Haute école spécialisée bernoise
Haute école technique et informatique
Section microtechnique
Laboratoire de l'informatique technique

Programmation proche du hardware avec le kit de développement pour microcontrôleur C8051F

Ce manuscrit traite l'introduction de la programmation proche du hardware en C. Ce cours est basé sur le kit de développement pour microcontrôleur de C8051F, fourni par « Silicon Laboratories ». Des bonnes connaissances dans la programmation en C sont nécessaires pour la poursuite de ce cours.

Nom du fichier: ProgrammationProchHW.doc
Définition le 16. mars 2007
Définie par: Elham Firouzi
Version 1

Table des matières

1 Les microcontrôleurs 4
1.1 Les différents types d'ordinateur 4
1.1.1 Les ordinateurs universels (PC) 4
1.1.2 Les super ordinateurs 4
1.1.3 Les micro ordinateurs 4
1.1.4 Ordinateur temps réels 4
1.2 Le processus technique 5
1.3 Transfert de données entre l'ordinateur temps réels et le processus technique 6
2 L'architecture des microprocesseurs 7
2.1 Structure d'un système à microprocesseur 7
2.2 Le Processeur 8
2.3 Le système de bus 9
2.3.1 Les bus d'adresse, de données et de contrôle 9
2.4 Domaine d'adressage 11
2.4.1 Le décodeur d'adresse 12
2.5 La mémoire 13
2.5.1 Les technologies 13
2.5.2 L'organisation de la mémoire 14
2.5.3 Les modèle de stockage et les formats des données 14
2.5.4 Little Endian / Big Endian 15
2.6 La périphérie 16
2.6.1 Les registres 16
2.6.2 Les amplificateurs de sortie 17
2.6.3 Les entrées et sorties digitales 18
2.6.4 L'interface sérielle 18
2.6.5 Le port sériels RS232 19
2.6.6 SPI 21
2.7 Les convertisseurs A/D 22
3 La famille des microcontrôleurs C8051F2xx 24
3.1 Le cœur du C8051F2xx 25
3.2 La mémoire du C08051F2xx 26
3.3 L'interface JTAG 27
3.4 Les entrées et le sorties du C08051F2xx 28
3.5 Les ports sériels du C08051F2xx 29
3.6 Le convertisseur analogique - digitale du C08051F2xx 29
4 La programmation proche du hardware 30
4.1 L'adressage direct 30
4.2 L'adressage indirect 31
4.3 Mise à un et mise à zéro des bits dans les registres 32
4.4 Désactivation des optimisations du compilateur 32
4.4.1 Exemple 32
4.5 Déviation des fonctions d'entrée et de sortie 34
4.6 Adaptation supplémentaire de la bibliothèque standard 35
5 Les interruptions 36
5.1 Définition d'une routine d'interruption 36
5.2 Blocage des interruptions 37
5.3 Traitement des interruptions avec le contrôleur C8051F 39
5.3.1 Les vecteurs d'interruption 39

6 Le code de démarrage 41

1 Les microcontrôleurs

1.1 Les différents types d'ordinateur

1.1.1 Les ordinateurs universels (PC)

Les ordinateurs universels sont utilisés pour des applications de tous les jours. Ces derniers sont standardisés et sont produits en très grande quantité. Il existe des programmes très variés pour ce type d'ordinateur (traitement de texte, tabulateur, environnement de développement, jeux etc.).

1.1.2 Les super ordinateurs

Les super ordinateurs sont utilisés pour des travaux qui exigent des puissances de calcul élevées (météo, modélisation des processus physiques etc.). Ces derniers sont produits en très petite quantité et ne disposent pas de programmes déjà existants. Ces ordinateurs, ainsi que les programmes qui doivent y être exécutés, sont produits pour des applications très spécifiques.

1.1.3 Les micro ordinateurs

Les micro-ordinateurs sont des petits ordinateurs, qui sont utilisés pour des applications de contrôle. Ces derniers possèdent souvent des entrés et des sorties supplémentaires, afin de pouvoir réaliser ces opérations de contrôle. Ces ordinateurs sont fournis sous forme de carte électronique ou de module (avec ou sans boîtiers industriels). Les composants, qui ne sont pas nécessaires, comme par exemple la carte graphique, le clavier, le disque dure etc., ne sont pas disponibles. Toutefois, certain système permettent d'ajouter ces derniers sous forme d'option. Le principe ici est la simplicité avant tout.

1.1.4 Ordinateur temps réels

Les ordinateurs temps réels sont utilisés pour piloter des processus techniques. Ces ordinateurs doivent respecter les critères suivants :

- Maintient de l'ordre temporel
- Temps de réaction et de traitement rapide et prédéfinie
- Comportement temporel reproductible et prédéfinie

Les catégories d'ordinateur dépendent de la complexité du processus technique:

- Super ordinateur
- PC / SPS
- Microcontrôleur

Les micros ordinateurs peuvent être répartis dans deux catégories :

- Single Chip µC (intégration sur un seul chip)
- Single Board µC (circuit intégré avec des éléments périphériques)

Contrairement aux ordinateurs temps réels, les ordinateurs standard (comme par exemple PC avec traitement de texte Microsoft) ne remplissent pas les critères temps réels définis ci dessus. Cela résulte du système d'exploitation.

Ce cours traite uniquement les micro-ordinateurs, qui sont utilisés dans les applications suivantes :

- Télécommunication (centrale, terminaux...)

- Technique médicinale (mesure du sucre, appareil auditif etc.)
- Sécurité (équipement d'alerte incendie, traverse de chemin de fer etc.)
- Automatisation industrielle (senseur, actuateur, équipement de pilotage etc.)
- Industrie automobile, chemin de fer, avion
- Electronique de consommation (appareil photo, imprimante, hi fi etc.)

Les micro-ordinateurs, qui sont utilisés dans les applications de ci-dessus, sont souvent qualifiés de contrôleurs embarqués. En effet, ces derniers sont « embarqués » dans un équipement.

1.2 Le processus technique

La figure suivante illustre l'interface entre l'ordinateur temps réel et le processus technique.

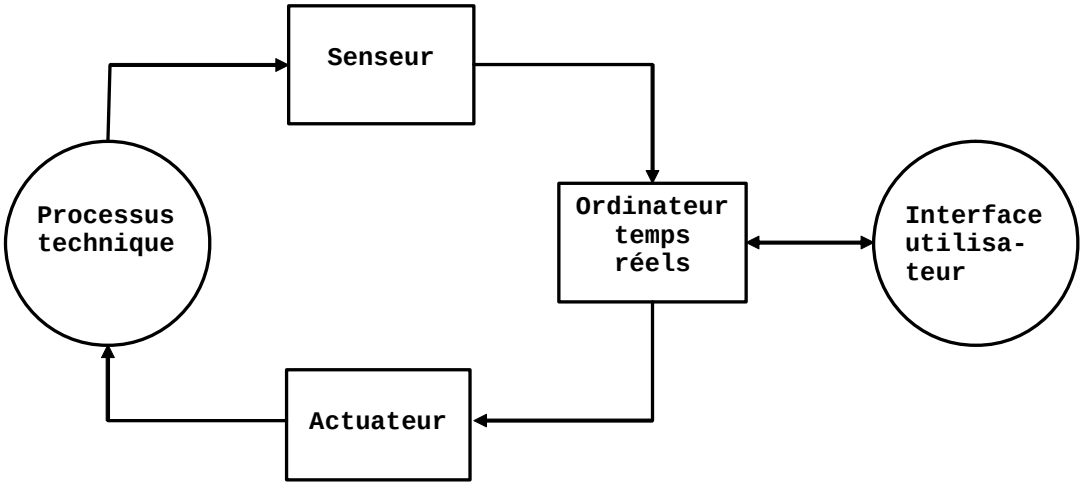

Figure 1 : Interface entre l'ordinateur temps réel et le processus technique

Le processus technique est la source et la destination des données techniques. L'ordinateur lit l'état du processus à l'aide des senseurs et il influence cet état à l'aide des actuateurs. L'utilisateur a la possibilité d'influencer le processus technique à l'aide de l'interface utilisateur (configuration, paramétrisation). Il peut également se renseigner sur l'état du processus technique à l'aide de cette interface.

Exemple de senseur :

- Interrupteur mécanique, inductive ou optique
- Convertisseur A/D
- Sonde de température, sonde de pression

Exemple d'actuateur :

- Soupape
- Moteur
- Ecran, LED
- Relais
- Convertisseur D/A

1.3 Transfert de données entre l'ordinateur temps réels et le processus technique

La figure suivante montre en détail les transferts de données entre l'ordinateur temps réels et le processus technique.

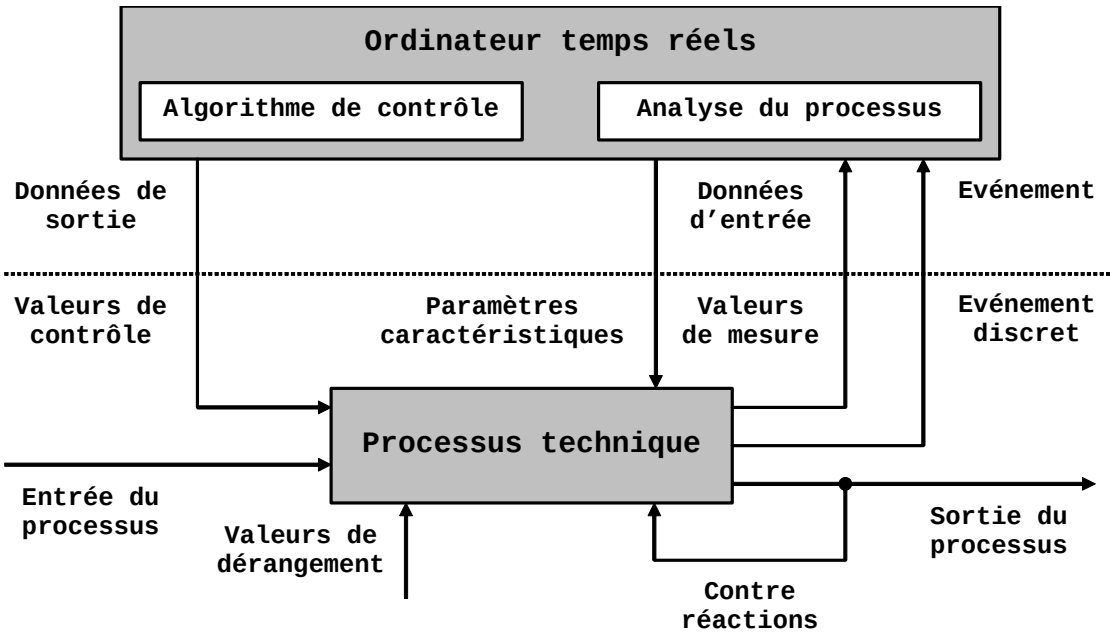

Figure 2 : Transfert de données entre un ordinateur temps réels et le processus technique

Le processus technique possède des entrées et des sorties, qui peuvent être soit de l'information ou du matériel. Les contre réactions ou les valeurs de dérangement influencent également le processus technique. Des paramètres caractéristiques peuvent être transmis au processus technique. L'ordinateur temps réels analyse le processus technique en lisant les valeurs de mesure à des instants précis, qui sont définis par les occurrences des événements. Les algorithmes de contrôle permettent alors de définir des valeurs de contrôle afin d'assurer le fonctionnement correct du processus technique.

2 L'architecture des microprocesseurs

2.1 Structure d'un système à microprocesseur

Un système à microprocesseur comprend un processeur (Central Processing Unit : CPU), une mémoire programme pour le stockage des instructions à exécuter (Read Only Memory : ROM), une mémoire des données pour le stockage des variables (Random Access Memory : RAM) et les éléments périphériques (voir Figure 3).

Le processeur accède à ces composants à l'aide d'un système de bus unique. Les instructions et les données doivent donc être chargées de façon séquentielle. C'est-à-dire qu'il faut au moins deux cycles d'horloge pour lire une instruction et les données à traiter.

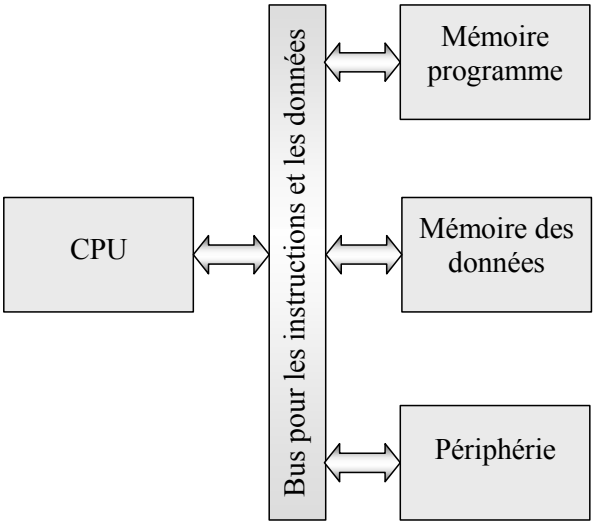

Figure 3 : Architecture de Von Neumann

La mémoire programme contient les instructions du programme. Cette dernière est en générale non volatile, c'est à dire que son contenu n'est pas perdu lorsque l'alimentation est coupée. Quant à la mémoire des données, elle contient les variables du programme et est en générale volatile.

Le CPU lit les instructions à partir de la mémoire programme et, en fonction de ces dernières, traite les variables (lecture et écriture), qui sont stockées dans la mémoire des données ou les composants périphériques.

2.2 Le Processeur

Le Processeur (CPU) est le cœur de l'ordinateur. Il contrôle le déroulement du programme et traite les données. La Figure 4 illustre la structure générale du CPU. Ce dernier peut varier en fonction du type et du fabriquant de l'ordinateur.

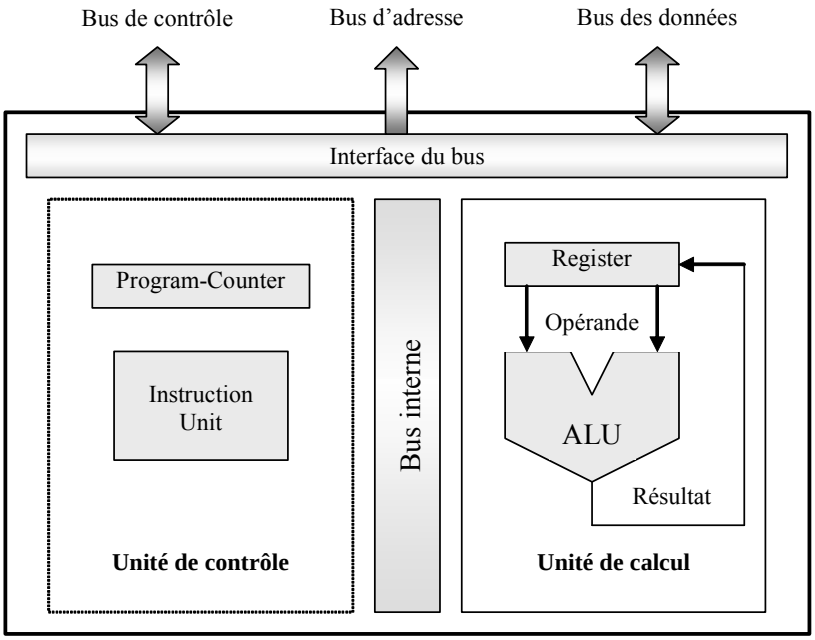

Figure 4 : Structure du CPU

L'**unité de contrôle** est responsable de l'exécution du programme. Il est composé des éléments suivants :

- Instruction Unit (dispositif de commande) : Il interprète les instructions et est responsable de leurs exécutions.
- Program-Counter (**PC**): Il contient l'adresse de l'instruction suivante à exécuter, qui est stockée dans la mémoire programme.

L'**unité de calcul** est responsable du traitement des données. Il contient les composants suivants :

- **ALU** (Arithmetic Logical Unit), qui exécute les opérations arithmétiques et logiques. L'ALU n'exécute que des opérations avec des nombres entiers. Pour des instructions à virgule flottante ou des instructions mathématiques plus complexes on emplois souvent un FPU (Floating Point Unit).
- Les **registres** de données, qui sont destinés aux stockages des opérandes et des résultats des opérations (accumulateur). Un registre est une case mémoire rapide, qui se trouve à l'intérieure du processeur. Ce dernier peut être accédé directement, c'est à dire sans système de bus, par le CPU.

Le **bus interne** relie les unités de contrôle, de calcul et l'interface du bus. Les opérations, qui doivent être exécutés dans l'unité de calcul, sont définies par l'unité de contrôle. L'unité de calcul livre à son tour des informations sur son état actuel à l'unité de contrôle. L'interface du bus est responsable du contrôle des unités externes (mémoire et périphérie).

2.3 Le système de bus

2.3.1 Les bus d'adresse, de données et de contrôle

Différents bus relient le CPU aux composants externes. Ces bus permettent d'échanger des informations concernant l'adresse et les données des variables.

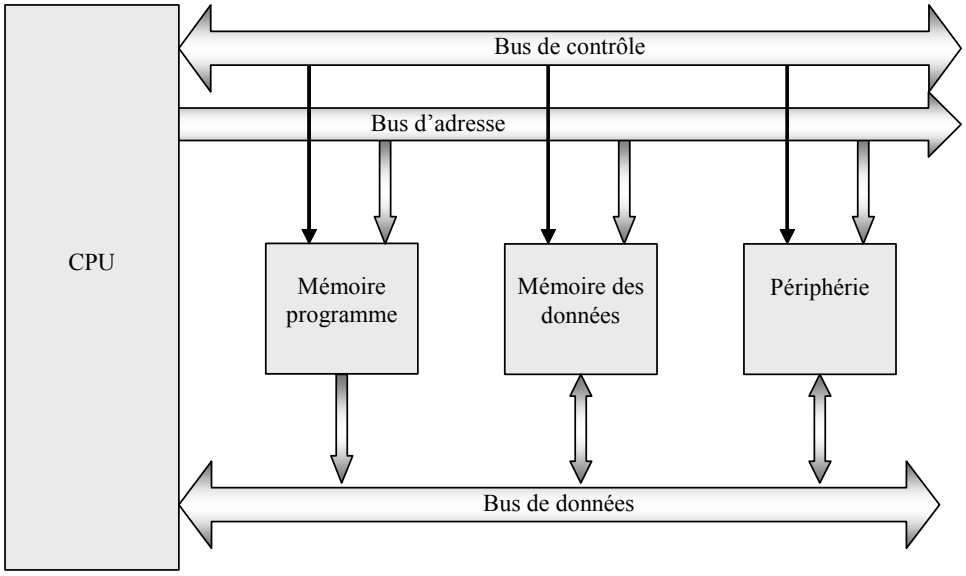

Figure 5 : Bus d'adresse, de données et de contrôle

Le **bus d'adresse** permet de transmettre les adresses (pour la sélection des cases mémoires) aux composants externes (mémoire et périphérie), qui sont connectés à le CPU. Ces adresses sont déterminées par l'unité de contrôle de le CPU. **Avec qui?**

Le **bus de données** permet de transmettre les instructions et les données du programme. Ces dernières sont **quoi?**

a) lues par le CPU à partir des composants de stockage externes (RAM ou ROM) ou périphériques.
b) générées par le CPU et écrites dans les composants de stockage externes ou périphériques.

Le **bus de contrôle** permet de transmettre des informations supplémentaires pour la gestion de la communication (read/write, reset, interrupts, requests et acknowledge, handshake etc.). **comment?**

Exemple :
Le fonctionnent du système de bus peut être illustré plus en détail, à l'aide d'un transfert de données depuis la périphérie, en passant par le CPU, à la mémoire des données. Exemple : lecture de la température à l'aide d'une sonde x (périphéries) et stockage de cette dernière dans la variable "`temp`", qui se situe dans le RAM. En C cella serait définie de la manière suivante :

```
int temp = Temperatur_Sensor_x;
```

1) Le CPU dépose l'adresse de l'instruction à lire sur le bus d'adresse. Cette adresse est stockée dans le compteur de programme (Program counter).

2) Le CPU fixe le sens du transfert « lecture », à l'aide du bus de contrôle.
3) La mémoire programme fournit l'instruction stockée dans la case mémoire, qui est sélectionnée par le bus d'adresse, sur le bus de donnée.
4) Le CPU lit cette instruction.
5) Le CPU fournit l'adresse du composant périphérique, à partir de laquelle les données doivent être lues, sur le bus d'adresse.
6) Le CPU fixe le sens du transfert « lecture », à l'aide du bus de contrôle.
7) La périphérie fournit l'information, contenue dans la case mémoire sélectionnée, sur le bus de données.
8) Le CPU lit les données.
9) Le CPU sélectionne la case de destination « `temp` » dans la mémoire de donnée, à l'aide du bus d'adresse.
10) Le CPU fournit les données sur le bus de données.
11) Le CPU fixe le sens du transfert « écriture », à l'aide du bus de contrôle.
12) La mémoire des données saisit les données sur le bus de données, et les stocke dans la case mémoire sélectionnée par le bus d'adresse.

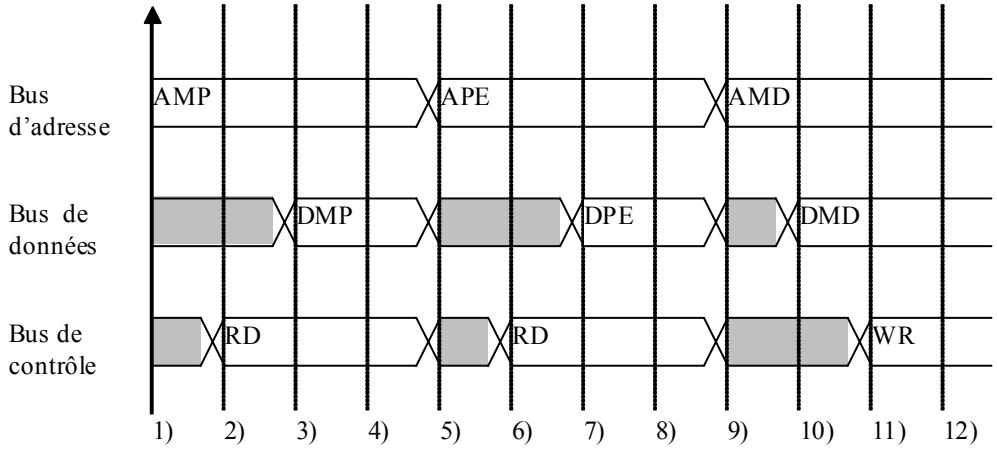

Figure 6 : Le timing du bus

Les abréviations suivantes ont été utilisées dans la Figure précédente :

AMP : Adresse de la mémoire programme
APE : Adresse de la périphérie
AMD : Adresse de la mémoire des données
DMP : Donnée de la mémoire programme
DPE : Donnée de la périphérie
DMD : Donnée de la mémoire des données
RD : Read, lire
WR : Write, écrire

De plus les notations suivantes illustrent l'état de la tension du bus :

La tension du bus est stable et valable, elle est soit high ou low

La tension du bus n'est pas valable ("don't care")

La tension du bus change

2.4 Domaine d'adressage

La mémoire d'un processeur est composée d'un certain nombre de cases mémoire, dont la taille correspond à 1 byte (8 bits). Toutes ces cases mémoire possèdent une adresse distincte, qui s'étend de 0 jusqu'à la taille de la mémoire -1.

L'adresse est transmise de façon binaire à l'aide du bus d'adresse. La largeur du bus d'adresse limite par conséquence la zone d'adressage. Par exemple, avec 16 bits, il n'est possible de générer que des valeurs allant de 0 à 2^{16} - 1. Ce qui correspond à une zone d'adressage de 64 kilo bytes (en informatique 1 kilo représente 1024). Avec 24 bits cette limite atteint 16M bytes (1 Méga correspond à 1024 kilo). Alors qu'avec 32 bit, il est possible d'adresser 4G bytes (1 Giga correspond à 1024 Méga). Toutefois, toutes les cases mémoires adressables ne sont pas forcément disponibles, car la zone d'adressage peut contenir des lacunes, ou il n'y a rien.

L'attribution effective des zones d'adressage à la RAM, à la ROM et aux composants périphériques est fixée par le circuit électronique. Un plan de mémoire permet d'illustrer ces zones mémoires. Un système microprocesseur contient généralement une ROM (composant, qui permet de stocker les instructions du programme), une RAM (composant, qui permet de stocker les données et les variables) et des éléments périphériques (composants, qui permettent de communiquer avec le monde extérieur). Le système peut également contenir des zones d'adressage vides, qui sont prévues pour les éventuelles extensions.

Exemple d'un plan de mémoire

0xFFFFFF **0xF80000**	**ROM**
0xF7FFFF **0xF00000**	**Non utilisé**
0xEFFFFF **0xEFFFE0**	**Timer**
0xEFFFDF **0xEFFF80**	**UART**
0xEFFF7F **0x028000**	**Non utilisé**
0x027FFF **0x020000**	**RAM** **(Alimenté par accu)**
0x01FFFF **0x000400**	**RAM**
0x0003FF **0x000000**	**RAM** **(Table des vecteur)**

Figure 7: Plan de mémoire

La représentation du plan de mémoire s'effectue généralement à partir du bas, en commençant par l'adresse 0. Mais le plan de mémoire peut être également représenté dans l'autre sens.

2.4.1 Le décodeur d'adresse

La tâche du décodeur d'adresse est de sélectionner un des composants externes. Admettons par exemple qu'un système à microcontrôleur soit composé de plusieurs composants de stockage (RAM et ROM). Dans ce système, du bus d'adresse ne suffit pas pour sélectionner un des composants. Par conséquent, il faut utiliser des signaux de sélection (Chip Select, CS) supplémentaires. Ces signaux sont fournis par le décodeur d'adresse en fonction des bits de poids plus fort du bus d'adresse.

Les décodages d'adresse simples sont réalisés avec de la logique discrète (AND, OR ou 1 of X decoder). Les composants nécessaires pour réaliser ce genre de décodeur ne sont pas très chers. Toutefois, la logique discrète nécessite plus de place sur la platine et elle n'est pas flexible. Par conséquent, les erreurs de conception ou les éventuelles adaptations du plan de mémoire nécessitent souvent une nouvelle platine.

A cause des problèmes de la logique discrète décrits ci-dessus, les décodeurs d'adresse sont souvent réalisés à l'aide de la logique programmable (GAL, PAL, CPLD etc.). Le décodeur d'adresse correspondant au plan de mémoire de la **Fehler! Verweisquelle konnte nicht gefunden werden.** peut être réalisé de la manière suivante :

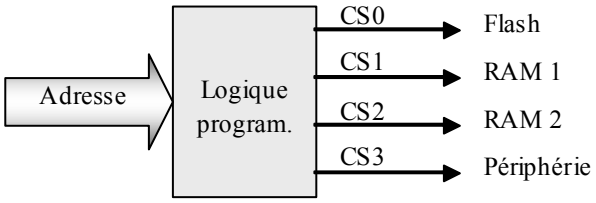

Figure 8 : Décodeur d'adresse programmable

2.5 La mémoire

2.5.1 Les technologies

Fondamentalement il existe les mémoires non volatiles, qui sont utilisée uniquement en lecture, et les mémoires volatiles, qui peuvent être utilisées en lecture ou en écriture.

Mémoires non volatiles

Ces mémoires conservent leur contenu, même si leur alimentation est coupée. Les différents types de mémoire non volatile sont les suivants :

- ROM (Read Only Memory) : Programmation par le fabriquant à l'aide de masques.
- EPROM (Erasable Programmable ROM) : Ces mémoires peuvent être programmées plusieurs fois par l'utilisateur (outils de programmation spécifique) et effacés à l'aide de rayon UV.
- OTP (ou OTP ROM, One Time Programmable ROM) : Ces mémoires sont construites comme les EPROM, toutefois sans fenêtre. Les OTP ne peuvent être programmées qu'une seule fois par l'utilisateur.
- EEPROM (Electrical Erasable Programmable ROM) : Ces mémoires peuvent être effacées et reprogrammées case par case durant leur fonctionnement. Les EEPROM ont une capacité réduite (quelques kilos bytes) et sont, par conséquent, utilisées pour stocker les données de production, comme par exemple les numéros de série etc.
- FLASH (la définition exacte est FLASH EEPROM) : Ces mémoires peuvent être effacés électriquement par bloque durant leur fonctionnement (pas de possibilité d'effacer uniquement des cases, comme avec les EEPROM). Les FLASH possèdent une grande capacité de stockage (quelques Mégas bytes) et sont, par conséquent, souvent utilisées pour stocker le code du programme.

Mémoire volatile

Ces mémoires perdent leur contenu dès que leur alimentation est coupée. Ces mémoires sont en générale qualifiées de RAM (Random Access Memory). Random veut dire que les cases mémoires peuvent être accédées de façon aléatoire. Il existe les catégories suivantes :

- DRAM (Dynamic RAM) : Ces mémoires stockent l'information à l'aide de condensateur et doivent, par conséquent, être rafraîchies périodiquement.
- SRAM (Static RAM) : Ces mémoires possèdent une structure plus complexe que les mémoires DRAM mais, par contre, elles ne doivent pas être rafraîchies périodiquement. Les SRAM sont plus rapide, nécessitent moins de courant et sont plus cher que les DRAM.
- SDRAM (Synchronous Dynamic RAM) : Ces mémoires sont une version cadencée des DRAM. La fréquence d'horloge est prédéfinie par le bus du système. Chez les DDR-SDRAM (Double Data Rate SDRAM), les accès mémoires sont possibles aux flans montant et aux flans descendants de l'horloge.

Aujourd'hui il est également possible de travailler avec des technologies RAM non volatile (FeRAM, MRAM). Toutefois, ces dernières possèdent une faible capacité de stockage et sont relativement chères.

2.5.2 L'organisation de la mémoire

Les mémoires sont organisées comme un Tableau, qui contient des cases mémoires. La largeur de ces cases dépend du type de la mémoire et correspond à 1, 8 ou 16 bits.

La Figure suivante montre la structure d'une mémoire organisée en byte :

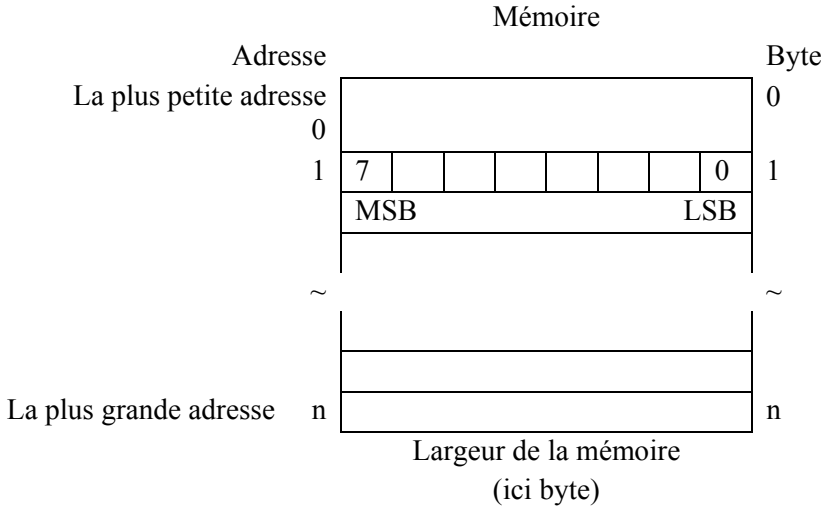

Figure 9 : Structure d'une mémoire organisée en byte

Le nombre de bit d'adresses, qui sont utilisés pour adresser la mémoire, dépend de la taille de la mémoire. Avec m bits d'adresse, il est possible d'adresser 2^m cases mémoires. Par exemple avec 16 bits d'adresses, il est possible d'adresser un do maire de 64 k (ce qui est souvent utilisé avec de microcontrôleur à 8 bits).

2.5.3 Les modèle de stockage et les formats des données

La mémoire du microprocesseur est organisée de façon bytes. Ce qui veut dire, que les variables à 16 ou 32 bits doivent être stockées à l'aide de plusieurs bytes. Il est recommandé de stocker les valeurs 16 bits à des adresses paires et des valeurs 32 bits à des adresses multiples de 4 (cela est même une obligation avec certain processeur). Ce qui permet d'effectuer les opérations de lecture et d'écriture en une seule étape avec des bus de données de 16 ou de 32 bits.

2.5.4 Little Endian / Big Endian

Il existe deux formats de stockages des valeurs, qui sont composées de plusieurs bytes. Avec le format petit boutiste (en anglais little endian), c'est toujours le byte de poids plus faible qui est stocké en premier. Avec le format gros boutiste (en anglais big endian) c'est le contraire. Les figures suivantes illustres ces deux formats de stockage avec les processeurs 8, 16 et 32 bits :

Microprocesseur 32 bits

Little Endian

Valeur 32 bits (long)			
Bit 0-7	Bit 8-15	Bit 16-23	Bit 24-32

Big Endian

Valeur 32 bits (long)			
Bit 24-32	Bit 16-23	Bit 8-15	Bit 0-7

Microprocesseur 16 bits

Little Endian

Valeur 32 bits (long)			
Bit 16-23	Bit 24-32	Bit 0-7	Bit 8-15

Big Endian

Valeur 32 bits (long)			
Bit 24-32	Bit 16-23	Bit 8-15	Bit 0-7

Microprocesseur 8 bits

Little Endian

Valeur 32 bits (long)			
Bit 24-32	Bit 16-23	Bit 8-15	Bit 0-7

Big Endian

Valeur 32 bits (long)			
Bit 24-32	Bit 16-23	Bit 8-15	Bit 0-7

Figure 10 : Modèle de stockage et format de données

2.6 La périphérie

Les composants suivant sont considérés comme périphériques :

a) Les composants qui permettent d'accéder aux données de l'environnement du système. Ces données sont lues avec des senseurs ou d'autre type d'interfaces.
b) Les composants qui permettent de transmettre les données à l'environnement du système. Ces données sont transmises avec des actuateurs ou d'autres types d'interface.

La périphérie représente l'interface entre le processeur et son environnement. Les composants périphériques sont en général connectés au bus du processeur.

2.6.1 Les registres

Les registres sont utilisés pour fixer le mode de fonctionnement ou comme mémoire tampon pour les valeurs d'entrée et de sortie des composants périphériques. Ils forment donc l'interface entre le processeur et sa périphérie.

Un registre est un regroupement de cellules de stockage selon une certaine fonctionnalité. Ces derniers sont le plus souvent réalisés avec des flip-flops, ou parfois avec des cellules DRAM.

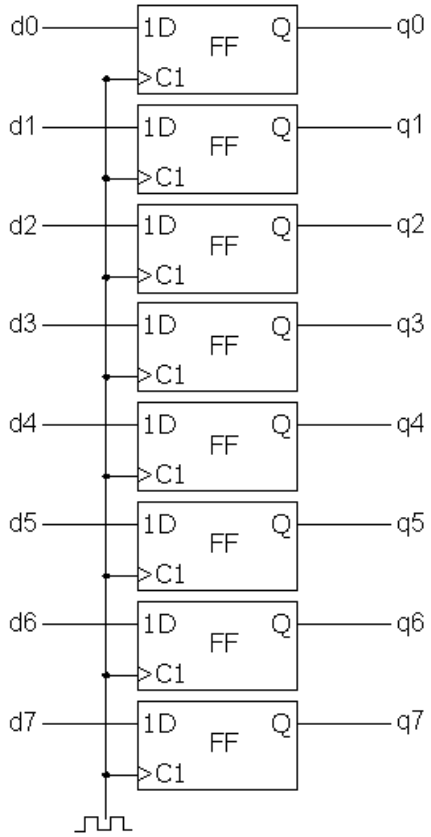

Figure 11 : Registre à 8 bits, réalisé avec des flip-flops

2.6.2 Les amplificateurs de sortie

Les signaux de sortie d'un chip, qui sont fournis à l'aide des pins, doivent être amplifiés. Il existe différents types d'amplificateur de sortie, dont le choix dépend de l'application.

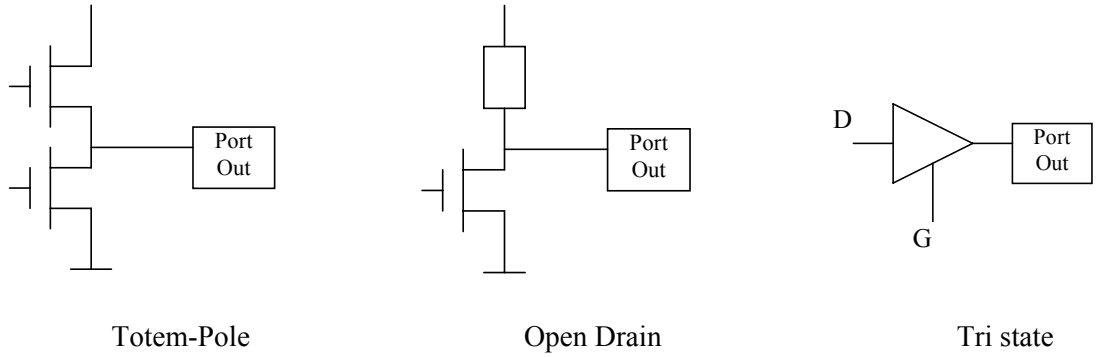

Figure 12 : Amplificateur de sortie

Les étages totem pole

Les étages totem pole fournissent toujours un signal à leur sortie. Par conséquent, ils ne peuvent pas être connectés en parallèle !

a) La logique low est représentée le plus souvent par « 0 »

b) La logique high est représentée le plus souvent par « 1 »

Open Drain

Les sorties Open Drain (avec la technologie FET) ou Open Collector (avec la technologie bipolaire) possèdent un seul transistor, qui permet de tirer une résistance (Pull-up) contre la masse. Les résistances pull-up peuvent être sois internes ou externes. Les sorties Open Drain et Open Collector peuvent être connectées en parallèle, comme par exemple avec les sources d'interruption :

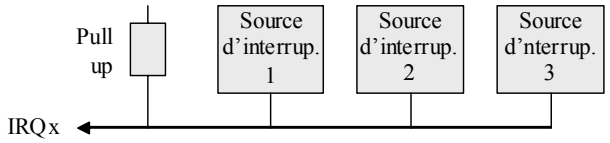

Figure 13 : Source d'interruption avec des sorties Open Drain

Tri state

Les sorties tri state peuvent avoir un état à haute impédance. L'état de sortie active ou « tri state » est activée à l'aide d'un signal de contrôle supplémentaire (G).

En générale les sorties, qui fournissent des signaux sur un bus, sont réalisées avec des tri states, comme par exemple les entrées et sorties des données (I/O) d'une RAM.

2.6.3 Les entrées et sorties digitales

Les entrées et sorties digitales sont utilisées respectivement pour la lecture et l'écriture des signaux digitales. Il existe deux possibilités pour fournir des entrées sorties digitales :

1) Utilisation de pins du microcontrôleur, appelés GPIO (General Purpose Input Output). Le nombre de ces pins varie en fonction de la famille des microcontrôleurs. L'accès à ces derniers s'effectue à l'aide de registres.
2) Utilisation de composants périphériques supplémentaires (FPGA, PIO, …), qui mettent à disposions des I/O. La communication avec ces composants s'effectue avec le système de bus.

Les entrées et sorties digitales sont utilisées pour connecter les éléments d'affichage (lampes, LED etc.), les relais, les moteurs, les soupapes etc.

Les sorties digitales possèdent souvent un étage d'amplification, afin qu'elles puissent fournir le courant nécessaire à la transmission (ex. contrôle des moteurs, des soupapes etc.). Les entrées digitales possèdent souvent des filtrés (circuit RC et trigger de Schmitt). Elles sont également protégées contre les sur tensions (comme par exemple à l'aide de diodes Transzorb et des varistances).

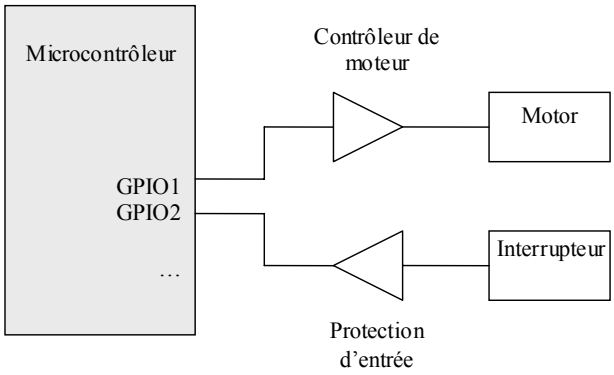

Figure 14 : Exemple d'utilisation de GPIO

2.6.4 L'interface sérielle

Les interfaces sérielles sont utilisées pour transmettre des données de façon sérielle (ex. RS232, I^2C, SPI, FireWire, USB etc.). Un registre à décalage est nécessaire pour convertir les données, depuis le format parallèle en séquence de bits, afin de pouvoir réaliser la transmission.

Les propriétés les plus importantes de la transmission sérielle sont les suivants :

- Le nombre de bit par seconde (baud rate)
- La tension
- Le protocole

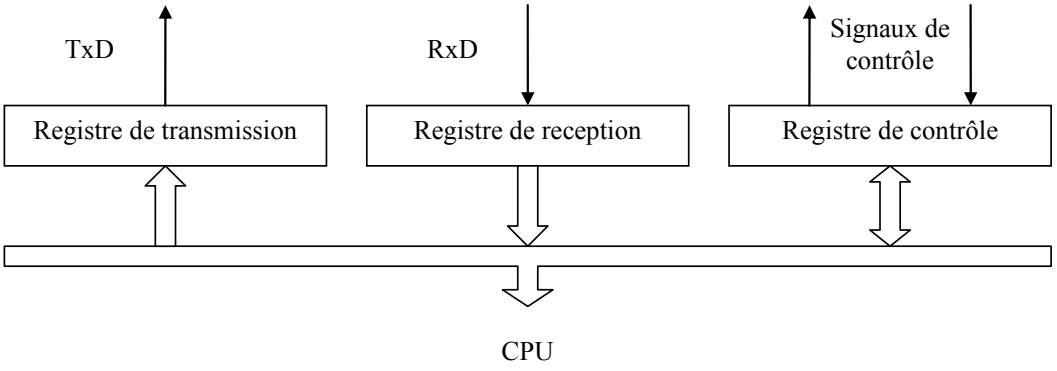

Figure 15 : Principe de fonctionnement d'une interface sérielle

2.6.5 Le port sériels RS232

L'interface RS232 est très importante dans le domaine des systèmes embarqués. Cette interface permet par exemple de gérer un affichage LCD ou une ligne de commande (Command Line Interface). La plupart des microcontrôleurs possèdent déjà une interface RS232. Toutefois, les tensions de sortie de ces dernières correspondent au niveau TTL. Par conséquent, un convertisseur de tension externe doit être ajouté au système.

Les bits de données sont transmis de façon sérielle. Le début de la transmission est marqué par un bit de démarrage, qui a la valeur logique 0. Les bits de données sont transmis à la suite de ce bit de démarrage, en commençant par le bit de poids le plus faible (Least Signifiant Bit ≡ LSB). La fin de la transmission est marquée par un ou deux bits d'arrêt, qui ont la valeur logique 1. Un bit de parité peut être inséré en option entre le dernier bit de donnée et le bit d'arrêt.

Le canal de transmission possède par défaut la valeur logique 1. Ce qui permet de repérer le bit de démarrage.

La norme RS232 requiert pour le niveau logique 0, une tension de transmission de +12 V et pour le niveau logique 1, une tension de transmission de - 12 V.

La diagramme de la tension en fonction du temps est le suivant pour un exemple quelconque :

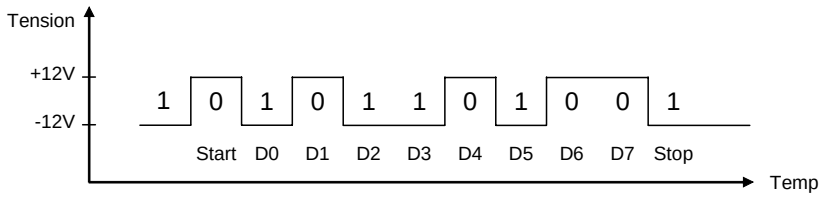

Figure 16 : Transmission de donnée selon RS232

Le schéma bloc de l'interface RS232 est typiquement le suivant :

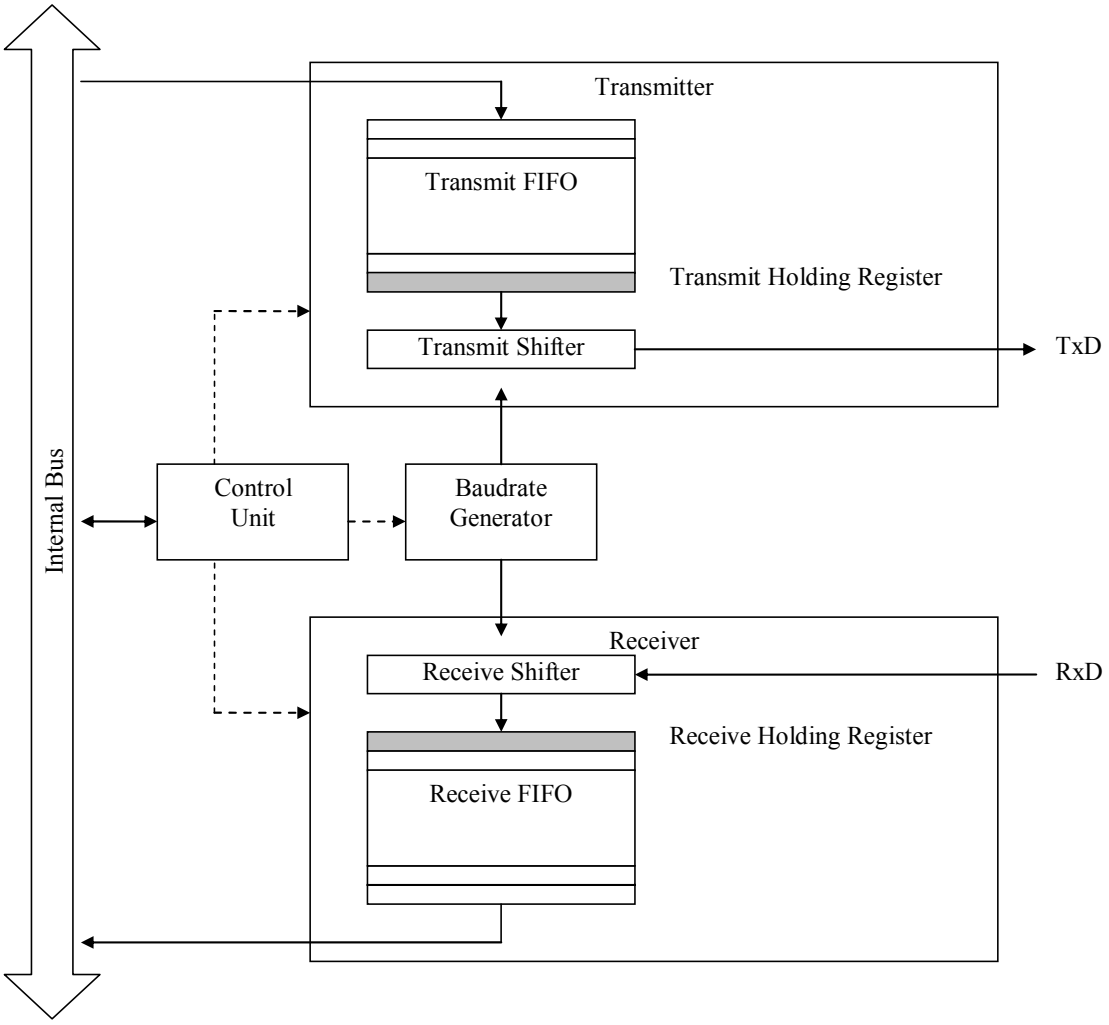

Figure 17 : Schéma bloc d'une interface RS232 commerciale

La connexion entre un émetteur et un récepteur nécessite deux lignes de connexion, une ligne de données et une ligne de masse. Une ligne supplémentaire est nécessaire pour une communication dans les deux sens. Ces connexions sont désignées avec les abréviations suivantes : `GND` pour la masse (pin numéro 5), `TXD` pour la sortie des données (pin numéro 3) et `RXD` pour l'entrée des données (pin numéro 2). Les numéros de pin correspondent à ceux d'un connecteur RS232 à 9 broches (voir figure 17). Il faut toujours transposer `TXD` de l'émetteur avec `RXD` du récepteur.

L'interface sérielle possède également d'autres lignes de commande, qui ne sont pas décrites dans ce document. Ces lignes de commande sont utilisées pour le « handshaking », et permettent de stopper l'émetteur lorsque les données arrivent trop rapidement. Le handshaking peut être désactivé à l'aide du code ou par transposition des lignes de contrôle.

L'émetteur et le récepteur doivent utiliser le même format de transmission afin qu'ils puissent se comprendre. Ce qui veut dire que le nombre de bits par seconde (Baud rate), le nombre de bits de données à transmettre et le mode de parité doivent être configuré de la même manière.

Figure 18: Connecteur à 9 pôles

2.6.6 SPI

L'interface SPI (Serial Peripheral Interface) est un système de bus sériel à haut débit, destiné à la communication entre le microcontrôleur et la périphérie. Ce dernier est souvent utilisé pour la communication avec des extensions I/O, des affichages LCD ainsi que des convertisseurs A/D et D/A. Il peut également être utilisé pour la communication entre microcontrôleurs.

La Figure 19 montre le principe de fonctionnement du SPI.

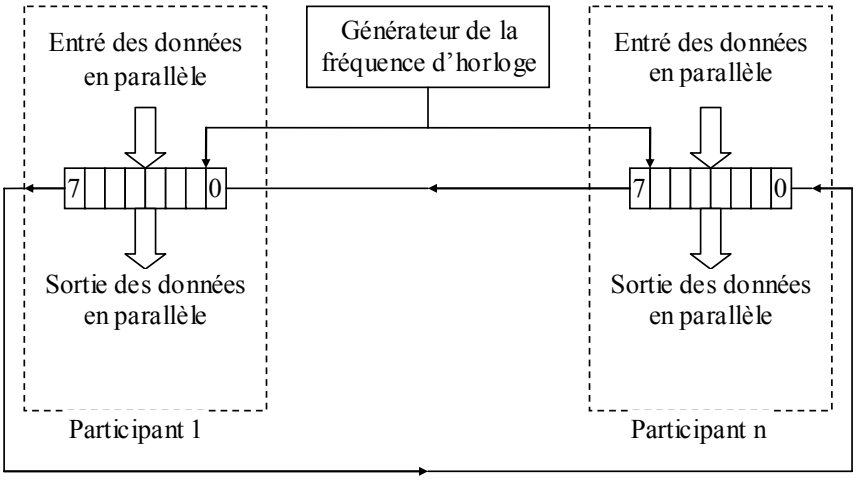

Figure 19 : Principe du SPI

L'interface SPI est toujours utilisée en mode maître esclave. Le maître est alors responsable de la génération de la fréquence d'horloge. Le SPI peut travailler de façon duplexe à l'aide de deux lignes de transmission : MOSI (Master Out Slave In) et MISO (Master In Slave Out). Les esclaves peuvent être connectés soit de façon parallèle (c'est à dire que toutes les sorties des esclaves sont rassemblée et connectées à l'entré MISO du maître) ou de façon sérielle (la sorite d'un esclave est connectée à l'entrée du prochain esclave et la sortie du dernier esclave est connecté à l'entrée MISO du maître).

Le microcontrôleur écrit les données à transmettre dans un tampon de transmission. Ces dernières sont sérialisées à l'aide d'un registre à décalage (comme une transmission RS232). Les données reçues sont également converties à l'aide d'un registre à décalage. Le microcontrôleur peut alors lire ces données de façon parallèle dans le tampon de réception. Du fait que les interfaces SPI ont une bande passante relativement élevé, les tampons de transmission et de réception contiennent souvent plusieurs bytes.

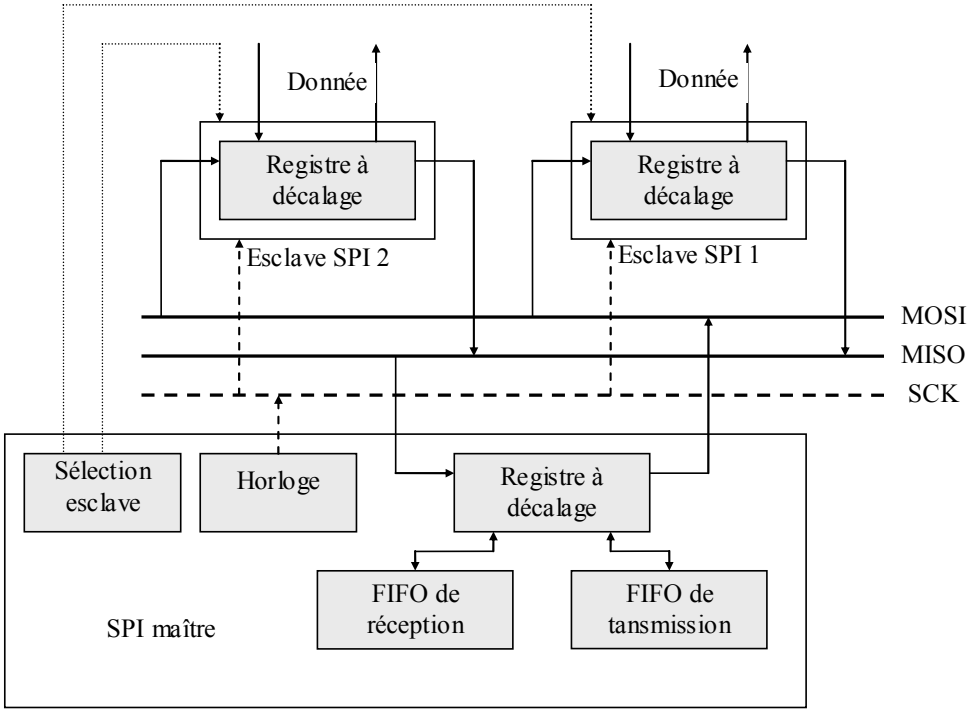

Figure 20 : SPI avec un maître de deux esclaves

2.7 Les convertisseurs A/D

Les convertisseurs A/D mesure une tension analogue à leur entrée, compare cette valeur avec une tension de référence afin de fournir une valeur digitale. Le microcontrôleur peut lire cette valeur soit à l'aide du système de bus ou avec une interface sériel (comme par exemple le SPI).

Les convertisseurs A/D possèdent souvent plusieurs entrées analogues, qui peuvent être sélectionnées à l'aide d'un multiplexeur.

Les caractéristiques les plus importants pour un convertisseur A/D sont les suivants:

- Largeur de bit (les valeurs typiques sont 8, 10, 12 ou 16 bits)
- Temps de conversion

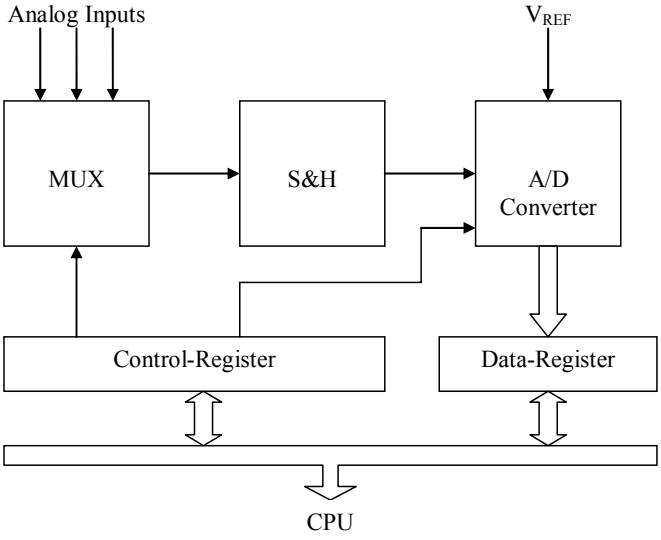

Figure 21 : Schéma bloc d'un convertisseur A/D

3 La famille des microcontrôleurs C8051F2xx

La famille 8051 est la famille la plus connue et la plus répandue des contrôleurs 8 bits. Plusieurs fabricants fournissent des dérivées de ce microcontrôleur, qui se distinguent par leurs modules périphériques. Le 8031 d'INTEL est considéré ici comme étant le précurseur.

Tous les microcontrôleurs de la famille C8051F2xx, qui sont produit par « Silicon Laboratories », possèdent le cœur 8051. Les opérations arithmétiques et logiques y sont effectuées sur 8 bits. Cette famille de microcontrôleur possède également 8 kilo bytes de mémoire flash, 256 ou éventuellement 1280 bytes de RAM et des interfaces sérielles (UART et SPI). Quant aux différents packages, ils sont composés de 22 à 32 pins d'entrée sortie, dont certains peuvent être assignés aux modules périphériques (UART, SPT etc.).

Certaine version de la famille C8051F possèdent également des convertisseurs analogique digitale (CAD). Tous leurs pins d'entrée sortie peuvent être configurés comme entrée analogique pour ces convertisseurs.

	MIPS (Peak)	Flash Memory	RAM	SPI	UART	Timers (16-bit)	Digital Port I/O's	ADC Resolution (bits)	ADC Max Speed (ksps)	ADC Inputs	Voltage Comparators	Package
C8051F206	25	8 k	1280	✓	✓	3	32	12	100	32	2	48TQFP
C8051F220	25	8 k	256	✓	✓	3	32	8	100	32	2	48TQFP
C8051F221	25	8 k	256	✓	✓	3	22	8	100	22	2	32LQFP
C8051F226	25	8 k	1280	✓	✓	3	32	8	100	32	2	48TQFP
C8051F230	25	8 k	256	✓	✓	3	32	—	—	—	2	48TQFP
C8051F231	25	8 k	256	✓	✓	3	22	—	—	—	2	32LQFP
C8051F236	25	8 k	1280	✓	✓	3	32	—	—	—	2	48TQFP

Figure 22: Les différents processeurs de la famille C8051F2xx

3.1 Le cœur du C8051F2xx

Le cœur des microcontrôleurs de la famille C8051F2xx est composé des composants suivants :

1) CPU (8051 Core)
2) 8 kilo bytes de mémoire flash
3) 256 bytes de SRAM plus éventuellement 1024 bytes de XRAM
4) Des interfaces sérielles (UART et SPI)
5) Trois compteurs 16 bits (Timer 0, Timer 1 et Timer2)
6) Un comparateur de tension et un convertisseur AD en option
7) 128 bytes de registres de configuration (Special Function Register : SFR)
8) Quatre ports d'entrée sortie de 8 pins chacun

Un bus interne relie le CPU avec les différents composants de stockage et un bus SFR relie le CPU avec les registres de configuration.

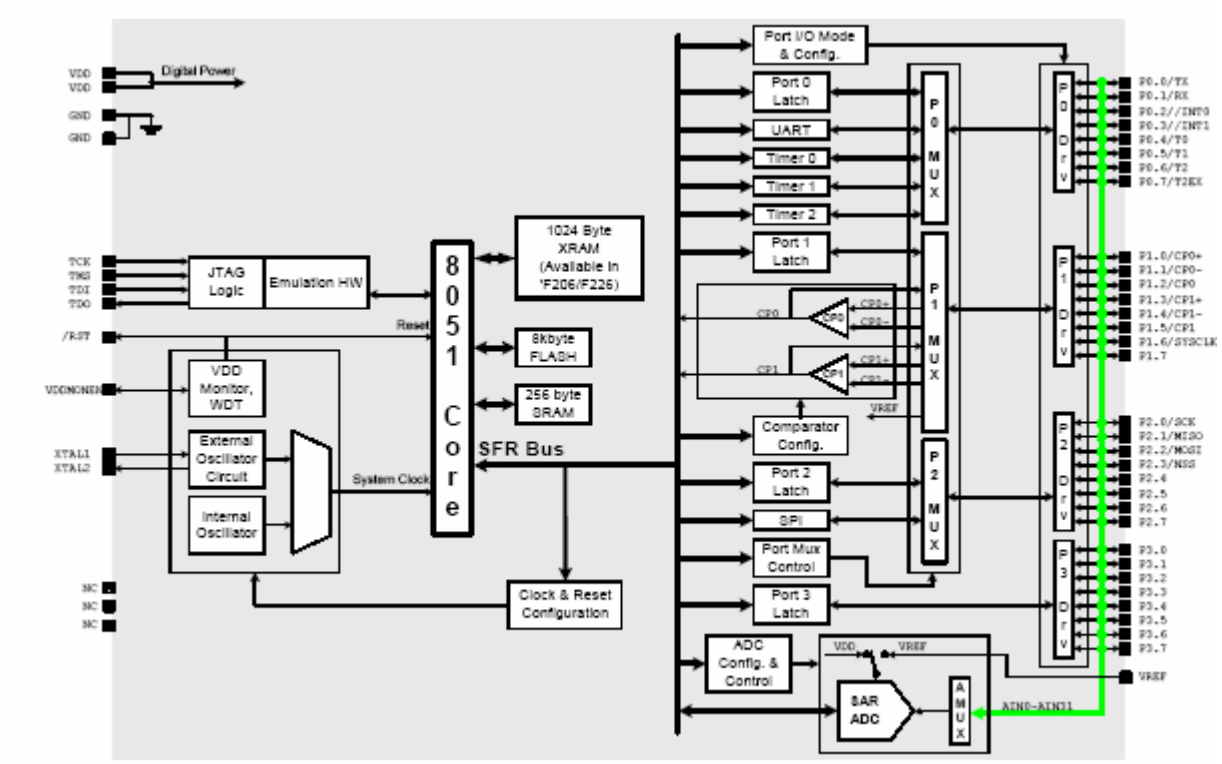

Figure 23 : Schéma bloc du contrôleur C8051F2xx

3.2 La mémoire du C08051F2xx

La mémoire vive du C8051F2xx est composée de 256 bytes. La zone d'adressage supérieure de cette mémoire est partagée entre les registres de configuration (Special Function Register) et la SRAM. Le mode d'adressage permet d'y différencier l'accès. L'adressage direct permet ainsi à accéder aux registres de configuration et l'adressage indirect à la RAM. Les 32 bytes de la zone d'adressage inférieure sont composés de 4 jeux de registres fantômes (banked register) d'utilité générale, et les 16 bytes suivants peuvent être adressé à la fois de façon byte et bit. Une mémoire vive externe de 1024 bytes est également mise à disposition en option avec les familles F206, F226 et F236.

La mémoire programme est composée de deux mémoires flash de 8 kilos et 128 bytes. La zone mémoire entre 0x1E00 et 0x1FFF est réservée pour le fabricant. La figure suivante illustre le plan de mémoire des composants de stockage :

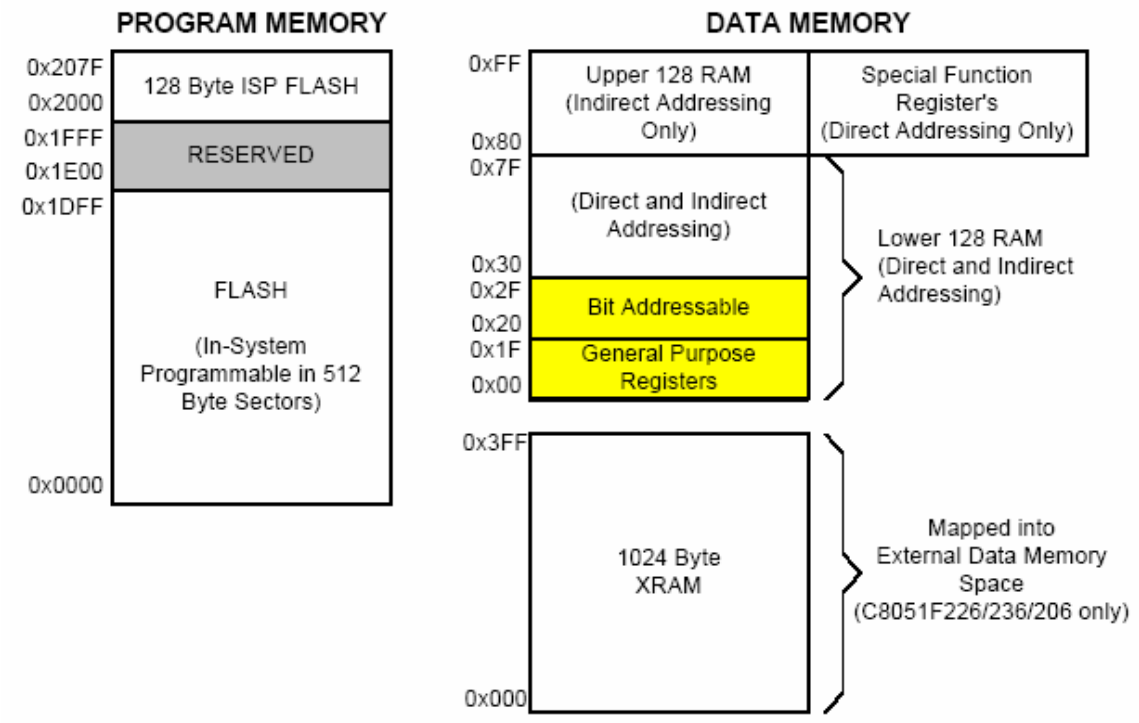

Figure 24 : plan de mémoire des microcontrôleurs de la famille C8051F2xx

3.3 L'interface JTAG

Les contrôleurs de la famille C8051F2xx possèdent une interface JTAG et de la logique de déboguage, qui permettent de développer et tester les applications en temps réel.

Les applications peuvent ainsi être programmées à l'aide d'environnements de développement moderne sur les PC. Ensuite elles peuvent être téléchargées sur les cartes de développement afin d'y être testées en temps réel. Ces testes peuvent être également effectuées pas à pas à l'aide de points d'arrêt. Ces derniers peuvent être insérés à n'importe quel endroit dans le programme et permettent, comme leur nom l'indique, d'arrêter momentanément l'exécution du programme.

La Figure 25 montre la connexion entre le PC et la carte de développent à l'aide d'un adaptateur sériel, dont le rôle est de convertir uniquement le format du signal de déboguage, depuis RS323 à JTAG.

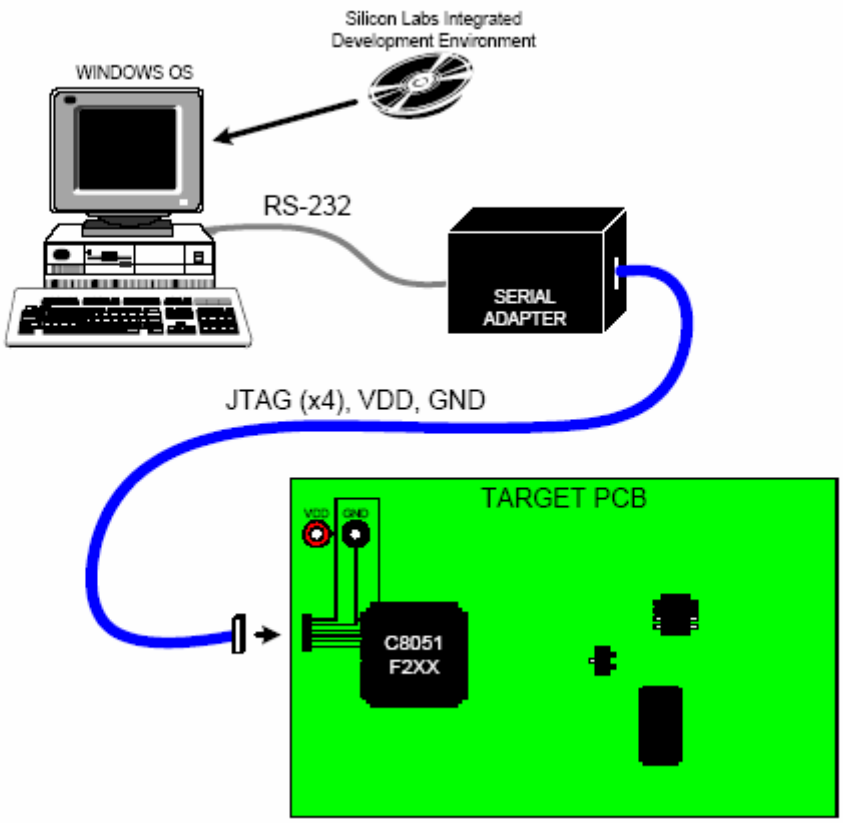

Figure 25 : Environnement de développement de "Silicon Laboratories"

3.4 Les entrées et le sorties du C08051F2xx

Tous les contrôleurs de la famille 8051 possèdent 4 ports standards, contenant chacun 8 pins. L'étage d'amplification de sortie de ces pins peut être configuré soit comme « push pull » ou « open drain ».

Les composants digitaux (le timer, l'horloge du système, le comparateur de tension, le convertisseur AD, les interfaces sériels SPI et UART,) peuvent être connectés à leurs pins à l'aide de multiplexeurs. Ces multiplexeur peuvent être configuré à l'aide de registres de configuration (Special Function Register : SFR).

Un multiplexeur (en abrégé: MUX) est un circuit de commutation, qui permet de sélectionner un signal parmi un certain nombre de signaux (comme par exemple, pour les accès aux cases mémoire ou la sélection analogique ou numérique des canaux d'entrée).

Les modes d'entrée des 32 pins peuvent être configurés soit en analogique ou en numérique (voir Figure 26).

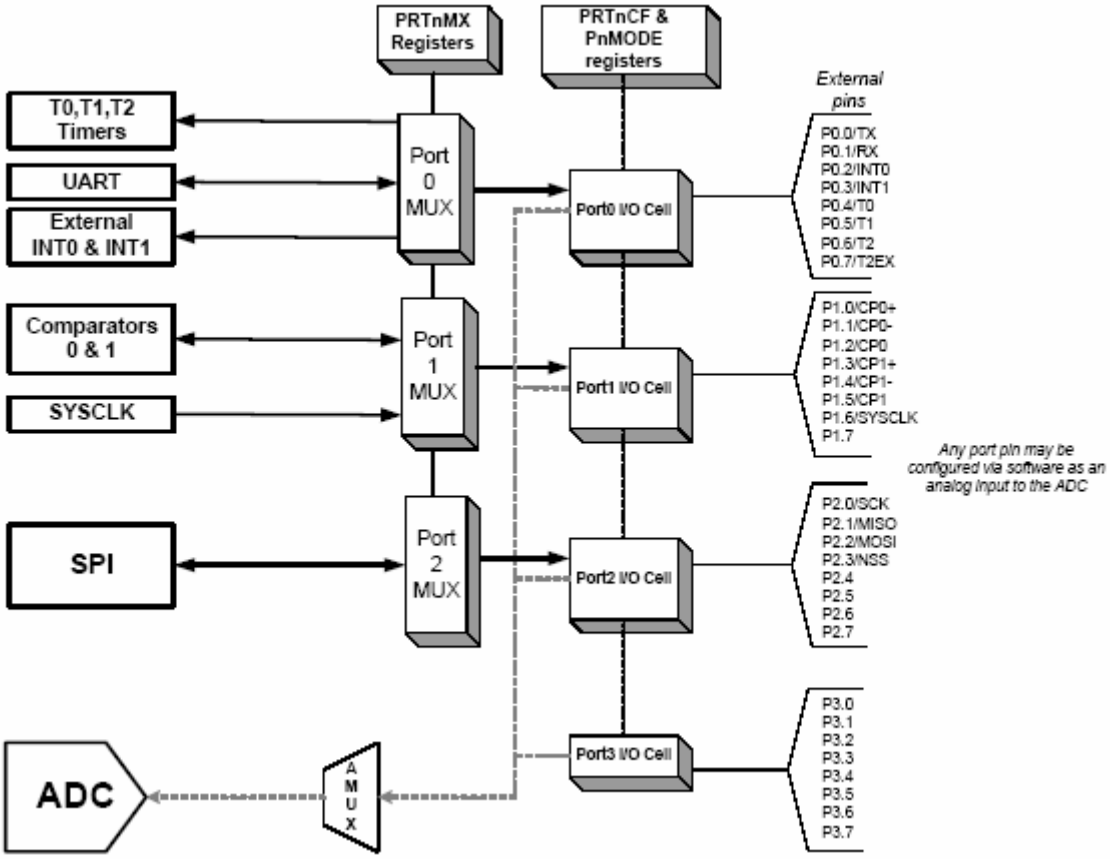

Figure 26 : Schéma bloque des entrées et des sorties

3.5 Les ports sériels du C08051F2xx

Les microcontrôleurs des familles C8051F206, C8051F220/1/6 et C8051F230/1/6 possèdent un UART, capable de transmettre en mode duplexe, et un bus sériel du type SPI. Tous les deux bus sont implémentés en hardware et sont en mesure de générer des interruptions. Ce qui permet de réduire au minimum les interventions de le CPU. De plus, les deux buses ne partagent pas les mêmes ressources hardware, comme par exemple les timers, les interruptions ou les ports d'entrée sortie. Ils peuvent par conséquent être utilisés simultanément (la fréquence d'horloge pour la transmission UART peut être générée soit avec le Timer1, le Timer2 ou le SYSCLK).

3.6 Le convertisseur analogique - digitale du C08051F2xx

Le C8051F220/1/6 possède un convertisseur AD de 8 bis. La fréquence de conversion maximale est de 100 kilo échantillons par seconde. La tension de référence peut soit être la tension d'alimentation ou une référence externe (VREF). Le système peut désactiver le convertisseur afin d'entrer dans un mode d'économie d'énergie. Un amplificateur à gain programmable permet d'amplifier le signal du multiplexeur. Les facteurs d'amplification peuvent être définis de 0.5 à 16 par pas de puissance de 2.

La conversion peut être démarrée de deux façons ; soit par commande software ou avec le Timer2. Ce qui permet de démarrer les conversions soit avec des évènements du type software ou de façon continue. La fin de la conversion peut être marqué soit à l'aide d'une interruption ou avec un bit d'état. Le résultat de la conversion peut alors être lu à partir d'un registre de configuration.

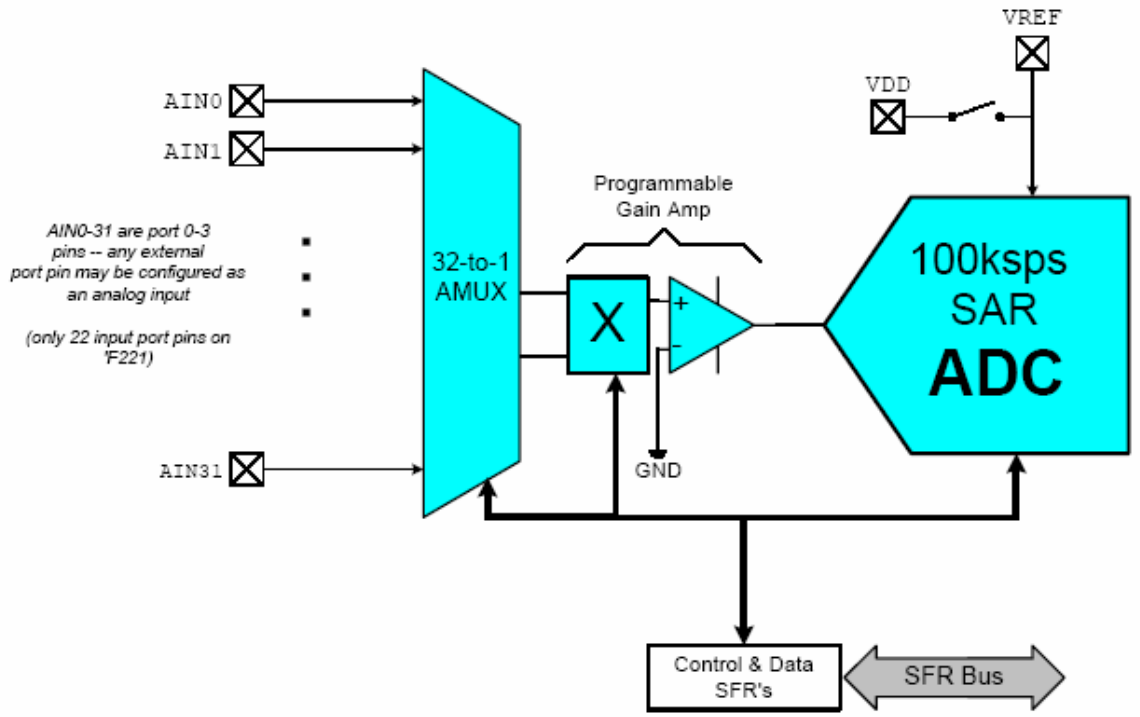

Figure 27 : Le convertisseur AD des microcontrôleurs C8051F220/1/6

4 La programmation proche du hardware

Lorsque que l'on veut définir un programme pour micro contrôleur, il devient alors primordial de pouvoir accéder directement au hardware. En effet, les diverses fonctionnalités des composants hardware ne peuvent être définie qu'avec des registres de configurations. Ces registres sont des cases mémoires, qui possèdent des adresses prédéfinies. C'est-à-dire qu'ils occupent certaines zones du plan de mémoire comme les mémoires RAM et ROM. La fonctionnalité et la signification de ces registres sont fournies par la documentation du microcontrôleur.

La Figure 28 décrit par exemple le registre de configuration « **PRT0MX** », qui contient 8 bits. Ces derniers permettent de définir la fonctionnalité des 8 pins d'entrée sortie du port 0.

R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	Reset Value
T2EXE	T2E	T1E	T0E	INT1E	INT0E	-	UARTEN	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE1

Bit7: T2EXE: T2EX Enable Bit
0: T2EX unavailable at Port pin.
1: T2EX routed to Port Pin.
Bit6: T2E: T2 Enable Bit
0: T2 unavailable at Port pin.
1: T2 routed to Port Pin.
Bit5: T1E: T1 Enable Bit
0: T1 unavailable at Port pin.
1: T1 routed to Port Pin.
Bit4: T0E: T0 Enable Bit
0: T0 unavailable at Port pin.
1: T0 routed to Port Pin.
Bit3: INT1E: /INT1 Enable Bit
0: /INT1 unavailable at Port pin.
1: /INT1 routed to port pin.
Bit2: INT0E: /INT0 Enable Bit
0: /INT0 unavailable at Port pin.
1: /INT0 routed to Port Pin.
Bit1: UNUSED. Read = 0, Write = don't care.
Bit0: UARTEN: UART I/O Enable
0: UART I/O unavailable at port pins.
1: TX, RX routed to pins P0.0 and P0.1, respectively.

Figure 28 : Description du registre de configuration PRT0MX

4.1 L'adressage direct

Dans le mode d'adressage direct, l'accès aux cases mémoire s'effectue en indiquant leur nom symbolique. Les variables du type registre permettent de réaliser ce type d'adressage. Ces dernières sont définies avec des mots clés spécifiques à l'environnement de développement, qui ne font pas partie du standard ANSI-C.

L'exemple suivant montre la définition et l'utilisation d'une variable du type registre dans l'environnement de développement « Silicon Laboratories ».

```
/*************************************************/
/* Definition of the register variable PRT0MX  */
/* with the key word sfr.                       */
/* The initialisation value 0xE1 corresponds   */
/* here to its address within the memory map!  */
/*************************************************/
sfr PRT0MX = 0xE1;

/*************************************************/
/* Direct access to the register variable      */
/* Set the functionality of Port0 to IO        */
/*************************************************/
PRT0MX = 0x00;
```

4.2 L'adressage indirect

Dans le mode d'adressage indirect, l'accès aux cases mémoire s'effectue en indiquant leur adresse. Cette dernière est copiée dans un registre spécial du processeur, dont la taille est identique à celle du bus d'adresses. Les pointeurs permettent de réaliser ce type d'adressage selon le standard ANSI-C.

L'avantage, par rapport à l'adressage direct, est que l'adresse peut être manipulée, comme par exemple pour accéder à une suite de données consécutives en mémoire. Ceci est particulièrement utile lorsqu'on manipule des données stockées dans un tableau.

L'environnement de développement du C8051F2xx contient les types de données suivants : `char` 8 bits, `short int` 16 bits, et `long` 32 bits. Les registres 8 bits peuvent ainsi être adressés avec un pointeur du type `unsigned char`, les registres 16 bits avec un pointeur du type `unsigned short int` et les registres 32 bits avec un pointeur `unsigned long`.

Les exemples suivants permettent d'écrire la valeur **0x12** à l'adresse **0x00F1** (qui pourrait représenter le port parallèle d'un composant externe).

Compact

```
*((unsigned char *) 0x00F1) = 0x12;
/* Convert the value 0x00F1 into an address and accede to it */
```

Via pointeur

```
/* Definition of the pointer variable */
unsigned char *Parallelport;
/* Inisialisation of the pointeur with the address 0x00F1 */
Parallelport = (unsigned char *) 0x00F1;
/* Write the value into the memory case,
   which is addressed by the pointer */
*Parallelport = 0x12;
```

Avec une macro

```
/* Definition of the macro instruction,
   which could be done in a header file */
#define Parallelport *((unsigned char *) 0x00F1)
/* Call the macro */
Parallelport = 0x12;
```

4.3 Mise à un et mise à zéro des bits dans les registres

Les mises à un et les mises à zéro ciblées des bits dans un registre peut être réalisées avec les opérateurs logiques standards (et '**&**', ou '**|**', ou exclusif binaire '**^**' et inverser '**~**'). Ces opérateurs permettent d'effectuer des opérations au niveau des bits.

Dans l'exemple suivant, le bit 4 du registre de configuration **PRT0MX** est d'abord mis à un et ensuite remis à zéro.

```
/***********************************************/
/* Configure pin 0.4 for the timer 0           */
/* Set the bit 4 of PRT0MX                     */
/***********************************************/
PRT0MX = PRT0MX |  0x40;

/***********************************************/
/* Configure pin 0.4 as standard IO            */
/* Clear the bit 4 of PRT0MX                   */
/***********************************************/
PRT0MX = PRT0MX & ~0x40;
```

4.4 Désactivation des optimisations du compilateur

Tous les compilateurs essayent de générer un code, qui est le plus optimal possible. Cette optimisation comprend entre autre l'élimination de code superflu ou redondant. ANSII C met à disposition le mot clé **volatile** afin de désactiver ces optimisations. **volatile** est par conséquent très utile pour l'accès direct hardware ou pour les interruptions.

4.4.1 Exemple

Le code suivant :

```
while (Parallelport == 0) {
   i++;                /* wait until the parallel port is ready */
}
```

peut être optimisé par le compilateur de la manière suivante :

```
if (Parallelport == 0) {        /* test if parallel port is read*/
   while (1) {
      i++;                    /* if yes, then never ending loop */
   }
}
```

Le compilateur optimise le code de ci-dessus afin de réduire de nombre d'accès à la variable **Parallelport**. Ce qui permet d'améliorer considérablement la vitesse d'exécution du code. Cela est tout à fait correct, car du point de vue du compilateur, la variable **Parallelport** reste constante. Les deux codes sont d'ailleurs équivalents, tant que **Parallelport** ne change pas.

Un autre exemple est la séquence :

```
Parallelport = 0;

while (Parallelport == 0) {
   i++;                /* wait until the parallel port is ready */
}
```

Cette dernière sera optimisée en :

```
Parallelport = 0;

while (1) {
   i++;                /* wait until the parallel port is ready */
}
```

Ici le compilateur sait que la variable **Parallelport** possède la valeur 0. Par conséquent, le test avec **if** devient superflu.

Un dernier exemple est la séquence suivante pour la transmission des bits d'information :

```
Parallelport = 0; /* Clear all bits */
Parallelport = 1; /* Set the clock bit */
Parallelport = 0; /* Clear the clock bit */
Parallelport = 2; /* Set the data bit */
Parallelport = 3; /* Set the clock bit */
Parallelport = 2; /* Clear the clock bit */
Parallelport = 0; /* Clear all bits */
```

Le compilateur optimisera cette séquence da la manière suivante :

```
Parallelport = 0; /* Clear all bits */
```

Ici la dernière instruction fixe définitivement la valeur de **Parallelport**. Toute les valeurs précédentes sont sur écrite par cette dernière.

Afin de forcer le compilateur à exécuter toutes les opérations de lecture et d'écriture sans optimisation, il faut définir les variables de façon volatile.

La définition de la variable **Parallelport** doit être complétée avec le mot clé **volatile** dans les exemples de ci-dessus.

Compact

```
*((volatile unsigned char *) 0x00F1 ) = 0x12;
```

Via pointeur

```
/* Definition of the pointer variable */
volatile unsigned char *Parallelport;
/* Inisialisation of the pointeur with the address 0x00F1 */
Parallelport = (volatiale unsigned char *) 0x00F1;
/* Write the value into the memory case,
   which is addressed by the pointer */
*Parallelport = 0x12;
```

Avec une macro

```
/* Definition of the macro instruction,
   which could be done in a header file */
#define Parallelport *((volatile unsigned char *) 0x00F1)
/* Call the macro */
Parallelport = 0x12;
```

Toutes les variables, qui peuvent changer sans que le compilateur le remarque ou dont le changement peut influencer le hardware, doivent toujours être définies de façon volatile. C'est-à-dire, toutes les variables qui ont un rapport avec le hardware, ou qui peuvent être changée dans une routine d'interruption.

Le code peut souvent fonctionner sans le mot clé **volatile**. Toutefois un complément ou une modification du code peuvent engendrer des problèmes, qui seront difficilement localisables.

Les optimisations du compilateur peuvent également être d'autres natures, que celles décrites ci-dessus. Par exemple, le contenu d'une variable peut être stocké dans un registre du processeur tout au début de la fonction. L'accès à ce type de registre est en générale beaucoup plus rapide que l'accès à la mémoire des données. Tous les traitements de la variable s'effectueront alors à l'aide de ce registre. A la fin de la fonction, le contenu du registre sera écrit dans la variable. Le code devient ainsi beaucoup plus rapide. Mais si cette optimisation se fait toujours au détriment de l'accès direct hardware. Par conséquent, le code ne pourrait plus fonctionner correctement.

4.5 Déviation des fonctions d'entrée et de sortie

Les microprocesseurs ne possèdent ni de système de fichiers ni de fonctions d'entrée et de sortie. Les bibliothèques standards permettent néanmoins d'adapter ces fonctions au hardware. Dans le cas le plus simple, il suffit de redéfinir les fonctions **putc()**, **getc()** et éventuellement **ungetc()**. Les autres fonctions de la librairie standard d'entrée et de sortie (**stdio.h**) utilisent ces fonctions.

Si l'on veut utiliser un système de fichier avec plusieurs canaux d'entrée et de sortie, on doit également programmer les fonctions **fopen()**, **fclose()** et éventuellement d'autres fonctions spéciales comme **ftell()** ou **fseek()**.

L'argument pointeur sur fichier (**FILE ***) peut être ignoré dans les fonctions **getc()** et **putc()** lorsque l'on ne travaille pas avec un système de fichier. La déviation des entrées et des sorties sur notre système de développement peut être définie de la manière suivante :

```
int putc (int c, FILE *stream)
{
   SendByteToSerialPort(c);
   return c; /* Should return EOF when an error has occured */
}

int getc (FILE *stream)
{
   /* return next character (or EOF when an error has occurred)*/
   return GetByteFromSerialPort();
}
```

4.6 Adaptation supplémentaire de la bibliothèque standard

La disponibilité de toutes les fonctionnalités de la librairie standard pour un système de développement nécessite des adaptations supplémentaires.

Il faut adapter les fonctions pour la gestion dynamique de la mémoire (**`stdlib.h`**) et pour la gestion du temps (**`time.h`**). Naturellement, seulement en cas de nécessité.

Les fonctions **`malloc()`** et **`free()`** doivent être modifiés pour la gestion dynamique de la mémoire. Toutefois, les options du relieur suffisent dans la plupart des systèmes de développement. Dans ce cas, il suffit de définir le domaine de la mémoire destiné à sa gestion dynamique.

Il est recommandé de consulter la documentation technique de l'environnement de développement pour adapter les fonctions destinées à la gestion du temps (**`time.h`**).

5 Les interruptions

Les interruptions sont des moyens importants pour réaliser des tâches en temps réels. Une interruption est générée par un évènement (souvent externe) et interrompt le programme principal à un instant donnée.

Lorsqu'apparaît une interruption, un sous programme est appelé afin de traiter cette dernière. Dans ce contexte on parle de routine de service d'interruption (Interrupt Service Routine : ISR). La demande d'interruption est appelée requête d'interruption. Lorsque toutes les instructions de la routine de service d'interruption ont été exécutées, le programme principal continue à l'endroit ou il a été interrompu, comme si rien n'a eu lieu.

Exemple d'interruption de la vie de tous les jours

Je suis en train de lire un livre dans le salon. Tout à cas, le téléphone sonne. J'interromps mon activité et je vais de décrocher le téléphone. A la fin de la communication, je raccroche le téléphone et je continue mon activité originale. Il se peut qu'un événement plus important a lieu pendant la communication téléphonique (par exemple quelqu'un pourrait sonner à la porte). En fonction de l'importance de cet événement, je vais également être obligé d'interrompre la communication téléphonique.

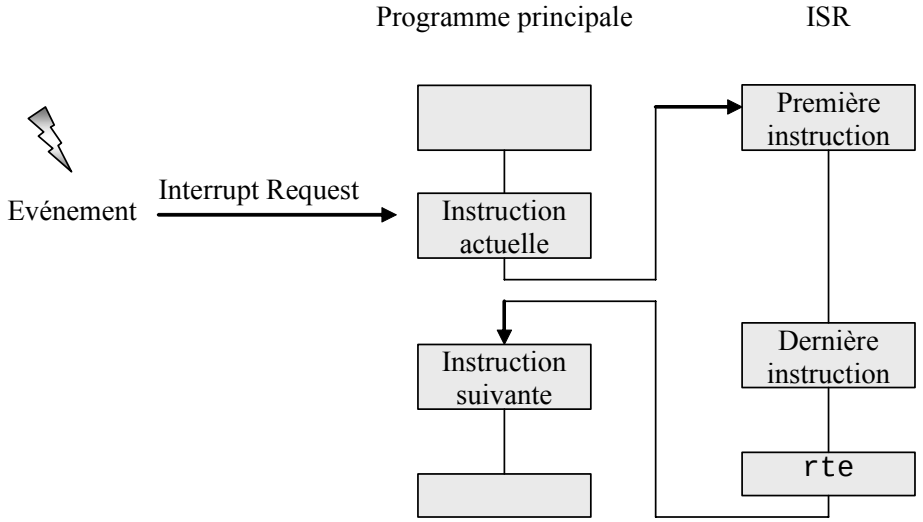

Figure 29 : Fonctionnement du programme à la suite d'une requête d'interruption

Le « contexte » est l'état actuel du programme principal (registre du processeur, le stack etc.). La routine d'interruption doit assurer qu'après son exécution le programme principal continue comme si rien n'est arrivé. Comme si vous avez noté la page que vous êtres en train de lire, avant d'aller répondre au téléphone.

5.1 Définition d'une routine d'interruption

Le compilateur doit générer un code spécial pour les routines d'interruption car ces derniers peuvent interrompre le programme principal à n'importe quel instant. Avec les appelles de sous programmes standard, le compilateur connaît parfaitement l'instant et les valeurs des registres à sauver. Avec les routines d'interruption cela n'est pas connu d'avance. Par conséquent, avant d'exécuter les instructions de la routine d'interruption, il est prudent de sauver toutes les données importantes au début de la routine de service afin que ces derniers puissent être restitués à la fin de la routine. Cela n'a pas été standardisé en C. Par conséquent, chaque environnement de développement possède ses propres

compléments d'instruction. Certains utilisent des mots clés comme par exemple `interrupt`, `__int`, `interrupt_nn` ou `#pragma interrupt`. La syntaxe exacte doit être recherchée dans la documentation du compilateur.

Dans l'environnement de développement du C8051Fxx, les routines d'interruption doivent être définies à l'aide du mot clé `interrupt`, complété avec le numéro du vecteur d'interruption :

```
/* the ISR will be called every 100 ms from timer interrupt */

void TimerInterruptHandler (void) interrupt 5
{
    Timeout--;
    if (Timeout <= 0) {
        Alarm();
        Timeout = 9999;
    }
}
```

5.2 Blocage des interruptions

Du fait qu'une interruption peut apparaître à n'importe quel instant, le programmeur doit faire attention aux opérations, dont l'exécution ne devrait pas être interrompue. Pour cela il existe des fonctions, qui permettent de bloquer ou de libérer les interruptions. Bien que les interruptions bloquées soient ignorés momentanément, ils ne sont pas oubliés par le processeur. Si tôt que les interruptions bloquées sont de nouveau libérées, elles seront traitées par le processeur. Le blocage des interruptions devrait être aussi court que possible. Si non des nouvelles interruptions peuvent apparaître avant que les anciennes soient traitées. Ce qui peut entraîner des comportements erronés du programme.

```
void TimerInterruptHandler (void) interrupt 5
{
    Timeout--;
}

void Trouble (void)
{
    if (Timeout > 0) {
        Magic = 1.0 / (float) Timeout;
    }
}
```

Dans l'exemple de ci-dessus une division par zéro peut apparaître, pourquoi ?

Car l'interruption peut apparaître juste après le test avec `if` et réduire la valeur de `Timeout` de 0 à 1, avant que la division ne soit exécutée par l'ordinateur.

Afin d'éviter cette erreur il faut modifier le code de la manière suivante :

```
void TimerInterruptHandler (void) interrupt 5

{
    Timeout--;
}

void Trouble (void)
{
    DisableInterrupts ();
    if (Timeout > 0) {
        Magic = 1.0 / (float) Timeout;
    }
    EnableInterrupts ();
}
```

Ce qui est beaucoup meilleur. Mais que se passe-t-il lorsque les interruptions sont préalablement bloquées et que ces derniers sont libérés. Il existe pour cela des fonctions, qui permettent de déterminer si les interruptions sont préalablement bloquées ou pas. De code devrait être définie de façon optimale de la manière suivante :

```
void TimerInterruptHandler (void) interrupt 5
{
    Timeout--;
}

void Trouble (void)
{
    int OldInterruptState;
    OldInterruptState = GetInterruptState();
    DisableInterrupts();

    if (Timeout > 0) {
        Magic = 1.0 / (float) Timeout;
    }

    /* Enable Interrupts only if they were enabled before */
    if (OldInterruptState == 1) {
        EnableInterrupts ();
    }
}
```

Attention

Même les morceaux de code qui peuvent paraître comme étant apparemment sur, comme par exemple **`a++`** ou **`a = a + 1`**, doivent être protéges des interruptions. Car ces instructions haut niveau sont souvent traduites en une suite d'instructions assembleur (lecture, incrémentation et écriture), qui ne doivent également pas être interrompues. Avec certain processeur, il existe des instructions dites « atomiques », qui ne peuvent pas être interrompues. Mais pour pouvoir utiliser ces instructions, il faut activer certaines options du compilateur.

5.3 Traitement des interruptions avec le contrôleur C8051F

Le traitement des interruptions dans notre système se déroule exactement de la manière décrite ci-dessus. Chaque source d'interruption peut être libérée ou bloquée avec des bits de configuration, qui sont contenus dans les registres de fonctions spéciales (**IE**, **EIE1** et **EIE2**). Toutefois, les interruptions doivent être libérées de façon globale en activant le bit **EA** dans le registre **IF**.

Les contrôleurs C8051F possèdent deux niveaux de priorités d'interruption, une haute et une basse. Une interruption de basse priorité peut être interrompue par une interruption de haute priorité. Par contre, une interruption de haute priorité ne peut pas être interrompue. Le niveau de priorité peut être définie avec les bits de priorité, qui se situent dans les registres de fonction spéciales (**IP**, **EIP1** et **EIP2**). Si deux requêtes d'interruption apparaissent au même instant, la requête avec la plus haute priorité sera traitée en premier. Si les deux requêtes possèdent le même niveau de priorité, le traitement aura lieu en fonction d'un ordre de priorité fixé d'avance, selon le tableau 9.4 de la documentation concernant le C8051F2xx.

L'exemple de l'alinéa précédant peut être adapté à notre système de la manière suivante :

```
void TimerInterruptHandler (void) interrupt 5
{
    Timeout--;
}

void Trouble (void)
{
    unsigned char OldStateIE;
    /* Save old state of Interrupt Enable (IE) register */
    OldStateIE = IE;

    /* Disable the interrupts by clearing the
       Enable All (EA) interrupts bits
      in the Interrupt Enable (IE) Register */
    EI = EI & 0x7F;

    if (Timeout > 0) {
        Magic = 1.0 / (float) Timeout;
    }

    /* Restore old state of Interrupt Enable (IE) register */
    IE = OldStateIE;
}
```

5.3.1 Les vecteurs d'interruption

La définition des routines d'interruption n'est pas suffisante. Le processeur doit également avoir, quelle routine doit être appelée lorsqu'apparaît une requête d'interruption. Le principe est le même pour toutes les familles de processeur. Soit le système possède une tabelle des vecteurs d'interruption, qui contient toutes les adresses des routines de service d'interruption. Ou alors, le système fait des sauts de programme à des adresses prédéfinies. Dans le premier cas, le programmeur doit écrire

l'adresse de la routine de service dans la tabelle des vecteurs d'interruption. Dans le second cas, il doit déposer le code de la routine de service d'interruption à l'adresse prédéfinie dans la mémoire.

Le C8051F travaille avec un tableau de vecteurs d'interruptions. Ce tableau contient 21 vecteurs d'interruption, dont 9 sont dédiés aux composants périphériques, comme par exemple : le module temporel (Timer 2), la communication sériel, les convertisseurs AD et les comparateurs (voir tableau 9.4 de la documentation du C8051F2xx).

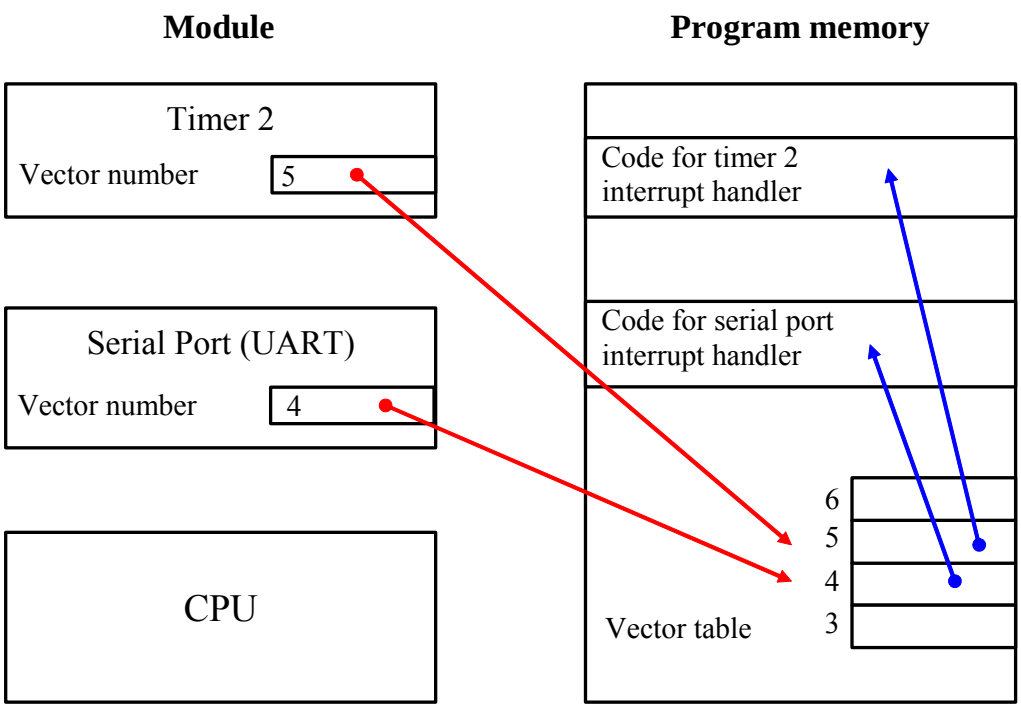

Figure 30 : Principe d'appel de la routine de service d'interruption

Lorsque qu'un composant périphérique génère une requête d'interruption, il transmet simultanément son vecteur d'interruption. A l'aide de ce vecteur d'interruption, le CPU peut chercher l'adresse de la routine de service dans la tabelle des vecteurs d'interruption.

Dans notre environnement de développement, le mot clé **`interrupt`** permet d'écrire l'adresse de la routine de service d'interruption dans la tabelle des vecteurs d'interruption. Ce dernier doit être complété avec le numéro du vecteur d'interruption. Dans les exemples de ci-dessus, l'adresse de la routine de service d'interruption est stockée dans le vecteur numéro 5 de la tabelle des vecteurs d'interruption. En effet, ce vecteur d'interruption est réservé pour le module temporel 2 :

```
void TimerInterruptHandler (void) interrupt 5
{
    Timeout--;
}
```

6 Le code de démarrage

Certaines tâches d'initialisation doivent être réalisées avant le démarrage de la fonction principale main. Ces tâches sont en générale très spécifiques au processeur et doivent être programmé en partie en assembleur. Ces tâches comprennent l'initialisation des variables globales et statiques, l'initialisation des fonctions des bibliothèques standards, la configuration minimale du hardware et initialisation des registres spéciaux du processeur et finalement initialisation du pointeur de pile (stack) et de trame (frame) pour la gestion des appelle de fonction et des variables locales.

Le déroulement d'un système à microcontrôleur autonome est le suivant (on admet que le programme se trouve dans la ROM).

Procédure :

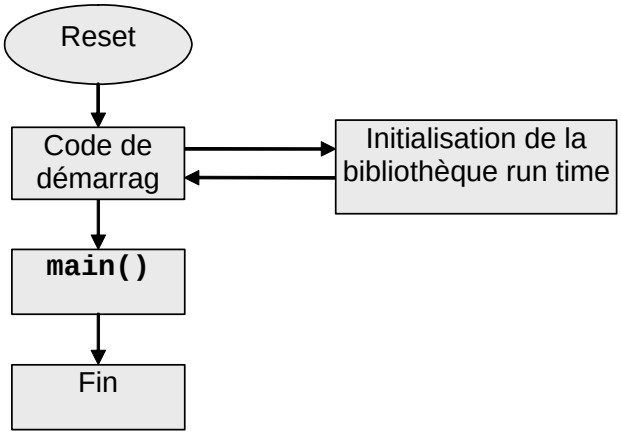

Figure 31 : Diagramme de flux de l'initialisation d'un système à microprocesseur

Il n'y pas de retour depuis la fonction principale **`main()`** dans les programmes destinés aux microcontrôleurs. Ce dernier est composé en générale d'une boucle infinie, qui réalise les diverses tâches de contrôle.

Si par malheur la fonction principale s'achève, le système pourrait entrer dans un état indéfini. En générale ce dernier entre dans une boucle infinie. Toutefois, il serait peut être plus intelligent de le redémarrer dans ces cas.

Dans les PC, un programme se termine à la fin de la fonction principale. Toutes les fenêtres de l'application sont alors fermées et le système d'exploitation attend sur des nouvelles commandes de l'utilisateur.

Les microcontrôleurs ne contiennent en générale pas de système d'exploitation. Le développeur doit dans ce cas définir par lui-même ce qui doit être fait (en tous cas quelque chose, qui ne limite pas la sécurité et la stabilité du processus. Un système de navigation devrait par exemple se réinitialiser ou stopper le véhicule. Dans tous les cas il ne devrait pas s'arrêter et laisser le véhicule continuer dans une direction donnée)