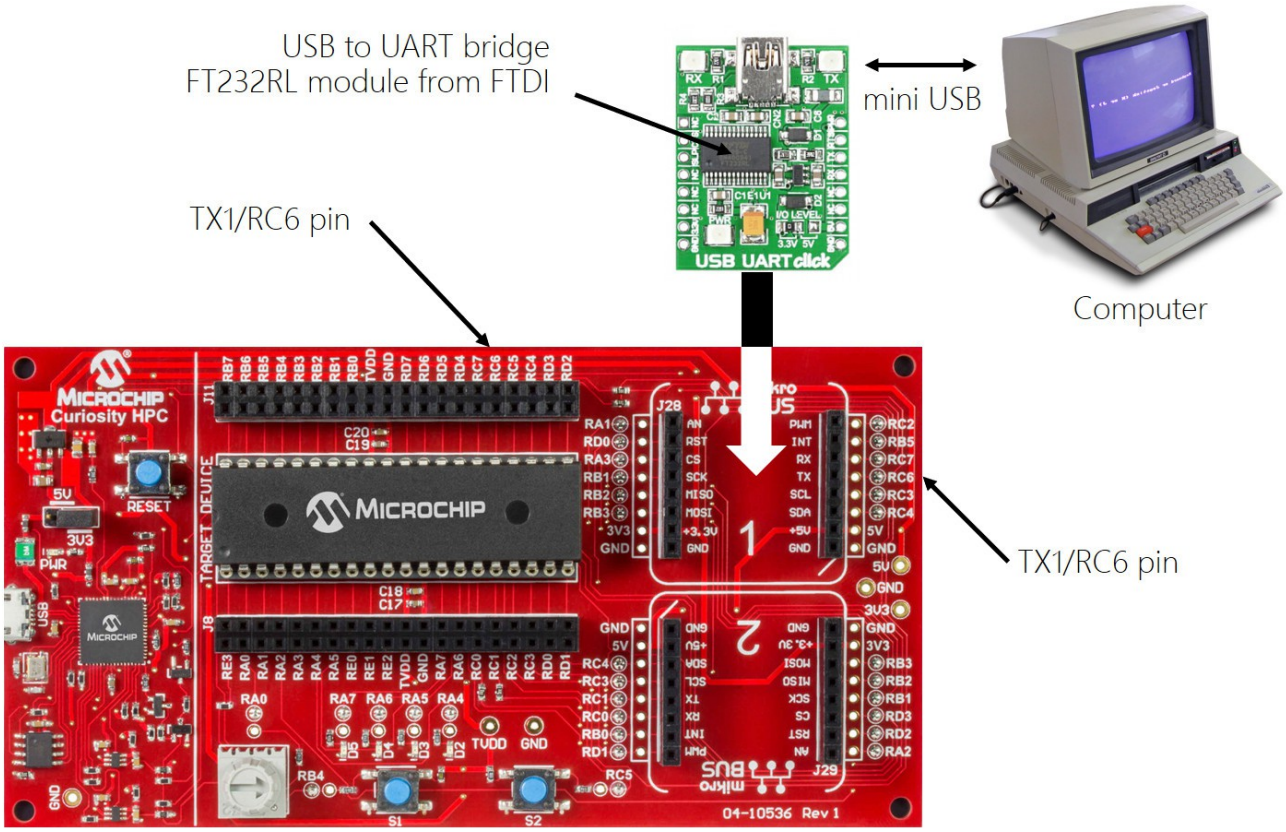
- Valider également votre configuration avec le simulateur proposé par MPLABX :

  - *Clic droit sur le nom du projet*
  - *Properties*
  - *Hardware Tool → Simulator → Apply*
  - *Conf : [default] → Simulator*
  - *Option categories → UART1 IO Options → Enable UART1 IO → Apply → OK*
  - *Programmer en mode debug → Observer la fenêtre de sortie UART 1 Output*

- Une fois validé, désélectionné le simulateur à sélectionné à nouveau la carte *Curiosity*

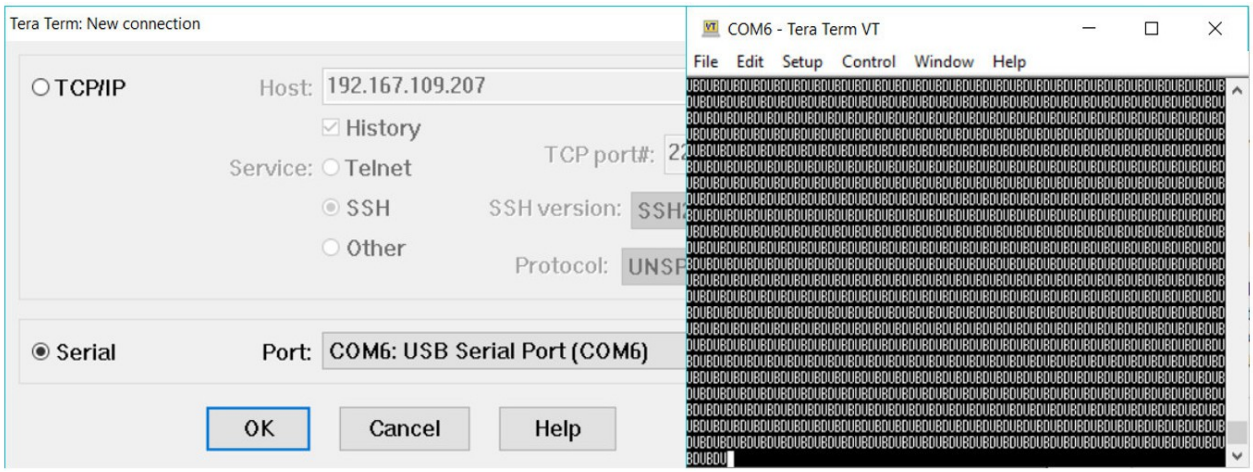
Interface de communication série sur ordinateur

Nous allons maintenant valider nos développements en déployant une communication série avec un ordinateur. Vérifier que le module *click board* de *Mikroelektronika USB to UART* soit bien physiquement connecté au socket *mikro BUS 1* dédié sur la carte Curiosity HPC. Vérifier également la connexion en USB avec l'ordinateur.



- Ouvrir un terminal de communication série sur ordinateur et validé le bon fonctionnement du programme. Utiliser par exemple TeraTerm sous Windows :

    - *Exécuter l'application C:\Program Files (x86)\teraterm\ttermpro.exe*
    - *Serial → sélectionner votre interface de communication → OK*
    - *Setup → Serial Port... → valider la configuration de la communication série → OK*
    - *Il ne reste plus qu'à observer les retours dans la console. De même, tout caractère saisi sur cette même console sera envoyé par le module série virtuel depuis l'ordinateur (port COM sous Windows)*



Transmission de chaînes de caractères

- Modifier le fichier source *uart1_puts.c* afin de valider l'envoi de chaînes de caractères par UART1. Modifier la fonction *main* du programme en conséquent.

```
uart1_puts(''DUB'') ;
```

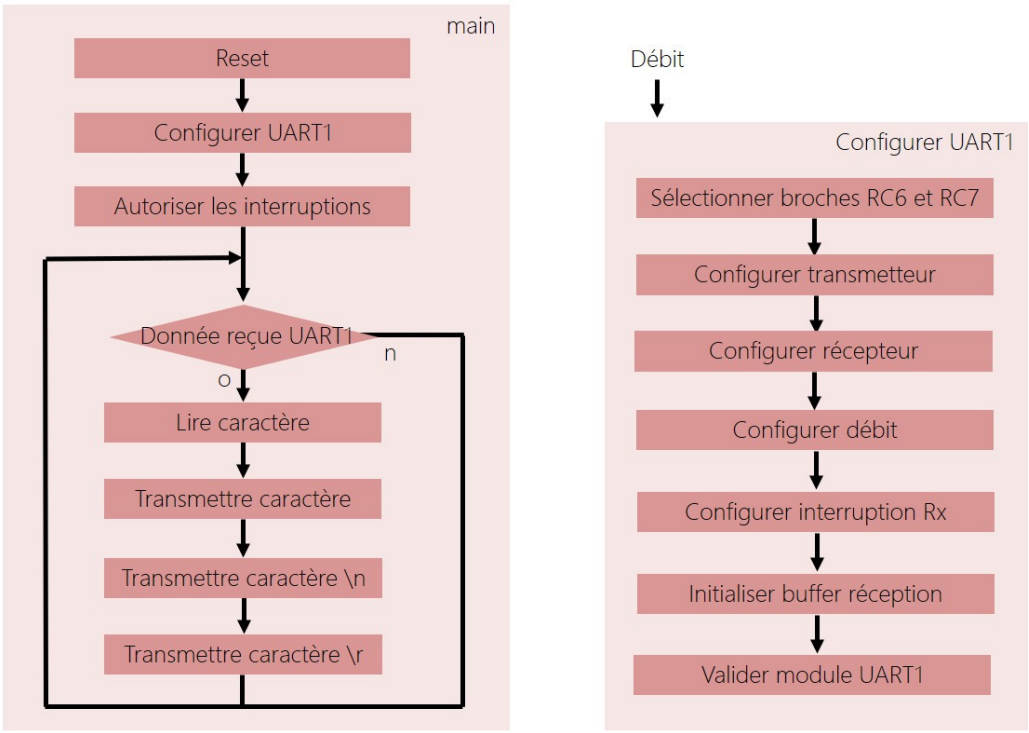
- Valider le bon fonctionnement de votre programme en utilisant une console série sur ordinateur.

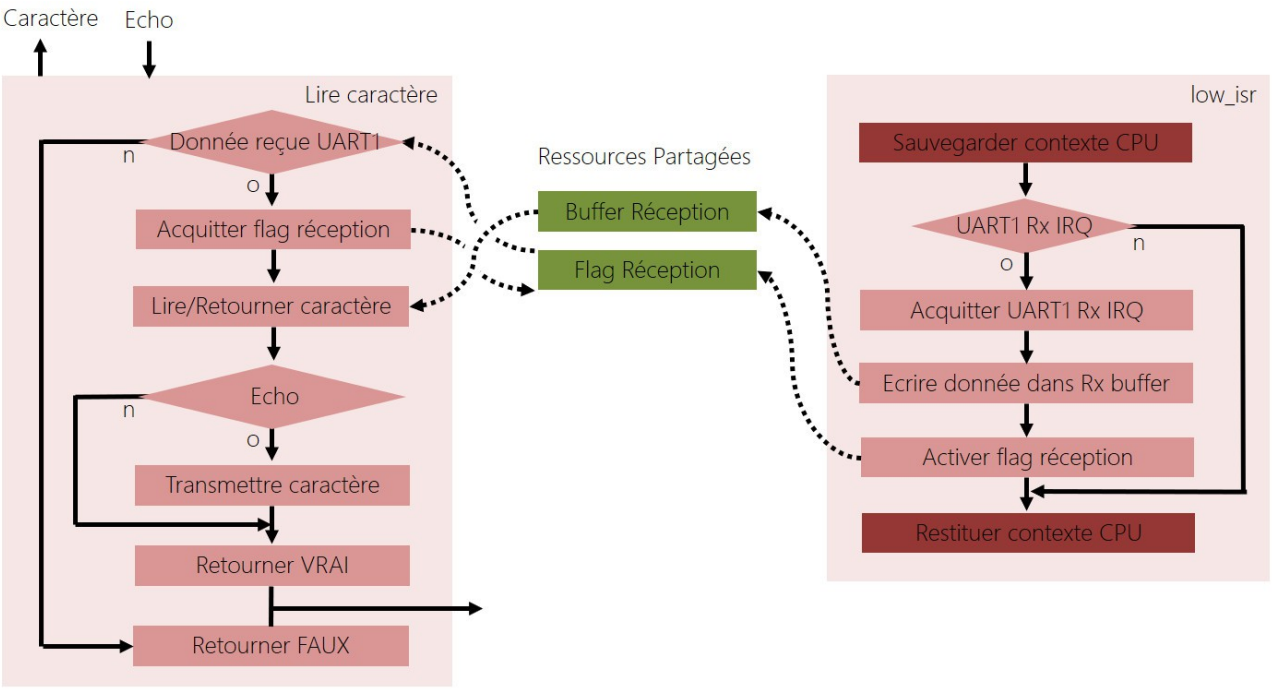
UART1 en réception

- Modifier les sources *uart1_init.c* et *uart1_getc.c* afin de configurer l'UART1 pour que le périphérique puisse implémenter la réception de données en respectant la configuration spécifiée dans le fichier d'en-tête *bsp/uart1/include/uart1.h.* S'aider de la documentation (cartouche doxygen) de la fonction d'initialisation dans ce même header. Nous aurons à configurer l'interruption en réception en priorité basse ainsi que le récepteur de l'UART1. Pour information, en générale sur MCU pour acquitter une interruption en réception pour un UART il nous suffit et lire et donc vider le registre de réception. C'est le cas sur MCU PIC18. Attention, les fonctions du fichier *uart1_getc.c* doivent probablement être les fonctions les plus complexes à écrire de la trame de TP. Être donc attentif !

- Quelle est la broche de réception Rx utilisée par défaut par l'UART1 ?

- Développer une application de test implémentant le logigramme suivant. L'utilisation de fonctions d'interruption ou ISR dans une application amène le nouveau paradigme de programmation événementielle. En effet, nous ne pouvons prédire le réveil d'une ISR (sur événement physique externe ou interne), une routine d'interruption est donc entièrement indépendante de la fonction main. Pour la première fois en 1A, nous allons devoir utiliser des variables globales (ressources partagées) afin de faire communiquer ISR et fonctions appelées depuis le *main*. Nous aurons besoin d'échanger deux informations. La donnée reçue et un flag précisant qu'une donnée est bien arrivée. En langage C, toujours déclarer une variable globale dans un fichier source, le plus souvent dans le source où est présent l'écrivain.





- Valider le bon fonctionnement de votre programme en utilisant une console série sur ordinateur.

- Après validation, envoyer maintenant le contenu d'un fichier fichier texte depuis la console. Voici la procédure sous TeraTerm :

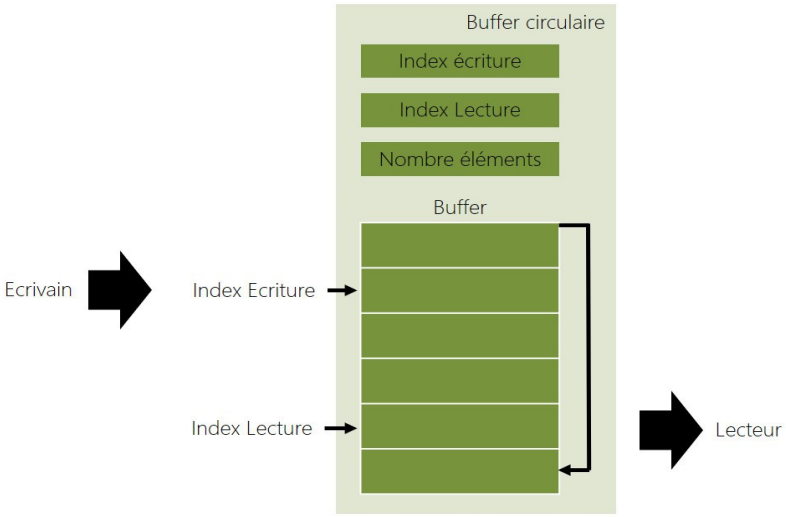
    - *Se placer sur la console TeraTerm*
    - *File → Send file...*
    - *Sélectionner le fichier texte suivant disco/bsp/uart1/test/rx_test_file1-> Ouvrir*

- Pourquoi n'observe-t-on qu'un caractère sur trois en retour sur la console ?

- Proposer des solutions à ce problème

Buffer circulaire de réception

Actuellement, le buffer d'échange entre l'ISR et la fonction de lecture de caractère ne fait qu'un octet. Nous allons le remplacer par un buffer de taille configurable (à la compilation) géré circulairement. Vous pouvez observer une définition de type structuré représentant un objet buffer circulaire dans le fichier d'en-tête *uart1.h*. Dans notre cas, l'écrivain sera l'ISR et le lecteur la fonction *getc*. Lorsque l'écrivain écrit dans le buffer, il incrémente l'index d'écriture ainsi que le nombre d'éléments présents dans le buffer. Si le buffer est plein, il stop les écritures. Les donnée sont alors perdues. Lorsque que l'index d'écriture pointe la fin du buffer, il bascule en position initiale. Il est alors géré circulairement. A chaque récupération, le lecteur incrémente l'index de lecture et décrémente le nombre d'éléments dans le buffer. Si le buffer est vide, le lecteur passe son tour.



- Implémenter un buffer circulaire de 20 éléments entre l'ISR et fonction *getc*. Valider le fonctionnement du programme par envoi du fichier texte *rx-test-file1.txt* depuis le terminal de communication série.

- Après validation, envoyer maintenant le contenu du fichier texte *rx-test-file3.txt* depuis la console. Pourquoi certains caractères sont-ils perdus durant la transmission ?

- Proposer des solutions à ce problème.

Contrôle de flux logiciel *(facultatif)*

A faire par les étudiants en avance. A ce niveau des développements, deux solutions s'offrent à nous si nous ne souhaitons plus perdre de données à la réception. La première consisterait à étendre la taille du buffer circulaire. Néanmoins cette solution ne résout pas le problème mais repousse seulement les limites. Pour information, selon l'utilisation de l'UART dans une application, ces limitations peuvent être amplement suffisantes (ce sera souvent le cas). Néanmoins, une solution optimale serait d'implémenter un contrôle de flux. Le concept est simple, si le buffer de réception est plein, le récepteur demande explicitement à l'émetteur d'arrêter de transmettre. Dès que suffisamment de place ait été libéré dans son buffer, le récepteur prévient l'émetteur de reprendre sa transmission. Ce contrôle des flux de communication peut être matériel ou logiciel :

  - *Contrôle de flux matériel :* Utilisation de deux broches et conducteurs complémentaires (CTS et RTS). Les deux signaux sont croisés et permettent par simple logique de stopper ou valider les communications.

  - *Contrôle de flux logiciel :* Sans utiliser de broches complémentaires, le contrôle de flux se fait par envoi de caractères spéciaux (XON ou 0x11, et XOFF ou 0x13). Si l'émetteur reçoit le caractère XOFF, il stoppe les transferts. Dès qu'il recevra le caractère XON, il reprendra les échanges.

- Implémenter et valider un contrôle de flux logiciel. La solution est simple, 5 lignes de codes. Ne donc pas se perdre dans l'implémentation. Les limites seront les suivantes :

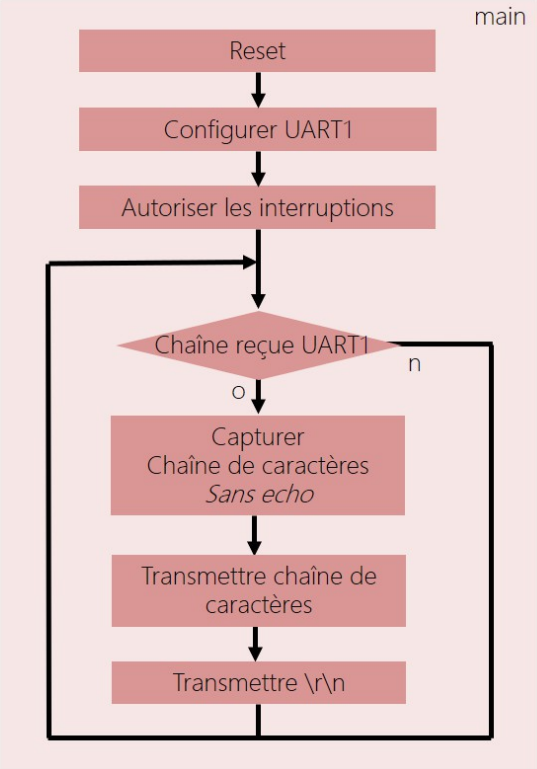
  - Envoi du caractère XOFF depuis l'ISR (écrivain) dès que le buffer circulaire de réception est rempli à 4 éléments du débordement (BUFF_SIZE - 4). La taille minimale du buffer sera fixée à 6 éléments.

  - Envoi du caractère XON depuis la fonction *getc* (lecteur) dès que la moitié du buffer a été vidée.

Réception de chaîne de caractères

- Modifier le fichier source *uart1_gets.c* afin d'assurer la réception de chaînes de caractères. Depuis la console, une action sur la touche Entrée (Enter ou '\r') signifiera la fin d'une saisie de chaîne de caractères. A la réception, la fonction devra remplacer le caractère *\r* par un *\0*. Si la chaîne de caractères capturée dépasse une taille fixée en entrée de fonction (paramètre de fonction), l'écriture du tableau de destination se stoppera et se clôturera également par l'écriture d'un \0.

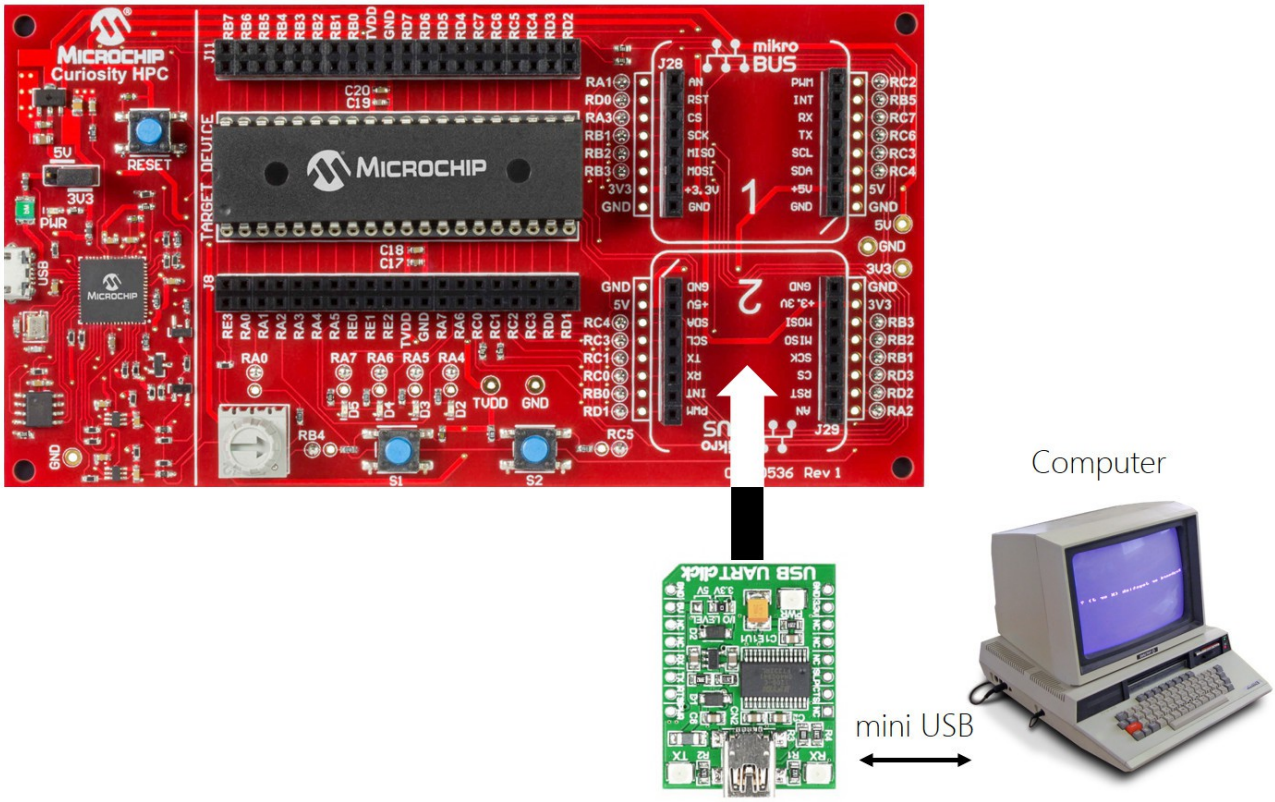
- Développer une application de test implémentant le logigramme suivant :



- Valider le bon fonctionnement de votre programme en utilisant une console série sur ordinateur.

- Envoyer maintenant le contenu du fichier texte *rx-test-file3.txt* depuis la console. Si le contrôle de flux logiciel est implémenté ainsi qu'un buffer circulaire de taille suffisante, alors la magie devrait opérer !

Configuration et validation UART2



- Vérifier que le module *click board* de *Mikroelektronika USB to UART* soit bien physiquement connecté au socket *mikro BUS 2* dédié sur la carte Curiosity HPC. Créer un projet MPLABX nommé *uart2_test* (ou autre nom) dans le répertoire *disco/bsp/uart2/test/pjct*. Réitérer l'ensemble des développements (quasiment que de la recopie avec changement d'indice), des tests et des validations précédents pour l'UART2.

6. BLUETOOTH