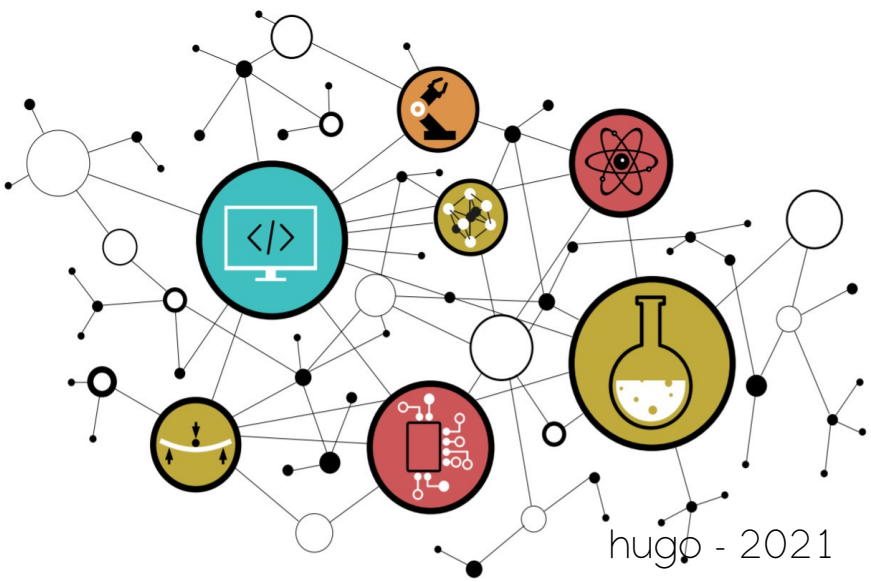
TRAVAUX PRATIQUES

MODULE DE BROCHE GPIO
ET ASSEMBLEUR PIC18

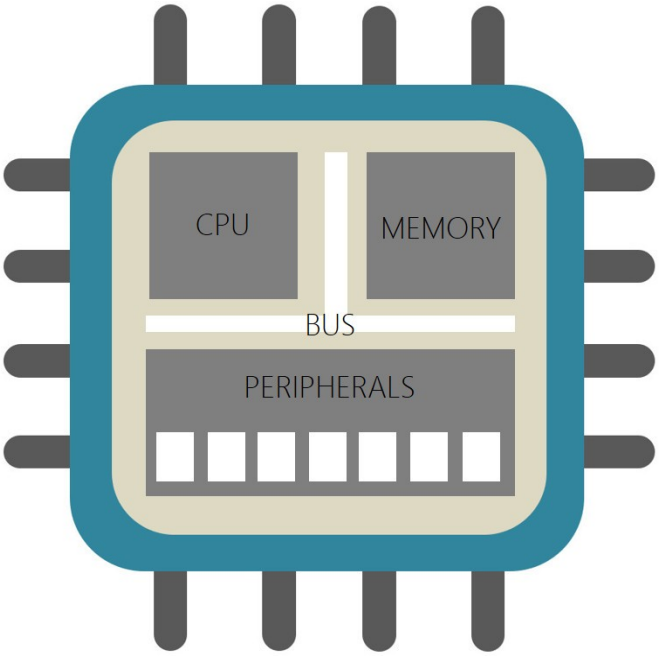


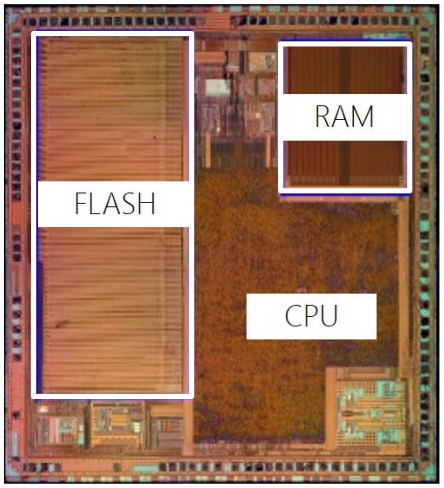
hugo - 2021

SOMMAIRE

MODULE DE BROCHE GPIO ET ASSEMBLEUR PIC18

2. MODULE DE BROCHE GPIO ET ASSEMBLEUR PIC18





Die MCU 32bits Microchip
MCU PIC32MX340F512
MIPS32 M4K Core 5-stage Pipeline
Memory 512Ko Flash / 32Ko SRAM

Au fil des introductions de chaque document de TP, nous allons vous présenter les architectures et les fonctionnements génériques de périphériques standards (GPIO, Timer et UART) présents sur la grande majorité des MCU du marché (Micro Controller Unit ou Microcontrôleur). Pour chaque périphérique, une illustration technologique sur PIC18 (solution MCU 8bits propriétaire Microchip) sera également proposée ainsi qu'un exemple de configuration bas niveau en assembleur PIC18.

Rappelons que sur processeur MCU, la mémoire (program Flash et data SRAM) et le CPU (Central Processing Unit) sont respectivement chargés de stocker l'information (programme et données) puis de la traiter. La mémoire morte Flash non-volatile stocke de façon persistante le programme (ou code) et la mémoire vive SRAM volatile stocke à l'exécution les données en cours de manipulation par le CPU. D'une implémentation technologique à une autre, l'architecture et donc l'empreinte silicium de ces deux éléments fondamentaux pendront plus ou moins de place sur le *die* (puce silicium). A titre indicatif, l'exemple ci-dessus montre les contraintes d'intégration d'un MCU 32bits Microchip du marché. Les services proposés par un processeur seront toujours le fruit de compromis technologiques liés à l'intégration sur silicium. D'un point de vu étymologique, l'ensemble CPU/Mémoire représente déjà à lui seul un processeur (stockage et traitement de l'information). Une fois l'information stockée puis traitée, l'application sera chargée de l'échanger vers l'extérieur du système. Les périphériques entrent alors en jeu.

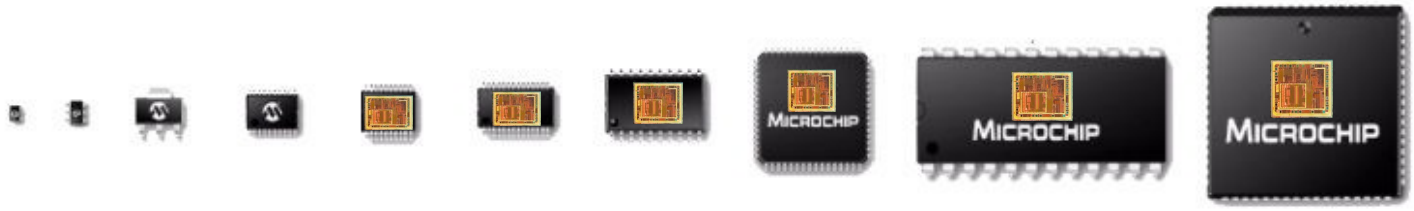
Un périphérique est un service matériel proposé par la machine. Les fonctions matérielles spécialisées périphériques à l'ensemble CPU/Mémoire, plus communément nommées ''périphériques'', jouent généralement l'un des trois rôles suivants dans le système :

- *Communiquer :* Fonction spécialisée d'échange et de partage d'information de machine à machine implémentant un protocole de communication souvent normalisé voire standard (UART, SPI, I2C, USB, Ethernet, CAN, etc). Un protocole de communication assure une encapsulation de l'information ou charge strictement utile (payload).avant transmission ou réception par trames de communication.

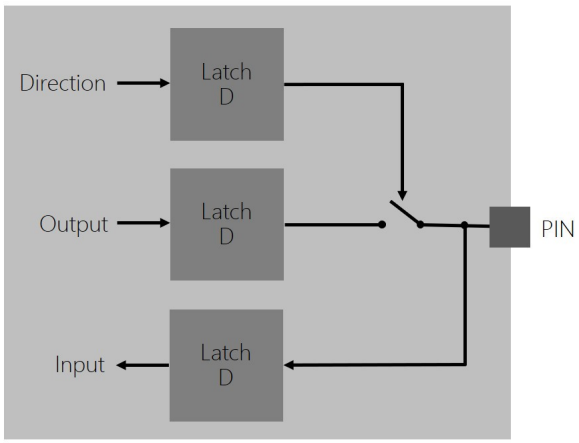
- *Convertir :* Fonction spécialisée de conversion (GPIO, ADC, DAC, PWM, etc) entre les domaines de l'électronique analogique (signaux physiques continus) et numérique (signaux logiques discrets)

- *Traiter (voire accélérer) :* Fonction spécialisée interne (Timer, DMA, Crypto, FFT, etc) de traitement (comptage, copie, calcul, etc). Ceci permet d'aider le CPU à se dédier à la supervision du système par exécution du programme applicatif voire dans certains cas à exécuter du calcul algorithmique.

2.1. Module GPIO ou General Purpose Input Output

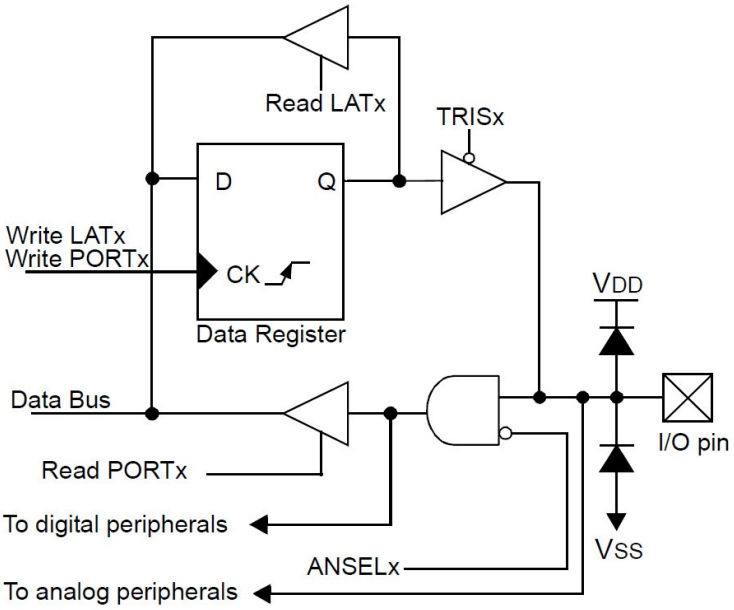


Une broche (ou pin) est une interface physique présente et visible à l'œil nu sur le boîtier d'un MCU ou microcontrôleur. Il est à noter qu'une même puce de silicium (die), un MCU par exemple, peut être encapsulée dans différentes technologies de boîtier (DIP, SOIC, BGA, QFN, etc). Chaque technologie offre son lot d'avantages et d'inconvénients (coût, encombrement, facilité de maintenance, procédé d'intégration, etc). Une broche assure une interface physique avec le module GPIO présent quant à lui sur la puce de silicium constituant le processeur MCU (cf. schéma fonctionnel ci-dessous). Sur tout processeur du marché (MCU, DSP, MPPA, FPGA, etc), une broche d'entrée/sortie généraliste ou GPIO (General Purpose Input Output) offrira toujours un service de configuration de la direction (entrée ou sortie) de type TOR (Tout Ou Rien, par exemple 0V ou Vcc) puis un service d'utilisation en lecture ou en écriture. L'objectif étant de pouvoir lire ou écrire l'état d'une broche. Ceci peut notamment servir d'interface avec des fonctions périphériques externes au processeur. Ces fonctions matérielles externes seront portées sur le PCB (Printed Circuit Board ou circuit imprimé). Par exemple, les GPIO's peuvent permettre d'interfacer des LED ou différents actionneurs (afficheur, servomoteur, contacteur, etc), mais également à lire l'état de boutons poussoirs, d'interrupteurs voire de différents capteurs du marché (capteur électromécanique de fin de course, capteurs optiques, etc).

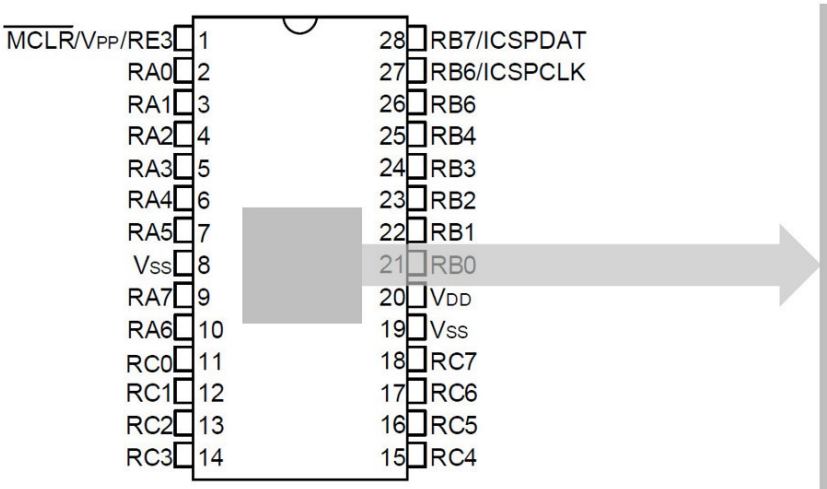


Un module GPIO associé à une broche est une fonction matérielle interne à un MCU. Il est constitué sur silicium d'un circuit logique. La figure ci-dessus ne présente par exemple que le schéma symbolique pour la gestion d'une seule broche. Chacun des états logiques de configuration et d'utilisation d'une broche sera sauvegardé par des bascules (bascule D le plus souvent) assurant la mémorisation logique de ces mêmes états. Nous nommons un ensemble de bascules un registre. Pour un MCU, un PORT (de broches) est un ensemble de broches gérées par registre. De façon générale sur un processeur, un registre peut s'agir de registre de configuration (à écrire), d'utilisation (à lire et écrire) voire d'état (à lire). D'une implémentation technologique à une autre les registres auront des fonctions, des rôles, des noms et des tailles différentes (exemples des technologies MCU Microchip PIC18, MCU STMicroelectronics STM32, MCU Texas Instruments MSP430, etc). Ces aspects sont liés à la technologie utilisée. Une lecture et analyse des documentations techniques du constructeur est donc indispensable dans ce domaine.

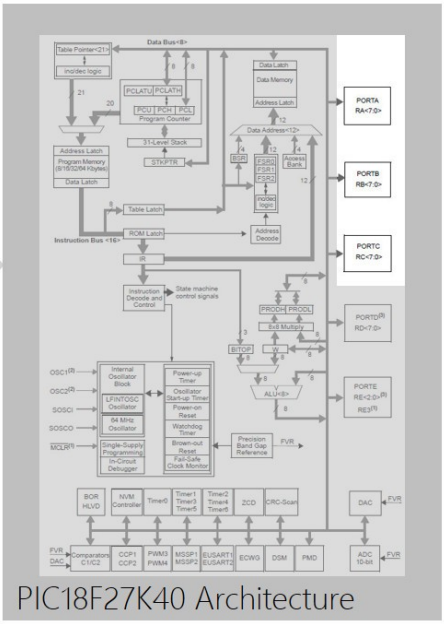
2.2. Module GPIO sur MCU PIC18



Le schéma précédent est extrait de la documentation technique ou datasheet d'un MCU PIC18F27K40 proposé par Microchip. Il présente plus en détail l'architecture matérielle réelle cachée derrière chaque GPIO et broche. Un PORT d'entrée sortie (PORTA, PORTB et PORTC sur PIC18F27K40 avec boîtier 28 broches) représente un ensemble de 8 broches ou pins de type GPIO. Les ports sont manipulables par utilisation de registres 8bits (regroupement en tout de 8 GPIO par PORT).



MCU PIC18F27K40
Packages SPDIP, SSOP, SOIC



PIC18F27K40 Architecture

Cependant, en fonction des besoins dans l'application, ces broches peuvent être configurées et utilisées pour d'autres usage. Les broches sont alors physiquement routées en interne vers d'autres fonctions matérielles périphériques (Timer, UART, ADC, SPI, I2C, etc). Tout ceci doit explicitement être programmé par le développeur et est documenté dans la datasheet du composant.

2.3. Configuration des GPIO sur PIC18

Sur PIC18, les registres de configuration en entrée/sortie des ports sont nommés TRISx (x=A,B, C sur PIC18F27K40) pour Tri-state (3 états : haut ou 1, bas ou 0, et haute impédance ou Z). Les registres d'écriture sont nommés LATx (pour Latch car associé à une bascule D latch) et les registres de lecture PORTx. En résumé, voici les usages des 3 familles de registres 8bits pour la gestion des GPIO sur MCU PIC18 :

- TRISx : Configuration (0=Output et 1=Input) de la direction des broches du PORT x (x=A,B ou C)
- LATx : Écriture (0=0v et 1=Vcc) de l'état logique des broches du PORT x (x=A,B ou C)
- PORTx : Lecture (0=0v et 1=Vcc) de l'état logique des broches du PORT x (x=A,B ou C)

Ouvrir la datasheet des processeurs PIC18Fx7K40 et parcourir le chapitre 15 relatif aux GPIO (I/O Ports) afin d'observer la configuration des registres (mcu/tp.doc/datasheets). La séquence assembleur PIC18 ci-dessous présente des exemples de configuration et de gestion des ports en assembleur PIC18.

```
; all PORTA pins are inputs

MOVLW       0xFF
MOVWF       TRISA


; all PORTB pins are outputs

MOVLW       0x00
MOVWF       TRISB


; pin RC3 is an output

BCF       TRISC, 3


; pin RC0 is an input

BSF       TRISC, 0
```

```
; read all PORTA pin and save value in W (Work) CPU register

MOVF       PORTA, 0
or
MOVFF      PORTA, WREG


; pins RB0-RB3 are set to low level and RB4-RB7 to high

MOVLW       0xF0
MOVWF       LATB


; set RC3 to high level

BSF       LATC, 3


; test if RC0 input level is low and perform action

BTFSC       PORTC, 0
<do_this_if_high>
<do_this_if_low>
```

2.4. Assembleur ou langage d'assemblage PIC18

```
main:   movlw     7
        movwf     data_address_in_data_memory
        movwf     TRISA
        goto      main
        return
```

L'assembleur (assembly) ou langage d'assemblage (assembly langage) est le langage de programmation de plus bas niveau sur la machine. Il est la conversion directe lisible par l'homme (mode texte) du programme exécutable par le CPU de la machine (code binaire). Il est de ce fait, le langage le moins universel au monde, car dépendant du jeu d'instructions supporté par le CPU cible. Entre les marchés des ordinateurs et des systèmes embarqués, un grand nombre de fabricants implémentent des modèles d'exécution CPU (RISC ou CISC, Von Neumann ou Harvard, 8-16-32-64bits, entier voire flottant, scalaire voire vectoriel, VLIW ou superscalaire, etc) sur des technologies différentes (x86, x64, ARM, PIC18, RISC-V, C6000, etc). L'assembleur présenté ci-dessus est par exemple de l'assembleur 8bits PIC18 développé par la société Américaine Microchip pour leur propre gamme de MCU 8bits PIC18. Il s'agit d'un assembleur propriétaire contrairement au solutions Open Hardware RISC-V actuellement rencontrées sur le marché. Comme tout langage de programmation, un programme assembleur se lit de haut en bas, à l'image du modèle d'exécution séquentiel d'un CPU. Même si l'assembleur n'est pas universel, certaines représentations conceptuelles liées au langage peuvent néanmoins être généralisées :

```
label:    instruction    opérandes
```

- Label : un label ou étiquette, est une référence symbolique (simple chaîne de caractères) représentant l'adresse mémoire logique (emplacement) de la première instruction suivant celui-ci. Les labels sont remplacés par les adresses logiques voire physiques à l'édition des liens. Un label se termine par : afin de le différencier d'éventuelles directives d'assemblage voire d'instructions.

- Instruction : une instruction est un traitement élémentaire à réaliser par le CPU. Par exemple, charger (load) une donnée depuis la mémoire vers le CPU ou sauver (store) une donnée depuis le CPU vers la mémoire, réaliser une opération arithmétique ou logique, déplacer une donnée de registre à registre, réaliser un saut dans le programme, etc. L'ensemble des instructions exécutables par un CPU est nommé jeu d'instructions (ISA ou Instruction Set Architecture).

- Opérandes : les opérandes, lorsque l'instruction en utilise, sont les données ou les emplacements de données (registres ou adresses en mémoire principale) manipulées par l'instruction. Nous distinguons les opérandes sources, utilisées comme entrées avant l'exécution de l'instruction, de l'opérande de destination pour sauver le résultat. Les méthodes d'accès aux données en assembleur sont nommées des modes d'adressage :

    - Mode d'adressage registre : l'opérande est un registre CPU dans lequel est sauvé une donnée. *Ce mode d'adressage n'existe pas sur PIC18 car le CPU PIC18 n'intègre qu'un seul registre de travail, le registre W (WORK). Tous les autres registres du processeur MCU sont accessibles par adresse unique associée à chaque registre.*

    - Mode d'adressage immédiat : l'opérande est une constante dont la valeur sera sauvée dans le code binaire de l'instruction. *Par exemple, les instructions movlw 0 ci-dessus. MOVLW signifie MOV (déplacer) L (littéral ou immédiat, donc une constante) dans le registre CPU nommé W (WORK)*

    - Mode d'adressage direct (accès en mémoire donnée) : l'opérande est directement l'adresse de la case mémoire de la donnée. *Par exemple, les instructions movwf data_address_in_data_memory et movwf TRISA ci-dessus.*

    - Mode d'adressage indirect (accès en mémoire donnée) : l'opérande est l'adresse de la case mémoire de la donnée stockée indirectement dans un registre CPU.

2.5. My first MPLABX project from scratch

Ce premier exercice a pour objectif la prise en main des outils de développement, matériel comme logiciel. Nous réaliserons également des analyses bas niveau assembleur des programmes développés. Nous profiterons de ce premier TP d'analyse pour réaliser 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 document nommé *mcu-tp-1-prelude.pdf*
- Si ce n'est pas déjà fait, télécharger l'archive de travail *mcu.zip* sur la plateforme moodle. L'extraire à la racine de votre lecteur réseau école ou dans un répertoire dédié sur votre machine personnelle. Par exemple, ne pas l'extraire dans le répertoire téléchargement. Ordonnez dors et déjà vos développements, cela fait partie du métier d'ingénieur. Tous les développements et créations de projets de l'ensemble des travaux pratiques durant ce semestre se feront dans le répertoire *mcu/tp/disco* (disco comme discovery).

> Je répète une fois de plus le point précédemment cité car je le vois apparaître bien trop souvent chaque année. Aucun développement, ni aucun projet MPLABX n'a à être créé à l'extérieur du répertoire *mcu/tp/disco* durant cette trame de TP. Soyez vigilant sur ce point durant la création de vos projets sous l'IDE MPLABX. Ceci permet d'éviter une dispersion de vos développements durant votre processus de création et facilite le partage de projet logiciel dans une optique de travail collaboratif en équipe. Ce qui sera le cas en entreprise. Votre expérience future vous montrera que ce point peut faire gagner ou perdre des semaines voire des mois de développement sur des projets collaboratifs en ingénierie.

Durant ce premier exercice décorrélé du reste de la trame de TP, nous allons juste créer un programme simple *from scratch* (en partant de zéro) et manipuler quelques GPIO (broches). L'objectif étant juste de comprendre l'architecture minimal d'un programme C sur nos outils de développement (IDE MPLABX et toolchain C XC8) et notre architecture processeur (MCU 8bits PIC18F27K40).

- Créer un fichier *main.c* dans le répertoire suivant *disco/apps/fromscratch/main.c. Sous Windows :*
  - *Clic droit → Nouveau → Document texte → créer le fichier main.txt.*
  - *Aller dans l'onglet Affichage de la fenêtre courant de l'explorateur de fichiers → Afficher / Masquer → [x] Extensions de noms de fichiers*
  - *Modifier l'extension du fichier main.txt en main.c*
- Créer un projet MPLABX nommé *fromscratch* dans le répertoire *disco/apps/fromscratch* . Inclure le fichier *disco/apps/fromscratch/main.c* puis éditer un programme C minimal (cf. ci-dessous). S'aider des tutoriels dans *mcu/tp/doc/tutos*
  - *Compiler le projet : fenêtre Projects > clic droit sur le nom du projet > Clean and Build*

```
main(){}
```

- Modifier le programme de façon à allumer puis éteindre successivement dans un *while(1)* les trois LED D2, D3 et D4 de la carte *curiosity HPC*. Compiler et analyser l'erreur de sortie !

```
main(){
    TRISA = 0x00 ;
    while (1){
        LATA = 0x70 ;
        LATA = 0x00 ;
    }
}
```

- Résoudre l'erreur de compilation en ajoutant le fichier d'en-tête constructeur pour notre MCU

```
#include <pic18f27k40.h>
```

Nous allons maintenant configurer le cadencement de l'horloge de travail du processeur (horloge de référence). Chez Microchip sur processeur PIC, nous parlons de bits de configuration.

- Ouvrir la documentation technique du PIC18F27K40 (répertoire *disco/bsp/doc/datasheets*) et analyser le schéma structurel matériel du sous système d'horloge intégré sur la puce silicium du MCU : *Figure 4-1.Simplified PIC MCU Clock Source Block Diagram*

- Sous MPLABX, adapter la configuration du CPU de façon à travailler à vitesse maximale (64MHz) en utilisant le résonateur interne au MCU (technologie RC ou MEMS) et non externe sur carte (Quartz)

  - *Production → Set Configuration Bits → Modifier le champ CONFIG1L :*
    FEXTOSC = OFF et RSTOS = HFINTOSC_64MHz
  - *Cliquer sur Generate Source Code to Output*
  - *Copier/coller le retour de la console en en-tête de votre fichier source (cf. ci-dessous).*
  - *Compiler le projet : fenêtre Projects > clic droit sur le nom du projet > Clean and Build*

```
/* copy and paste bits configuration here ! */

#include <pic18f27k40.h>

...
```

Test sur carte physique Curiosity HPC

- Ouvrir les propriétés du projet : *fenêtre Projects > clic droit sur le nom du projet > Properties*
- Sélectionner la carte Curiosity HPC : *Hardware Tool > Microchip Kits > Curiosity > Apply > OK*
- Compiler et exécuter le programme : *fenêtre Project > clic droit sur le nom du projet > Run*

Test sur simulateur MPLABX IDE (sans carte de développement)

- Ouvrir les propriétés du projet : *fenêtre Projects > clic droit sur le nom du projet > Properties*
- Sélectionner le simulateur : *Hardware Tool > Simulator > Apply > OK*
- Compiler et exécuter le programme : *fenêtre Projects > clic droit sur le nom du projet > Debug*
- Arrêter la simulation en cliquant sur le bouton carré rouge STOP (ou [Shift] + [F5])

Test sur simulateur de carte PICSimLab

- Sous MPLABX, compiler le programme : *fenêtre Projects > clic droit sur le nom du projet > Clean and Build*
- Sous MPLABX, observer après compilation le dossier où a été sauvé le fichier .hex de sortie:

```
BUILD SUCCESSFUL
Loading code from /<your_computer_path>/dist/default/production/<your_project_name>.hex
```

- Ouvrir le simulateur de carte PICSIMLab : *File > Load Hex > charger le fichier .hex généré. Ne pas hésiter à utiliser le module oscilloscope pour analyser la sortie : Modules > Oscilloscope*

Vous venez de créer, de configurer et d'exécuter en partant de rien votre premier projet sur processeur PIC18. Vous avez ajouté le fichier d'en-tête constructeur afin de pouvoir utiliser les noms des registres du processeur dans votre programme. Puis, vous avez configuré la vitesse de travail du CPU afin de maîtriser la vitesse d'exécution des instructions par le CPU. Pour conclure, vous avez réalisé 3 techniques de test disponibles pour cette trame d'enseignement

2.6. Analyse en assembleur et debug

- Compiler puis charger le programme en mode *debug* dans le MCU cible sur carte curiosity HPC ou sur simulateur MPLABX. S'aider des tutoriels dans *mcu/tp/doc/tutos*
  - *Fenêtre Projects > clic droit sur le nom du projet > Debug*
- Ouvrir et déplacer dans l'environnement graphique de l'IDE une fenêtre permettant d'observer le code binaire désassemblé (conversion binaire vers assembleur PIC18 du firmware par l'IDE) :
  - *Window → PIC/target Memory Views → Program Memory*
- Parcourir la totalité de la mémoire flash (mémoire programme) et retranscrire le code binaire du programme assembleur générés ci-dessous en respectant le *mapping* (adresses) ou modèle mémoire choisi par le *linker* (éditeur de liens) pour placer les instructions du programme en mémoire.

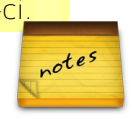
Address	Binary Opcode	Label	Disassembly Listing

- Analyser et comprendre 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 assembleur généré. Mettre un point d'arrêt sur le *nop*

```
asm("nop");
```

  - Point d'arrêt : *Double clic dans la fenêtre d'édition au numéro de ligne désiré*
  - Exécuter le programme pas à pas : *icône Step Into ou touche F7*
- En analysant la console de sortie de l'IDE (Build, Load,...) après compilation, préciser dans quel répertoire sur votre ordinateur a été rangé le *firmware* exécutable de sortie après édition des liens

Voilà, ce premier exercice est maintenant terminé. Il n'avait pour objectif que de permettre une meilleure compréhension de l'architecture minimale d'un programme sous IDE MPLABX, d'utiliser les outils de debug pour tester vos programmes dans la suite de la trame et de comprendre le fonctionnement à l'étage assembleur d'un programme C élémentaire. A partir de maintenant nous allons travailler dans l'arborescence de fichiers créée spécifiquement pour notre BSP (Board Support Packag pour carte Curiosity HPC et MCU PIC18F27K40 – *mcu/tp/disco/bsp*). La conception du BSP étant déjà réalisée vous aurez à vous concentrer sur les phases de développement, test unitaire et test d'intégration de celui-ci.

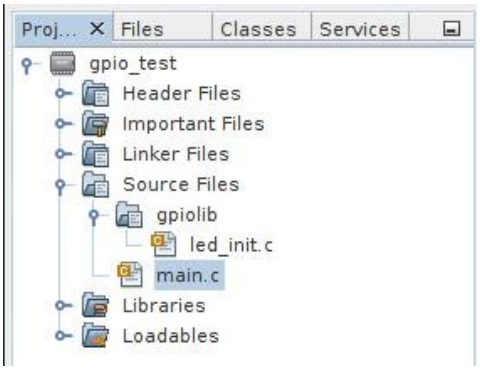


2.7. BSP et fonctions pilotes C/ASM

- Créer un projet MPLABX nommé *gpio_test* 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 des tutoriels dans *mcu/tp/doc/tutos*

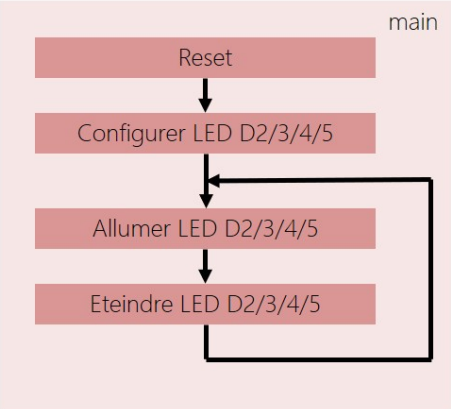
- Sur quelles broches du MCU sont physiquement connectées les LED D2, D3, D4 et D5 ? *S'aider de la documentation utilisateur (user guide) de la carte Curiosity HPC de Microchip présente dans mcu/tp/doc/datasheets*

- Créer un répertoire logique *gpiolib* dans le répertoire Sources Files du projet sous l'IDE (clic droit *New Logical Folder*). A réaliser et renommer pour chaque exercice. Ajouter le fichier *src/led_init.c* au projet et vérifier la bonne compilation de celui-ci.



- Modifier le fichier *led_init.c* afin de 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 en langage C). *Cet exercice n'a pour objectif que de vous montrer différentes manières en langage C pour réaliser un même traitement.*

- Développer une application de test implémentant le diagramme de séquence suivant :

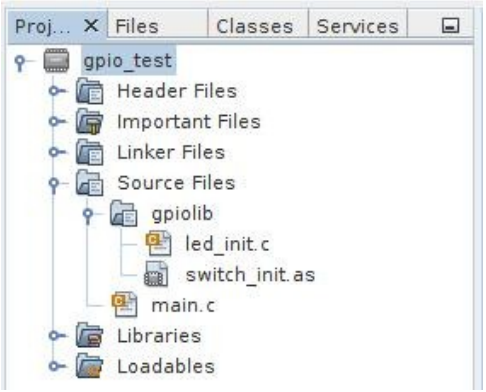


- Analyser le programme binaire généré. Observer à l'oscilloscope le signal en sortie du MCU envoyé à la LED D2. *Quelle solution (fonction ou macro) vous semble la plus efficace et pourquoi (exemple de question de recruteur en entretien d'embauche) ?*

2.8. Gestion des boutons poussoirs

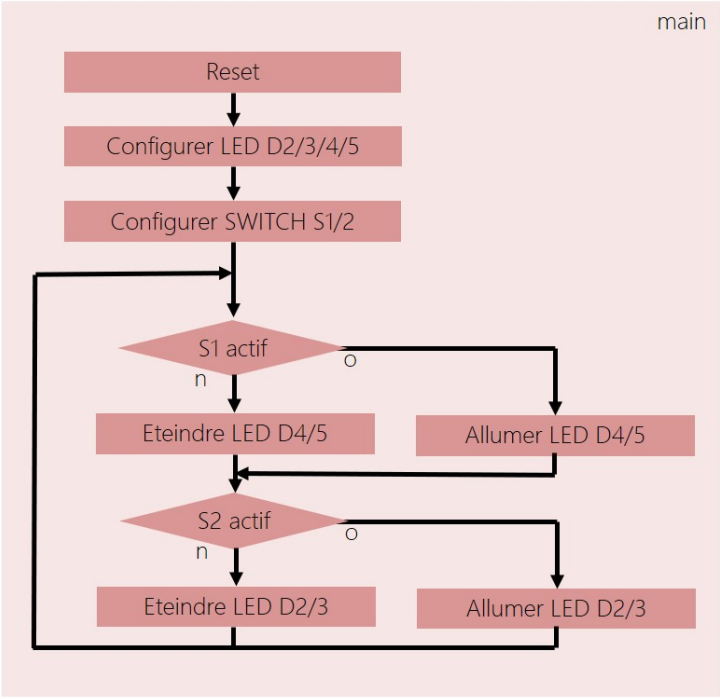
- Sur quelles broches du MCU sont physiquement connectés les boutons poussoirs S1 et S2 ? *S'aider de la documentation utilisateur (user guide) de la carte Curiosity HPC de Microchip présente dans mcu/tp/doc/datasheets*

- Ajouter le fichier assembleur *src/switch_init.as* au projet et vérifier la bonne compilation de celui-ci.



- Modifier le fichier assembleur *switch_init.as* afin de 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 en langage C). *Cet exercice n'a pour objectif que de vous montrer différentes manières en langage C pour réaliser un même traitement. Il vous faudra également configurer le registre ANSELB. Quel est son rôle ?*

- Développer une application de test implémentant le diagramme de séquence suivant :



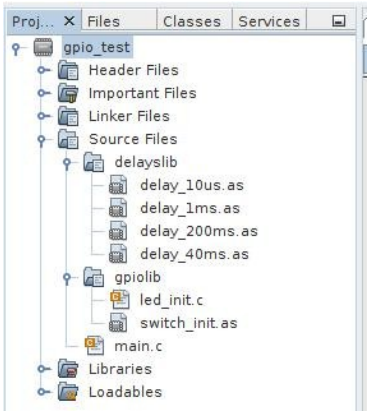
- Valider le bon fonctionnement de votre application

2.9. Délais logiciel

Dans une application, nous cherchons tant que possible à éviter les temporisations et délais logiciels. Il s'agit de boucles de décrémentation voire d'incrémentation exécutées par le CPU correspondant à des fonctions d'attentes. Nous préférons dédier le CPU à la supervision du système (application) ou aux opérations de calcul (algorithmique). Le reste du temps, le système doit être au repos (veille) pour des considérations énergétiques. Nous verrons la mise en veille du CPU dans le prochain exercice sur les interruptions et les modules périphériques Timers. Néanmoins, dans certains cas, comme les phases d'initialisation de certains périphériques au démarrage, ces fonctions de délais ou temporisations logicielles peuvent être utiles. Nous en verrons des applications par la suite.

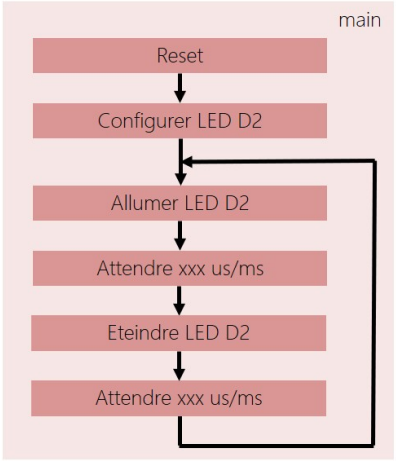
- Quelle est la fréquence de travail du CPU ? Quel fichier d'en-tête de notre BSP devrait-on modifier pour jouer sur ce paramètre (le chercher et spécifier son nom) ?

- Créer un répertoire logique *delayslib*. Ajouter les fichiers *bsp/common/*delay_*xxx.as* au projet et vérifier la bonne compilation du projet.



- Modifier le fichier *delay_10us.as* afin de réaliser une temporisation logicielle en assembleur de 10us. *S'aider de documentation du MCU PIC18F27K40 (chapitre 36 – Instruction Set Summary) et notamment de l'instruction decfsz.*

- Développer une application de test implémentant le diagramme de séquence suivant :



- Valider les durées de temporisation à l'oscilloscope puis réitérer le travail pour chaque fonction d'attente *delay_1ms*, *delay_40ms* et *delay_200ms*

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


notes