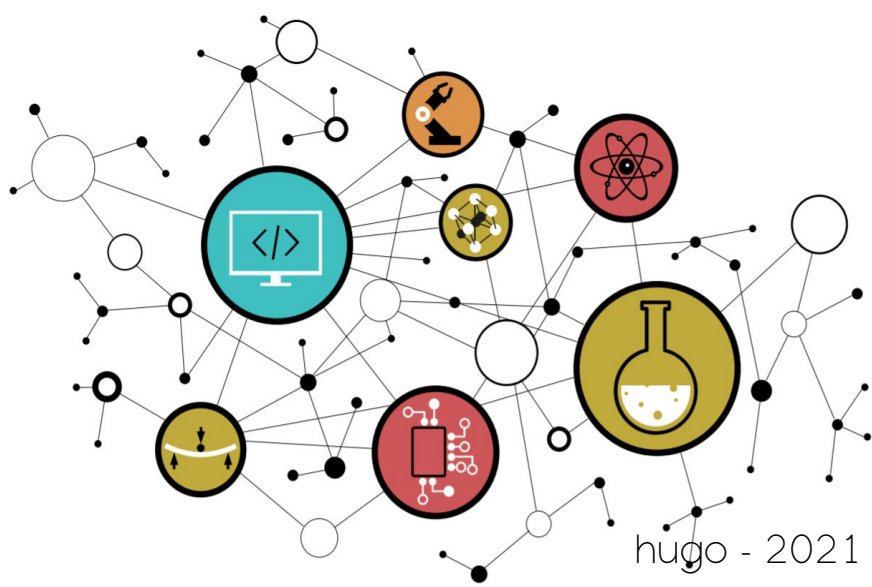


TRAVAUX PRATIQUES

SYNTHÈSE



SOMMAIRE

La trame de TP minimale que nous considérons comme être les compétences minimales à acquérir afin d'accéder à aux métiers de base du domaine suit le séquentiel suivant : chapitres 1, 2, 3, 4, 5 et 8. Le reste de la trame ne sera pas évalué et représente une extension au jeu de compétences minimales relatif au domaine en cours d'étude (* devant chapitres facultatifs voire complémentaires). Libre à vous d'aller plus loin selon votre temps disponible et votre volonté de mieux comprendre et maîtriser ce domaine !

1. PRÉLUDE

- 1.1. Présentation des Systèmes Embarqués
- 1.2. Objectifs pédagogiques
- 1.3. Quelques ressources internet

2. MODULE DE BROCHE GPIO ET ASSEMBLEUR PIC18

- 2.1. Introduction : *Module GPIO ou General Purpose Input Output*
- 2.2. Introduction : *Module GPIO sur PIC18*
- 2.3. Introduction : *Configuration des GPIO sur PIC18*
- 2.4. Introduction : *Assembleur ou langage d'assemblage PIC18*
- 2.5. My first MPLABX project from scratch
- 2.6. Analyse assembleur et debug
- 2.7. BSP et fonctions pilotes C/ASM
- 2.8. Gestion des boutons poussoirs
- 2.9. Délais logiciel en assembleur

3. MODULE DE COMPTAGE TIMER ET GESTION DES INTERRUPTIONS

- 3.1. Introduction : *Interruption matérielle*
- 3.2. Introduction : *Source et requête d'interruption IRQ*
- 3.3. Introduction : *Logique et démasquage d'interruption*
- 3.4. Introduction : *Vecteur d'interruption*
- 3.5. Introduction : *Fonction d'interruption ISR*
- 3.6. Introduction : *Gestion des interruptions sur PIC18*
- 3.7. Introduction : *Gestion du RESET sur PIC18*
- 3.8. Introduction : *Module périphérique de comptage Timer*
- 3.9. Introduction : *Module périphérique Timer0 sur PIC18*
- 3.10. Introduction : *Configuration du Timer0 sur PIC18*
- 3.11. Configuration du Timer0 et interruption
- 3.12. Analyse assembleur et commutation de contexte
- 3.13. Mise en veille du CPU

4. MODULE DE COMMUNICATION UART ET LIAISON SÉRIE

- 4.1. Introduction : *Protocole de communication d'une liaison série asynchrone*
- 4.2. Introduction : *Module périphérique UART*
- 4.3. Introduction : *Norme RS232*
- 4.4. Introduction : *Module périphérique UART sur PIC18*
- 4.5. Introduction : *Configuration du module UART sur PIC18*
- 4.6. Module périphérique UART1 en transmission
- 4.7. Terminal de communication série sur ordinateur
- 4.8. Transmission de chaînes de caractères
- 4.9. Module périphérique UART1 en réception
- 4.10. Buffer circulaire de réception
- 4.11. Contrôle de flux logiciel
- 4.12. Réception de chaînes de caractères
- 4.13. Module périphérique UART2
- 4.14. Bridge de communication UART1 vers UART2

5. MODULE AUDIO BLUETOOTH EXTERNE

- 5.1. Introduction : *Bluetooth*
- 5.2. Configuration du module Audio Bluetooth RN52

* 6. MODULE DE COMMUNICATION I2C ET AFFICHEUR LCD

* 7. MODULE DE CONVERSION ADC

8. CONCEPTION D'UNE APPLICATION ET ORDONNANCEMENT

- 8.1. Introduction : *Application, ordonnancement et philosophie Unix*
- 8.2. Conception et ordonnancement de l'application
- 8.3. Cahier des charges du POC (Proof Of Concept)
- 8.4. Développement du POC (Proof Of Concept)
- 8.5. Evolutions et améliorations

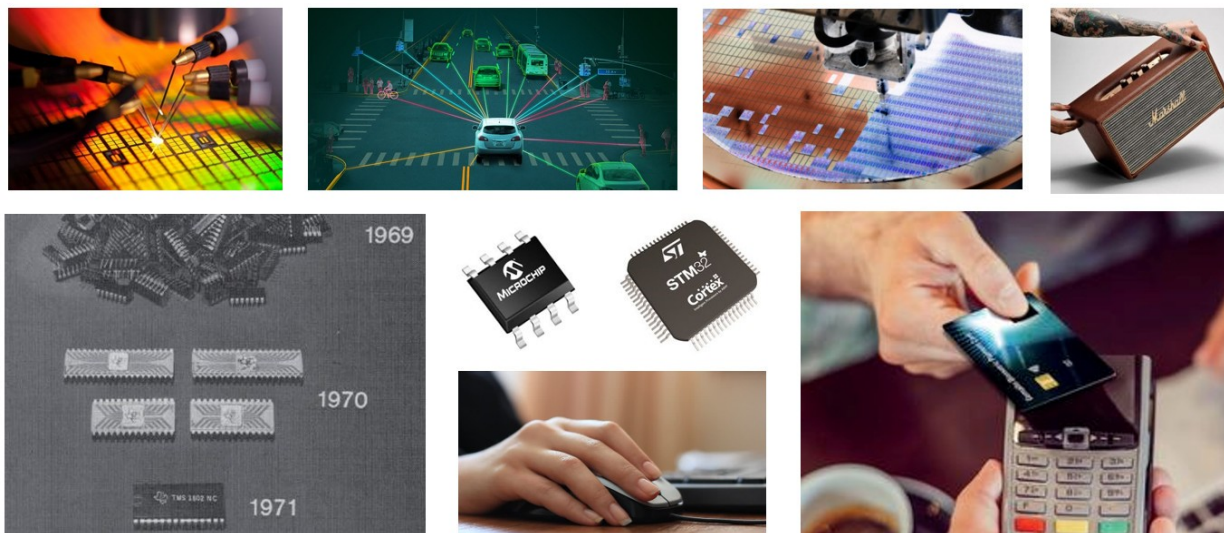
* 9. DOCUMENTATION TECHNIQUE ET LIVRABLES

- 9.1. Introduction : *Livrables et documentation technique*
- 9.2. Doxygen et documentation d'une bibliothèque
- 9.3. Documentation technique

PRÉLUDE

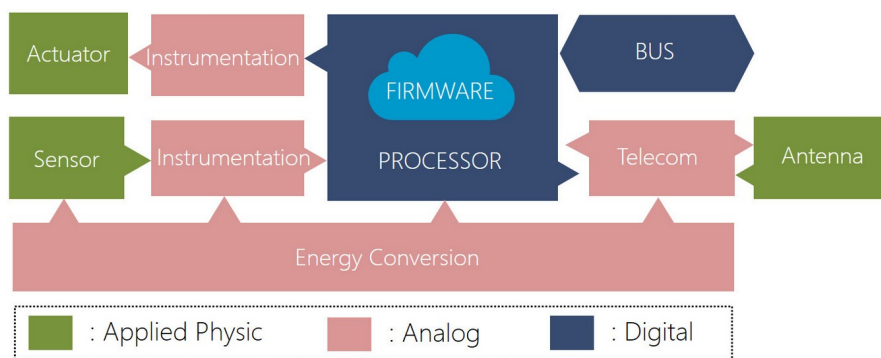
1. PRÉLUDE

1.1. Présentation des Systèmes Embarqués

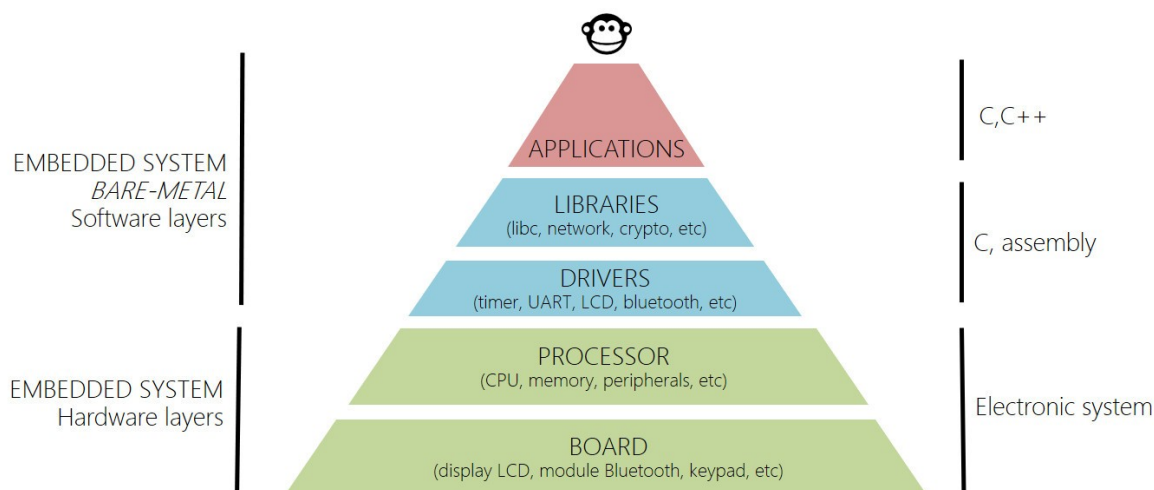


Un système embarqué peut-être vu comme le système (ensemble physique, électronique et informatique) embarqué dans le produit (souris, clavier, montre, voiture, fusée spatiale, carte bancaire, etc la liste est très longue). A l'image des produits, la conception d'un système embarqué peut offrir de multiples facettes. Il peut être communicant (téléphone mobile, Ebooks, montre connectée, IOT, etc) en utilisant divers protocoles de communication sans fil ou filaire (WIFI, Bluetooth, 2/3/4G, LORA, Ethernet, USB, etc), autonome sur réserve par stockage d'énergie électrique (tablette, souris, voiture, etc), portable dans la main (carte bancaire, clé de voiture, etc), faiblement encombrant (lecteur MP3, carte monétique, etc) comme très encombrant (fusée spatiale, avion, voiture, etc). Un système embarqué peut être vu comme l'ensemble des sous composantes ou sous systèmes suivants. Ce "tout" forme un système embarqué complet de stockage, d'échange et de traitement de l'information :

- *Système Physique (hardware/matériel)* : Interfaces avec l'environnement extérieur au produit (capteurs, actionneurs et antennes). Ponts entre l'environnement physique et le périmètre d'action du produit répondant à un besoin (capteurs de température, pression, humidité, luminosité, bouton poussoir, buzzer, LED, antennes radio, WIFI, Bluetooth, etc)
- *Système Électronique Analogique (hardware/matériel)* : Conditionnement et mise en forme des signaux de mesure et de contrôle (chaîne d'instrumentation) ainsi que des chaînes de communication (wire/filaire et wireless/sans fil).
- *Système Électronique de Puissance (hardware/matériel)* : Mise en forme et dimensionnement des entrées électriques d'énergie (redresseur, hacheurs flyback, forward, etc). Stockage de l'énergie électrique sur système embarqué autonome (batterie).
- *Système Électronique Numérique (hardware/matériel)* : Stockage numérique des données (mémoire donnée) et des programmes (mémoire programme). Traitement des données par exécution du programme puis mise en forme de l'information (processeur). Interfaces numériques de communication et d'échange avec l'extérieur (fonctions périphériques)
- *Système Informatique (software/logiciel et firmware/micrologiciel)* : Couche système (système d'exploitation du matériel), couche bibliothèque (utilitaires et fonctionnalités pour les applications) et couche applicative (missions de supervision du système appliquées à des besoins spécifiques voire problématiques de calcul)

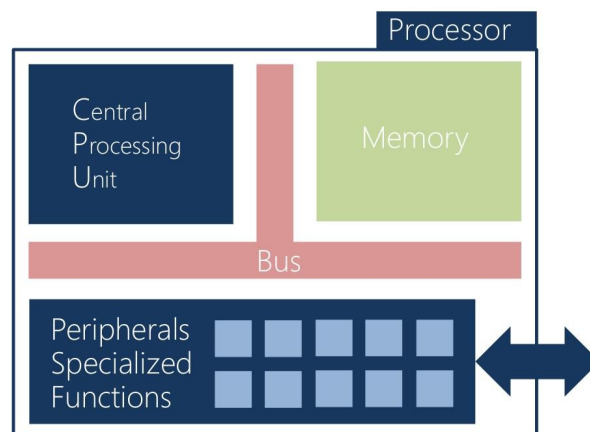


Au cœur de la très grande majorité des produits électroniques actuels se trouvent enfouis un (souris, montre, carte monétique, etc) voire plusieurs processeurs (voiture, avion, etc). Un système embarqué, peut d'ailleurs interagir avec plusieurs sous systèmes embarqués (un ordinateur et une souris par exemple). Différentes familles de processeurs cohabitent sur le marché (MCU, GPP/MPU, AP/MPU, DSP, MPPA, FPGA, GPU, etc), chacune répondant à des besoins et exigences (application/supervision ou calcul/algorithmique, consommation, coût, encombrement, performance, etc). Durant cet enseignement, nous nous intéresserons à l'architecture processeur la plus répandue en volume sur le marché, celle des MCU (Micro Controller Unit ou microcontrôleur).



Un système numérique de traitement de l'information peut souvent être représenté en couches matérielles comme logicielles. Ce modèle représente grossièrement les dépendances des différentes fonctionnalités matérielles et logicielles entre elles (application, bibliothèques, pilotes, périphériques internes et externes, etc) et donc le chemin de l'information dans le système (montante/entrante ou descendante/sortante). L'objectif premier restant de développer une application logicielle, répondant à un besoin et supervisant le système matériel dédié. Contrairement à un ordinateur cherchant la généricité (système d'exploitation multi-applicatifs, interfaces utilisateur génériques et standards, etc), un système embarqué est développé pour répondre spécifiquement et de façon optimale à un besoin. Une souris n'est pas une montre !

Nous allons tout au long de cet enseignement nous efforcer de comprendre puis maîtriser au mieux les différentes couches de ce modèle. Les solutions matérielles de prototypage ayant été spécifiées (MCU 8bits PIC18F27K40 et carte curiosity HPC de Microchip), une analyse attentive des documentations techniques (datasheet) sera nécessaire afin de développer les couches pilotes (drivers) puis applicative répondant à nos besoins. L'application terminale de l'enseignement visant à concevoir et développer un application audio Bluetooth. Cependant, une grande partie de notre travail consistera à développer un BSP spécifique (Board Support Package, bibliothèque de fonctions pilotes dédiées à notre carte et MCU).

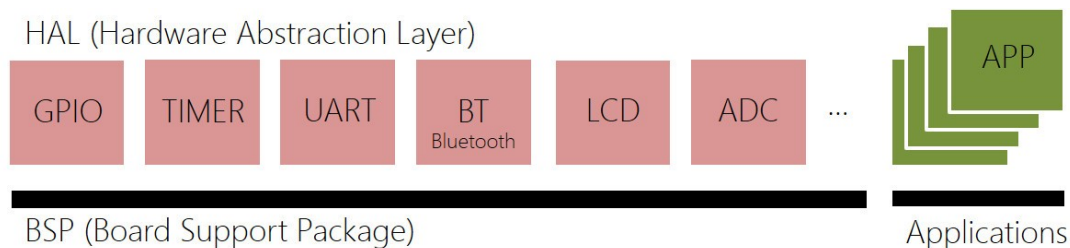


Un MCU (Micro Controller Unit) ou microcontrôleur est une famille de processeur numérique présente sur le marché de l'électronique. Le premier MCU produit et commercialisé date de 1971 par Texas Instruments. Cette famille de processeur possède notamment comme particularité d'intégrer sur une même puce de silicium (on chip) l'ensemble des éléments (CPU, bus, mémoires et périphériques) faisant d'elle un processeur complet sur puce, contrairement aux ordinateurs où les composants silicium électroniques (GPP/MPU, mémoires vive DRAM et de masse HDD/SSD, chipset, ec) sont distincts et portés sur une carte mère (circuit imprimé).

Les MCU sont dédiés et spécialisés aux applications exigeant un faible encombrement spatial (processeur complet intégré sur puce), une faible consommation énergétique (souvent mono CPU à qq10-100MHz) et un faible coût financier (entre de qq0,1€ à qq1€ l'unité). Il s'agit des processeurs les plus fabriqués en volumes dans le monde chaque année avec près de 25 milliards d'unités en 2020 (source IC Insights). Il s'agit de composants numériques de stockage, d'échange et de traitement de l'information. Rappelons les rôles de chaque entité d'un processeur architecturé autour d'un CPU :

- *CPU (Central Processing Unit) – Traiter l'information* : Composant chargé de récupérer séquentiellement par copie (fetch) une à une les instructions du programme binaire exécutable (firmware) présent en mémoire. Une fois récupérée, chaque instruction est décodée (decode), exécutée (execute) puis le résultat sauvé dans la machine (writeback). Hormis lorsqu'il est forcé en veille, un CPU réalisera sans arrêt les étapes séquentielles suivantes Fetch/Decode/Execute/WriteBack. Le pipeline matériel d'un CPU correspond à sa capacité à réaliser ces traitements en parallèle. Il existe plusieurs modèles architecturaux de fonctionnement de CPU (RISC/CISC, Harvard/VonNeumann/Hybrid, Scalar/Superscalar/VLIW, etc). Tous seront découverts en formation dans la suite du cursus.
- *Mémoire – Stocker l'information* : La mémoire est chargée de stocker l'information. L'information peut être de deux natures différentes dans la machine, instructions du programme (code) ou données (data)
 - *Mémoire programme* : mémoire non-volatile (persistante), elle sera souvent nommée historiquement mémoire Flash sur MCU. Elle sera le plus souvent accessible en lecture seule par le CPU à l'usage (ReadOnly). Les technologies les plus répandues sont les mémoires flash NOR et flash NAND.
 - *Mémoire donnée* : mémoire volatile le plus souvent de technologie SRAM sur MCU. Elle sera accessible en Lecture et écriture par le CPU à l'usage (Read/Write)
- *Périphériques – Échanger/Convertir/Traiter l'information* : 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 ou échanger l'information* (par protocole de communication), *convertir l'information* (du domaine analogique vers numérique) ou *traiter l'information* (fonction de traitement matérielle spécialisée)

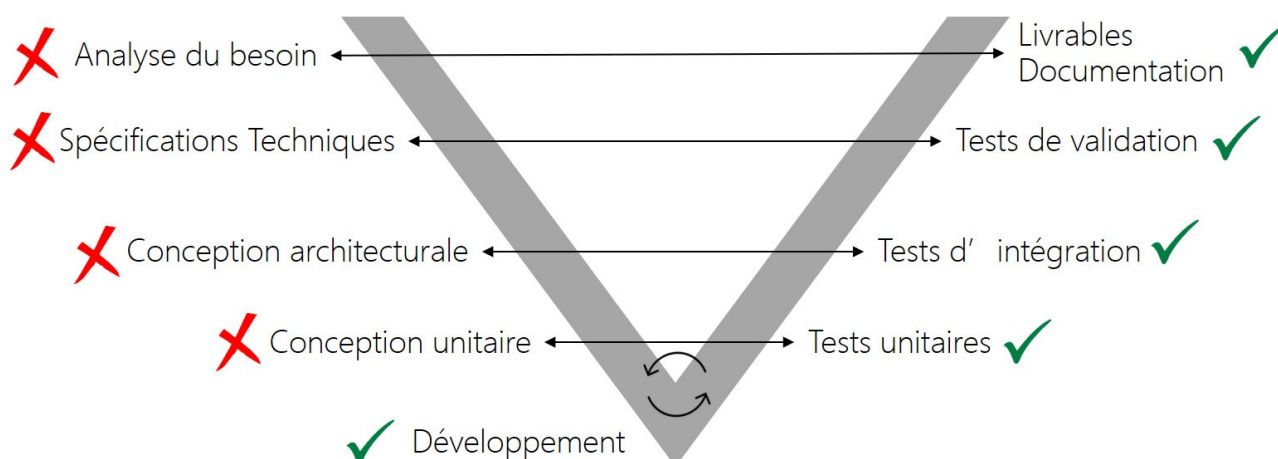
1.2. Objectifs pédagogiques



Les objectifs de cet enseignement sont multiples. Pour une grande partie des réalisations, nous aurons à développer un *BSP (Board Support Package)* ou *HAL (Hardware Abstraction Layer) 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 développer 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 (GPIO, Timer, UART, I2C, etc) voire externes (LED, switch, module Audio Bluetooth, potentiomètre, afficheur alphanumérique LCD, etc).

Le BSP sera dédié à notre processeur de travail (MCU 8bits PIC18F27K40 Microchip) et notre carte (Curiosity HPC Microchip). Une migration de technologie processeur et/ou carte nécessiterait des ajustements et de nouveaux développements. Une fois le BSP développé, testé, validé, documenté et la bibliothèque statique générée, nous développerons une application audio Bluetooth *bare-metal*. Dans le domaine de l'embarqué, *Baremetal* signifie nu sans système d'exploitation (OS ou Operating System ni RTOS ou Real Time OS). L'application implémentera un *scheduler offline*, ce point sera vu plus en détail dans la suite de la trame. En fin d'enseignement, nous ne pourrions alors qu'imaginer l'infini potentiel créatif s'ouvrant devant nos yeux !

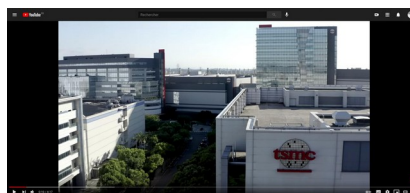
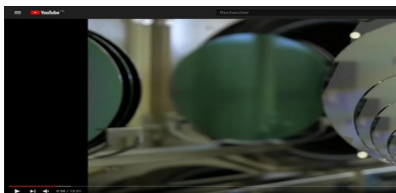
La chronologie de la trame de TP est construite autour d'un workflow typique rencontré en industrie autour de ce type de développements. Avant de développer tout applicatif, nous aurons à valider unitairement toutes les fonctionnalités et interfaces de l'application (matérielles et logicielles). Nous pourrions ensuite nous focaliser sur les phases d'intégration successives. Même si des méthodologies agiles de gestion de projet seront utilisées lors de projets en équipe en 2^{ème} et 3^{ème} année en majeure, celui découvert ici présente un cycle en V (V et agilité peuvent cohabiter). De même, il ne serait pas raisonnable ni pertinent de demander à un élève ingénieur en formation 1^{ère} année de réaliser la conception d'un projet de cette taille sans avoir le recul suffisant sur le domaine. Les spécifications techniques et les conceptions ont donc été réalisées (matériel et logiciel, architecturales et unitaires), vous aurez donc à vous focaliser sur les phases de développement, de test unitaire, de test d'intégration et de validation. La conception sera découverte en majeure durant la 2^{ème} année et la 3^{ème} année. Ce point nécessite une très bonne assise des compétences de 1^{ère} année.



1.3. Quelques ressources internet

Voici quelques ressources en ligne pouvant vous aider à une meilleure contextualisation du domaine, des acteurs, compréhension des processus de fabrication et des outils que nous utiliserons durant cet enseignement.

How microchips are made – Infineon (13mn) and TSMC foundry world leader (4mn)



<https://www.youtube.com/watch?v=bor0qLifjz4>

<https://www.youtube.com/watch?v=Hb1WDxSoSec>

How PCB and MotherBoards are made – PCBWay (13mn) and GigaByte (2mn)



https://www.youtube.com/watch?v=_GVk_hEMjzs&t=637s

<https://www.youtube.com/watch?v=bR-DOeAm-PQ>

Microchip Company – Microchip (5mn)

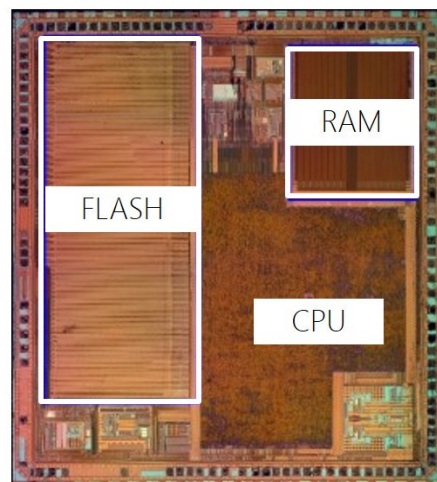
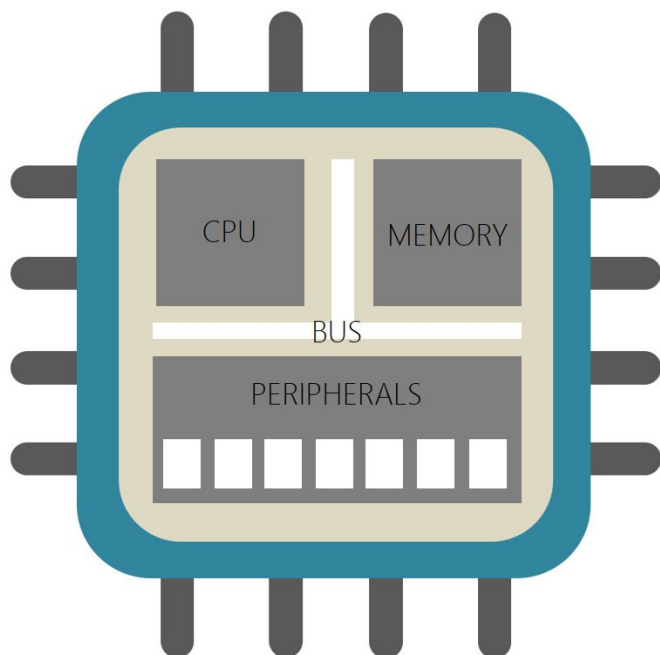


<https://www.youtube.com/watch?v=-N20QMlgh4Q>

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 prendront 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.

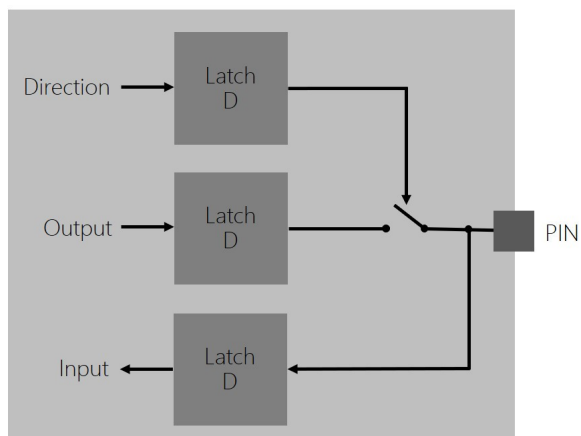
Un périphérique est un service matériel proposé par la machine. Les fonctions matérielles spécialisées périphériques à l'ensemble CPU/Mémoire, plus communément nommées "périphériques", jouent généralement l'un des trois rôles suivants dans le système :

- *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

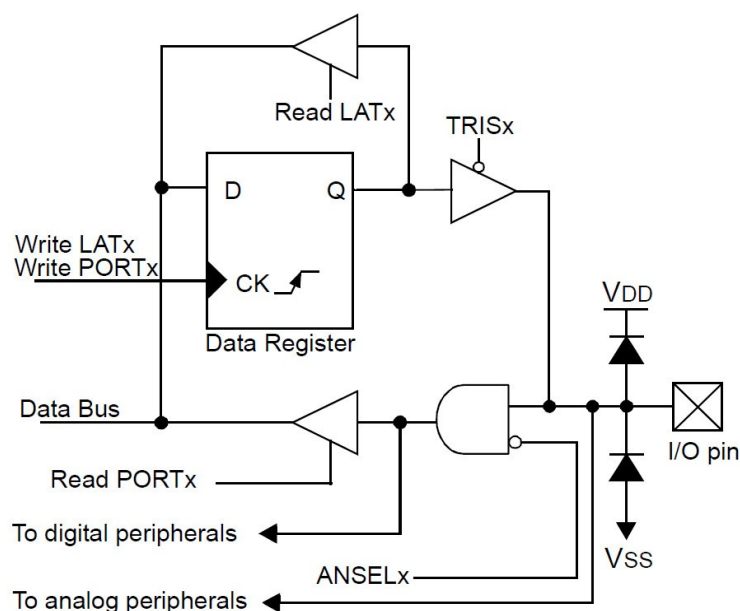


Une broche (ou pin) est une interface physique présente et visible à l'œil nu sur le boîtier d'un MCU ou microcontrôleur. Il est à noter qu'une même puce de silicium (die), un MCU par exemple, peut être encapsulée dans différentes technologies de boîtier (DIP, SOIC, BGA, QFN, etc). Chaque technologie 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).

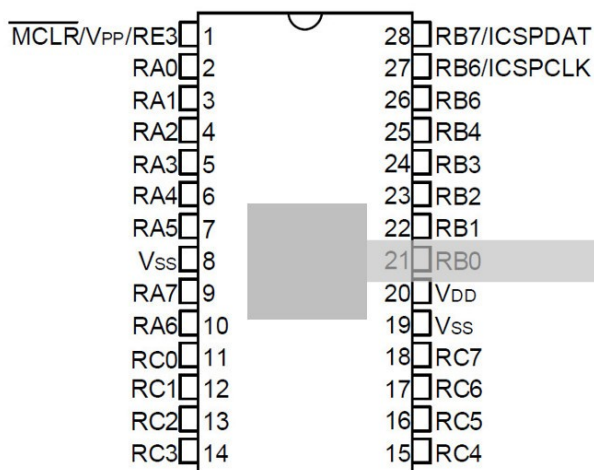


Un module GPIO associé à une broche est une fonction matérielle interne à un MCU. Il est constitué sur silicium d'un circuit logique. La figure ci-dessus ne présente par exemple que le schéma symbolique pour la gestion d'une seule broche. Chacun des états logiques de configuration et d'utilisation d'une 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.

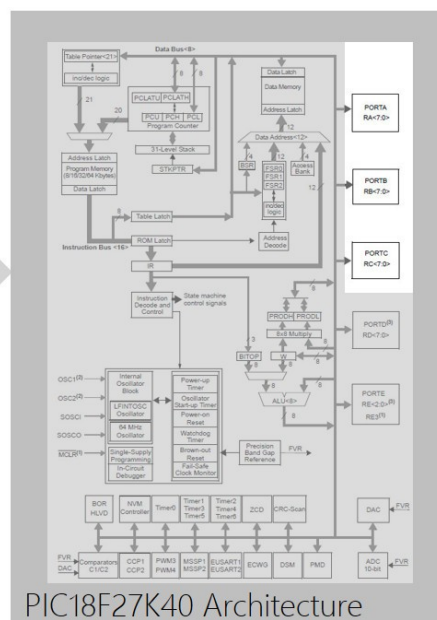
2.2. Module GPIO sur MCU PIC18



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.

<pre>; 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</pre>	<pre>; 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></pre>
---	--

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 aux 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 :

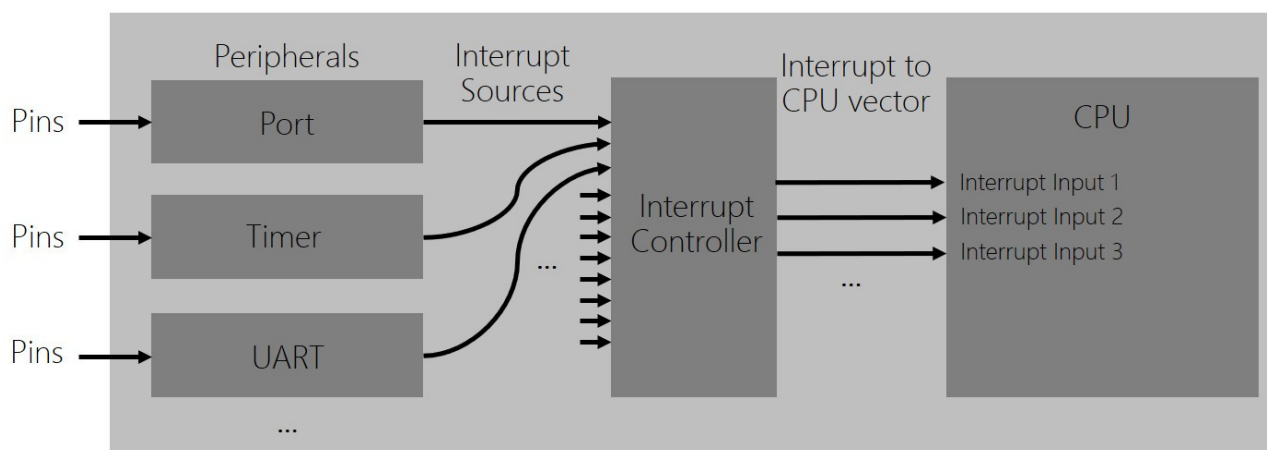
label:	instruction	opérandes
--------	-------------	-----------

- **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.

MODULE DE COMPTAGE TIMER ET GESTION DES INTERRUPTIONS

3. MODULE DE COMPTAGE TIMER ET INTERRUPTIONS

3.1. Interruption matérielle



Le concept d'interruption matérielle est essentiel sur tout processeur conçu autour d'un CPU. Il s'agit de la capacité des fonctions périphériques à interrompre le CPU en cours de traitement pour lui affecter une nouvelle mission potentielle. Une interruption matérielle sera toujours rattachée à une fonction matérielle périphérique et correspond à l'occurrence d'un événement physique dans le système. Elle est toujours envoyée par un périphérique vers le CPU. Une fois configuré pour une mission dédiée, un périphérique est un composant indépendant et autonome dans le processeur lui-même. Chaque périphérique possède donc un rôle et une tâche spécifique dans le système (communiquer, convertir ou traiter).

Un module périphérique Timer est par exemple conçu autour d'un compteur ou décompteur numérique. Sa mission est de compter à la place du CPU. Les modules UART, SPI, I2C ou USB sont par exemple des périphériques de communication permettant d'échanger de l'information avec l'extérieur du processeur. Communication de machine à machine. Lorsqu'un événement physique relatif à la tâche d'un périphérique se produit (fin de comptage pour un Timer, réception ou fin d'émission d'une donnée pour un UART, etc), celui-ci va tenter d'interrompre le CPU pour le prévenir de cet événement. Une interruption est alors envoyée par le périphérique vers le CPU. Une interruption est un simple signal physique électrique booléen tout ou rien. A la réception de l'interruption, le CPU sera soit en cours d'exécution d'une instruction (*fetch/decode/execute/writeback*), soit en veille (*inactif*). Pour que le CPU soit sensible et donc voit l'interruption, le développeur aura à configurer le contrôleur d'interruption du processeur. Nous allons découvrir tout ceci dans les pages qui suivent.

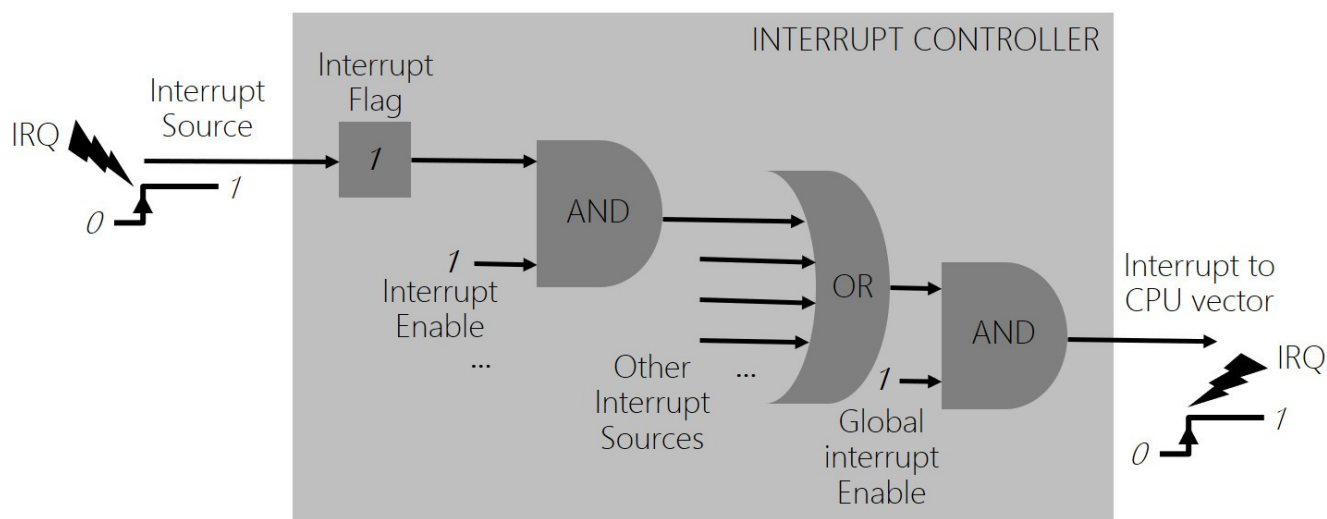
Une interruption est donc un signal physique partant d'un périphérique et arrivant (sous conditions) jusqu'au CPU. Les entrées d'interruption d'un CPU sont le plus souvent classées par niveau de priorité. Il est à noter qu'une "Belle" application, ne travaillera et ne réalisera des actions qu'au moment opportun, lorsqu'un traitement est strictement nécessaire. Le reste du temps, le système de supervision devra rester au repos (veille). La mise en veille CPU correspond à sa capacité à ne pas exécuter d'instruction (CPU inactif). Alors, seuls les périphériques restent à l'écoute des événements extérieurs (voire intérieurs) au système. Ils servent donc d'interfaces entre le système embarqué et son environnement physique extérieur.

3.2. Source et requête d'interruption IRQ



Dans la majorité des cas, il existe au moins autant de sources d'interruption que de périphériques dans le processeur. Une source d'interruption est un signal physique unidirectionnel (conducteur sur le die) allant d'un périphérique au contrôleur d'interruption. Une interruption (par abus de langage) ou requête d'interruption ou IRQ (Interrupt Request) correspond au passage d'un état logique inactif à actif sur une source d'interruption. Un périphérique envoie alors une requête au CPU et demande à interrompre son traitement en cours afin de lui affecter une nouvelle mission. Faut-il encore que le CPU y soit sensible !

3.3. Logique de démasquage d'interruption



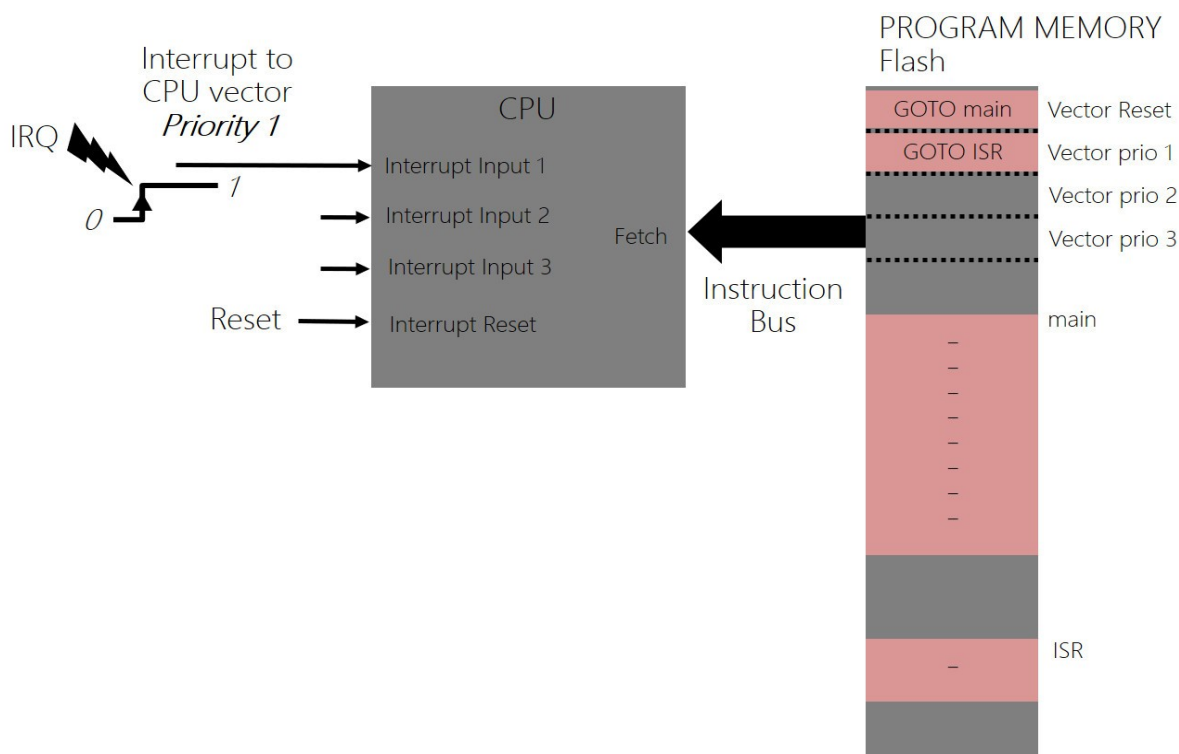
Il est à noter que par défaut à la mise sous tension, généralement le CPU n'est sensible à aucune source d'interruption, hormis celle du reset (bouton poussoir externe voire reset logiciel). Même si des périphériques devaient envoyer des IRQ (Interrupt Request), ils ne pourraient interrompre l'exécution du programme en cours. En effet, le développeur en charge du développement devra démasquer les sources d'interruption une à une en fonction des besoins spécifiques de l'application. Il s'agit d'une logique combinatoire de démasquage à réaliser afin de configurer le contrôleur d'interruption (cf. Interrupt Controller ci-dessus).

Comme pour tous les périphériques, cette configuration se fera par écriture dans des registres. Il y aura en général à minima la validation de la source d'interruption et la validation globale des interruptions pour l'ensemble du processeur. De même, les requêtes d'interruption ou IRQ seront mémorisées (cf. interrupt flag ci-dessus – simple bascule D) et auront à être acquittées explicitement par mise à zéro dans l'application par le développeur. Ces acquittements se feront dans les fonctions d'interruption (ou ISR) et seront présentés par la suite.

Si une IRQ parvient à traverser le contrôleur d'interruption, elle forcera le CPU à arrêter les traitements en cours et à exécuter un code spécifique (vecteur d'interruption). La commutation est matériellement câblée dans le CPU. Il est souvent associé un niveau de priorité (voire de sous priorités) à un vecteur d'interruption. Ceci permet à l'ingénieur de rendre plus ou moins prioritaires des périphériques (Timer, UART, USB, Ethernet, etc) et donc les tâches à réaliser par l'application. Le problème se produira lorsque plusieurs périphériques enverront des requêtes simultanément.

3.4. Vecteur d'interruption

Un vecteur d'interruption est une "petite" zone en mémoire programme. Un vecteur d'interruption est donc chargé de mémoriser "quelques" instructions binaires. Ces instructions permettent une redirection vers la fonction d'interruption ou ISR (Interrupt Service/Software Routine). D'où le nom de vecteur. Les ISR sont quant à elles des fonctions logicielles accessibles depuis l'application et présentent dans le programme en cours de développement. Il est potentiellement possible d'avoir autant d'ISR que de vecteurs d'interruption dans une application. Rappelons que généralement il est associé un niveau de priorité à un vecteur d'interruption et donc aux ISR liées. Ces priorités sont également configurables par registres.



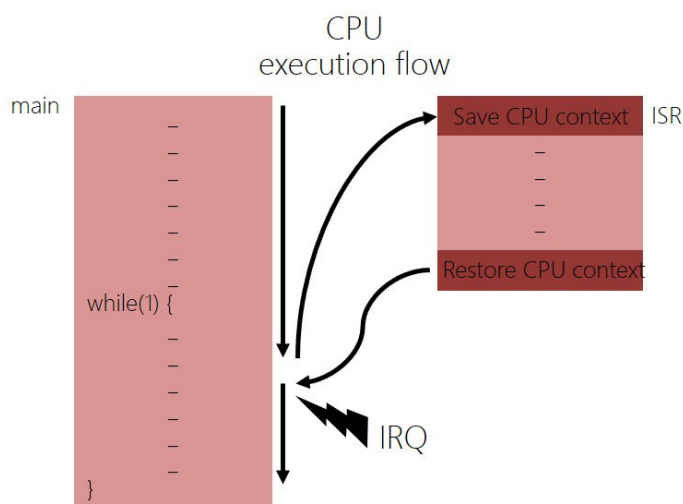
Attention, le schéma ci-dessus représente deux concepts radicalement différents (cf. version numérique du document) :

- **Architecture matérielle (hardware)** en gris pour les fonctions électroniques matérielles (CPU et mémoire programme seulement) et les flèches noires pour les bus et conducteurs physiques
- **Micrologiciel (firmware)** en rose correspondant au code binaire utile généré suite au processus de compilation et d'édition des liens sur ordinateur

Le firmware est donc mémorisé en mémoire programme non-volatile, souvent nommée mémoire flash sur MCU. Il s'agit de l'application logicielle embarquée dans le système matériel. Il s'agit de la mission du système embarqué répondant à un besoin. Sans système d'exploitation (scheduler), si aucun périphérique ne cherche à interrompre le CPU, alors celui-ci n'exécutera que du code de fonctions appelées depuis la fonction main. Dans l'exemple ci-dessus, dès que le CPU capte l'IRQ sur son entrée d'interruption de priorité 1, il commence à aller chercher puis exécuter le code présent dans le vecteur d'interruption de priorité 1 (simple instruction GOTO ISR). Puis par redirection le CPU ira exécuter le code de la fonction ISR associée. Dès que l'ISR est terminée, il reprend l'exécution de l'application exactement à l'instruction à laquelle il avait été interrompu par l'IRQ. Nous nommons ce phénomène commutation de contexte (sauvegarde et restauration). Il sera illustré sur PIC18 par la suite dans cet enseignement. De même, les vecteurs d'interruption sont généralement situés aux adresses les plus basses de la mémoire programme et sont sur certaines architectures translatables à d'autres adresses mémoire. L'ensemble des vecteurs d'interruption est nommée table des vecteurs d'interruption.

3.5. Fonction d'interruption ISR

Une ISR (Interrupt Service/Software Routine) ou fonction d'interruption est une fonction logicielle telle que vous avez pu en rencontrer en langage C dans toute application. À ceci près, qu'elle ne doit jamais être appelée de façon explicite depuis l'application. Il s'agit de programmation événementielle. Les ISR se réveilleront de façon asynchrone (non prédictible lorsqu'il s'agit de périphériques de communication) sur événement matériel en suivant la logique précédemment présentée (IRQ, démasquage puis vecteur d'interruption). Une ISR possédant le plus haut niveau de privilège d'exécution sur un processeur à CPU (MCU, AP, GPP, DSP, MPPA, etc), il faut toujours penser à passer le moins de temps possible à l'intérieur en déportant les traitements longs dans les fonctions appelées depuis le main (code applicatif). Sans système d'exploitation, utiliser alors des variables globales pour les échanges.



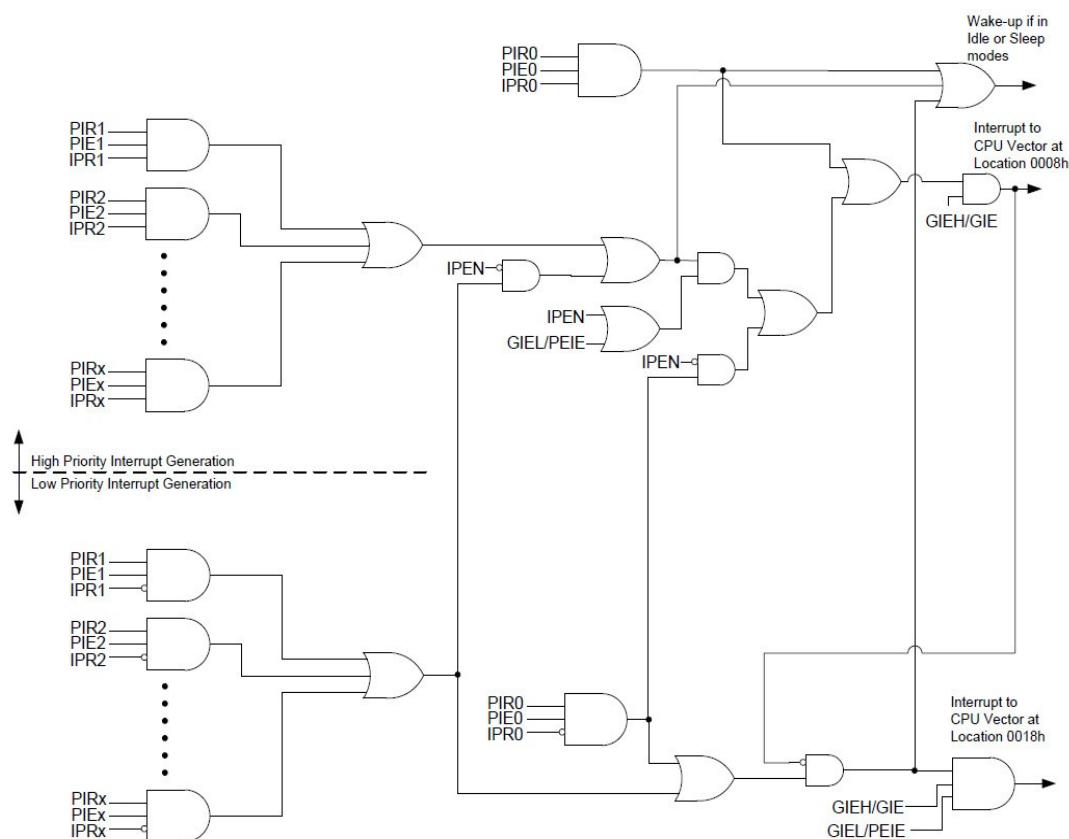
En respectant le code couleur précédemment utilisé (cf. version numérique pdf), les concepts maintenant présentés sont représentatifs du logiciel développé par le développeur et du firmware embarqué. Sans système d'exploitation, les développements sont alors nommés *Baremetal*, soit nu directement sur le processeur sans système logiciel exploitant la machine (scheduler pour le CPU, gestionnaire mémoire par MMU/MPU pour la mémoire, drivers ou pilotes pour les périphériques, etc). L'application doit alors impérativement implémenter un `while(1)`. Nous verrons en travaux pratiques comment développer une application en implémentant un *scheduler offline*.

Les ISR étant spécifiques à une architecture, leur déclaration sera différente d'un processeur à un autre et donc d'une chaîne de compilation à une autre. Le plus souvent, un qualificateur de fonction lui est associé. Le compilateur a besoin de ce qualificateur afin d'implémenter les sauvegardes et restaurations de contexte en en-tête et pied de fonction. L'exemple ci-dessous illustre une déclaration d'ISR de priorité haute sur PIC18 en langage C sur toolchain XC8. Pour information, il y a seulement deux niveaux de priorité (haute et basse) et donc deux vecteurs d'interruption sur PIC18 (en plus de celui du reset).

<pre>void main (void) { - while(1) { - - } }</pre>	<pre>void __interrupt(high_priority) ISR (void) { - - - }</pre>
--	---

Vous constaterez que le code C ainsi que le schéma ci-dessus utilisent deux illustrations différentes pour présenter un même concept (texte vs graphique). L'une est une implémentation technologique en langage C et l'autre une représentation conceptuelle sous forme de dessin. La majorité des exercices demandés dans la trame de travaux pratiques seront représentés graphiquement de façon générique. Il sera à votre charge de réaliser les implémentations technologiques C ou assembleur sur PIC18 sous environnement de développement MPLABX et chaîne de compilation XC8.

3.6. Gestion des interruptions sur PIC18



Les MCU 8bits PIC18 de Microchip ont tous en commun le même CPU, les mêmes bus, et donc le même jeu d'instructions assembleur (ISA) ainsi que la même chaîne de compilation (XC8). Il existe néanmoins un très large choix de PIC18 différents au catalogue de Microchip. Ils sont tous différenciés par leurs ressources mémoire (Flash et SRAM) ainsi que par leur jeu de périphériques (UART, SPI, I2C, USB, CAN, etc). Un PIC18 intègre en général un large jeu de périphériques et chaque périphérique possède une voire plusieurs sources d'interruption. Il existe donc un certain nombre de registres de configuration pour les interruptions. Nous pouvons classer tous ces registres 8bits de configuration des interruptions sur PIC18 en 4 sous familles :

- **INTCON** : Configuration globale pour l'ensemble du système (bits GIEH, GIEL, IPEN, etc)
- **PIEx (x=0 à 7)** : Configuration des bits de validation ou démasquage (interrupt enable)
- **PIRx (x=0 à 7)** : Lecture et acquittement des bits d'interruption IRQ (interrupt flag)
- **IPRx (x=0 à 7)** : Configuration des bits de priorité (ISR priorité basse ou haute)

De même, pour chaque périphérique, 3 bits seront alors à configurer (xxx dépend du périphérique à configurer). Ces bits correspondent à des champs de bits dans les registres précédemment cités.

- **xxxIE** : Interrupt Enable (validation de l'interruption afin de rendre le CPU sensible à l'IRQ)
- **xxxIF** : Interrupt Flag (mémorisation de l'IRQ pour acquittement dans l'ISR)
- **xxxIP** : Interrupt Priority (configuration du vecteur d'interruption de priorité basse ou haute)

L'exemple suivant présente une configuration d'interruption en assembleur pour le Timer0 :

<code>; clear flag, enable and set low priority interrupt</code>	<code>; global interrupt enable for all MCU system</code>
<code>BCF PIR0, TMR0IF</code>	<code>BSF INTCON, IPEN</code>
<code>BSF PIE0, TMR0IE</code>	<code>BSF INTCON, GIEL</code>
<code>BCF IPR0, TMR0IP</code>	<code>BSF INTCON, GIEH</code>

Program memory mapping of PIC18F27K40

Observons ci-dessous un extrait de datasheet présentant l'organisation de la mémoire programme d'un PIC18F27K40, soit le MCU utilisé en TP. Nous pouvons également constater les emplacements et tailles des 3 vecteurs d'interruption des MCU PIC18 :

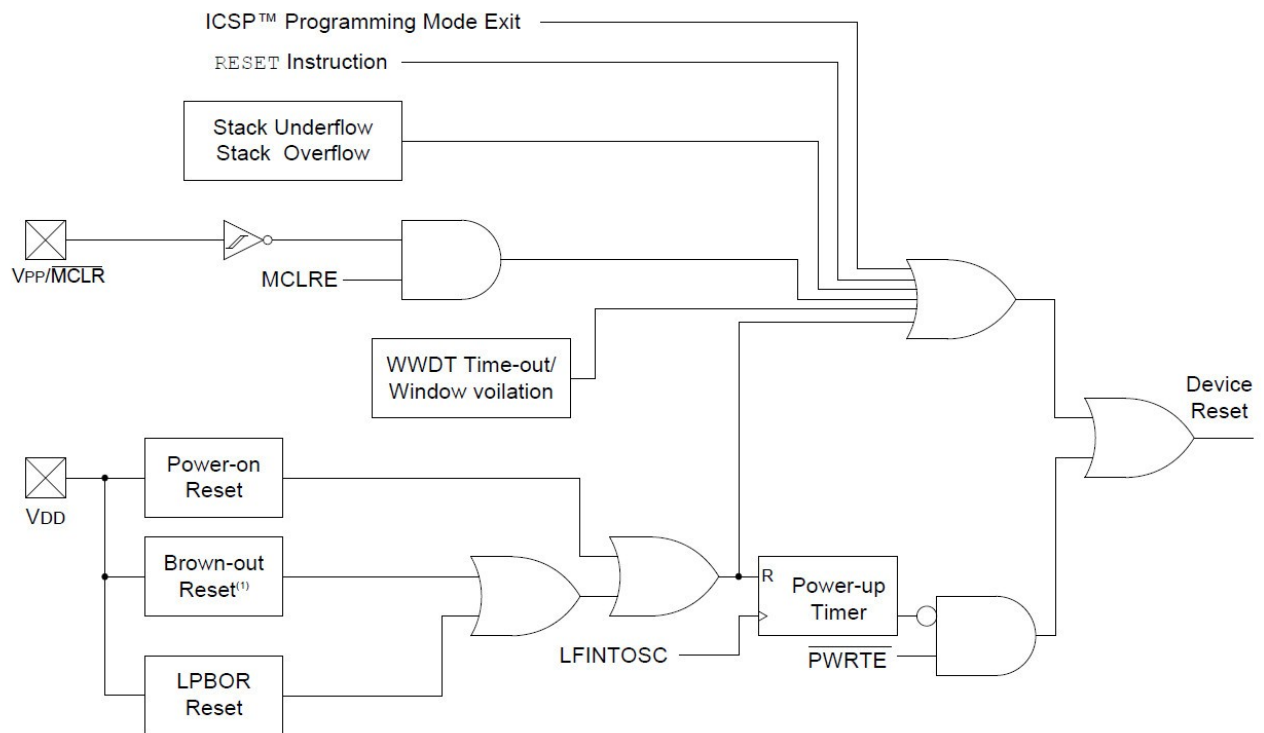
- Adresse 0x000000 : *reset*
- Adresse 0x000008 : *vecteur de priorité haute*
- Adresse 0x000018 : *vecteur de priorité basse*

Ces emplacements sont les mêmes pour tous les PIC18 de Microchip du marché. Chaque vecteur d'interruption ne propose que quelques octets de stockage. Pour information, un MCU PIC18F27K40 possède 128ko de mémoire programme Flash (ci-dessous 64KW soit 64KWords - 1Word=2octets sur PIC18) et 1ko de mémoire donnée non-volatile EEPROM. Ces deux technologies de mémoire sont non-volatiles et à ne pas confondre avec la mémoire donnée volatile de technologie SRAM (4Ko sur PIC18F27K40)

Address	Device				
	PIC18(L)Fx4K40	PIC18(L)F25/45K40	PIC18(L)F65K40	PIC18(L)Fx6K40	PIC18(L)Fx7K40
Note 1	Stack (31 Levels)				
00 0000h	Reset Vecor				
...	...				
00 0008h	Interrupt Vecor High				
...	...				
00 0018h	Interrupt Vecor Low				
...	...				
00 001Ah to 00 3FFFh	Program Flash Memory (8 KW)	Program Flash Memory (16 KW)	Program Flash Memory (16 KW)	Program Flash Memory (32 KW)	Program Flash Memory (64 KW)
00 4000h to 00 7FFFh					
00 8000h to 00 FFFFh		Not Present ⁽²⁾	Not Present ⁽²⁾	Not Present ⁽²⁾	
01 0000h to 01 FFFFh					
02 0000h to 1F FFFFh	Not Present ⁽²⁾				Not Present ⁽²⁾
20 0000h to 20 000Fh	User IDs (8 Words) ⁽³⁾				
20 0010h to 2F FFFFh	Reserved				
30 0000h to 30 000Bh	Configuration Words (6 Words) ⁽³⁾				
30 000Ch to 30 FFFFh	Reserved				
31 0000h to 31 00FFh	Data EEPROM (256 Bytes)		Data EEPROM (1024 Bytes)		
31 0100h to 31 01FFh	Unimplemented				
30 000Ch to 30 FFFFh	Reserved				
3F FFFCh to 3F FFFDh	Revision ID (1 Word) ⁽⁴⁾				
3F FFFEh to 3F FFFFh	Device ID (1 Word) ⁽⁴⁾				

← PIC18F27K40

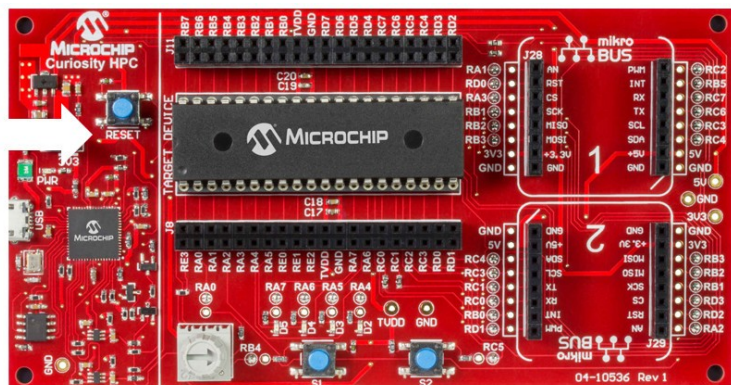
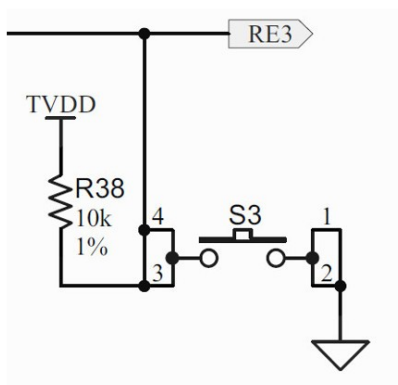
3.7. Gestion du RESET sur PIC18



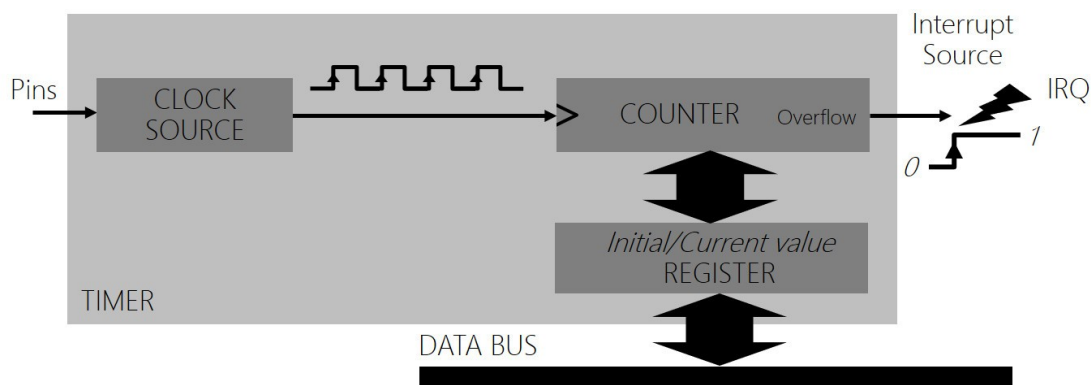
Nous pouvons observer ci-dessus la logique de démasquage du RESET sur PIC18F27K40. Par exemple, si un niveau logique bas est appliqué sur la broche externe RE3 nommée Vpp/MCLR (Master Clear) et si le bit de configuration MCLRE (MCLR Enable) est actif (fait par défaut à la mise sous tension), alors un RESET processeur est forcé.

Le CPU exécute alors le code présent dans le vecteur d'interruption du RESET à l'adresse 0x000000 de la mémoire programme. En général, celui-ci appelle la fonction main (en assembleur) ou la fonction de startup (en langage C) utilisée par défaut par la chaîne de compilation.

Observons par exemple ci-dessous le schéma électrique de câblage du bouton poussoir du RESET sur la carte Curiosity HPC de Microchip utilisée en TP. La broche Vpp/MCLR est la broche RE3 sur le boîtier du processeur. Il est courant sur un processeur qu'une broche ait différents noms et donc différents rôles possibles. Cela dépend de certaines fonctionnalités additionnelles associées à la plupart des broches. Chaque rôle associé à une broche aura à être configuré par le développeur en fonction des besoins spécifiques de chaque application. Ces aspects seront abordés dans la suite des exercices. Toute fonctionnalité matérielle du MCU est bien entendu documentée dans la documentation technique du processeur.



3.8. Module périphérique de comptage Timer



Un Timer sera toujours conçu autour d'un compteur ou décompteur numérique. Il est donc dédié aux opérations de comptage et le plus souvent à la gestion de bases de temps dans les applications (acquisitions de mesures physiques à intervalles de temps régulier, tâches périodiques, etc). Quelque soit la technologie du Timer, nous pouvons jouer sur 3 éléments afin de configurer une référence temporelle :

- **Fréquence/Période de comptage** (horloge de référence du Timer – fonction matérielle Clock Source ci-dessus)
- **Valeur initiale de comptage** (fonction matérielle Initial/Current Value Register ci-dessus)
- **Nombre de bits utilisés par le compteur 8-16-32-64 bits** avant débordement (fonction matérielle Counter et overflow ci-dessus)

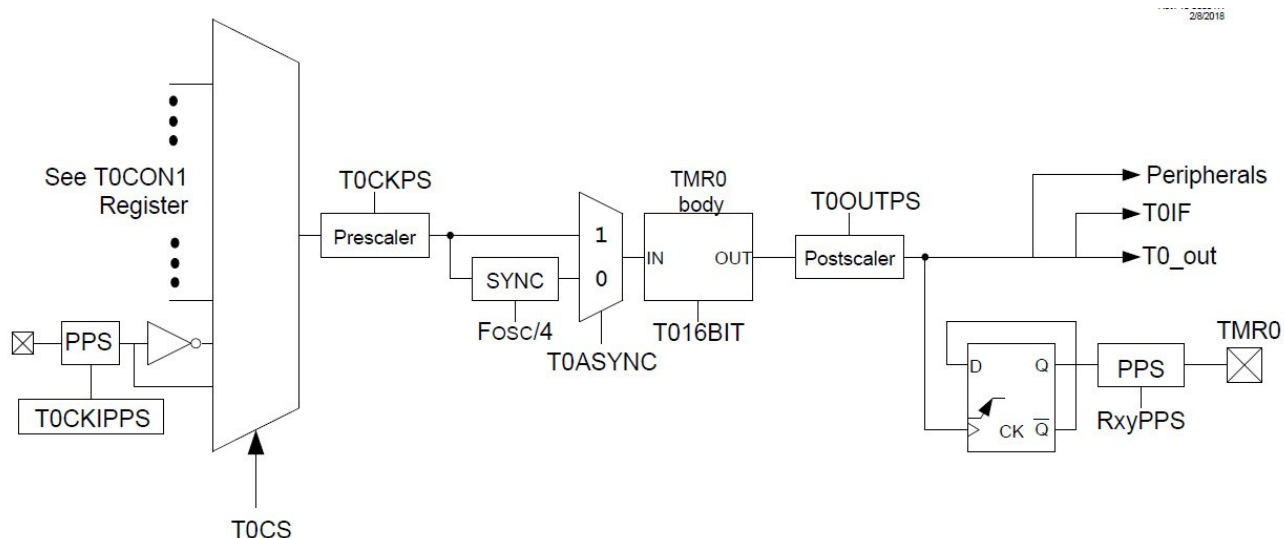
L'ensemble de ces trois éléments constituent en général un tout nommé Timer. Néanmoins, les Timers les plus élémentaires rencontrés sur processeur ne possèdent même pas de référence initiale de comptage ni d'horloge de référence configurable. Ils démarrent en partant de 0 à la mise sous tension et comptent à la fréquence de travail du CPU. Il sont souvent nommés TSC (Time Stamp Counter) et sont présents dans chaque CPU d'un grand nombre de processeurs du marché (Intel, AMD, TI C6000, ARM, etc). Ils sont souvent utilisés pour réaliser de la mesure de temps d'exécution de programme. Néanmoins, un Timer applicatif est plus riche en services matériels et fonctionnalités (comparateur numérique, rechargement automatique, module PWM, etc). Leur configuration peut sur certaines technologies être même relativement ardue.

Observons ci-dessous la configuration d'un Timer 8bits générique et calculons une valeur initiale de comptage à recharger après débordement afin d'obtenir une base de temps. Après comptage puis mise au niveau haut du bit de débordement (bit d'overflow), le Timer reprend le comptage à 0 par défaut (sans chargement de la valeur initiale). Par exemple sur 8bits ($255_{10} + 1_{10} = 1111\ 1111_2 + 1_2 = 1\ 0000\ 0000_2 = 256_{10}$ soit un résultat sur 9bits), le bit de débordement est le 9^{ième} bit actif. Exemple de calcul d'une valeur initiale de comptage sur Timer 8bits :

- Timer 8bis, comptage de 0 à 255 soit de 0x00 à 0xFF
- Hypothèse d'une horloge de référence de période 1ms
- Quelle serait la valeur initiale de comptage à charger au compteur afin d'obtenir un débordement après 100ms ?
- Réponse : *155 ou 0x9B (valeur à charger dans le registre initial de comptage)*
- Pourquoi : *Le Timer aura à compter 100 cycles de référence ($100 \times 1ms = 100ms$) avant débordement au 256^{ième} cycle. Nous devons donc débiter le comptage à la valeur 155*

Sur un Timer, le signal à la source de l'interruption ou IRQ n'est autre que le bit de débordement (overflow flag). Ce signal électrique est alors envoyé au CPU (cf. chapitre Interruption).

3.9. Module périphérique Timer0 sur PIC18



Le MCU PIC18F27K40 utilisé à l'école intègre 4 Timers 16bits (Timer0, Timer1, Timer3 et Timer5) ainsi que 3 Timers 8bits (Timers2, Timer4 et Timer6). Il intègre donc 7 Timers pouvant être utilisés pour différents besoins dans une application.

Le schéma bloc ci-dessus ne présente que la structure interne du Timer0. Les autres Timers du PIC18F27K40 offrent d'autres services matériels complémentaires et sont donc différents. Le Timer0 est configurable en mode 16bits ou 8bits. Seul le mode 16bits est présenté ci-dessus et sera utilisé en TP. Petit exercice, entourer sur le schéma les fonctions matérielles suivantes :

- Sous module compteur 16bits (COUNTER) ?
- Sous module de référence d'horloge (CLOCK SOURCE) ?
- Signal d'interruption IRQ envoyé au CPU par le Timer0 ?

Ouvrir la datasheet des processeurs PIC18Fx7K40 et parcourir le chapitre 18 relatif au Timer0 (Timer0 Module) afin d'observer la configuration des registres. Le Timer0 possède deux registres 8bits de configuration (T0CON0 et T0CON1) et deux registres 8bits de chargement de la valeur initiale de comptage (TMR0L et TMR0H). Analysons le registre 8bits T0CON1 chargé de configurer l'horloge de référence.

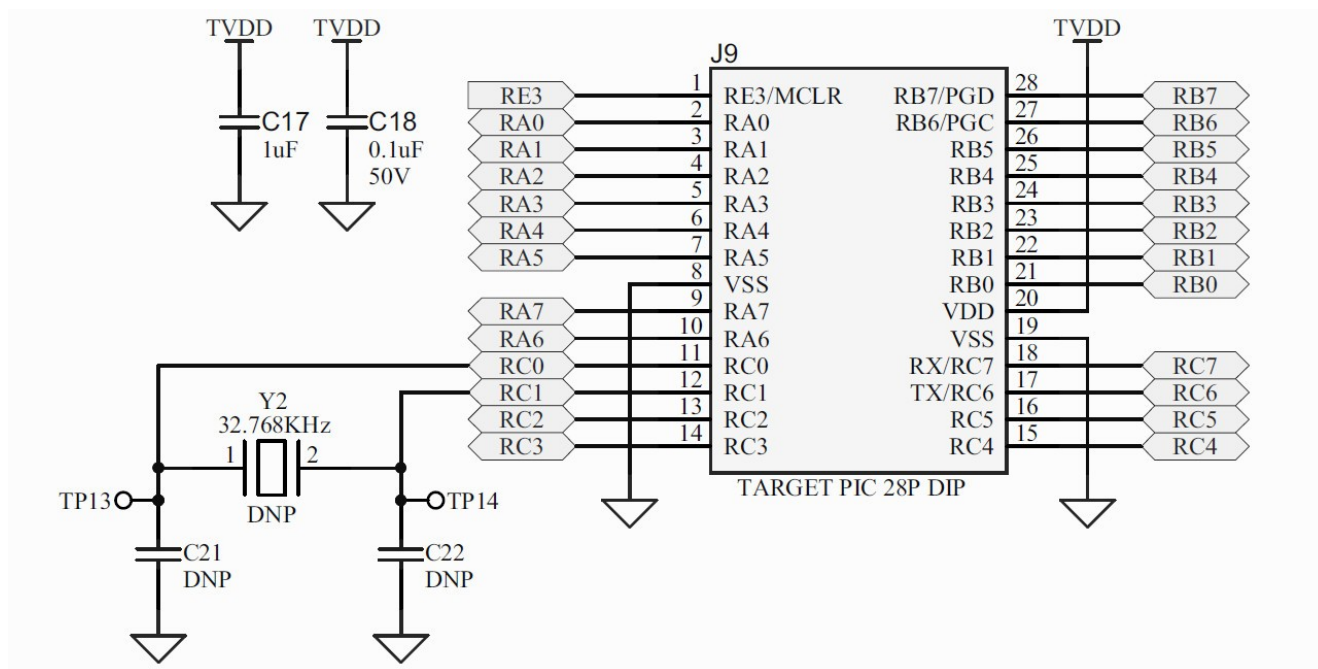
Nous pouvons observer sur le schéma ci-dessus que les champs T0CS (Timer0 Clock Source Select) et T0ASYNC (Timer0 Input Synchronization) permettent de piloter deux multiplexeurs d'aiguillage chargés de router la référence interne ou externe d'horloge. De même, le champ T0CKPS (Timer0 Clock Prescaler Select) permet de configurer un diviseur de fréquence (1:1, 1:2, 1:4, etc, 1:32768) avant d'entrer sur le compteur 16bits.

Name: T0CON1
Offset: 0xFD6

Timer0 Control Register 1

Bit	7	6	5	4	3	2	1	0
	T0CS[2:0]			T0ASYNC		T0CKPS[3:0]		
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

3.10. Configuration du Timer0 sur PIC18



Attention, la configuration proposée n'est pas celle demandée dans les TP. Elle sera donc à adapter. L'exemple suivant présente comment réaliser une base de temps très précise de 1 seconde. Il s'agirait d'une configuration de Timer si nous souhaitons réaliser une montre à quartz ou un réveil par exemple. En effet, les quartz 32.768KHz comme présenté ci-dessus sont typiquement ceux rencontrés dans les montres à Quartz. Les quartz possèdent une très faible dérive en fréquence par rapport à d'autres technologies de résonateurs (RC, MEMS, etc), typiquement proche de +/-10ppm (Particules Par Millions) soit +/- 0,00001% d'erreur et donc de précision.

Cet exercice pourrait d'ailleurs être réalisé sur la carte Curiosity HPC utilisée en TP à l'image de la capture du schéma électrique ci-dessus mais en utilisant néanmoins le Timer3 (broches utilisées par défaut sur la carte Curiosity HPC afin de relier le MCU au Quartz 32.768KHz).

```
; Timer configuration :
; Timer disable, 16bits mode, post-scaler 1:1
```

```
MOVLW    0b00010000
MOVWF    T0CON0
```

```
; Timer configuration :
; Pins selection RC0 and RC1
; Extern 32.768KHz Quartz resonator
; No CPU synchro, pre-scaler 1:1
```

```
MOVLW    0b00110000
MOVWF    T0CON1
```

```
; Timer initialization :
; Initial decimal value 32767 (0x7FFF)
; Always write TMR0H before TMR0L
```

```
MOVLW    0x7F
MOVWF    TMR0H
MOVLW    0xFF
MOVWF    TMR0L
```

```
; Interrupt configuration :
; Clear flag, enable
; Set low priority interrupt
```

```
BCF      PIR0, TMR0IF
BSF      PIE0, TMR0IE
BCF      IPR0, TMR0IP
```

```
; Interrupt configuration :
; Global interrupt enable
```

```
BSF      INTCON, IPEN
BSF      INTCON, GIEL
BSF      INTCON, GIEH
```

```
; Timer enable
; The Timer start to count only now !
```

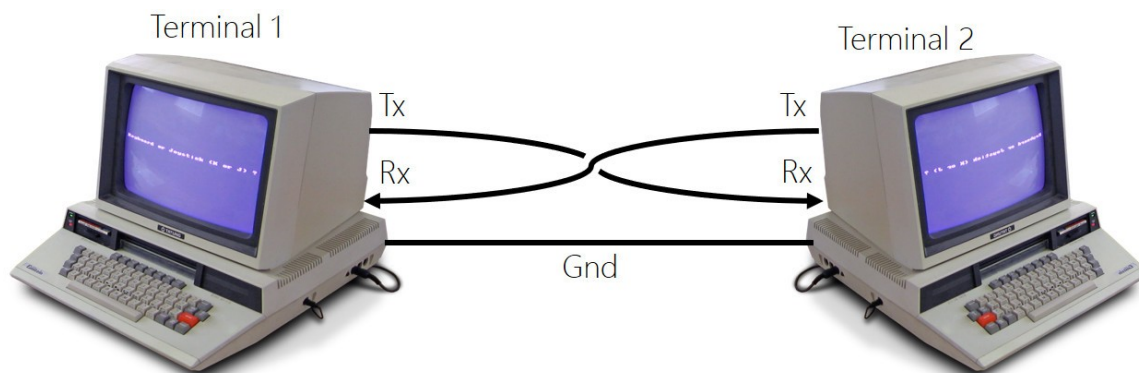
```
BSF      T0CON0, T0EN
```

```
; ... This program generate an
; interruption after 1s !
```


MODULE DE COMMUNICATION

UART ET LIAISON SÉRIE

4. MODULE DE COMMUNICATION UART



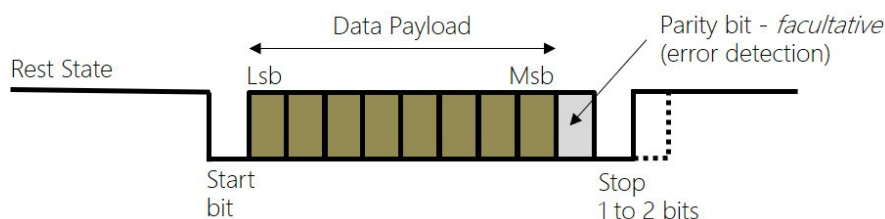
Le protocole de communication d'une liaison série asynchrone et les périphériques UART (Universal Asynchronous Receiver Transmitter) l'exploitant sont rencontrés depuis maintenant longtemps sur processeur numérique et ordinateur. A titre indicatif, la norme RS232 a été spécifiée en 1981 et n'a progressivement disparu sur ordinateur qu'à partir des années 2000 suite à l'arrivée de l'USB (norme USB1.0 Universal Serial Bus en 1996). Les modules de communications UART restent néanmoins encore très rencontrés en Systèmes Embarqués. Tout simplement car ils répondent toujours à un besoin (échanges bas débits de caractères en mode point à point. Il s'agit de communication Full Duplex (communication bidirectionnelle simultanée) utilisant 3 conducteurs physique. Les broches sont souvent nommées :

- **Rx** : Broche de réception croisée (toujours vu du récepteur). Tx vers Rx.
- **Tx** : Broche de transmission croisée (toujours vu du transmetteur). Tx vers Rx.
- **Gnd** : Fil de référence de masse (ground ou masse)

Les modules périphériques UART seront encore probablement utilisés très longtemps et sont bien connus des ingénieurs du domaine. L'UART est un périphérique spécialisé dans l'échange de caractères (données sur 8bits de façon générique) en topologie point à point entre 2 systèmes (peer to peer). Le tout avec des débits plutôt lents (de qq1KBps à qq100KBps) à notre époque (contrôle de procédé, test et prototypage, mesure, contrôle de module de communication, etc). Étant spécialisé dans l'échange de caractères, si un Homme cherche à interagir directement avec un périphérique UART depuis un ordinateur, celui-ci aura à configurer et utiliser un terminal asynchrone de communication (minicom, kermi, PuTTY etc sous GNU/Linux et TeraTerm, PuTTY, etc sous Windows). Rappelons de façon générique qu'un terminal ou une console est dédiée aux communications en mode texte par échange de phrases. Exemple ci-dessous d'interface de configuration sous terminal minicom sur système GNU/Linux.

```
+-----[configuration]-----+
| Filenames and paths          |
| File transfer protocols      |
| Serial port setup            |
| Modem and dialing            |
| Screen and keyboard          |
| Save setup as dfl             |
| Save setup as..              |
| Exit                          |
| Exit from Minicom            |
+-----+-----+-----+-----+
```

4.1. Protocole de communication d'une liaison série asynchrone



Ne pas confondre l'UART (périphérique matériel) et le protocole de communication (technique d'échange d'information). Une liaison série asynchrone est l'un des rares derniers protocoles asynchrones de communication rencontré sur le marché. Cela signifie que l'émetteur n'envoie aucune information concernant le débit ou vitesse de transfert, à l'instar des SPI et I2C (conducteur physique d'horloge dédié) ou encore de l'USB et l'Ethernet (champs d'horloge en en-tête des trames assurant la synchronisation d'une PLL à la réception).

Sur une liaison série asynchrone, l'état au repos de la ligne de communication est l'état logique haut (état de repos ou rest state ci-dessus). Une trame débutera toujours par 1 bit de start (état logique bas). Suivent 7 ou 8 bits de données utiles ou payload (charge utile), 1 bit de parité facultatif permettant une gestion élémentaire de détection d'erreur. Une trame se termine par 1 ou 2 bits de stop.

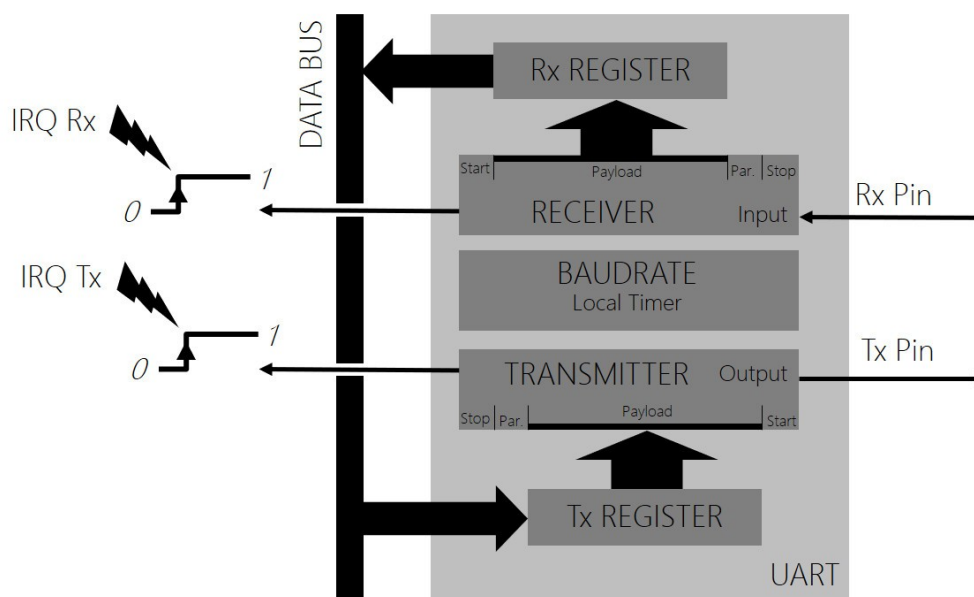
Néanmoins, la liaison série asynchrone est le plus souvent utilisée avec la configuration suivante : 1 bits de start, 8 bits de payload et 1 bit de stop (solution minimale et souvent suffisante pour du test). Une trame série complète d'informations est alors constituée de 10 bits dont 8 bits utiles (différence entre débit réel et utile). Par exemple, avec un débit configuré à 9600 Bd/s (Bps ou Baud/s ou symbole/s ou bits/s pour une liaison série asynchrone), l'envoi d'un bit dure environ 100 µs et donc l'envoi d'un caractère environ 1 ms.

Durant une communication de module UART à module UART, physique ou logique, ne jamais oublier de bien configurer récepteur et émetteur au même débit et avec le même protocole de communication (nombre de bits de donnée, nombre de bits de stop, gestion du bit de parité et contrôle de flux). Exemple ci-dessous d'interface de configuration sous terminal minicom sur système GNU/Linux :

- Débits 9600 Bps
- 8 bits de payload
- Pas de bit parité
- 1 bit de stop
- Pas de contrôle de flux matériel ni logiciel

```
+-----+
| A -   Serial Device       : /dev/ttyUSB0
| B - Lockfile Location    : /var/lock
| C -   Callin Program     :
| D -   Callout Program    :
| E -   Bps/Par/Bits       : 9600 8N1
| F - Hardware Flow Control : No
| G - Software Flow Control : No
|
| Change which setting? [ ]
+-----+
```

4.2. Module périphérique UART



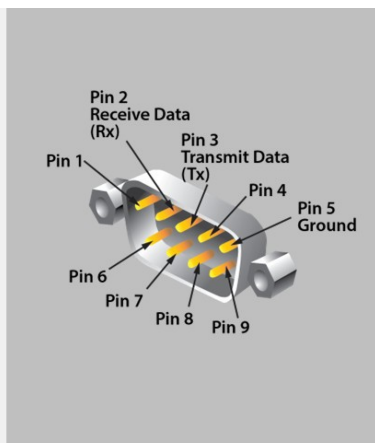
Un périphérique UART (Universal Asynchronous Receiver Transmitter) peut être vu comme deux périphériques dissociés. Un récepteur (receiver) et un transmetteur (transmitter). Néanmoins, les deux implémentent le protocole d'une liaison série asynchrone et leur configuration protocolaire est similaire (protocole et débit). Un Timer local dédié (Baurate ci-dessus) est souvent présent dans un UART afin de configurer le débit de communication.

Comme tout périphérique série de communication, le transmetteur et le récepteur sont conçus autour de registres à décalage. Le récepteur est chargé de récupérer les trames bit à bit dans son registre interne à décalage, de détecter d'éventuelles erreurs de transmission puis d'enlever l'enveloppe protocolaire de la trame pour ne récupérer que les données utiles (payload). A chaque nouvelle trame, la donnée utile est chargée dans un registre de travail (Rx register ci-dessus) afin d'être récupérée par le CPU et une requête d'interruption est envoyée au CPU (IRQ Rx) pour le prévenir de l'arrivée d'une donnée.

Le transmetteur fait quant à lui le travail inverse. L'application demande à envoyer une donnée par écriture dans le registre de transmission (Tx register ci-dessus, opération atomique de qq cycles CPU). Néanmoins, cela ne signifie pas que la donnée ait été transmise. Le transmetteur est alors responsable de charger la donnée utile dans son registre interne à décalage, de rajouter l'enveloppe protocolaire (start, stop voire parité) puis d'envoyer bit à bit les données en respectant le débit configuré. Une fois la donnée envoyée, une requête d'interruption est envoyée au CPU (IRQ Tx) pour le prévenir de la fin de transmission.

4.3. Norme RS232

DB-9 male to DB-9 female



DB-9 male to USB

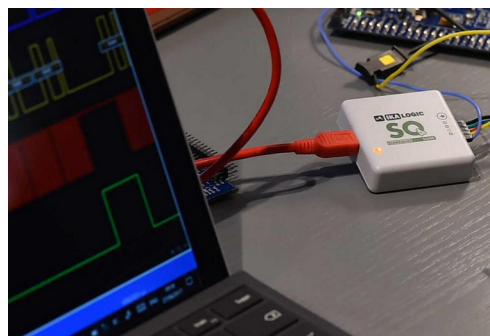
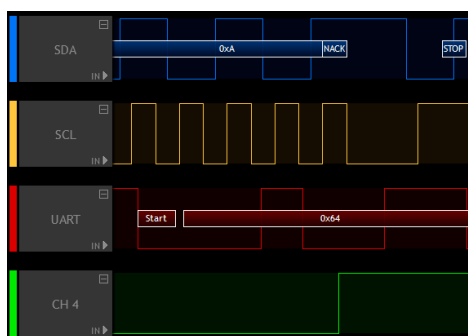


Ne pas confondre la norme RS-232 (nommée aussi EIA 232) avec le protocole d'une liaison série asynchrone et un périphérique UART. Cette norme est apparue en 1981 et a donné naissance aux interfaces port COM sur ordinateur sous Windows (remplacé par l'USB depuis les années 2000). Cette norme de communication de machine à machine utilise des UART et implémente une liaison série asynchrone, mais tend à standardiser les débits de communication, les connecteurs et câbles associés, les longueurs de câbles, les niveaux de tension sur les conducteurs physiques, etc.

Cette norme standardise donc des contraintes et des limites physiques permettant de faciliter l'interfaçage et la communication de machines entre elles. Observons quelques unes de ces limitations :

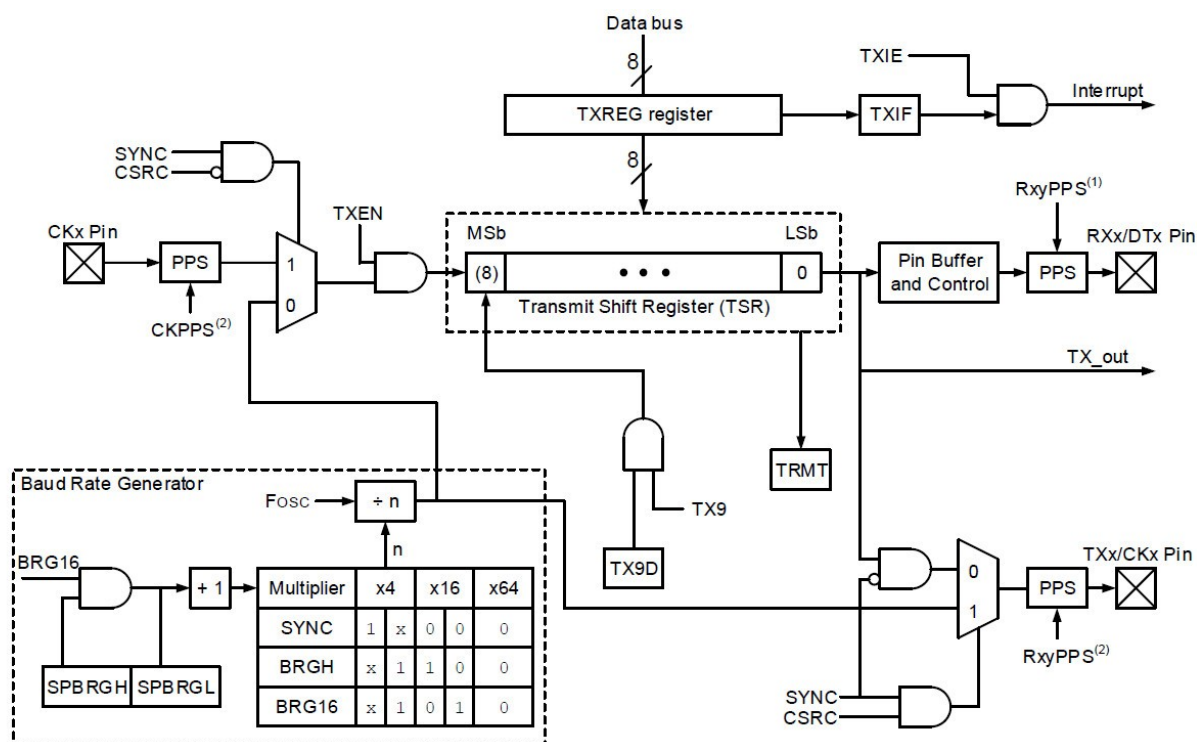
- Longueurs de câbles en fonction des débits : 60m (2,4KBd/s), 15m (9,6KBd/s) et 2,6m (56KBd/s)
- Niveaux de tensions (logique inversée) : niveau logique 0 (de +3V à +25V) niveau logique 1 (de -3V à -25V)
- Technique de codage des bits : NRZ (Non Return to Zero)
- Types de connecteurs : DB9 (9 broches), etc

Dans certaines phases de debug et de test, nous pouvons être amenés à utiliser des outils matériels d'analyse des trames circulant sur les bus de communication externes au MCU (broches Tx et Rx pour une liaison série asynchrone). Nous utilisons par exemple à l'école des solutions conçues en France par la société IKALOGIC ainsi que des options d'analyse proposés avec les oscilloscopes.



4.4. Module périphérique UART sur PIC18

Transmetteur UART liaison série asynchrone

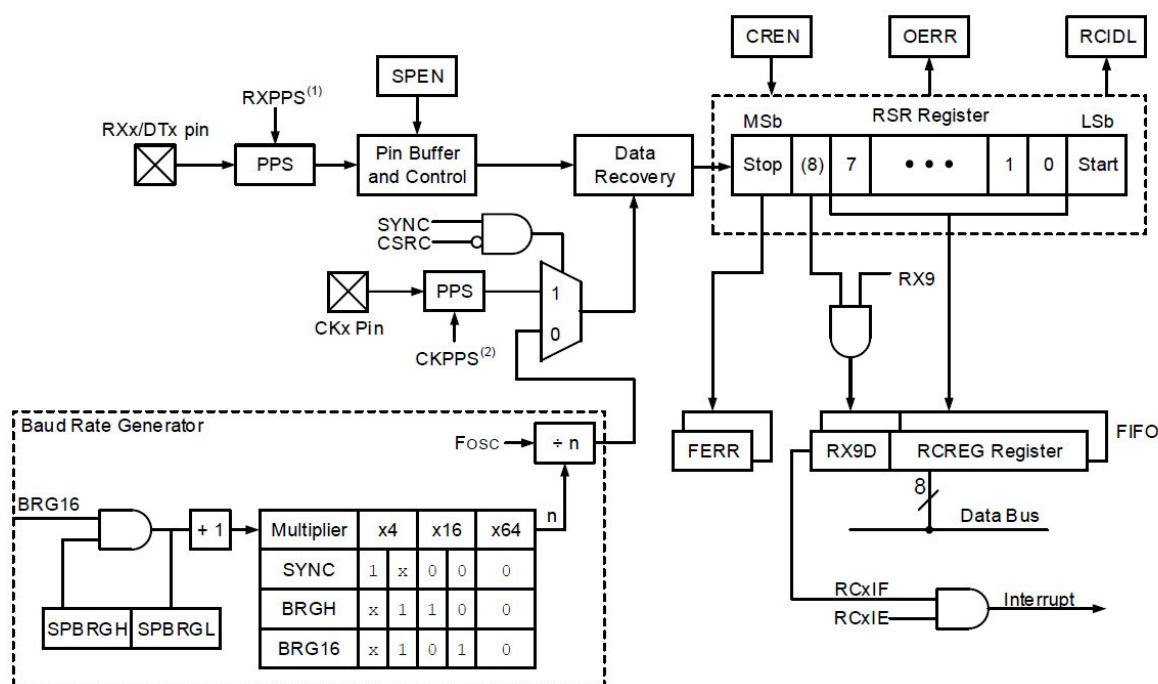


Le MCU PIC18F27K40 utilisé en TP intègre deux UART (UART1 et UART2). Les deux UART seront configurés et utilisés durant cet enseignement. Ces deux modules UART sont identiques. Seuls les noms des registres diffèrent (nommage avec indice 1 ou 2). Le schéma bloc ci-dessus présente la structure du transmetteur de ces UART. Petit exercice, entourer sur le schéma les fonctions matérielles suivantes :

- *Registre à décalage de transmission (envoi bit après bit cadencé sur l'horloge de référence)*
- *Registre de travail (pour écriture de la payload depuis l'application) permettant de charger le registre à décalage de transmission*
- *Ensemble assurant la référence d'horloge et donc le débit de la communication (Baud Rate Generator)*
- *Broche Tx de sortie*
- *Signal d'interruption IRQ envoyé au CPU (après envoi d'une information)*

Ouvrir la datasheet des processeurs PIC18F7K40 et parcourir le chapitre 27 relatif aux UART (EUSART Enhanced Universal Asynchronous Receiver Transmitter) afin d'observer la configuration des registres. Prenons l'exemple de l'UART1, sachant que l'UART2 possède une stratégie de configuration et d'utilisation similaire. L'UART 1 possède 2 registres de configuration récepteur/transmetteur (RC1STA et TX1STA), 3 registres de configuration du débit à l'image de la configuration d'un Timer (BAUDCON1, SP1BRGH et SP1B1GL) et deux registres de travail pour la gestion des payload (TX1REG et RC1REG). Analysons une partie du registre 8bits TX1STA chargé de configurer le transmetteur. Nous pouvons observer sur le schéma ci-dessus que le champ TXEN (Transmitter Enable) permet d'appliquer la référence d'horloge au registre à décalage de transmission (TSR ou Transmit Shift Register) et autorise donc une transmission. Le champ SYNC (Synchronous) permet potentiellement d'utiliser une référence d'horloge externe, comme de sortir sur broche cette même référence et de transformer la communication asynchrone en communication synchrone.

Récepteur UART liaison série asynchrone



Le schéma bloc ci-dessus présente la structure du récepteur UART sur PIC18. Petit exercice, entourer sur le schéma les fonctions matérielles suivantes :

- *Registre à décalage de réception (récupération bit après bit de la trame de communication)*
- *Registres en FIFO (First In First Out) de travail pour la réception (pour lecture de la payload par l'application)*
- *Ensemble assurant la référence d'horloge et donc le débit de la réception (Baud Rate Generator)*
- *Broche Rx d'entrée*
- *Signal d'interruption IRQ envoyé au CPU (après réception d'une information)*

Ouvrir la datasheet des processeurs PIC18F7K40 et parcourir le chapitre 27 relatif aux UART (EUSART Enhanced Universal Synchronous Asynchronous Receiver Transmitter) afin d'observer la configuration des registres. Analysons une partie du registre 8bits RX1STA chargé de configurer le récepteur. Nous pouvons observer sur le schéma ci-dessus que le champ SPEN (Serial Port Enable) permet de valider ou pas la connexion physique de la broche au registre à décalage de réception. Nous pouvons également constater que le récepteur est chargé de détecter les erreurs. Si une erreur de communication est détectée, alors l'un des champs FERR (Frame Error) et OERR (Overrun Error) est activé par le récepteur. Dans tous les cas, si une erreur de communication est détectée ou constatée dans l'application, la meilleure stratégie reste de demander un renvoi à l'émetteur.

Receive Status and Control Register

Bit	7	6	5	4	3	2	1	0
	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
Access	R/W	R/W	R/W	R/W	R/W	RO	R/HC	R/HC
Reset	0	0	0	0	0	0	0	0

4.5. Configuration du module UART sur PIC18

Attention, la configuration présentée ci-dessous n'est pas celle demandée dans les TP. Elle sera donc à adapter. L'exemple suivant présente une séquence assembleur de code configurant le transmetteur de l'UART1 avec un débit de 115200Bd/s pour un horloge système de 64MHz. Une fois configuré, le caractère 'D' est transmis en respectant le protocole d'une liaison série asynchrone sur la broche TX1/RC6 du processeur. Le bit ou flag TRMT (TSR Register is Empty) est mis à jour par l'UART1 et permet de savoir si des bits restent présents dans le registre à décalage de transmission (registre TSR).

```
; UART1 Transmitter configuration :
; uart1 enable, asynchronous mode
; 8bits data, no parity, high baudrate

MOVLW    0b00100100
MOVWF    TX1STA

; UART1 BaudRate Generator configuration :
; 16bits baudrate mode
; baudrate 115.200KBps (64MHz CPU clock)

MOVLW    0b00001000
MOVWF    BAUDCON1
MOVLW    0x8B
MOVWF    SP1BRG
MOVLW    0x00
MOVWF    SP1BRGH
```

```
; send 'D' ASCII code
; 'D' = 68 = 0x44

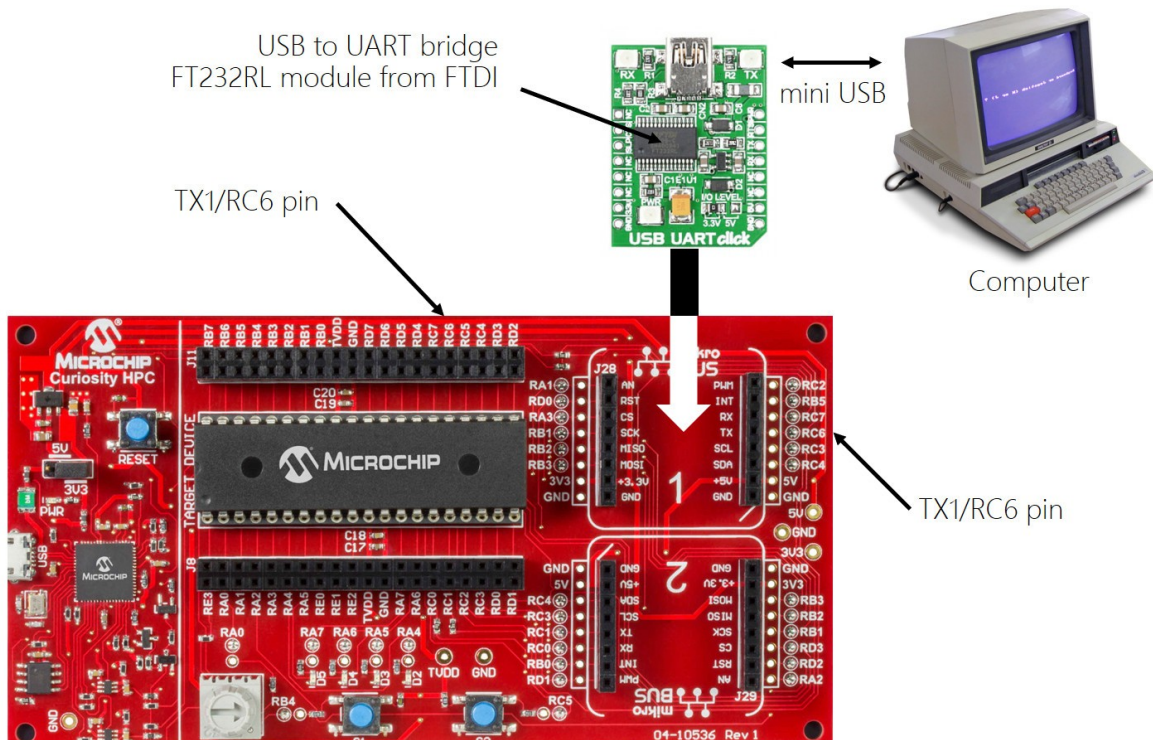
MOLW     0x44
MOWF     TX1REG

; wait for the end of transmission...

Label:
    BTFSS    TX1STA, TRMT
    GOTO     Label

; ... This program send 'D' character by UART1
; The transmission end after 86.806 us
; 115200 bit/s = 8,6806 us/bit
; Frame length : 10 bits
; 1 bit start + 8 bits payload + 1 stop
```

A notre époque, la plupart des ordinateurs modernes ne disposent plus de port COM avec connecteur DB-9 pour liaison série. Nous utilisons généralement des modules passerelles USB vers UART (TTL 0-5V) assurant une transposition de protocole mais ne modifiant pas la donnée échangée. Nous parlons alors de bridge ou de passerelle (exemple de votre box internet, par exemple ADSL vers WIFI). Un bridge UART1 vers UART2 sera d'ailleurs développé en TP. Sur le marché des modules "USB to UART", la société FTDI s'est spécialisée sur ce type de solution.



MODULE AUDIO BLUETOOTH

EXTERNE

5. MODULE AUDIO BLUETOOTH EXTERNE

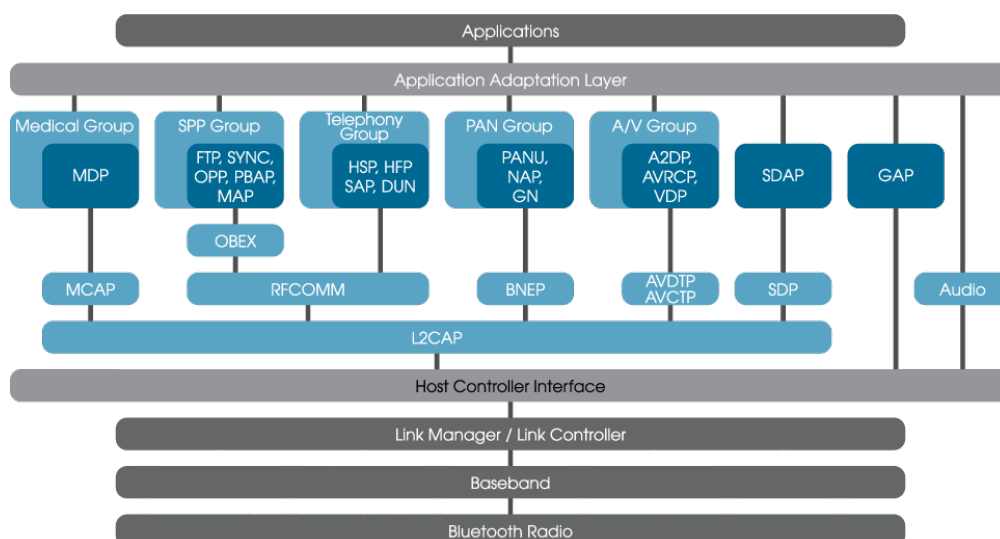
5.1. Introduction

Dans l'embarqué, la communication de machine à machine (M2M ou Machine to Machine) est un point souvent central dans la conception d'une application. Il s'agit notamment du cœur des systèmes dits nomades. Les solutions technologiques de communication les plus couramment rencontrées en embarqué à notre époque sont le WIFI, le Bluetooth, les technologies cellulaires (LTE, 4G, 3G et 2G), le NFC (Near Field Communication), LORA, Sigfox, etc. Aucune technologie n'est meilleure qu'une autre. Elles possèdent toutes des avantages et inconvénients. Elles répondent toutes à une famille de besoins parfois très différents.

Par exemple, diffuser un flux vidéo streaming en WIFI est techniquement et physiquement très différent d'une simple lecture de température puis échange toutes les minutes sur un appareillage de production dans une usine. Il existe donc des protocoles et solutions techniques adaptées à chaque besoin. Le tableau ci-dessous synthétise "approximativement" les périmètres d'actions des principales solutions technologiques actuelles du marché.

Protocol	Optimized for Battery Life	Nominal Range Limit	Typical Data Rate	Spectrum
Bluetooth		<10m	2Mbps	ISM 2.4GHz unlicensed
WIFI		<100m	>100Mbps	ISM 2.4GHz/5GHz unlicensed
LORAWAN		>10Km	<50Kbps	ISM 900MHz unlicensed
2G/3G		>30Km	<2Mbps	<i>Licensed cellular</i>
4G		>30Km	>100Mbps	<i>Licensed cellular</i>
NFC		<4cm	100Kbps	ISM 13.56MHz unlicensed

Le Bluetooth est par exemple une norme de communication permettant l'échange bidirectionnel de données sur de courte distance (<100m voire <qqm suivant la classe de fonctionnement) en utilisant des ondes radio dans la bande UHF (bande de fréquence autour de 2,4GHz). Comme la plupart des normes et protocoles de communication, le Bluetooth peut être représenté en couches protocolaires plus ou moins proches du monde physique (couche Radio). Nous parlons souvent de Stack (ou Pile) protocolaire en faisant référence à certaines bibliothèques logiciel. Dans l'exercice de la trame de TP, le module RN52 utilisé intègre et implémente le profil Audio A2DP (groupe A/V ou Audio/Vidéo), utilisant lui-même la couche liaison de multiplexage L2CAP, elle-même interfacée par les couches en bandes de bases et Radio.



Il existe sur le marché des modules de communication plus ou moins intégrés (emport potentiel de couches protocolaires ou stack Bluetooth) avec des échelles de coûts souvent liées à la richesse des fonctionnalités embarquées dans le module de communication (exemple du schéma fonctionnel ci-dessous). Dans le cadre de notre application, nous utiliserons une solution intégrée gérant déjà pleinement une partie du protocole Bluetooth désiré (profil Audio A2DP, couche liaison L2CAP, couche en bande de base et couche radio) et s'interfaçant par simple liaison série asynchrone et périphérique UART. Par exemple, le module Bluetooth RN52 de Microchip supporte déjà les couches protocolaires nécessaires à l'application (cf. schéma de droite ci-dessous). Nous n'aurons qu'à le configurer et le contrôler depuis le MCU par envoi de chaînes de caractères ASCII avec une communication par liaison série asynchrone via UART (cf. tableaux ci-dessous).



Nous pouvons par exemple observer ci-dessous un extrait de la documentation technique du module Bluetooth RN52 (cf. *mcu/tp/doc/datasheets*) présentant la synthèse de toutes les commandes de configuration et d'action supportées. Par exemple, si notre MCU envoie par UART les suites de caractères ASCII suivantes, le module RN52 réalisera les traitements demandés :

- **AV+** : incrémente le volume (Audio Volume +)
- **AT-** : rejoue la dernière piste Audio (Audio Track -)
- **SN, <string>** : change le nom du réseau Bluetooth créé par le module RN52 par <string>
- **etc**

SET COMMANDS

Command	Description
S , <hex16>	Audio output routing
S-, <string>	Sets the normalized name
S^, <dec>	Automatic Shutdown on Idle
S%, <hex16>	Extended Features
SA, <0, 1, 2, 4>	Authentication enable/disable
SC, <hex24>	Service class
SD, <hex8>	Discovery profile mask
SF, 1	Factory defaults
SK, <hex8>	Connection profile mask
SM, <hex32>	Microphone/LINEIN gain
SN, <string>	Device name
SS, <hex8>	Speaker Level
ST, <hex8>	Tone Level
STA, <dec>	Connection Delay
STP, <dec>	Pairing Timeout
SU, <hex8>	UART Baudrate

ACTION COMMANDS

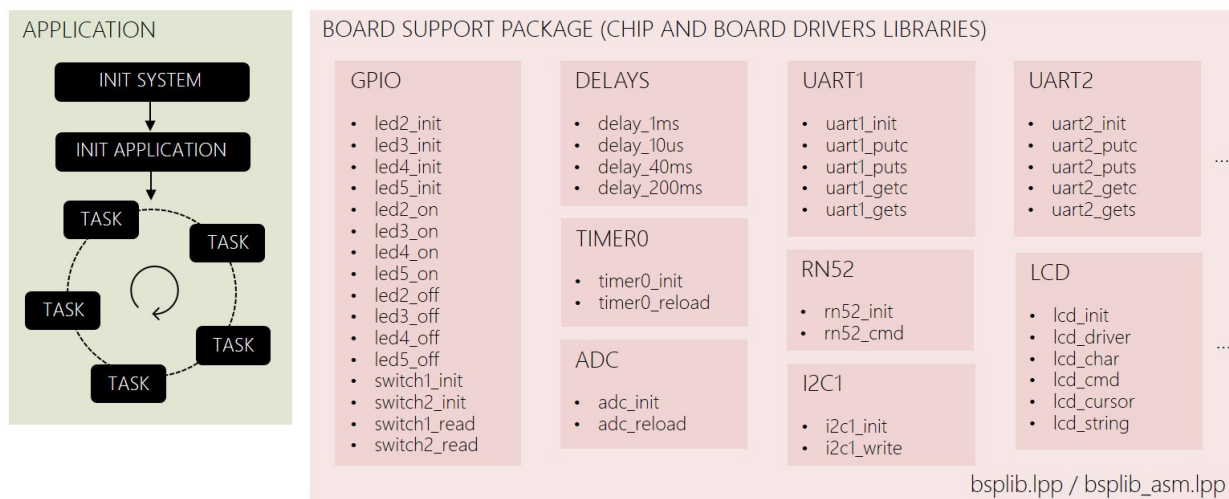
Command	Description
+	Toggle the local echo of RX characters in Command mode
@, <flag>	Toggle whether the module is discoverable
#, <0, 1>	Accept/reject pairing
\$	Put the module into DFU mode
A, <telephone number>	Initial a voice call to <telephone number>
AD	Retrieve track metadata information
AR	Redial last dialed number
AV+	Increase the volume (AVRCP command)
AV-	Decrease the volume (AVRCP command)
AT+	Play the next track (AVRCP command)
AT-	Play the previous track (AVRCP command)
AP	Pause or start playback (AVRCP command)
B	Reconnect Bluetooth® profiles to the most recently paired and connected device
C	Accept an incoming voice call
E	Terminate an active call or reject an incoming call
F	Release all held calls
I@	Read GPIO configuration
I@, <hex16>	Set GPIO configuration
I&	Reads current GPIO levels for input
I&, <hex16>	Set GPIO levels for output
J	Accept waiting calls and release active calls
K, <hex8>	Kill the currently active connection
L	Accept waiting calls and hold active calls
M, <flag>	Toggle the on hold/mute function
N	Add held call
O	Connect two calls and disconnect the subscriber
P	Activate Voice Command
Q	Query the current connection status
R, 1	Reboot
U	Reset Paired Device List (PDL)
T	Retrieves caller ID information
X, <0, 1>	Transfer call between HF and AG

CONCEPTION D'APPLICATION ET ORDONNANCEMENT

8. CONCEPTION D'UNE APPLICATION ET ORDONNANCEMENT

Toute application logicielle en Systèmes Embarqués débutera par la configuration séquentielle des services matériels nécessaires (périphériques internes et externes). En fonction de l'état des entrées du système (switch, bouton poussoir, capteurs divers, interfaces réseaux, etc) suivra ensuite l'initialisation de l'environnement logicielle de l'application (variables d'environnement) et la mise à jour des interfaces utilisateur (afficheur, LED, déploiement de réseau de communication, etc) avec l'état du produit par défaut au démarrage. L'application pourra alors débiter.

Une application *software* est une solution logicielle de supervision d'un produit répondant à un besoin (souris, clavier, lecteur MP3, assistance au freinage, etc) et a été développée pour une mission spécifique. Cette mission globale sera toujours la même pour un produit donné. Et peut intégrer des sous missions (lire les entrées du système, gérer l'affichage utilisateur, gérer les interfaces de communication, etc). Ces sous missions sont nommées tâches. Une application aura donc toujours plusieurs tâches à réaliser. A titre indicatif, le *scheduler* ou ordonnanceur d'un système d'exploitation est chargé de gérer et ordonner un environnement multi-tâches et de répartir les besoins (tâches ou *threads* souhaitant s'exécuter) sur les ressources d'exécution (un ou plusieurs CPU). Cependant, nous ne découvrirons ce type d'ordonnancement (*scheduling online*) qu'en 2^{ème} année à l'école. En première année, nous développerons une application dite *Baremetal*, soit nue sur le MCU.



L'un des objectifs premier de cette trame d'enseignement est le développement d'un BSP (Board Support Package) pour la carte Curiosity HPC sur PIC18F27K40 (développements, tests, validations voire documentation). Des projets de tests unitaires ont été réalisés par fonction périphérique et certaines intégrations partielles ont également été validées (UART1 avec UART2, ou Timer0 avec GPIO par exemples). A travers le développement d'une application, nous avons à valider l'intégration de l'ensemble des modules (GPIO, Timer0, UART1, UART2, I2C, etc) nécessaires à développement du produit (application Audio Bluetooth dans notre cas). La conception d'une application nécessite une grande attention afin de définir une architecture ainsi qu'une stratégie d'ordonnancement répondant de façon optimale au besoin et garantissant modularité, clarté, simplicité, évolutivité et robustesse au projet logiciel.

Nous allons nous attacher à développer une application conçue autour d'un *scheduler offline*. Le séquencement et les mesures temporelles des différentes tâches applicatives seront réalisées et validées avant la mise en production durant les phases de développement et de test. Nous maîtriserons donc le déterminisme à l'exécution de notre application. Ce type d'ordonnancement est souvent rencontré dans les systèmes critiques, solutions où le droit à l'erreur n'est pas permis malgré une complexité systémique pouvant être importante (par exemple dans le domaine de l'avionique avec Airbus). Les *scheduler offline* peuvent cependant être déployés sur de plus petits systèmes, par exemple dans des applications industrielles (Automate Programmable Industriel, exemple du compteur Linky, etc). Cependant, ce type d'ordonnancement est plus difficilement évolutif et maintenable que d'autres solutions dites *online* (ordonnancement et partage du temps CPU à l'exécution).

8.1. Philosophie Unix



L'un des plus grands défis de l'ingénieur est de converger vers la solution la plus simple durant la conception et le développement d'une solution (électronique, informatique, mécanique, etc). Nous parlons souvent de principe de parcimonie ou du rasoir d'Ockham. Nous pouvons voir ci-dessus Ken Thompson (à gauche, concepteur d'Unix et inventeur du langage B) et Dennis Ritchie (à droite, inventeur du langage C et codéveloppeur d'Unix), deux des pères des langages informatique et des systèmes d'exploitation ayant formalisé des bases philosophiques dans le développement des systèmes d'information. Ces règles peuvent être appliquées à bien des domaines. Vous trouverez par exemple ci-dessous 13 des 17 règles Unix proposées par Eric S. Raymond, un hacker Américain notamment connu pour avoir popularisé le terme "Open Source" par opposition à "free software", dont Richard Stallman est l'initiateur (le père du projet GNU et de la Free Software Foundation).

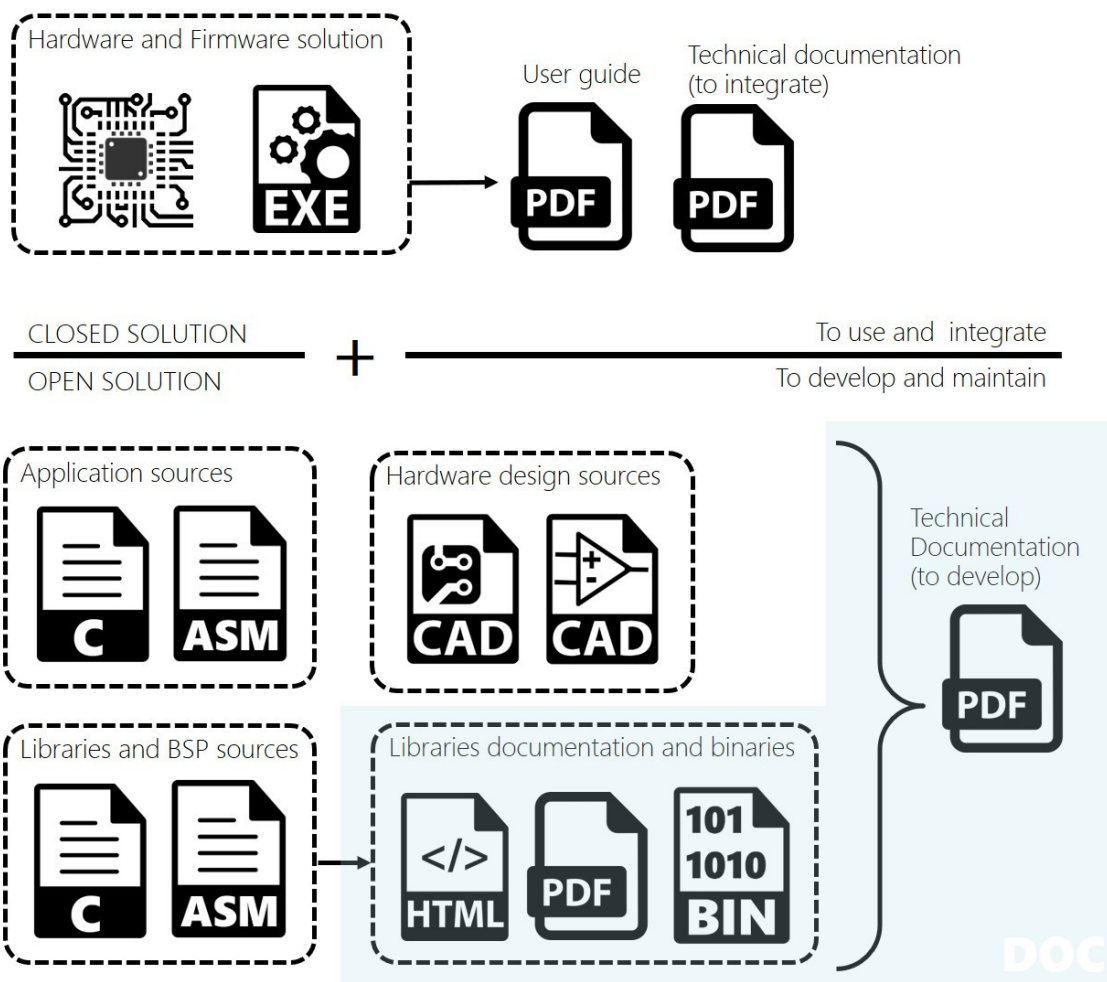
«La simplicité est la sophistication suprême » Léonard de Vinci

- Règle de Modularité : *Écrire des éléments simples reliés par de bonnes interfaces*
- Règle de Clarté : *La Clarté vaut mieux que l'ingéniosité*
- Règle de Séparation : *Séparer les règles du fonctionnement; Séparer les interfaces du mécanisme*
- Règle de Simplicité : *Concevoir pour la simplicité; ajouter de la complexité seulement par obligation*
- Règle de Parcimonie : *Écrire un gros programme seulement lorsqu'il est clairement démontrable que c'est l'unique solution*
- Règle de Transparence : *Concevoir pour la visibilité de façon à faciliter la revue et le déverminage*
- Règle de Robustesse : *La robustesse est l'enfant de la transparence et de la simplicité*
- Règle de Représentation: *Inclure le savoir dans les données, de manière que l'algorithme puisse être bête et robuste*
- Règle du Silence : *Quand un programme n'a rien d'étonnant à dire, il doit se taire*
- Règle de Dépannage : *Si le programme échoue, il faut le faire bruyamment et le plus tôt possible*
- Règle d'Optimisation : *Prototyper avant de figoler. Mettre au point avant d'optimiser*
- Règle de Diversité : *Se méfier des affirmations de « Unique bonne solution »*
- Règle d'Extensibilité : *Concevoir pour le futur, car il arrivera plus vite que prévu*
- etc

DOCUMENTATION TECHNIQUE ET LIVRABLES

9. DOCUMENTATION TECHNIQUE ET LIVRABLES

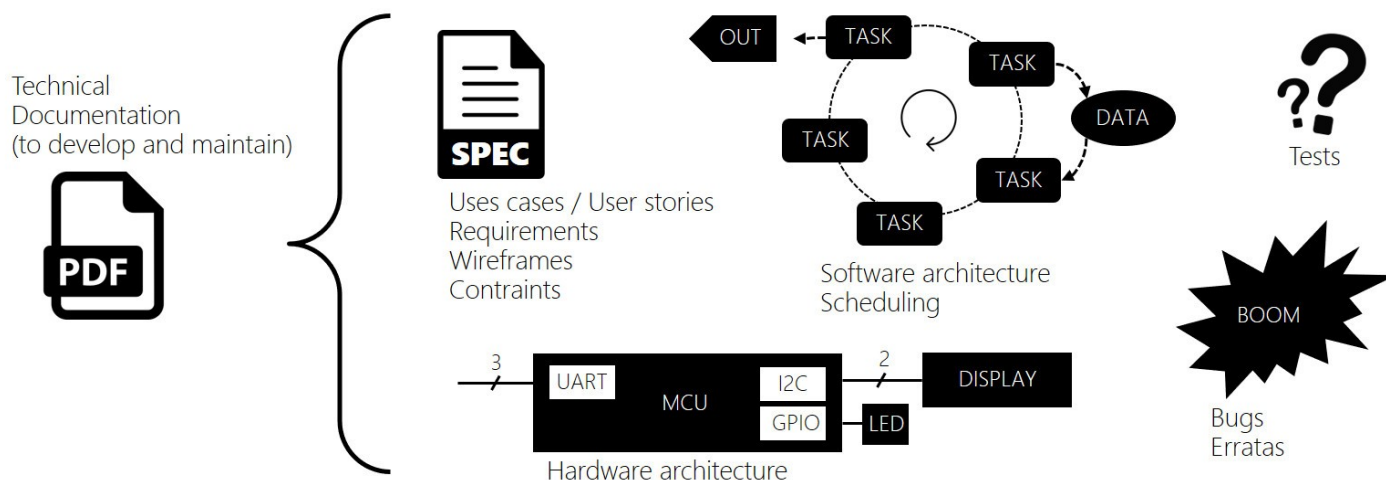
Le schéma ci-dessous présente l'ensemble des livrables constituant une solution (système embarqué complet). En fonction du contrat signé entre client (MOA ou Maîtrise d'ouvrage) et prestataire (MOE ou Maîtrise d'œuvre), certaines parties de la solution resteront fermées.



La documentation peut être un point essentiel dans le processus de développement, d'intégration, de validation et de maintenance d'une solution. Parlons d'histoires réelles et vécues par bien des ingénieurs. Un ingénieur développe durant des semaines voire des mois une solution pour un client (sous-traitance, équipe interne, société de services, etc). Celui-ci change de mission voire d'entreprise, ou le contrat n'inclue pas forcément la maintenance. Sa solution est fonctionnelle et a été intégrée à une version donnée du produit. De même, elle était maladroitement architecturée et développée (ingénieur junior, stagiaire, incompétences techniques, etc) et maladroitement documentée. Un jour, un autre ingénieur doit se plonger dans sa solution (déverminage, correctif, évolution, intégration vers une nouvelle version du produit, portage, etc). La perte de temps peut être considérable, des jours, des semaines voire bien plus, sachant que le problème peut se reproduire. Cette histoire est très courante en industrie, de nombreux retours de ce type nous sont faits chaque année.

Cependant, pour information, dans le domaine du logiciel, le concept de "code propre" émerge. Il met la solution logicielle en avant et au centre de tout, et part du principe qu'une solution bien architecturée, conçue et bien codée (modularité, clarté, simplicité, séparation, etc) se suffit à elle-même. Elle ne nécessitera alors que très peu de documentation. Selon votre majeure et expertise, une formation dédiée en génie logiciel, développement agile et en code propre sera réalisée.

9.3. Documentation technique



La documentation technique est directement destinée aux équipes techniques de développement et de maintenance du produit. L'objectif premier étant de permettre à un ingénieur développeur, intégrateur ou architecte d'appréhender le plus rapidement possible les architectures et solutions logicielles comme matérielles anciennement développées, ainsi que le positionnement du sous système dans un projet global de plus grande envergure.

Éditer une documentation technique présentant l'ensemble de vos développements (BSP et application Bluewave). Pour ce travail, la construction du plan et donc la définition de l'architecture du document est la première chose à réaliser. Ce travail demande un esprit de synthèse, une vision d'ensemble du projet et une approche progressive. Toujours se mettre à la place du lecteur (un ingénieur) qui découvre votre projet et qui doit comprendre rapidement où travailler et opérer. S'aider de représentations graphiques et de schémas commentés afin de présenter vos travaux de développement. Être concis et précis, aller à l'essentiel en s'efforçant d'être le plus rigoureux possible quant au vocabulaire et descriptions techniques utilisés. Voici les points à présenter :

- Présenter le *projet dans son ensemble* : Expression du besoin, périmètre d'action du produit, guide d'utilisation et cas d'usages (use cases and user stories), cahier des charges, spécifications techniques et contraintes (requirements), etc
- Présenter les *outils de développement et technologies* déployées sans oublier les *versions* de chaque outil (IDE, toolchains, sonde de programmation, processeurs, cartes, etc)
- Présenter à l'aide d'un schéma fonctionnel commenté l'*architecture matérielle du produit* (interfaces externes utilisateur, bus et signaux d'interconnexion avec le processeur, périphériques internes au processeur, brochages et nommage des signaux, etc). Mettre en documents annexes les schémas électriques et layout de routage (version PDF et sources CAO).
- Présenter à l'aide d'un schéma fonctionnel commenté l'*architecture logicielle de l'application* (tâches applicatives, ressources partagées, description du travail de chaque tâche, stratégie d'ordonnancement, etc)
- Présenter les *procédures de test* de la solution (boîte blanche et/ou boîte noire) ainsi que les *résultats des mesures physiques* liés au comportement de l'application (temps d'exécution des tâches applicatives, charge CPU, réactivités des interfaces, consommation énergétique en veille et en fonctionnement, etc)
- Présenter *explicitement et clairement les bugs et erratas connus* voire des méthodes de résolution ou de contournement
- Proposer des améliorations et évolutions techniques voire architecturales du produit (sans pour autant avoir essayé de les développer)