



3 IRC – Architecture des ordinateurs

Langage C et microcontrôleur 8051

Ver: 2018

membre de UNIVERSITÉ DE LYON



LYON
CPE
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

Objectifs de la seconde partie du module



Conception d'un mini système embarqué

- **Un système de ce type sera mis en œuvre durant le micro-projet de TP**

Définition d'un système embarqué



De nombreuses définitions du terme « système embarqué » existent...

Une définition parmi d'autres: un système embarqué est un dispositif à la fois matériel et logiciel qui contient au moins un processeur programmable (microprocesseur, microcontrôleur, DSP) et qui est mis en œuvre par des utilisateurs qui n'ont pas « conscience » de l'existence du processeur.

Autres notions fréquemment associées:

- **Contraintes temps réel, c'est-à-dire capacité de répondre à tout évènement dans un délai parfaitement maîtrisé.**
- **Sécurité de fonctionnement – sur certains systèmes, un plantage pourrait avoir des conséquences catastrophiques.**

Exemples de systèmes embarqués



Exemples:

- **Domestique:** lave-vaisselle, lave-linge, cafetières
- **Audio-vidéo:** lecteurs MP3, dictaphone, gadgets....
- **Téléphonie:** combinés, stations de base...
- **Automobile:** calculateurs de contrôle de freinage, d'allumage, de régulation de vitesse
- **Aéronautique:** calculateurs divers, altimètres, pilote automatique, environnement passager
- **Médical....**
- **Etc....**

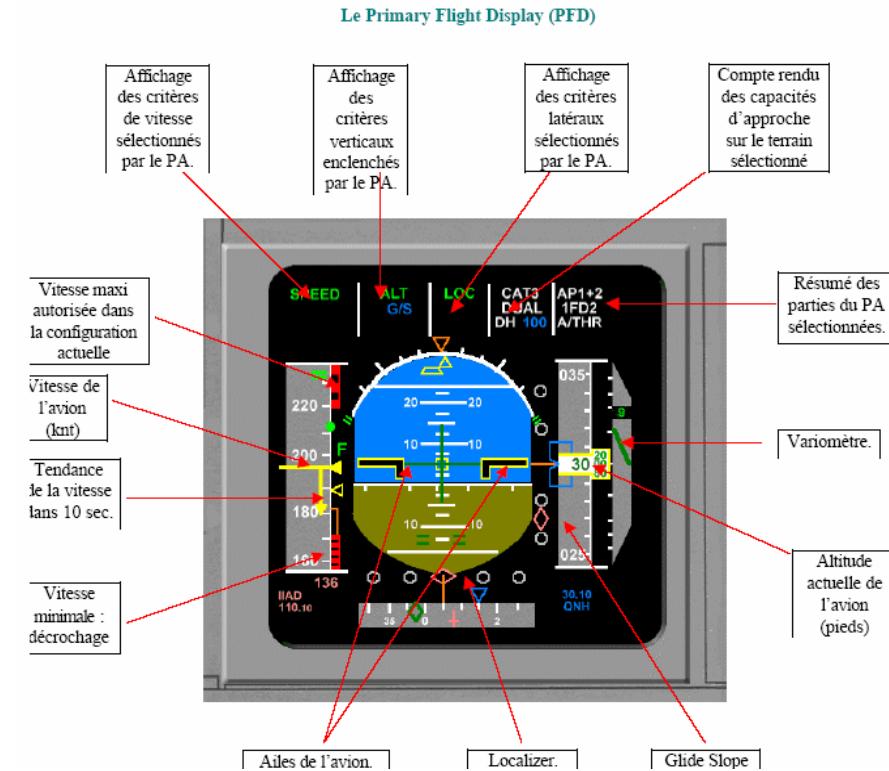
A contrario: micro-ordinateurs, tablettes, smartphones ne sont pas des systèmes embarqués...

Caractéristiques essentielles d'un « SE »

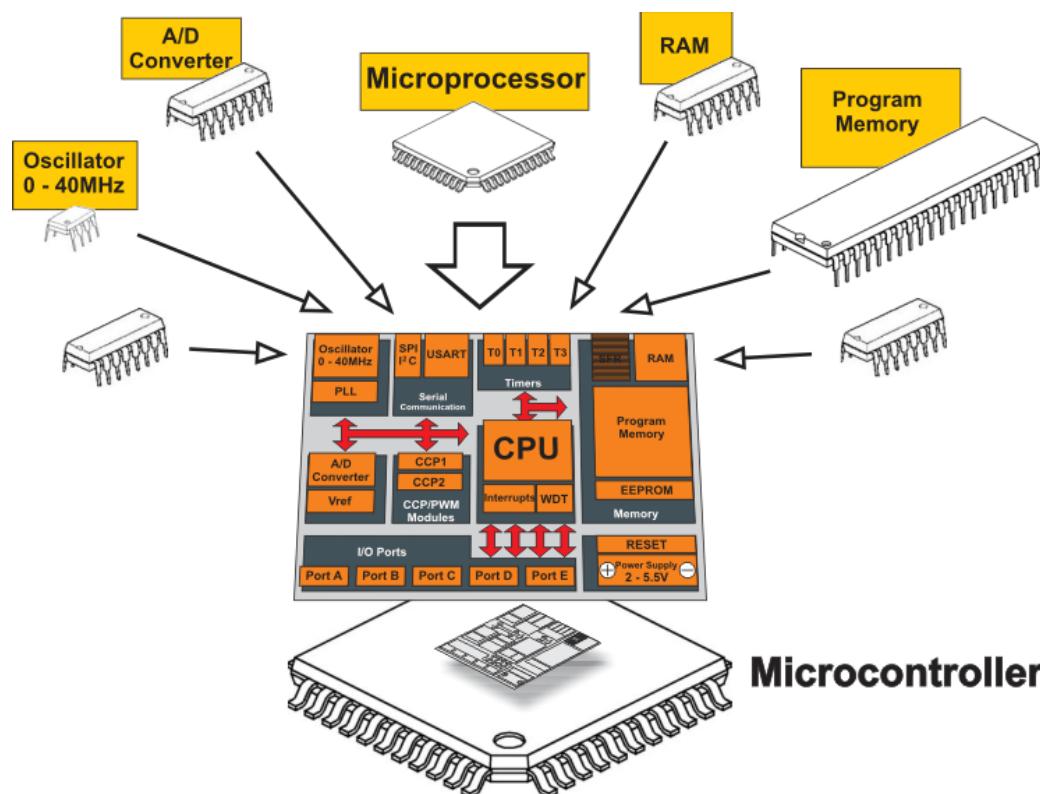
Du gadget à LEDS au Primary Flight Display (PFD)....

C'est le processeur (le microcontrôleur) qui constitue le cœur d'un système embarqué.

Son choix sera déterminant.



Un Microcontrôleur?



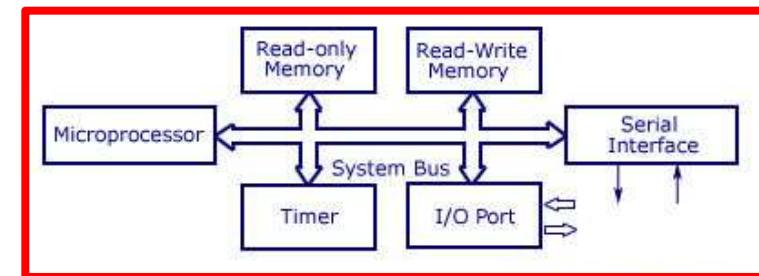
Source: <http://maxembedded.com/>

Intégration de toutes les fonctionnalités d'un système à microprocesseur sur une seule puce silicium

Objectifs:

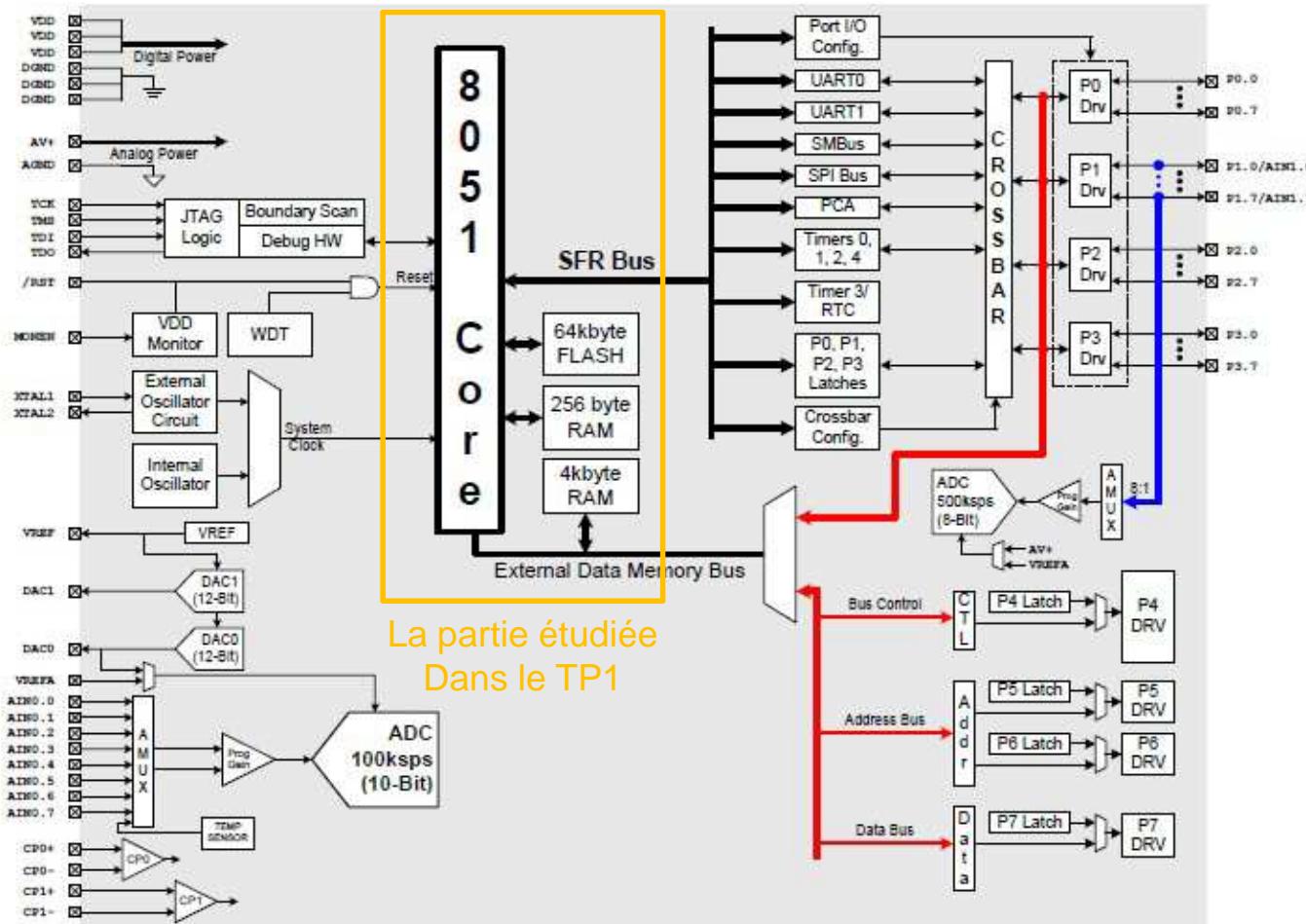
- **Minimiser l'encombrement**
- **Minimiser le coût**
- **Améliorer la fiabilité**

Schematic Arrangement of a Microprocessor Based System



www.CircuitsToday.com

Schéma bloc du 8051F020 utilisé en TP



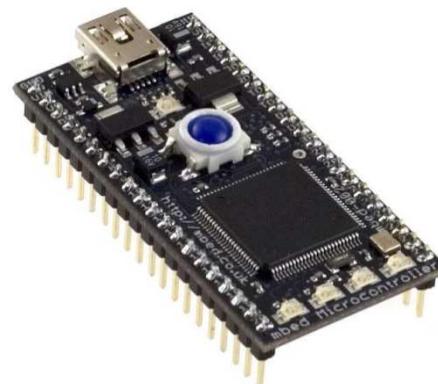
Les solutions sur « étagère »

Il existe des solutions qui permettent de concevoir rapidement un système embarqué. Ces solutions couvrent aussi bien l'aspect logiciel que matériel.

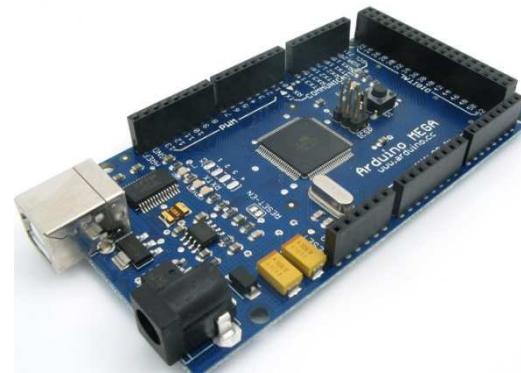
Des exemples:

- Arduino
- Mbed

ARM® mbed™



™
ARDUINO



Le point commun de ces solutions...

- **Masquage du bas-niveau pour simplifier le développement**
- **Mise en place de librairies de haut niveau pour gérer les périphériques**



Une approche simplifiée, accessible au plus grand nombre.



Mais: l'architecture du processeur et ses aspects bas-niveau sont ignorés, d'où une quasi impossibilité de faire face à un problème lié à l'utilisation des librairies.



Mais: mauvaise optimisation des librairies, l'utilisation de ces plateformes peut devenir délicate en cas de contraintes fortes en temps réel.

Conséquence: une connaissance du bas-niveau permet d'envisager des architectures logicielles optimisées et apportera une capacité à optimiser l'utilisation des plateformes telles que Arduino.

Notre Hypothèse de départ pour la réalisation du mini système embarqué



Cas classique pour un « petit » microcontrôleur »

- Utilisation du langage C
- Pas d'utilisation de librairie de gestion de périphériques
- Pas de système d'exploitation embarqué
- Programmation « from scratch »
- Mais avec l'aide de l'environnement de développement.

Source des informations: datasheet 8051F020, documentation du compilateur C51 Keil et documentation générique sur le langage C

Programmer en C sur un microcontrôleur...



PARCE QUE:

- Plus rapide de développer en C qu'en assembleur
- Permet d'atteindre un niveau d'abstraction plus élevé
- Code en C aisément réutilisable
- Langage qui reste proche de la machine

MAIS:

- Le processeur ne sait exécuter que de l'assembleur. (Etape de compilation nécessaire)
- En cas de problème à l'exécution, il est parfois nécessaire de maîtriser la chaîne de génération de code et l'architecture du processeur

Exemple élémentaire: Addition de 2 nombres 16-bits

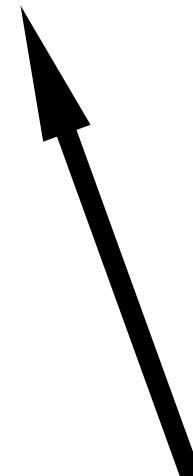
```
data_test    segment  DATA
              rseg      data_test
;Convention: little endian format
```

```
X:         DS     2
Y:         DS     2
Z:         DS     2
```

```
ADD_Program segment CODE
rseg      ADD_Program
```

```
MOV A,X
ADD A,Y
MOV Z,A
MOV A,X+1
ADDC A,Y+1
MOV Z+1,A
```

```
#include <c8051f020.h>
void main (void) {
int x, y, z;
z = x + y;
}
```



La version C

La version assembleur (de haut niveau)

Spécificité du code C sur microcontrôleurs



Il faut tenir compte:

- Faible mémoire
- Faible puissance de calcul (architectures 8 bits)
- Haut niveau de réactivité temporelle attendu
- Accès direct aux mémoire physiques et périphériques

A propos du code...



```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
```

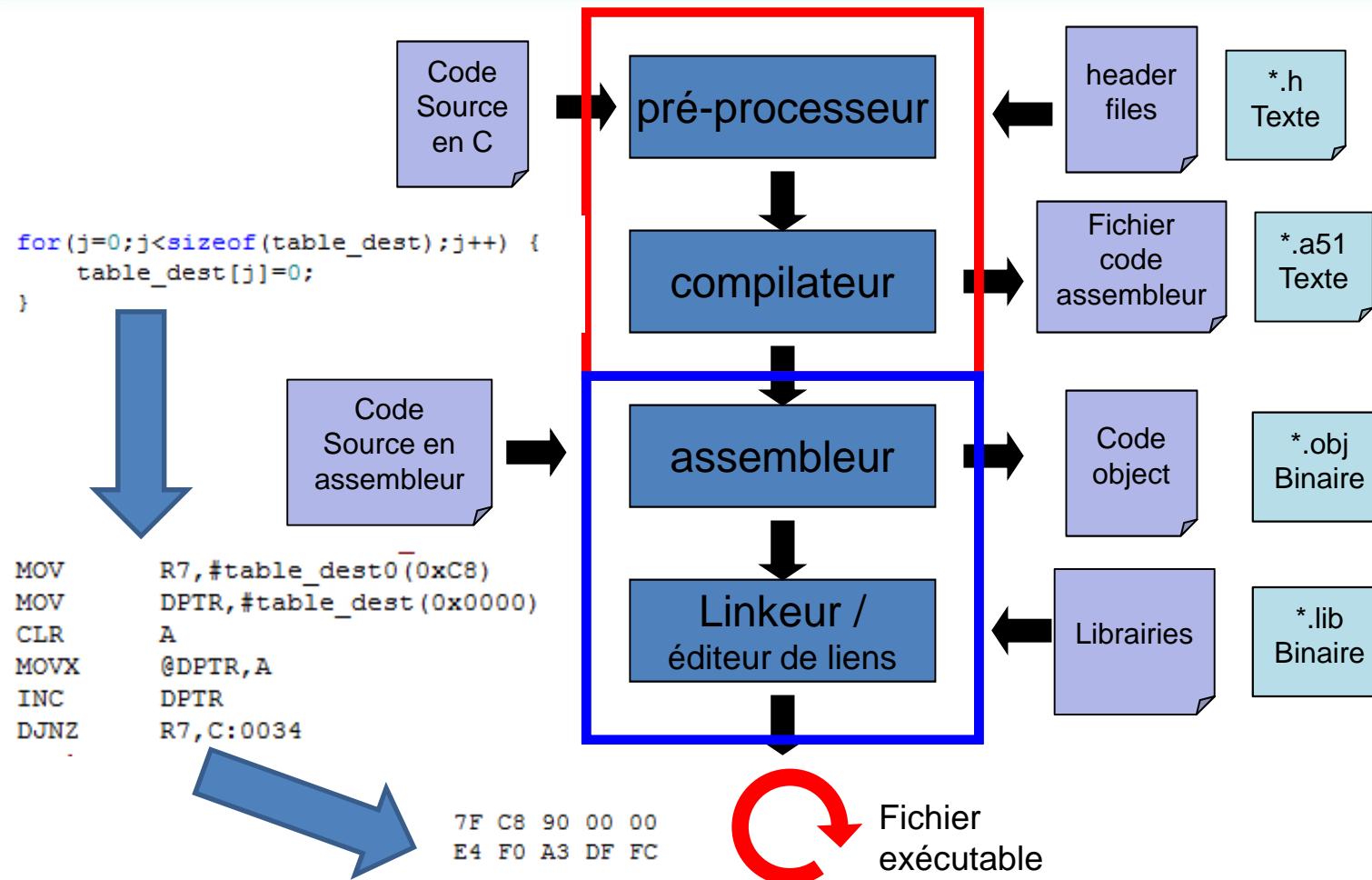
Pris en compte par le pré-processeur

```
main()
{
    int i;
    i = 5 * 2;
    printf("5 times 2 is %d.\n", i);
    printf("TRUE is %d.\n", TRUE);
    printf("FALSE is %d.\n", FALSE);
}
```

Pris en compte par le compilateur

Géré par des librairies de fonctions

Les étapes de la compilation du code



La base de la compilation



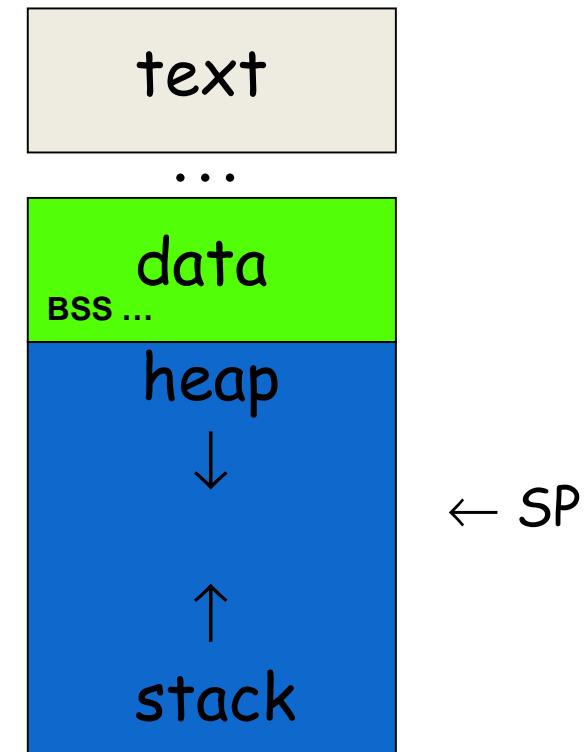
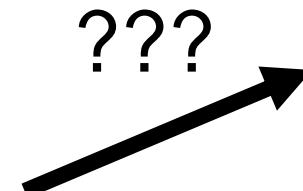
- Le **pre-processeur** gère
 - Les Macros (#define)
 - Les Inclusions (#include)
 - Les Inclusions conditionnelles (#ifdef, #if)
 - Les extensions du langage (en partie) (#pragma).
- Le **compilateur** transforme du code source en langage assembleur
- L'**assembleur** convertit le langage assembleur en code binaire (visualisé en hexadécimal) (code objet).
- Le “**linkeur**” (**éditeur de lien**) procéde à l’édition de liens des codes objets (qui se référencent). Il crée l’exécutable (éventuellement en lien avec un format d’exécutable)

Organisation classique de la mémoire en C dans un environnement avec OS

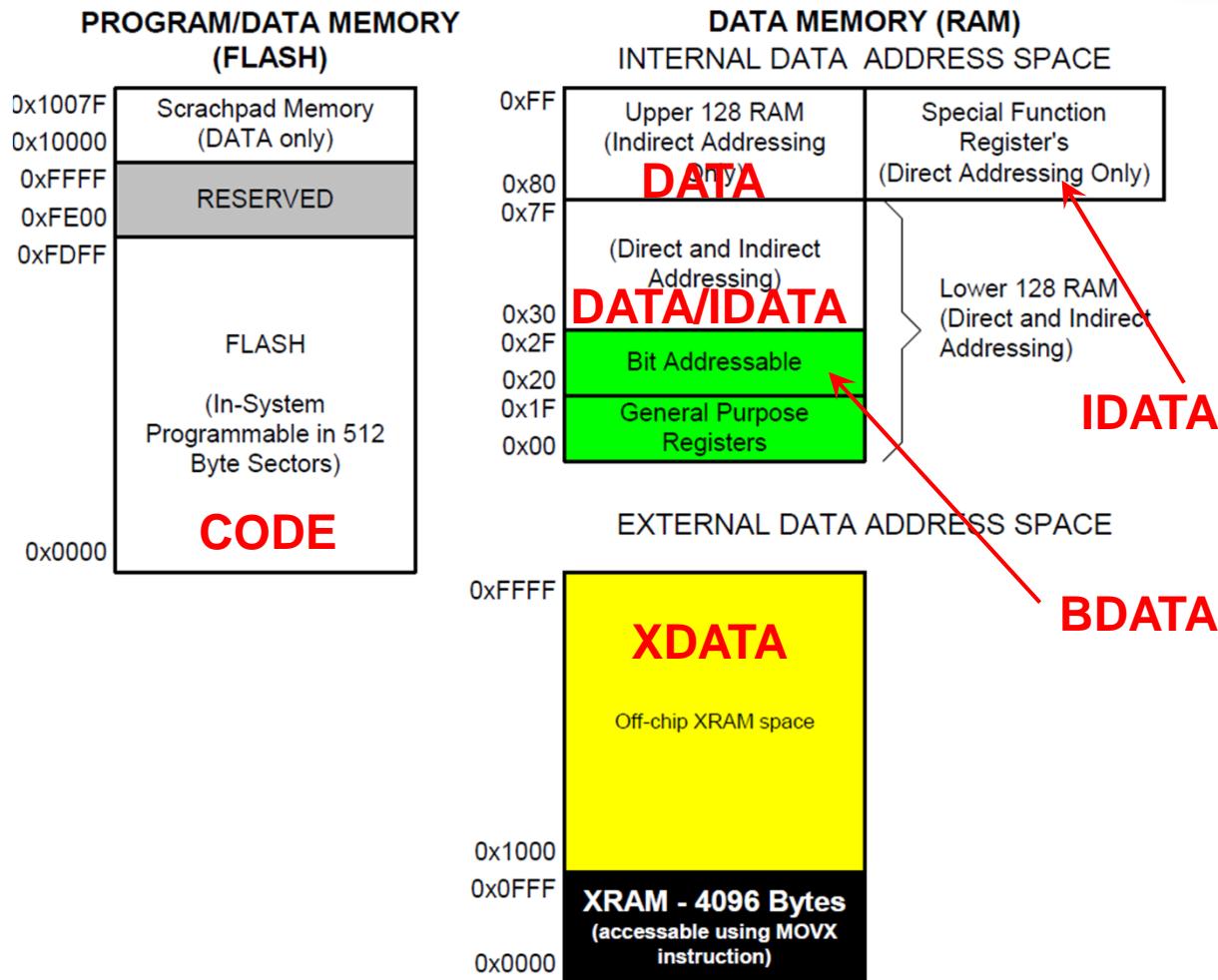
Zones mémoires:

- **Text** == code
- **Data** == variables globales et constantes globales (BSS)
- **Heap (Tas)** == données dynamiques
- **Stack (Pile)** ==
 - Variables locales
 - Paramètres des fonctions
 - Adresse de retour

```
char strng="hello";
int count, this, that;
main()
{
int i, j, k;
char *sp;
.....
for (i=0;i != 100;i++)
.....
sp= (char*) malloc(sizeof(i));
```



Organisation de la mémoire dans un environnement Microcontrôleur 8051

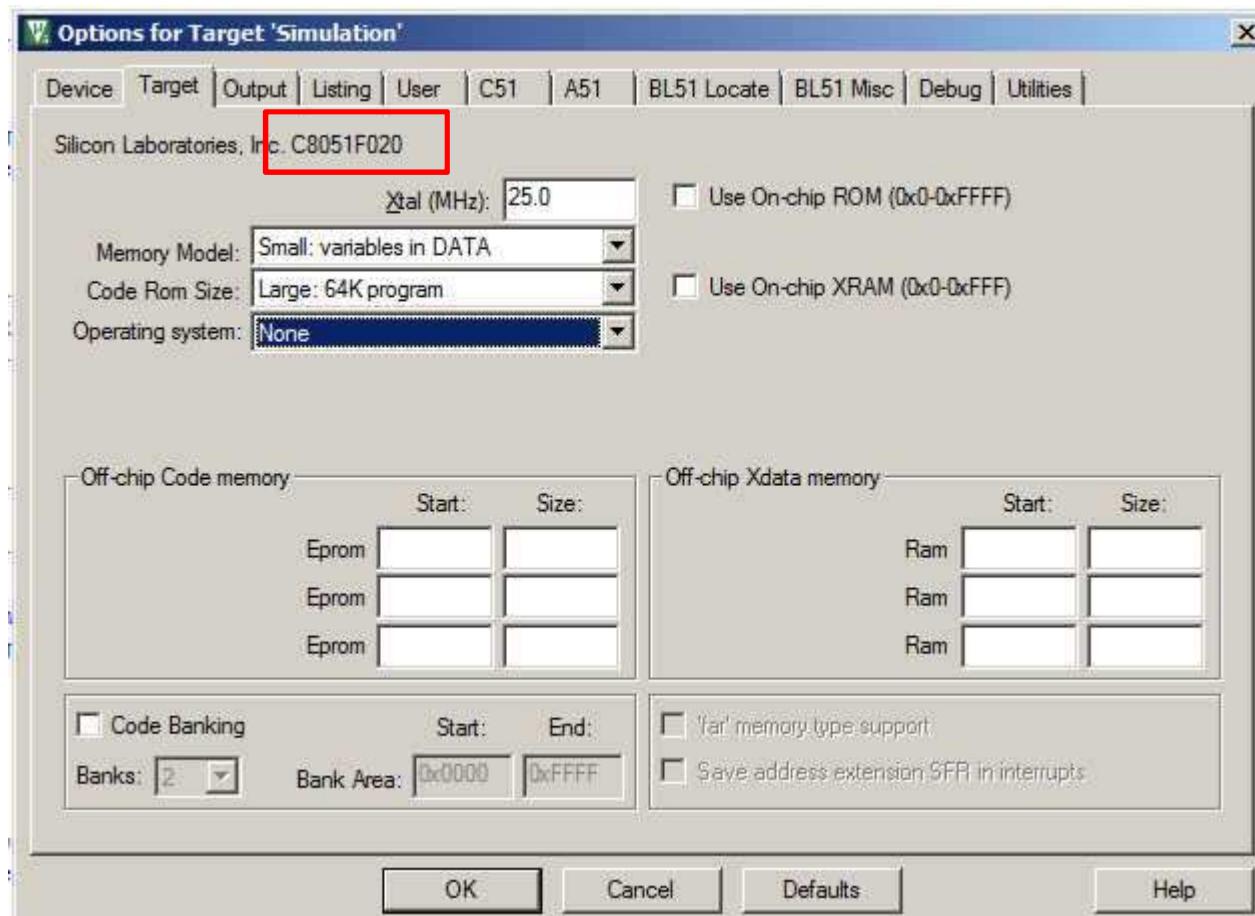


Localisation?

- **Text (Code)**
- **Data**
- **Heap (à éviter)**
- **Stack**

RAPPEL Assembleur:
Utilisation de segments
data – idata – xdata –
bdata - code

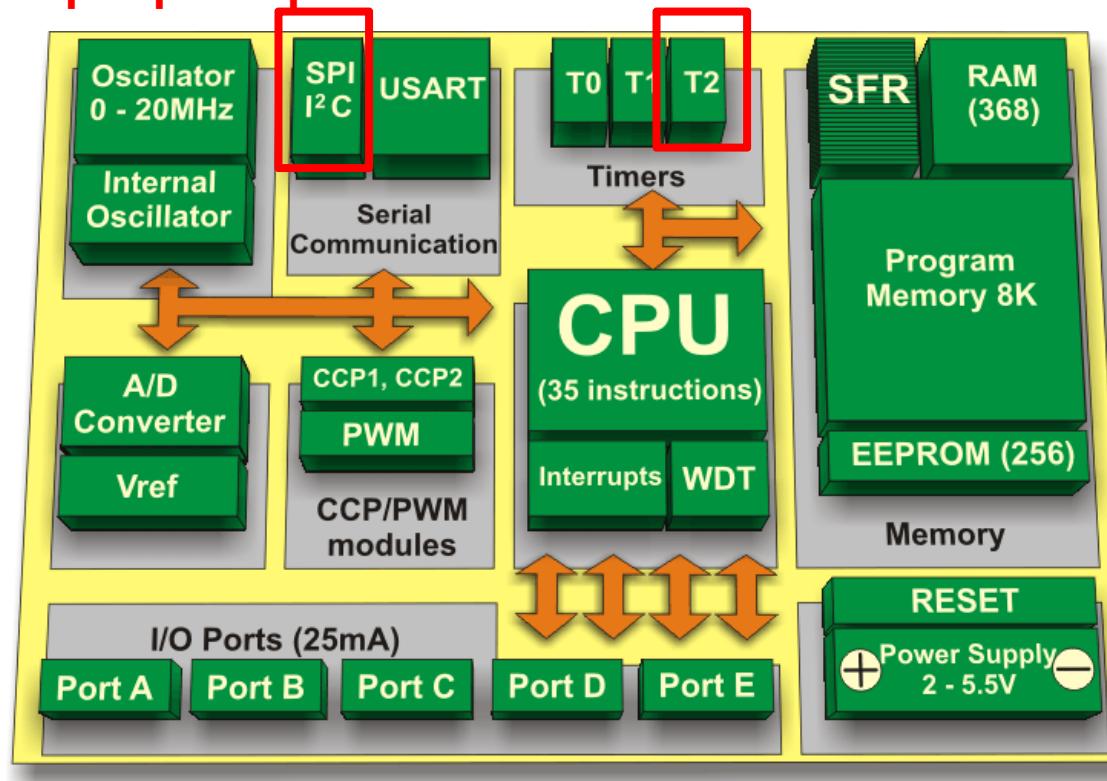
Gestion de la mémoire via l'environnement de développement



Coder une application: Par où commencer...

Un microcontrôleur = 1 Processeur + des Mémoires + des Périphériques

1 périphérique:

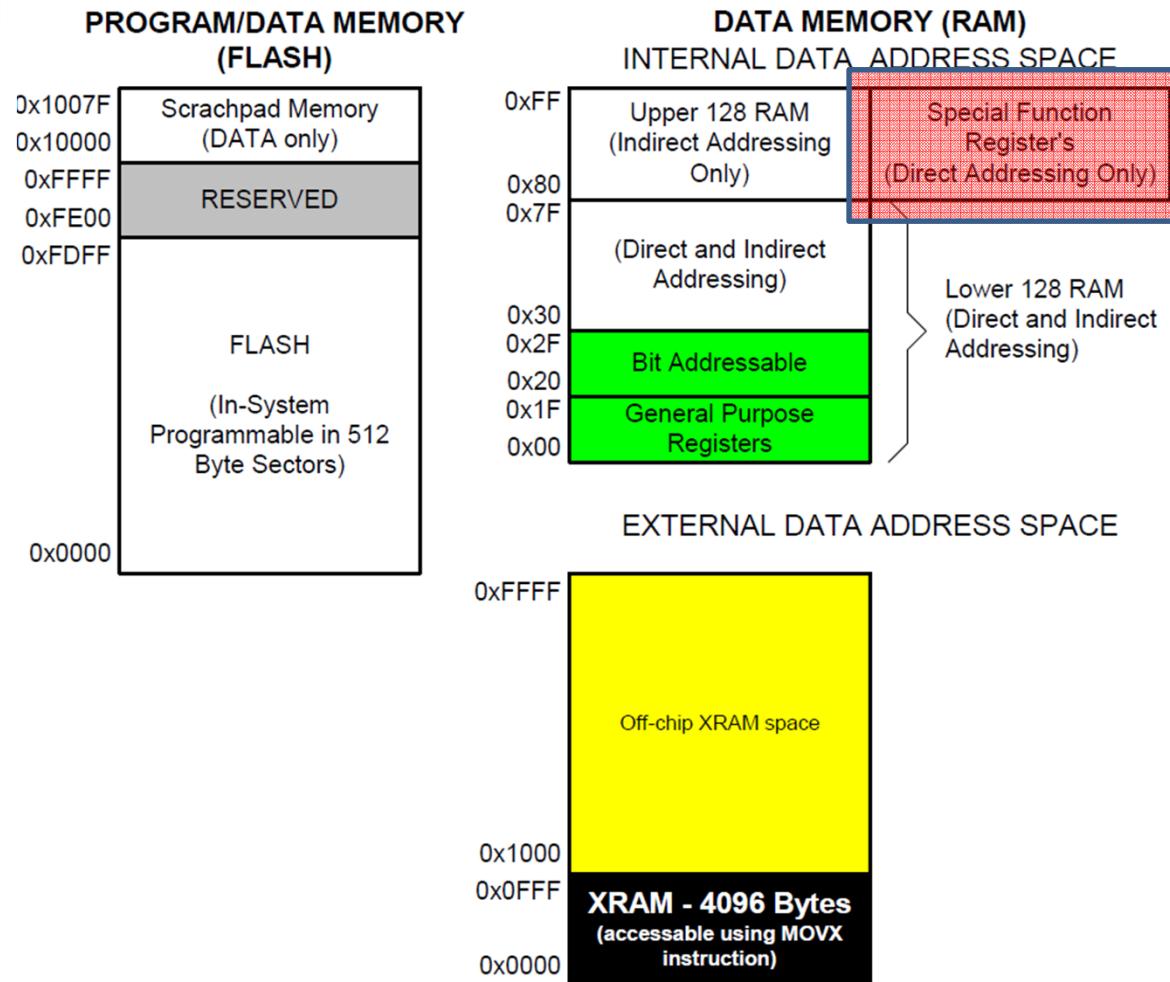


Source: mikroe.com

Avant de coder l'application, il faut initialiser, configurer ces 3 éléments.
Pour chaque périphérique:

- J'utilise ou pas?
- Si oui, je le configure pour le mode de fonctionnement désiré

Configurer Processeur, (Mémoire) et Périphérique....



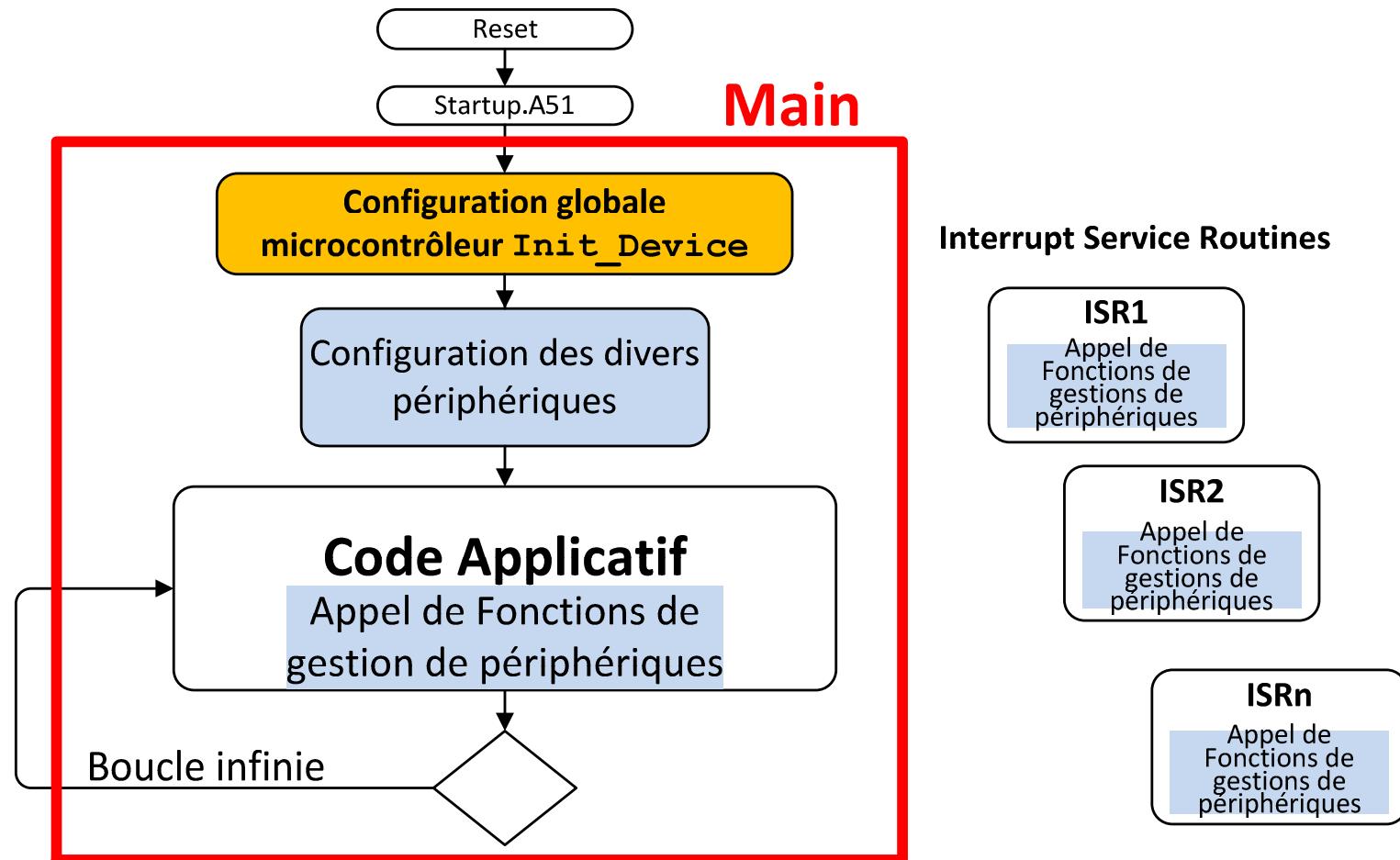
....Reviens à modifier le contenu d'un ou plusieurs registres présents dans l'espace mémoire SFR

C'est aussi à travers ces registres que l'on va assurer la configuration initiale du processeur

Cas du 8051F020: l'espace SFR

F8	SPI0CN	PCA0H	PCA0CPH0	PCA0CPH1	PCA0CPH2	PCA0CPH3	PCA0CPH4	WDTCN
F0	B	SCON1	SBUF1	SADDR1	TL4	TH4	EIP1	EIP2
E8	ADC0CN	PCA0L	PCA0CPL0	PCA0CPL1	PCA0CPL2	PCA0CPL3	PCA0CPL4	RSTSRC
E0	ACC	XBR0	XBR1	XBR2	RCAP4L	RCAP4H	EIE1	EIE2
D8	PCA0CN	PCA0MD	PCA0CPM0	PCA0CPM1	PCA0CPM2	PCA0CPM3	PCA0CPM4	
D0	PSW	REF0CN	DAC0L	DAC0H	DAC0CN	DAC1L	DAC1H	DAC1CN
C8	T2CON	T4CON	RCAP2L	RCAP2H	TL2	TH2		SMB0CR
C0	SMB0CN	SMB0STA	SMB0DAT	SMB0ADR	ADC0GTL	ADC0GTH	ADC0LTL	ADC0LTH
B8	IP	SADEN0	AMX0CF	AMX0SL	ADC0CF	P1MDIN	ADC0L	ADC0H
B0	P3	OSCXCN	OSCICN			P74OUT†	FLSCL	FLACL
A8	IE	SADDR0	ADC1CN	ADC1CF	AMX1SL	P3IF	SADEN1	EMI0CN
A0	P2	EMI0TC		EMI0CF	P0MDOUT	P1MDOUT	P2MDOUT	P3MDOUT
98	SCON0	SBUF0	SPI0CFG	SPI0DAT	ADC1	SPI0CKR	CPT0CN	CPT1CN
90	P1	TMR3CN	TMR3RLL	TMR3RLH	TMR3L	TMR3H	P7†	
88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	PSCTL
80	P0	SP	DPL	DPH	P4†	P5†	P6†	PCON
	0(8) (bit addressable)	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)

L'application globale en C



Objectifs de codage



Séparation des codes:

- Codes dépendant du processeur:
 - la configuration du microcontrôleur
 - la configuration des périphériques
 - l'utilisation des périphériques
 - Codes dépendant de l'application
 - Utiliser des fichiers source différents.
- Conséquence:** dans un code applicatif, on ne manipule jamais directement des registres.

Certaines familles de microcontrôleurs proposent des bibliothèques logicielles de gestion de périphériques

Le démarrage du microcontrôleur



Point de départ: le **Reset**.

- Point de vue processeur: Exécution du code à partir de l'adresse 0000
- Point de vue développeur d'application en C: exécution de la fonction **Main**
- Réellement: exécution du programme assembleur **startup.a51**, puis exécution du **Main** (dans l'environnement MicroVision)

Exécution du code de démarrage



Sur le 8051F020 – 1 seul mode de démarrage

- Code stocké dans la zone mémoire « code »
- Cette mémoire est une mémoire de techno « Flash », programmée via le dispositif de débogage.

Sur d'autres microcontrôleurs: – plusieurs modes de démarrage

- Multiboots
- Code spécifique de boot – Copie de mémoire

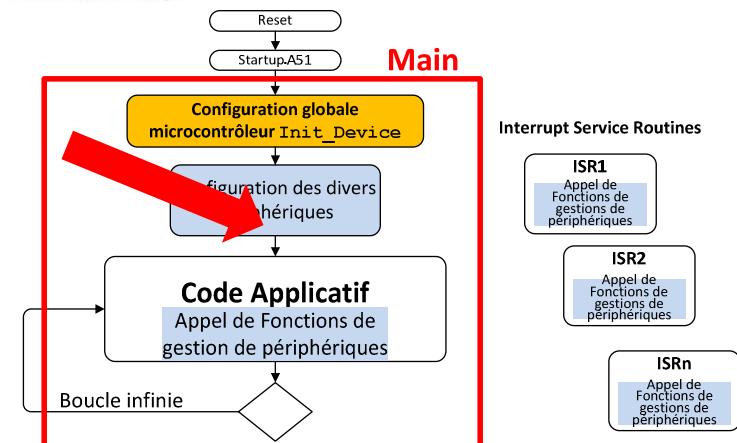
Cas de l'environnement Microvision: Le fichier Startup.a51 Configuration mémoire

Inséré automatiquement par l'environnement de développement MicroVision lors de la création du projet

Startup Code

Startup code is executed immediately upon reset of the target system. The Keil startup code performs (optionally) the following operations in order:

- Clears internal data memory
- Clears external data memory
- Clears paged external data memory
- Initializes the small model reentrant stack and pointer
- Initializes the large model reentrant stack and pointer
- Initializes the compact model reentrant stack and pointer
- Initializes the 8051 hardware stack pointer
- Transfers control to code that initializes global variables or to the main C function if there are no initialized global variables

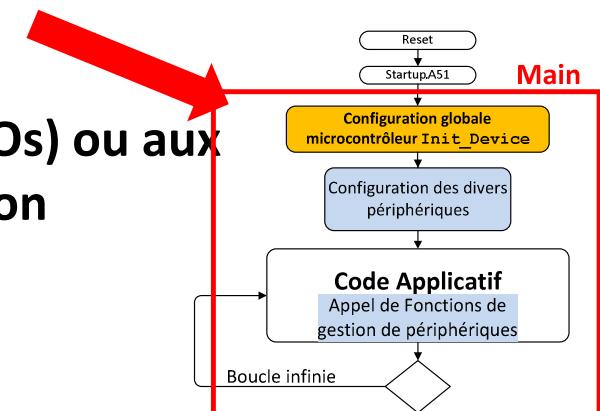


Configuration globale du microcontrôleur « Init_Device() »

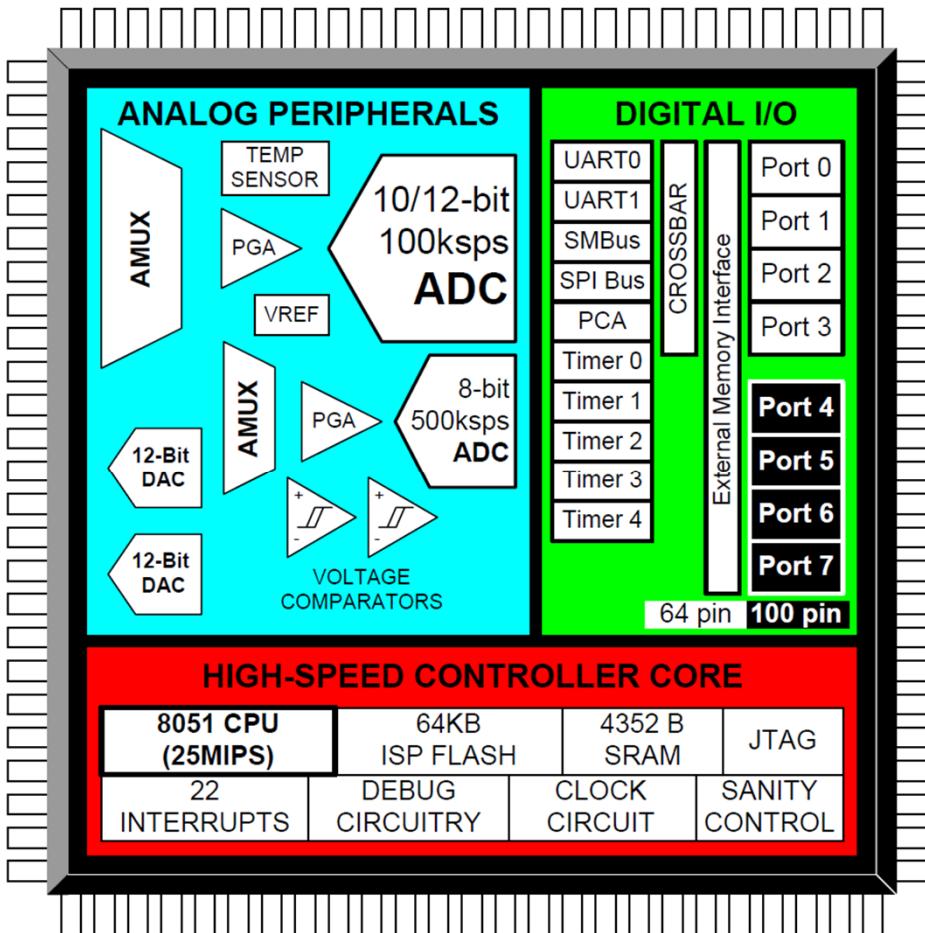
Fichier à Insérer: Son rôle est de configurer les fonctions de gestion du **processeur, des mémoires, et des périphériques « partagés »**

- Les sources de « Reset »
- Le chien de garde
- L' (ou les) horloge(s)
- Les mémoires physiques
- La gestion de la puissance
- **L'affectation des broches aux entrées-sorties (GPIOs) ou aux périphériques et éventuellement leur configuration**

Attention: les choix effectués dans cette étape peuvent conditionner le fonctionnement des divers périphériques



Affectation des broches dans un 8051F020

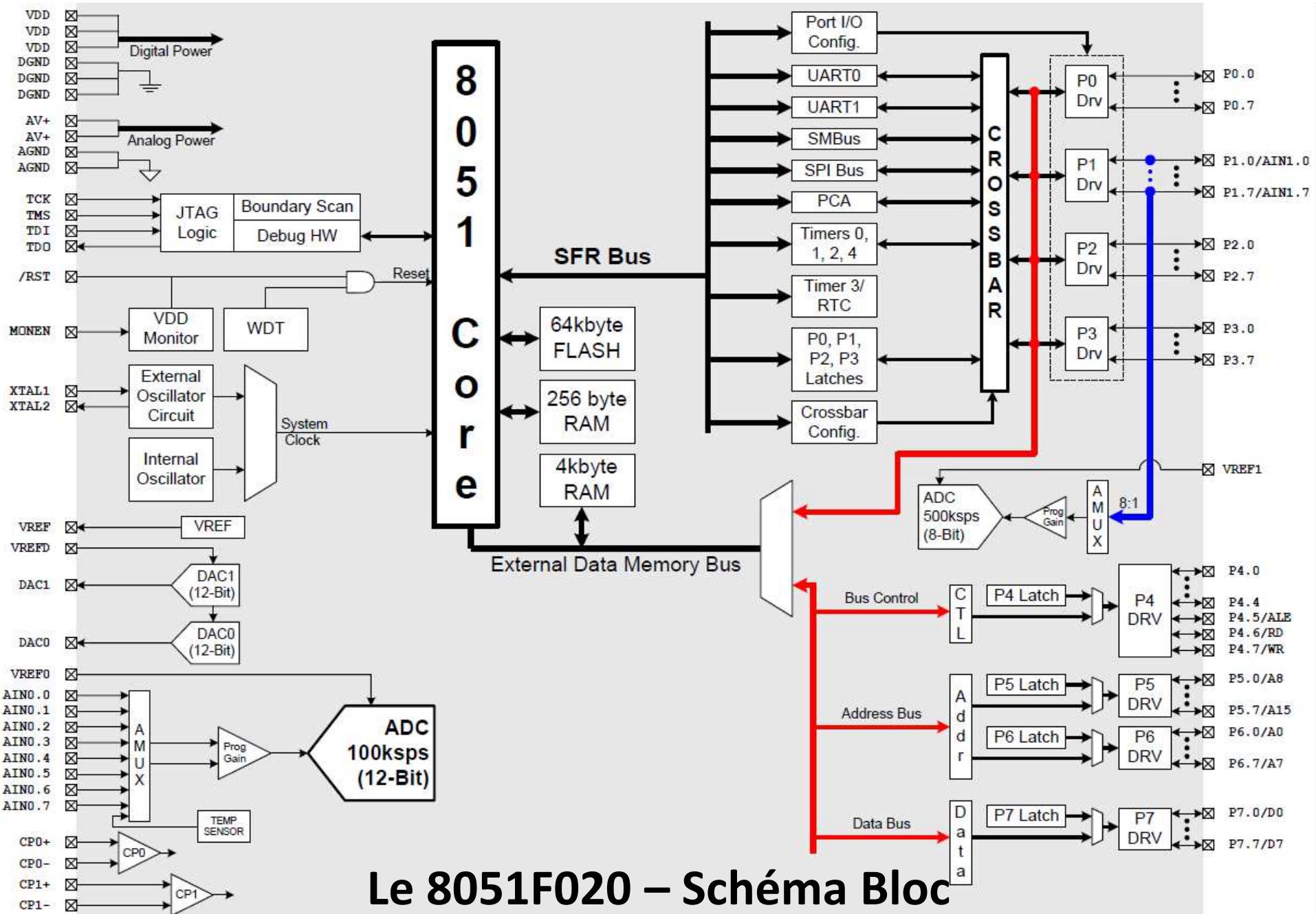


Quelques questions:

- Quels sont les périphériques disponibles?
- Comment les utiliser?
- Comment les connecter avec l'extérieur?

61	P0.1
60	P0.2
59	P0.3
58	P0.4
57	ALE/P0.5
56	/RD/P0.6
55	/WR/P0.7
54	AD0/D0/P3.0
53	AD1/D1/P3.1
52	AD2/D2/P3.2
51	AD3/D3/P3.3

v _{DD}	I _{DD}
DGND	38
A15m/A7/P2.7	39
A14m/A6/P2.6	40
A13m/A5/P2.5	41
A12m/A4/P2.4	42
A11m/A3/P2.3	43
A10m/A2/P2.2	44
A9m/A1/P2.1	45
A8m/A0/P2.0	46
AD7/D7/P3.7/I/E7	47
AD6/D6/P3.6/I/E6	48
AD5/D5/P3.5	49
AD4/D4/P3.4	50



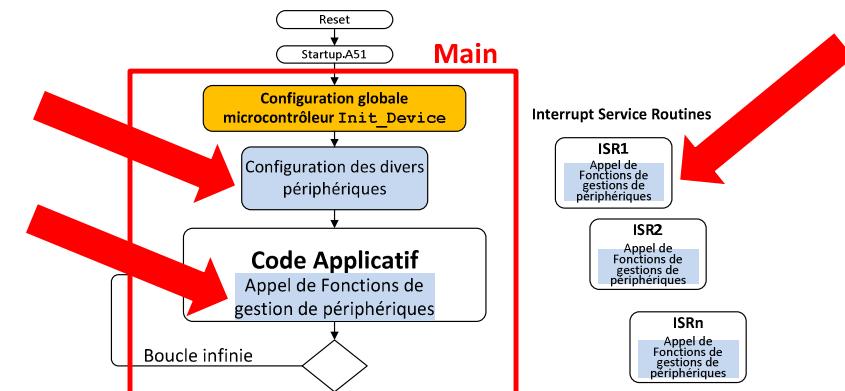
Le 8051F020 – Schéma Bloc

L'utilisation des périphériques

Toute application requiert l'utilisation d'un ou plusieurs périphériques.

Le code de gestion de ces périphériques sera scindé en 2 parties:

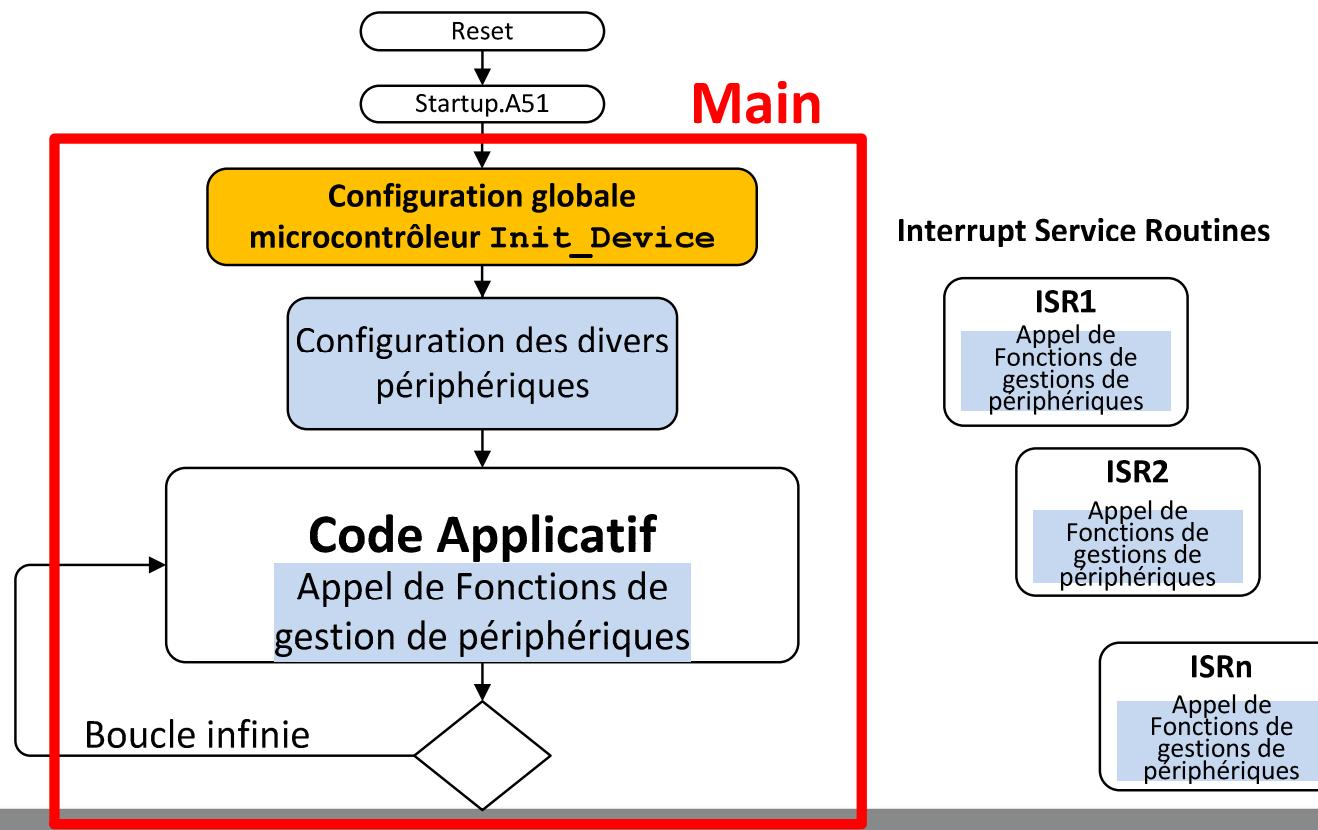
- La configuration
- L'utilisation



Bilan

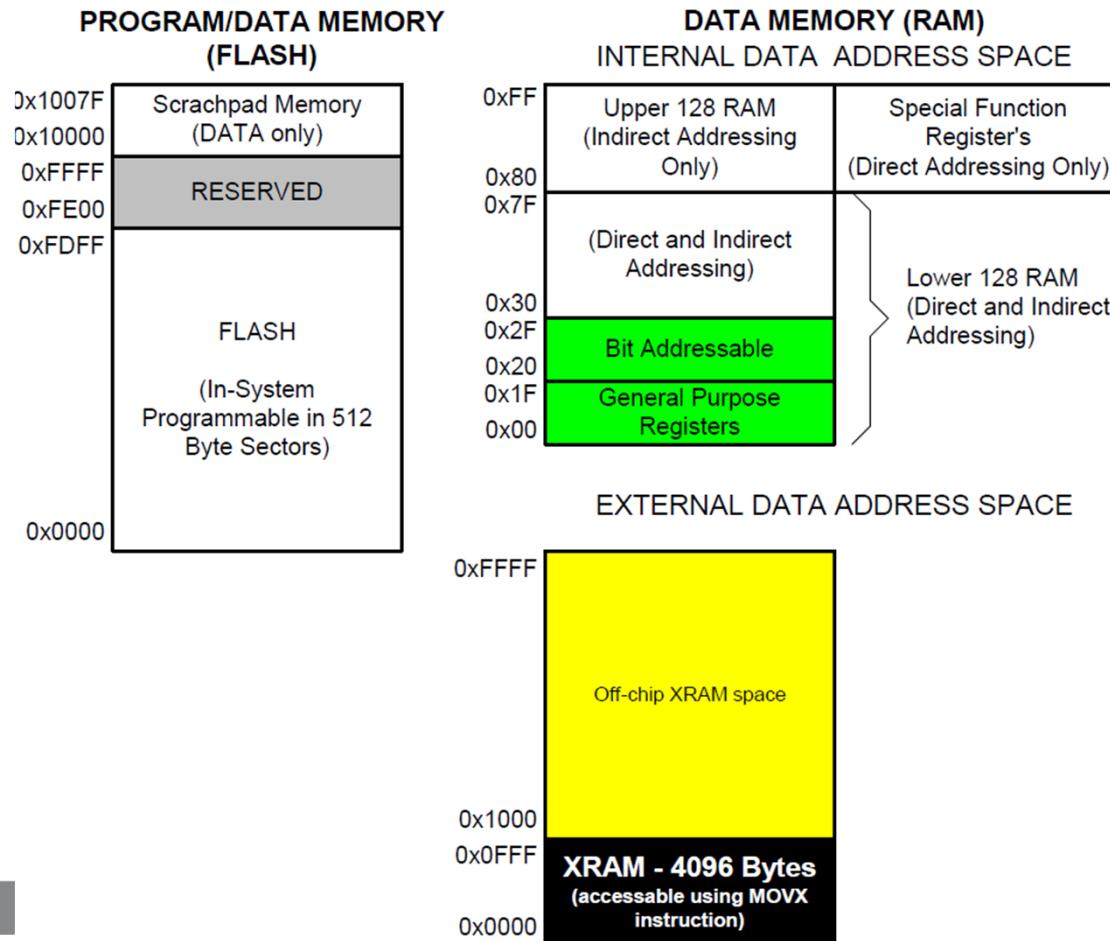
Développement d'une mini-application en langage C dans un environnement microcontrôleur

Structure de code typique:



Fonctionnalités propres au compilateur Keil C51

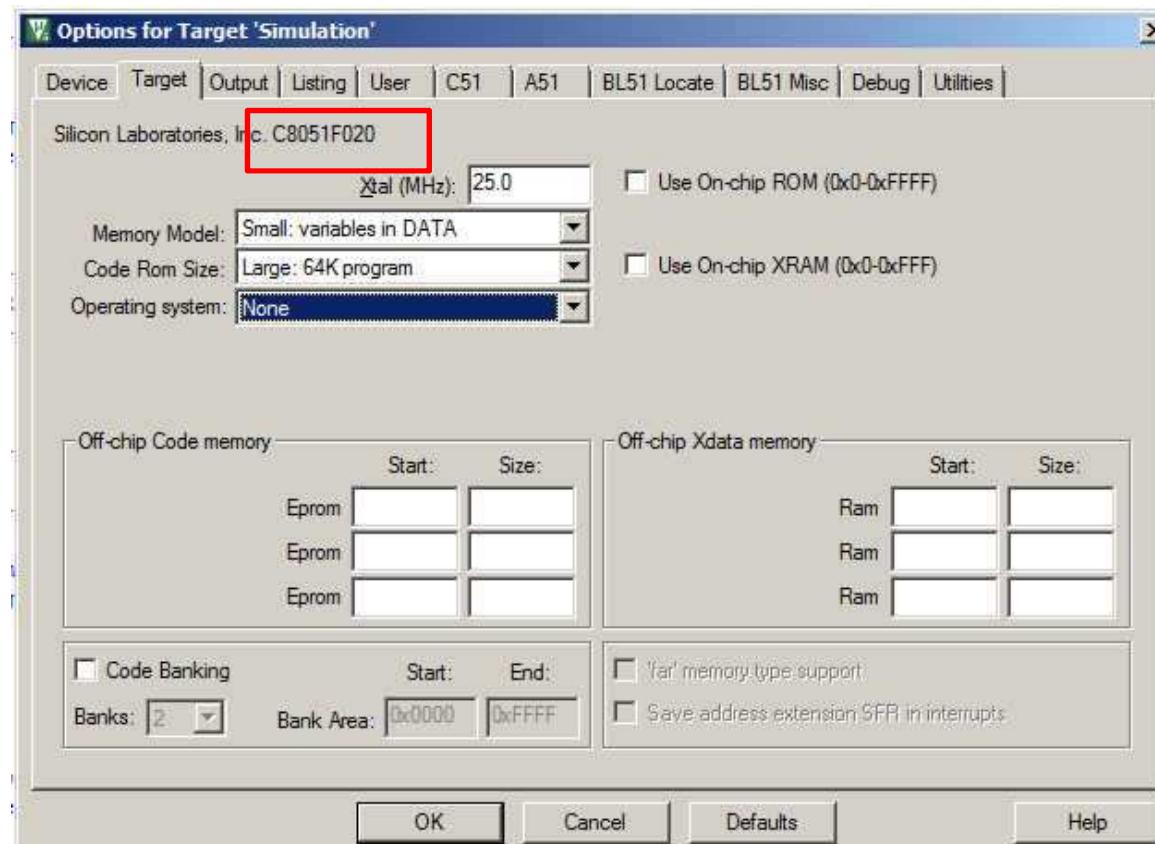
Prise en charge des multiples espaces mémoire du 8051



Rappel: Le 8051 possède des espaces mémoire, code, data, idata, xdata...

Où sont stockés les codes, les constantes et les données?

Gestion de la mémoire via l'environnement de développement



→ Sous microvision, cette configuration est stockée dans le fichier uvproj

Les modèles de gestion mémoire



Modèle SMALL (modèle par défaut)

- Par défaut, toutes les variables sont placées dans la mémoire interne (data, idata) ainsi que la pile.
- **Petite** mémoire, mais rapidité des temps d'accès

Modèle LARGE

- Par défaut, toutes les variables sont placées dans la mémoire externe (xdata). La pile matérielle reste en idata.
- Mémoire plus importante, mais lenteur des temps d'accès

Types de mémoire explicitement définis



Ajout de spécificateurs à la déclaration des variables

Memory Type	Description
code	Program memory (64 KBytes); accessed by opcode MOVC @A+DPTR.
data	Directly addressable internal data memory; fastest access to variables (128 bytes).
idata	Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes).
bdata	Bit-addressable internal data memory; supports mixed bit and byte access (16 bytes).
xdata	External data memory (64 KBytes); accessed by opcode MOVX @DPTR.
far	Extended RAM and ROM memory spaces (up to 16MB); accessed by user defined routines or specific chip extensions (Philips 80C51MX, Dallas 390).
pdata	Paged (256 bytes) external data memory; accessed by opcode MOVX @Rn.

As with the `signed` and `unsigned` attributes, you may include memory type specifiers in the variable declaration. For example:

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];
float idata x,y,z;
unsigned int pdata dimension;
unsigned char xdata vector[10][4][4];
char bdata flags;
```

Allocation de la mémoire (Keil)



Extension	Memory Type	Related ASM
data	Directly-addressable data memory (data memory addresses 0x00-0x7F)	MOV A, 07Fh
idata	Indirectly-addressable data memory (data memory addresses 0x00-0xFF)	MOV R0, #080h MOV A, @R0
xdata	External data memory	MOVX @DPTR
code	Program memory	MOVC @A+DPTR

La gestion mémoire optimale



- **Modèle SMALL**
- **Déclarations explicites des constantes en CODE et des variables de grande taille en XDATA**

Types de données

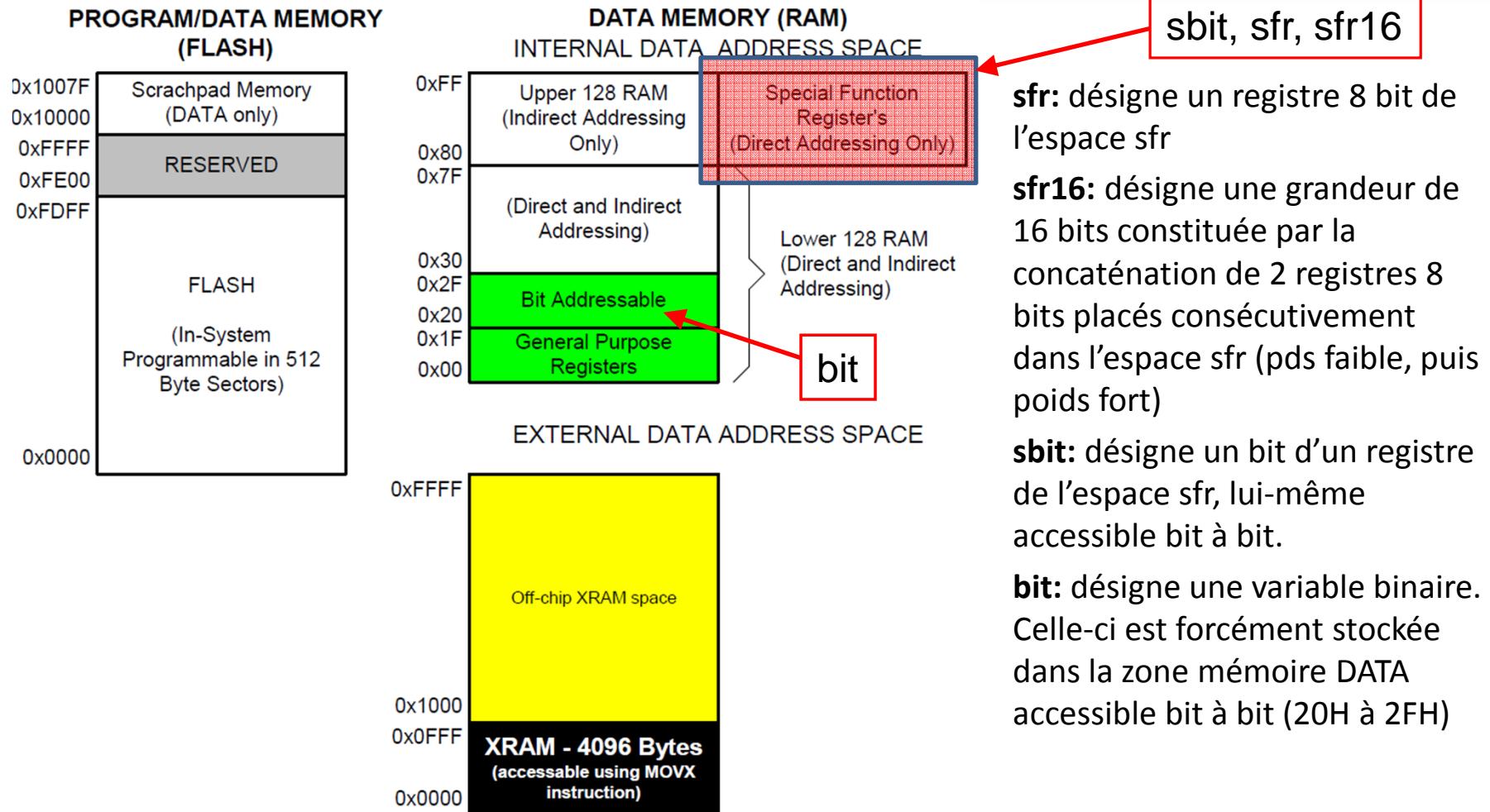
Data Types	Bits	Bytes	Value Range
bit	1		0 to 1
signed char	8	1	-128 – +127
unsigned char	8	1	0 – 255
enum	8 / 16	1 or 2	-128 – +127 or -32768 – +32767
signed short int	16	2	-32768 – +32767
unsigned short int	16	2	0 – 65535
signed int	16	2	-32768 – +32767
unsigned int	16	2	0 – 65535
signed long int	32	4	-2147483648 – +2147483647
unsigned long int	32	4	0 – 4294967295
float	32	4	$\pm 1.175494E-38$ – $\pm 3.402823E+38$
double	32	4	$\pm 1.175494E-38$ – $\pm 3.402823E+38$
sbit	1		0 or 1
sfr	8	1	0 – 255
sf16	16	2	0 – 65535

Ces types ne sont pas ANSI-C et sont spécifiques au compilateur

sbit, sfr, and sf16: pour l'accès à la zone SFR

Doivent être déclarés en global (ne permettent pas les indirections)

Localisation, bit, sbit, sfr, sfr16....



sfr: désigne un registre 8 bit de l'espace sfr

sfr16: désigne une grandeur de 16 bits constituée par la concaténation de 2 registres 8 bits placés consécutivement dans l'espace sfr (pds faible, puis poids fort)

sbit: désigne un bit d'un registre de l'espace sfr, lui-même accessible bit à bit.

bit: désigne une variable binaire. Celle-ci est forcément stockée dans la zone mémoire DATA accessible bit à bit (20H à 2FH)

Les déclarations SFR16

F8	SPI0CN	PCA0H	PCA0CPH0	PCA0CPH1	PCA0CPH2	PCA0CPH3	PCA0CPH4	WDTCN
F0	B	SCON1	SBUF1	SADDR1	TL4	TH4	EIP1	EIP2
E8	ADC0CN	PCA0L	PCA0CPL0	PCA0CPL1	PCA0CPL2	PCA0CPL3	PCA0CPL4	RSTSRC
E0	ACC	XBR0	XBR1	XBR2	RCAP4L	RCAP4H	EIE1	EIE2
D8	PCA0CN	PCA0MD	PCA0CPM0	PCA0CPM1	PCA0CPM2	PCA0CPM3	PCA0CPM4	
D0	PSW	REF0CN	DAC0L	DAC0H	DAC0CN	DAC1L	DAC1H	DAC1CN
C8	T2CON	T4CON	RCAP2L	RCAP2H	TL2	TH2		SMB0CR
C0	SMB0CN	SMB0STA	SMB0DAT	SMB0ADR	ADC0GTL	ADC0GTH	ADC0LTL	ADC0LTH
B8	IP	SADEN0	AMX0CF	AMX0SL	ADC0CF	P1MDIN	ADC0L	ADC0H
B0	P3	OSCXCN	OSCI0CN			P74OUT†	FLSCL	FLACL
A8	IE	SADDR0	ADC1CN	ADC1CF	AMX1SL	P3IF	SADE1	EMI0CN
A0	P2	EMI0TC		EMI0CF	P0MDOUT	P1MDOUT	P2MDOUT	P3MDOUT
98	SCON0	SBUF0	SPI0CFG	SPI0DAT	ADC1	SPI0CKR	CPT0CN	CPT1CN
90	P1	TMR3CN	TMR3RLL	TMR3RLH	TMR3L	TMR3H	P7†	
88	TCON	TMOD	TL0	TL1	TH0	TH1	CKCON	PSCTL
80	P0	SP	DPL	DPH	P4†	P5†	P6†	PCON
	0(8)	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)
	(bit addressable)							

Sfr16 possibles

Le fichier C8051F020.h

```

/* BYTE Registers */
sfr P0      = 0x80; /* PORT 0
sfr SP      = 0x81; /* STACK POINTER
sfr DPL     = 0x82; /* DATA POINTER - LOW BYTE
sfr DPH     = 0x83; /* DATA POINTER - HIGH BYTE
sfr P4      = 0x84; /* PORT 4
sfr P5      = 0x85; /* PORT 5
--> P6      = 0x86; /* PORT 6

/* IE 0xA8 */
sbit EA     = IE ^ 7;          /* GLOBAL INTERRUPT ENABLE */
sbit ET2    = IE ^ 5;          /* TIMER 2 INTERRUPT ENABLE */
sbit ES0    = IE ^ 4;          /* UART0 INTERRUPT ENABLE */
sbit ET1    = IE ^ 3;          /* TIMER 1 INTERRUPT ENABLE */
sbit EX1    = IE ^ 2;          /* EXTERNAL INTERRUPT 1 ENABLE */
sbit ET0    = IE ^ 1;          /* TIMER 0 INTERRUPT ENABLE */
sbit EX0    = IE ^ 0;          /* EXTERNAL INTERRUPT 0 ENABLE */

```

Des types spécifiques pour la manipulations des registres SFR: SFR, SFR16 et SBIT
Déclarés forcément en variables globales

Déclaration sfr16 (pas dans C8051F020.h)

```

sfr16 RCAP2 = 0xCA; //RCAP2L 0XCA,
//RCAP2H 0xCB

```

Exemple de manipulation de registre et de bit



SFR:

P4 = 0x22; (ASM: Mov 84H,#22H)

Sbit:

EA=1; (ASM: setb IE.7)

Bit:

Bit Flag0;
Flag0 = 0; (ASM CLR xx.yH)

SFR16:

sfr16 RCAP2 = 0xCA; //RCAP2L 0XCA, RCAP2H 0xCB
RCAP2 = 1234H; (ASM: MOV 0CAH,#34H
 MOV 0CBH,#12H)

Utilisation des entrés/sorties et des registres de configurations comme des variables



```
sbit LED = P1^6; /* now the functions may be written to use  
this location */  
sfr16 T2 = 0xCC /* Timer 2: T2L 0CCh T2H 0CDH */  
  
void main (void)  
{ /* forever loop, toggling pin 0 of port 1 */  
while (1==1)  
{ LED = !LED;  
T2 = 0;  
delay (500); /* wait 500 microseconds */  
}  
}
```

Le problème des fonctions mathématiques

Utilisation de calcul en réel (double, float ...) très gourmande en absence d'un coprocesseur arithmétique ou d'un processeur spécialisé

Solution : Se limiter à des entiers si possible non signés en fixant des limites inférieures, supérieures

Remarque: le type char peut être utilisé pour désigner un entier 8 bits

```
float time_sec;  
  
time_sec = 2,22;
```

Variable sur 4 octets

Utilisation d'une librairie de calculs en flottant

```
unsigned int time_msec;  
  
time_msec = 2220;
```

Variable sur 2 octets

```
unsigned char time_ctsec;  
  
time_ctsec = 222;
```

Variable sur 1 octet



Attention aux valeurs limites de la variable!!

Opérateurs logiques bit à bit



Operator	Description
&	Bitwise AND
	Bitwise OR (inclusive OR)
^	Bitwise XOR (exclusive OR)
<<	Left shift
>>	Right shift
~	One's complement

Utile pour les opérations de masquage (Manipulation de bits à l'intérieur de registres non accessibles bit à bit)

```
EIE2 = EIE2 & 0xBF; // Mise à zéro de EIE2. - Disable UART1 Interrupts  
EIE2 &= ~0x40; // Ecriture plus concise
```

```
TMR3CN = TMR3CN | 0x04; // Mise à un de TMR3CN.2 - Start Timer3  
TMR3CN |= 0x04; // Ecriture plus concise
```

Accès « octet » à l'intérieur d'un entier « Int » ou « long »



- Il est souvent nécessaire d'accéder individuellement aux différents octets à l'intérieur d'une variable de type INT ou LONG
- ‘C’ permet de le faire **efficacement** en utilisant une construction de type UNION
 - Tous les membres d'une UNION résident dans la même zone mémoire. Il y a donc plusieurs moyens d'accéder à la même zone mémoire

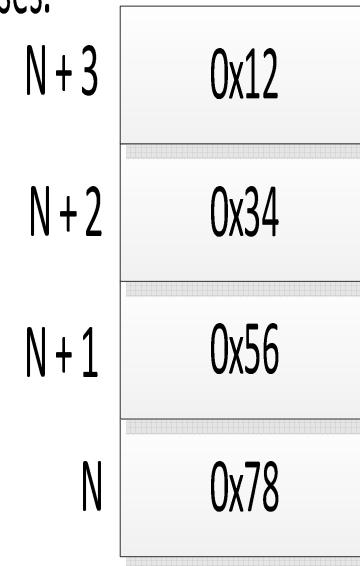
Accès « octet » à l'intérieur d'un Int ou d'un long - Exemple

```
typedef union {
    long t_long;
    int t_int[2];
    char t_char[4];
}tst_ok;

int val1,val2;
char val_c[4];
tst_ok to_test;

to_test.t_long=0x12345678;
val1= to_test.t_int[1]; // val1 = 0x1234
val2= to_test.t_int[0]; // val2 = 0x5678
val_c[0]=to_test.t_char[0]; // val_c[0] = 0x78
```

Contenu Mémoire
Adresses: Stockage en Little Endian



Affectation à une position absolue



- Dans le cas d'un accès à un dispositif externe interfacé dans l'espace mémoire du processeur.
- Utilisation du mot clé **_AT_**
- Selon: <[>*memory_type*<]> **type variable_name _at_ constant;**

```
char xdata text[256] _at_ 0xE000; /* array at xdata 0xE000 */
int xdata i1 _at_ 0x8000; /* int at xdata 0x8000 */
volatile char xdata IO _at_ 0xFFE8; /* xdata I/O port at 0xFFE8 */
```

Classes de mémorisation du langage C (non spécifique à Keil)



extern - `extern data-type name`

- Permet de déclarer une variable globale définie dans un autre fichier source.
- Quand une variable est déclarée extern, elle ne peut pas être initialisée, et il n'y a pas de réservation mémoire

static - `static data-type name <[>=value<]>;`

- En global (à l'extérieur d'une fonction), **static** définit une variable (ou fonction) non utilisable dans une autre unité de compilation (privée)
- En local (à l'intérieur d'une fonction), **static** définit une variable locale persistante. Utile en particulier dans les interruptions. Cette variable est initialisée au démarrage (comme les variables globales) et garde sa valeur entre 2 appels de fonction. Elle n'est pas réinitialisée à chaque entrée de fonction.

Qualificatifs de type (non spécifique à Keil)



const

- Désigne des objets considérés comme des constantes et qui ne peuvent donc être modifiés.
- S'applique à des variables en data, idata et xdata.
- La variable déclarée **const** doit être initialisée à la déclaration
- Souvent utilisé avec des pointeurs pour indiquer que l'on ne modifie pas l'objet pointé.

volatile

- Signale une variable modifiable par une tâche d'arrière-plan (background).
- Chacune des références à la variable en recharge le contenu depuis la mémoire plutôt que de profiter de situations dans lesquelles une copie est placée dans un registre.
- Prévient toute optimisation faite par le compilateur
- Dans le cas des ports d'entrée, la déclaration en volatile est parfois nécessaire

Les pointeurs – 1



Objectif: s'affranchir des divers espaces mémoires

-> Solution: les pointeurs génériques

```
char *s; /* string ptr */  
int *numptr; /* int ptr */
```

Information codée sur 3 octets: 1 octet pour le type de la mémoire, 1 octet pour le poids fort de l'adresse, 1 octet pour le poids faible de l'adresse

Mais: l'exécution est plus lente due à une gestion plus complexe du pointeur

Attention, les pointeurs génériques peuvent être placés dans un espace mémoire spécifié:

```
char * xdata strptr;      /* generic ptr stored in xdata */  
int * data numptr;        /* generic ptr stored in data */  
long * idata varptr;     /* generic ptr stored in idata */
```

Les pointeurs - 2



Pour palier la lenteur des pointeurs génériques: les pointeurs à mémoire spécifique

Ils sont déclarés pour gérer un seul type de mémoire

```
char data *str; /* ptr to string in data */
int xdata *numtab; /* ptr to int(s) in xdata */
long code *powtab; /* ptr to long(s) in code */
```

Mais: utilisation limitée à un seul type de mémoire.

Attention, les pointeurs à mémoire spécifique peuvent être placés dans un espace mémoire spécifié:

```
char data * xdata str;          /* ptr in xdata to data char */
int xdata * data numtab;        /* ptr in data to xdata int */
long code * idata powtab;       /* ptr in idata to code long */
```

Les pointeurs – Exemples de codes

Description	idata Pointer	xdata Pointer	Generic Pointer
Sample Program	char idata *ip; char val; val = *ip;	char xdata *xp; char val; val = *xp;	char *p; char val; val = *p;
8051 Program Code Generated	MOV R0,ip MOV val,@R0	MOV DPL,xp + 1 MOV DPH,xp MOVX A,@DPTR MOV val,A	MOV R1,p + 2 MOV R2,p + 1 MOV R3,p CALL CLDPTR
Pointer Size	1 byte	2 bytes	3 bytes
Code Size	4 bytes	9 bytes	11 bytes + library call
Execution Time	4 cycles	7 cycles	13 cycles

Ajouter de l'assembleur dans du code C (Keil)



```
#pragma asm  
Ajouter votre code ici  
#pragma endasm
```

Les deux grosses difficultés concernent:

- la gestion des arguments d'entrées et de sorties d'une « fonction »
- Attention à la manipulation de registres « réservés » par le compilateur

```
char Get_SW(void) {  
#pragma ASM  
    mov a, P3  
    anl a, #80h  
#pragma ENDASM  
}
```

```
void Set_LED(void) {  
#pragma ASM  
    setb P1.6  
#pragma ENDASM  
}
```

Appel de code Assembleur depuis le C



- Condition impérative: respecter les règles de passage de paramètres imposées par le compilateur.
- Passage de paramètres par registres par défaut (R1-R7) – Maximum 3 paramètres
- Sinon Passage de paramètre par mémoires en utilisant des segments mémoires spécifiques

Fonctions - Passage de paramètres par registre



Arg Number	char, 1-byte ptr	int, 2-byte ptr	long, float	generic ptr
1	R7	R6 & R7 (MSB in R6,LSB in R7)	R4–R7	R1–R3 (Mem type in R3, MSB in R2, LSB in R1)
2	R5	R4 & R5 (MSB in R4,LSB in R5)	R4–R7	R1–R3 (Mem type in R3, MSB in R2, LSB in R1)
3	R3	R2 & R3 (MSB in R2,LSB in R3)		R1–R3 (Mem type in R3, MSB in R2, LSB in R1)

Fonctions – Valeur renvoyée



- La valeur renvoyée est toujours renvoyée par registre

Return Type	Register	Description
bit	Carry Flag	Single bit returned in the carry flag.
char, unsigned char, 1-byte pointer	R7	Single byte type returned in R7.
int, unsigned int, 2-byte ptr	R6 & R7	MSB in R6, LSB in R7.
long, unsigned long	R4-R7	MSB in R4, LSB in R7.
float	R4-R7	32-Bit IEEE format.
generic pointer	R1-R3	Memory type in R3, MSB R2, LSB R1.

Appel de fonction - 1



- Quelques extensions existent lors de l'appel de fonction
 - Déclaration d'une fonction comme fonction d'interruption
 - Choix du banc de registre utilisé
 - Choix du modèle mémoire
 - Spécification de l'aspect “réentrant de la fonction”

Appel de fonction - 2



```
<[>return_type<]> funcname (<[>args<]>) <[>{small | compact | large}<]>
    <[>reentrant<]>
    <[>interrupt x<]>
    <[>using y<]>
```

Where

`return_type` is the type of the value returned from the function. If no type is specified, `int` is assumed.

`funcname` is the name of the function.

`args` is the argument list for the function.

`small` explicitly defines the function uses the [small memory model](#).

`compact` explicitly defines the function uses the [compact memory model](#).

`large` explicitly defines the function uses the [large memory model](#).

`reentrant` indicates that the function is recursive or [reentrant](#).

`interrupt` indicates that the function is an [interrupt function](#).

`x` is the interrupt number.

`using` specifies which register bank the function uses.

`y` is the register bank number.]

Fonctions réentrant



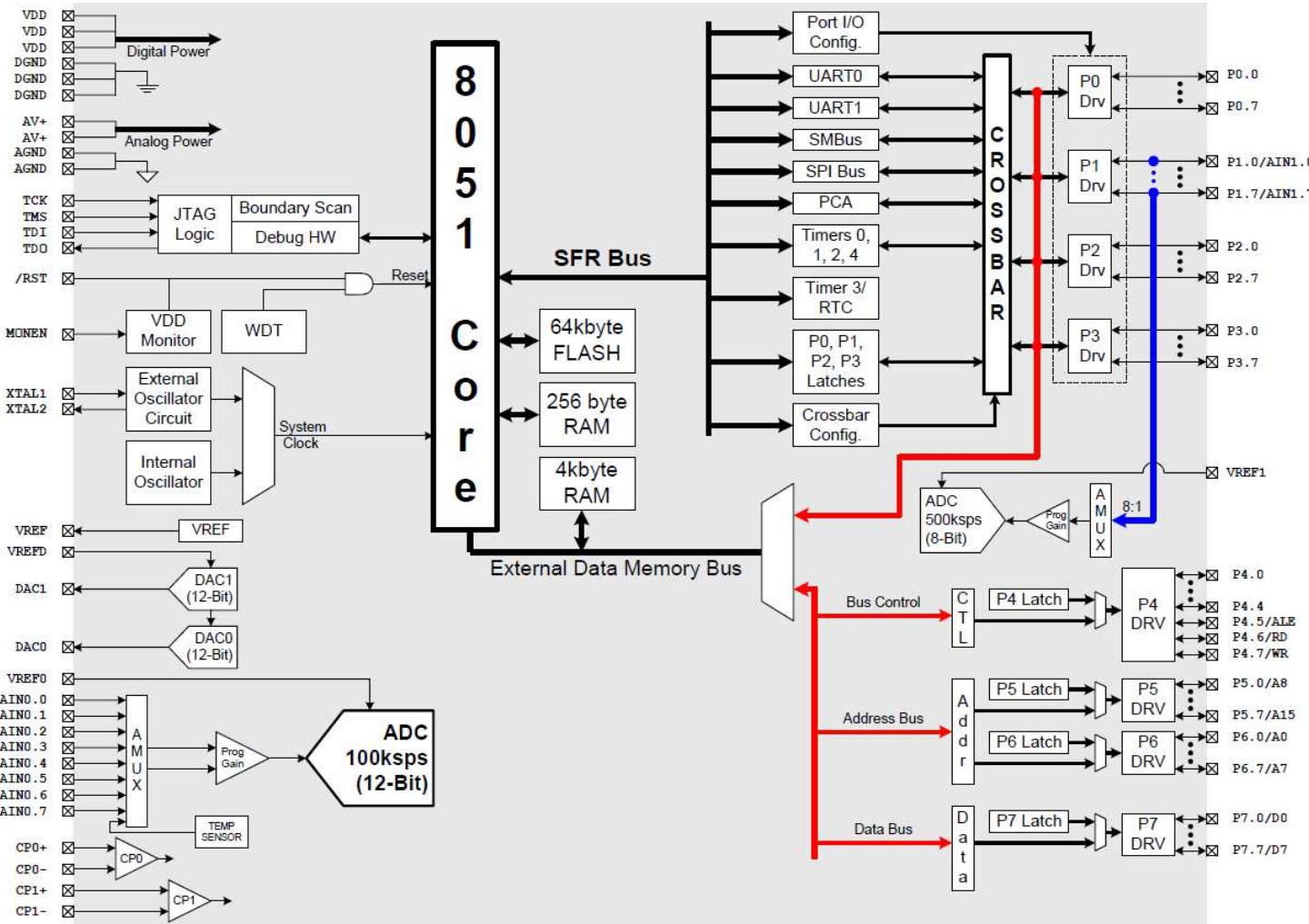
Keil C51 ne respecte pas la spécification du langage C,
les fonctions réentrant (récursives) doivent être définies explicitement

Création d'un espace de mémoire partagé différent pour chaque appel de la fonction.

```
/* Because this function may be called from both the main program */  
/* and an interrupt handler, it is declared as reentrant to */  
/* protect its local variables. */  
int somefunction (int param) reentrant{ ... return (param);}  
  
/* The handler for External interrupt 0, which uses somefunction() */  
void external0_int (void) interrupt 0{ ... somefunction(0);}  
  
/* the main program function, which also calls somefunction() */  
void main (void){ while (1==1) { ... somefunction(); }}
```

Mise en place des interruptions

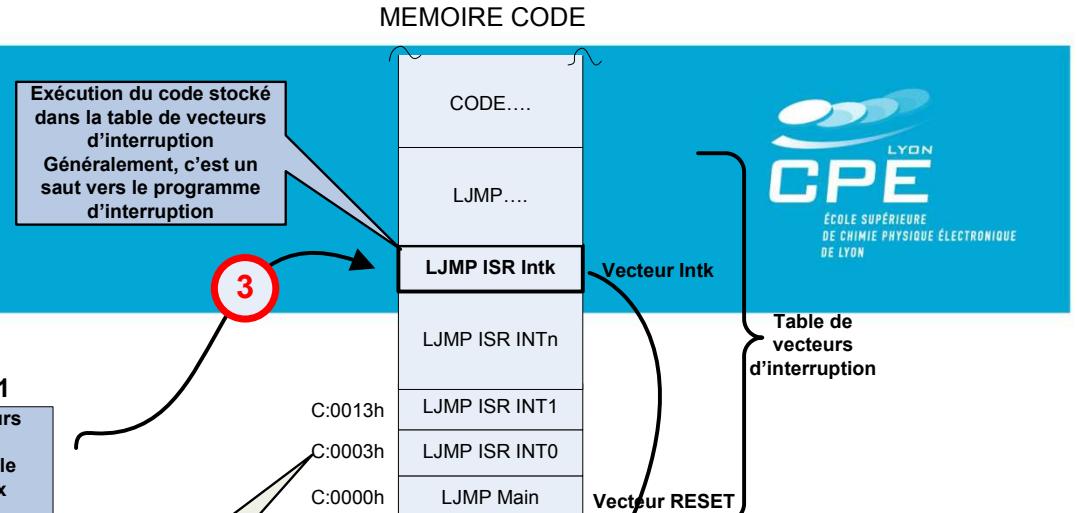
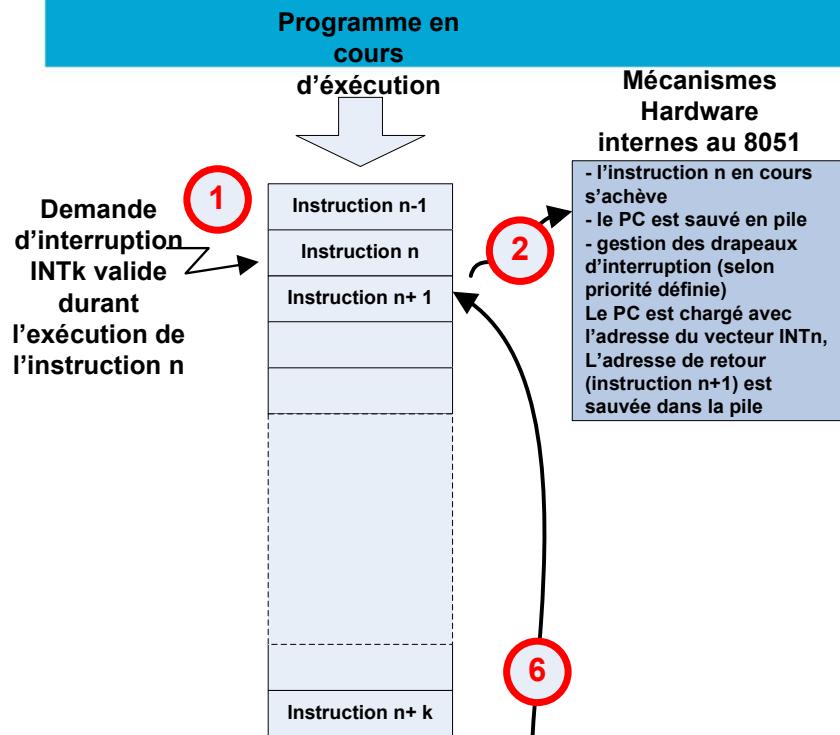
Interruptions Internes et interruptions externes



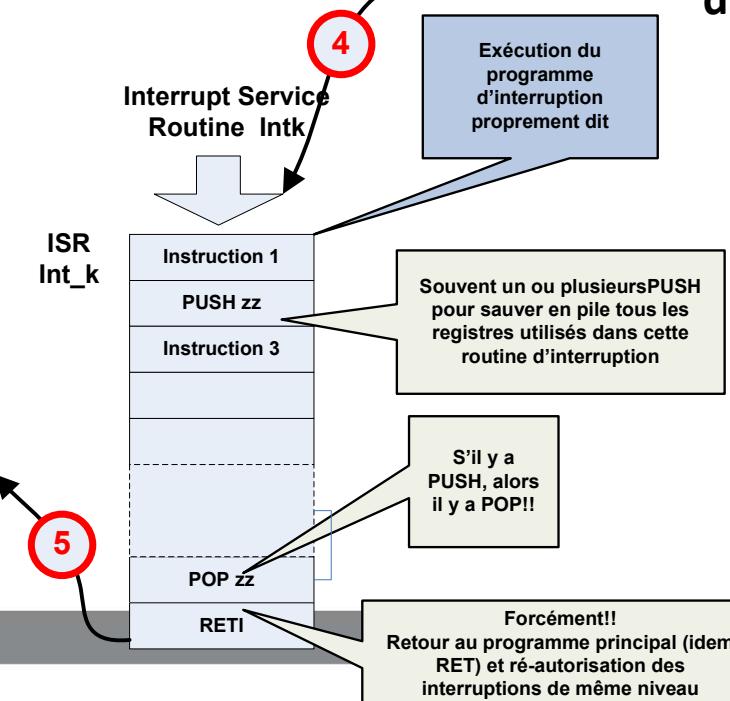
Interruptions internes: produites par les périphériques internes

Interruptions externes: produites par des signaux extérieurs (associés à des événements) connectés sur des entrées spécifiques d'interruption

Rappel: Cycle complet du déroulement d'une interruption dans un 8051



Mise en place d'une Interruption dans le 8051F020



NB: PC = Program Counter, autrement dit, le pointeur de programme

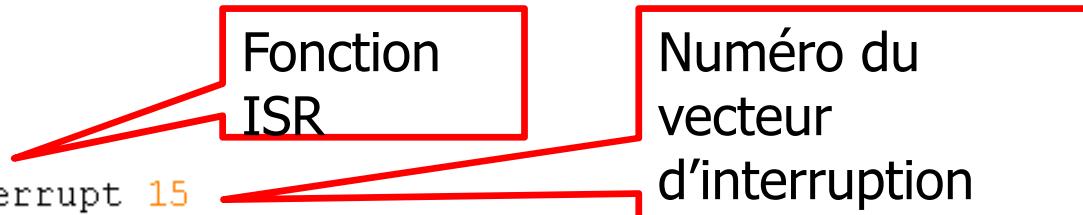
Procédure de mise en place d'une interruption en assembleur



1. Ecrire la routine d'interruption ISR (*Interrupt Service Routine*), à la manière d'un sous-programme. Par contre, cette routine doit obligatoirement se terminer par une instruction « **RETI** » au lieu de « **RET** ».
2. Placer dans la table de vecteurs d'interruption, à l'adresse réservée pour ce vecteur d'interruption, une ligne de code permettant le saut inconditionnel (type **JMP**) vers la routine ISR de traitement de l'interruption.
3. Configurer le périphérique pour qu'il soit en mesure de produire des demandes d'interruption au moment souhaité.
4. Configurer le bit (*Enable Flag*) dans un des registres de validation d'interruption (IE, EIE1 et EIE2) pour autoriser cette interruption.
5. Choisir le niveau de priorité donné à cette interruption par configuration du bit adéquat (*Priority Control*) dans un des registres de gestion des priorités (IP, EIP1 et EIP2).
6. Pour terminer, autoriser la prise en charge globale des interruptions en validant le bit EA du registre IE.

Mise en place d'une « Interrupt Service Routine (ISR) » en C Keil

- Les étapes 1 et 2 sont faites lors de la déclaration de la fonction d'interruption



```
void ADC0_ISR (void) interrupt 15
{
    static unsigned int_dec=INT_DEC;      // integrate/decimate counter
    .....                                     // we post a new result when
    .....                                     // int_dec = 0
    static long accumulator=0L;             // here's where we integrate the
    .....                                     // ADC samples
```

- Les étapes 3 à 6 sont à coder...

Table 12.4. Interrupt Summary

Interrupt Source	Interrupt Vector	Priority Order	Pending Flag	Bit addressable?	Cleared by HW?	Enable Flag	Priority Control
Reset	0x0000	Top	None	N/A	N/A	Always Enabled	Always Highest
External Interrupt 0 (/INT0)	0x0003	0	IE0 (TCON.1)	Y	Y	EX0 (IE.0)	PX0 (IP.0)
Timer 0 Overflow	0x000B	1	TF0 (TCON.5)	Y	Y	ET0 (IE.1)	PT0 (IP.1)
External Interrupt 1 (/INT1)	0x0013	2	IE1 (TCON.3)	Y	Y	EX1 (IE.2)	PX1 (IP.2)
Timer 1 Overflow	0x001B	3	TF1 (TCON.7)	Y	Y	ET1 (IE.3)	PT1 (IP.3)
UART0	0x0023	4	RI0 (SCON0.0) TI0 (SCON0.1)	Y		ES0 (IE.4)	PS0 (IP.4)
Timer 2 Overflow (or EXF2)	0x002B	5	TF2 (T2CON.7)	Y		ET2 (IE.5)	PT2 (IP.5)
Serial Peripheral Interface	0x0033	6	SPIF (SPI0CN.7)	Y		ESPI0 (EIE1.0)	PSPI0 (EIP1.0)
SMBus Interface	0x003B	7	SI (SMB0CN.3)	Y		ESMB0 (EIE1.1)	PSMB0 (EIP1.1)
ADC0 Window Comparator	0x0043	8	AD0WINT (ADC0CN.2)	Y		EWADC0 (EIE1.2)	PWADC0 (EIP1.2)
Programmable Counter Array	0x004B	9	CF (PCA0CN.7) CCFn (PCA0CN.n)	Y		EPCA0 (EIE1.3)	PPCA0 (EIP1.3)
Comparator 0 Falling Edge	0x0053	10	CP0FIF (CPT0CN.4)			ECP0F (EIE1.4)	PCP0F (EIP1.4)
Comparator 0 Rising Edge	0x005B	11	CP0RIF (CPT0CN.5)			ECP0R (EIE1.5)	PCP0R (EIP1.5)
Comparator 1 Falling Edge	0x0063	12	CP1FIF (CPT1CN.4)			ECP1F (EIE1.6)	PCP1F (EIP1.6)
Comparator 1 Rising Edge	0x006B	13	CP1RIF (CPT1CN.5)			ECP1R (EIE1.7)	PCP1F (EIP1.7)
Timer 3 Overflow	0x0073	14	TF3 (TMR3CN.7)			ET3 (EIE2.0)	PT3 (EIP2.0)
ADC0 End of Conversion	0x007B	15	AD0INT (ADC0CN.5)	Y		EADC0 (EIE2.1)	PADC0 (EIP2.1)
Timer 4 Overflow	0x0083	16	TF4 (T4CON.7)			ET4 (EIE2.2)	PT4 (EIP2.2)
ADC1 End of Conversion	0x008B	17	AD1INT (ADC1CN.5)			EADC1 (EIE2.3)	PADC1 (EIP2.3)
External Interrupt 6	0x0093	18	IE6 (P3IF.5)			EX6 (EIE2.4)	PX6 (EIP2.4)
External Interrupt 7	0x009B	19	IE7 (P3IF.6)			EX7 (EIE2.5)	PX7 (EIP2.5)
UART1	0x00A3	20	RI1 (SCON1.0) TI1 (SCON1.1)			ES1	PS1
External Crystal OSC Ready	0x00AB	21	XTLVLD (OSCXCN.7)			EXVLD (EIE2.7)	PXVLD (EIP2.7)

Interruptions dans le 8051F020



- Registres:
 - Validation Registres de périphériques
 - Autorisation IE + EIE1 + EIE2
 - Priorité IP + EIP1 + EIP2

Interruptions dans le 8051F020



Table 12.4. Interrupt Summary

Interrupt Source	Interrupt Vector	Priority Order	Pending Flag	Bit addressable?	Cleared by HW?	Enable Flag	Priority Control
Reset	0x0000	Top	None	N/A	N/A	Always Enabled	Always Highest
External Interrupt 0 (/INT0)	0x0003	0	IE0 (TCON.1)	Y	Y	EX0 (IE.0)	PX0 (IP.0)
Timer 0 Overflow	0x000B	1	TF0 (TCON.5)	Y	Y	ET0 (IE.1)	PT0 (IP.1)
External Interrupt 1 (/INT1)	0x0013	2	IE1 (TCON.3)	Y	Y	EX1 (IE.2)	PX1 (IP.2)
Timer 1 Overflow	0x001B	3	TF1 (TCON.7)	Y	Y	ET1 (IE.3)	PT1 (IP.3)
UART0	0x0023	4	RI0 (SCON0.0) TI0 (SCON0.1)	Y		ES0 (IE.4)	PS0 (IP.4)

Fichiers Configurables: Gestion des Entrées/Sorties



Basic I/O

The following files contain the source code for the low-level stream I/O routines. When you use µVision IDE, you can simply add the modified versions to the project.

C Source File Description

PUTCHAR.C Used by all stream routines that output characters. You may adapt this routine to your individual hardware (for example, LCD or LED displays).

The default version outputs characters via the serial interface. An **XON/XOFF** protocol is used for flow control. Line feed characters ('\n') are converted into carriage return/line feed sequences ('\r\n').

GETKEY.C Used by all stream routines that input characters. You may adapt this routine to your individual hardware (for example, matrix keyboards). The default version reads a character via the serial interface. No data conversions are performed.

Source: Documentation Microvision Keil

Fichiers Configurables: Allocation dynamique de mémoire



Memory Allocation

The following files contain the source code for the memory allocation routines.

C Source File Description

CALLOC.C This file contains the source code for the [calloc](#) library routine.

This routine allocates memory for an array from the memory pool.

FREE.C This file contains the source code for the [free](#) library routine.

This routine returns a previously allocated memory block to the memory pool.

INIT_MEM.C This file contains the source code for the [init mempool](#) library routine.

This routine allows you to specify the location and size of a memory pool from which memory may be allocated using the **malloc**, **calloc**, and **realloc** routines.

MALLOC.C This file contains the source code for the [malloc](#) library routine. This routine allocates memory from the memory pool.

REALLOC.C This file contains the source code for the [realloc](#) library routine. This routine resizes a previously allocated memory block.

Codage d'une application « système embarqué »

Sans utiliser de système
d'exploitation

... tout en donnant l'impression de gérer plusieurs tâches
simultanément...

Modèle Background

```
/* Background Boucle infinie
Programme principal */
void main (void)
{
    Initialisation;
    FOREVER { Handle events end treatments }
}
```

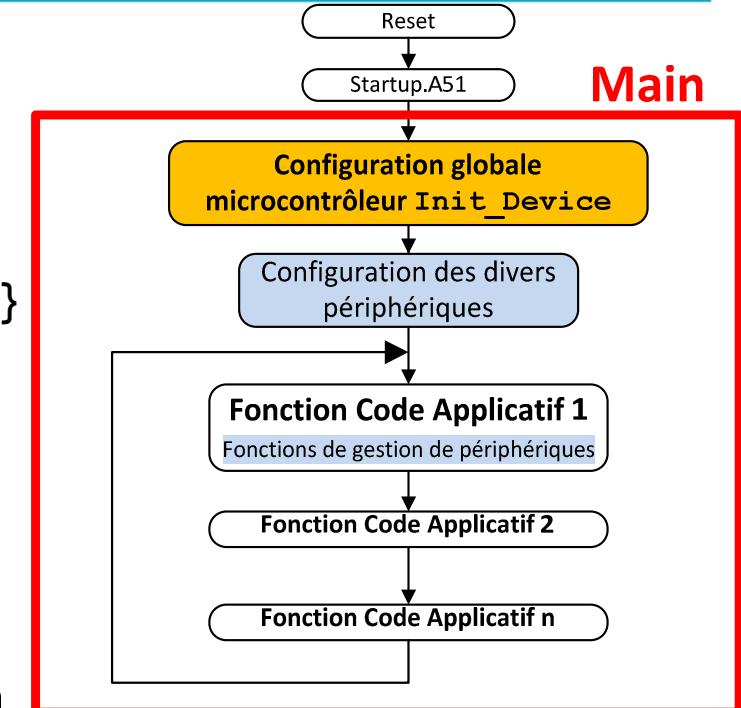
Fonctionnement par scrutation

Boucle infinie dans le programme principal

Le système ne sait pas gérer la sortie du main

Dans le cas d'un PC avec un système d'exploitation, un programme est chargé avant son exécution, le chargeur reprend la main à la fin de l'exécution.

Sur un PC il existe aussi un « main unique » au boot de la machine ...



Exemple de développement en Background



Hypothèse de base: il n'y a pas de scrutation d'attente bloquante
Envisageable pour des applications élémentaires.

- Mais, que se passe t'il si un sous-programme met plus de temps que prévu pour se terminer?
- De grande difficultés pour gérer des évènements sur des échelles de temps différentes

D'une manière globale: impossible de gérer de cette manière une application qui nécessite une connaissance précise du temps.

Exemple d'application: Boitier de gestion de capteurs d'alarme.

Modèle Foreground/Background (Interruptions)

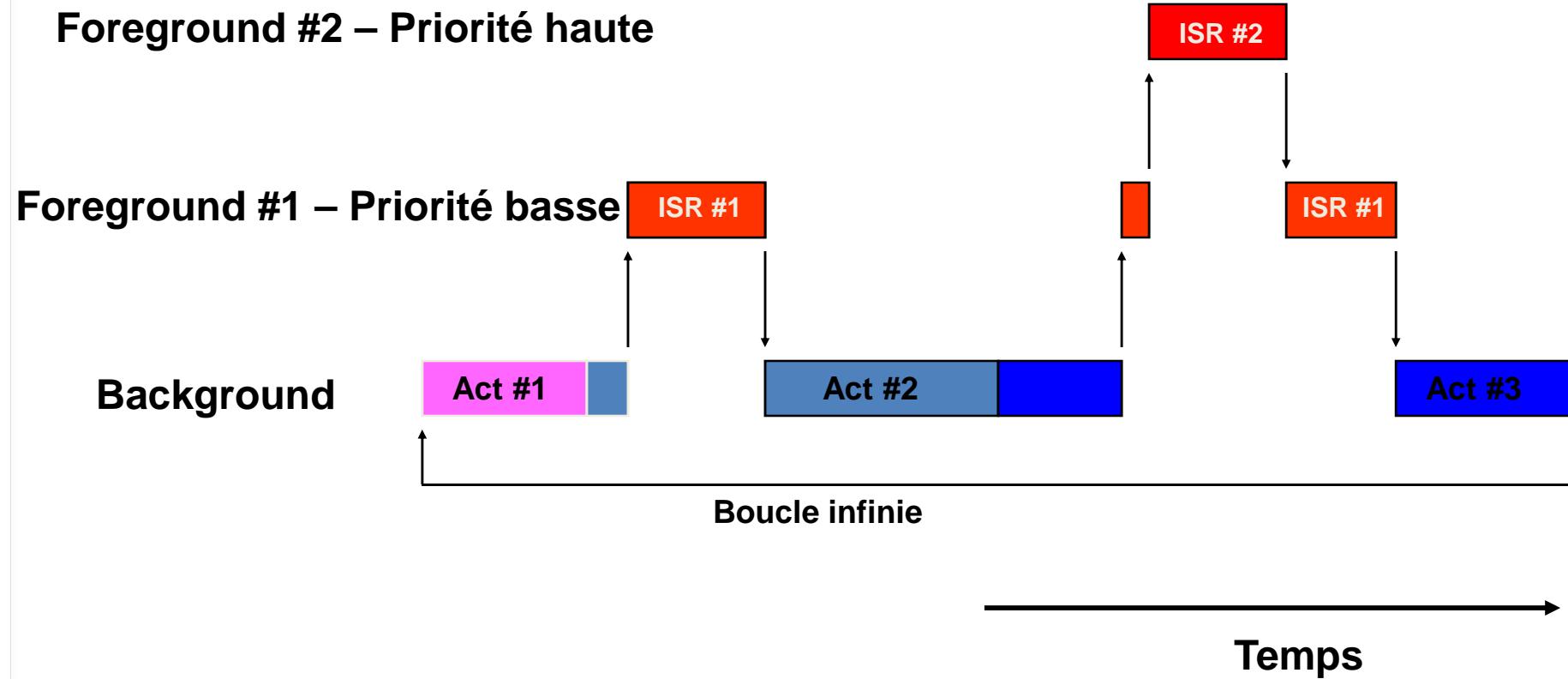


La majeure partie des applications embarquées est composé d'un main et d'un ensemble de routines d'interruptions

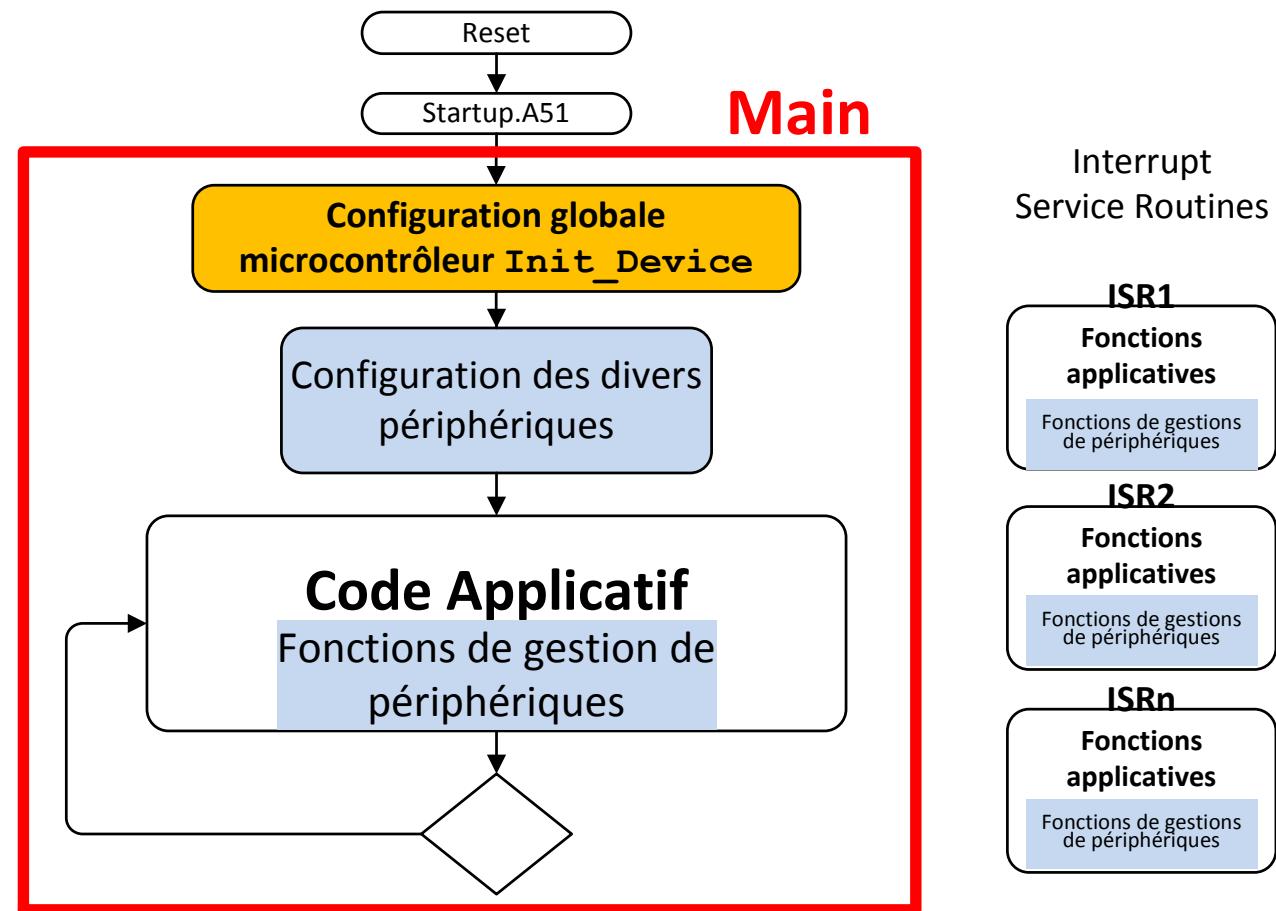
Les interruptions sont utilisées pour se dispenser des tâches de scrutation dans la partie background

```
/* Background Boucle infinie          /* Foreground
   Programme principal */             Gestion par interruption*/
void main (void)                   ISR (void)
{
    Initialisation;                {
    FOREVER {                      Handle asynchronous event;
        Handle events end treatments } }
    }                                ISR2 (void)
}                                    {
    Handle asynchronous event;     ...
}
```

Système Foreground/Background



Système Foreground/Background



Interruptions – Notions générales (Rappel)



Répondre rapidement aux événements extérieurs au processeur

- Les évènements extérieurs sont convertis par le matériel en interruptions qui se propagent dans le système.
 - Les routines d'interruption (ISRs Interrupt Service Routines) gèrent les interruptions
 - Certains systèmes (matériellement ou par logiciel ...) permettent la préemption des ISR par des ISR plus prioritaires
- Les ISRs se doivent se terminer le plus rapidement possible.
- Tous les évènements ne peuvent pas être associés à une interruption (mais toujours possible en rajoutant des composants électroniques)
- Une routine d'interruption peut être associée à plusieurs événements différents (il faut trouver le bon)

Echange d'informations et synchronisations



Utilisation « intensive » de variables globales (ne pose pas de problème en l'absence de mécanismes de protection mémoire)

- Echange de données entre les activités/fonctions
- Synchronisation d'une activité entre le main et les interruptions
- Utilisation des variables « statics » dans les interruptions (à la fois persistantes et privées)

Modèle Foreground/Background orienté gestion des activités



```
/* Background Boucle infini
Programme principale */
void main (void)
{
    Initialisation;
    FOREVER {
        Handle activity1
        input1
        treatment1
        output1
        Handle activity2
        Handle activity3
        ...
        Handle user interface;
        Handle communication requests;
        Other...
    }
}
```

```
/* Foreground
Gestion par interruption*/
ISR1 (void)
{
    Handle asynchronous event;
}
ISR2 (void)
{
    Handle asynchronous event;
}
...
Attention à ne pas bloquer le
CPU dans le code associé à
une activité, il faut penser à
« synchroniser » un traitement
dans le cycles à venir ...
```

Modèle Foreground/Background orienté gestion des entrées/sorties

```
/* Background Boucle infini
Programme principale */
void main (void)
{
    Initialisation;
    FOREVER {
        Read analog inputs (A1...An) ;
        Read discrete inputs ;
        Perform monitoring functions;
        Perform control functions;
        Update analog outputs (A1...An) ;
        Update discrete outputs;
        Scan keyboard;
        Handle user interface;
        Update display;
        Handle communication requests;
        Other...
    }
}
```

```
/* Foreground
Gestion par interruption*/
ISR (void)
{
    Handle asynchronous event;
}
ISR2 (void)
{
    Handle asynchronous event;
}
...
...
```

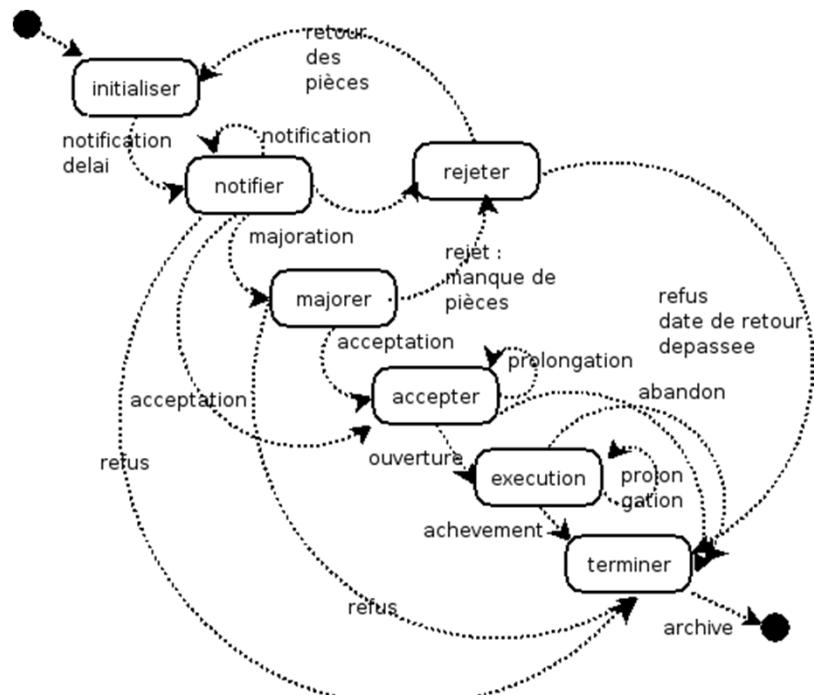
Nécessite de savoir décomposer chaque activité sous des formes équivalentes

...

Exécution sous forme d'un diagramme d'état-transition

```
/* Background Boucle infinie
Programme principale */
void main (void)
{
```

```
    Initialisation;
```



```
}
```

```
/* Foreground
Gestion par interruption*/
ISR (void)
{
    Handle asynchronous event;
}

ISR2 (void)
{
    Handle asynchronous event;
}

...
```

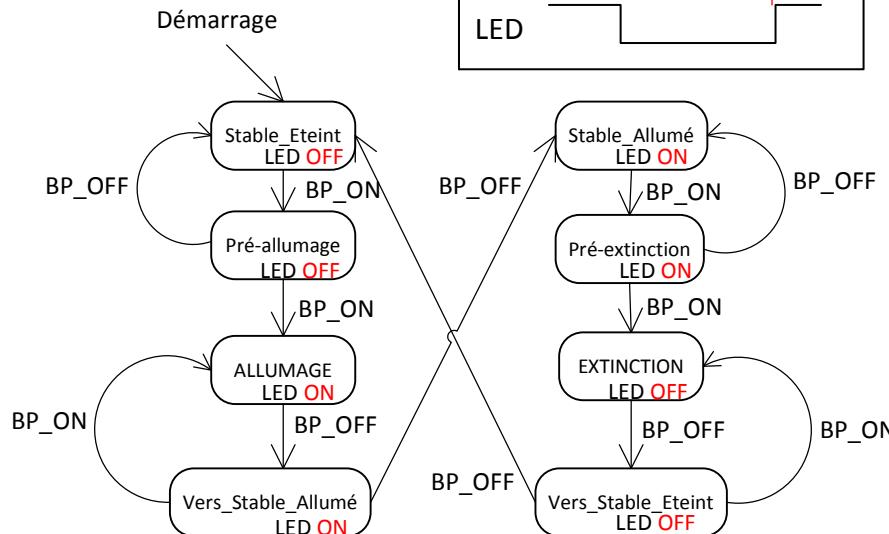
Implémentation d'une machine d'état

Machine d'état pour gérer l'allumage d'une LED

La transition entre les états peut dépendre:

- Du temps (parfois)
- Des entrées du système (rare)
- Du temps et des entrées du système (courant)

Machine d'état – Pilotage d'une LED avec un Bouton Poussoir



```

enum state_FSM_LED {Stable_Eteint,Pre_Allumage,Extinction,Vers_Stable_Eteint};
static enum state_FSM_LED FSM_LED = Stable_Eteint;
enum state_FSM_LED next_state;
bit Read_BP;

Read_BP = BP;

switch(FSM_LED) // Gestion de la machine d'état de Gestion de la LED
    // Prise en compte des rebonds éventuels sur le bouton poussoir
    // Changement des états
{
    case Stable_Eteint:
    {
        if (Read_BP == BP_ON) next_state = Pre_Allumage;
        else next_state = Stable_Eteint;
        break;
    }
    case Pre_Allumage:
    {
        if (Read_BP == BP_ON) next_state = Allumage;
        else next_state = Stable_Eteint;
        break;
    }
    case Extinction:
    {
        if (Read_BP == BP_OFF) next_state = Vers_Stable_Eteint;
        else next_state = Extinction;
        break;
    }
    case Vers_Stable_Eteint:
    {
        if (Read_BP == BP_ON) next_state = Extinction;
        break;
    }
    case Pre_Exstinction:
    {
        if (Read_BP == BP_ON) next_state = Extinction;
        else next_state = Stable_Allume;
        break;
    }
    default:
    {
        next_state = Stable_Eteint;
        break;
    }
}
FSM_LED = next_state;

if (FSM_LED == Stable_Eteint || // Commande de l'état
    FSM_LED == Pre_Allumage ||
    FSM_LED == Extinction ||
    FSM_LED == Vers_Stable_Eteint) LED = LED_OFF;
else LED = LED_ON;

```

Modèle Foreground exclusif

```
/* Background Boucle infinie
Programme principal */
void main (void)
{
    Initialisation;
    TimerConfiguration;
    FOREVER {
        Nothing or
        activity not possible in interruption
        or LPM (Low Power Mode) ...
    }
}
```

Il permet de faire de grosses économies d'énergie

Nécessite de penser intelligemment les traitements dans les interruptions.

```
/* Foreground
Gestion par interruption*/
ISR (void)
{
    Handle asynchronous event;
}
ISR2 (void)
{
    Handle asynchronous event;
}

ISR_timer (void)
{ Handle time
  Handle Activity1
  Handle Activity2
...
}
```

...

A mi-chemin du Background/Foreground et du Foreground Exclusif

Foreground/background system with a low-power sleep mode.

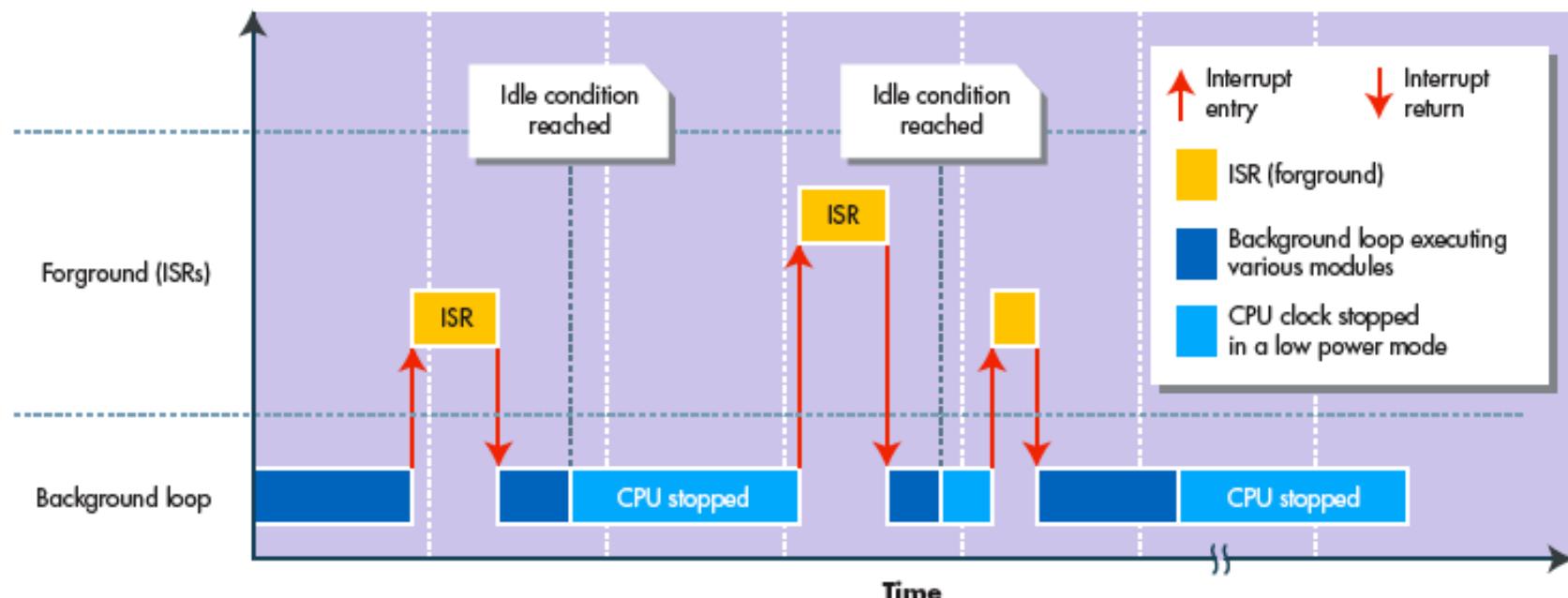


Figure 1

Source: <http://www.embedded.com/design/mcus-processors-and-socs/4007194/Use-an-MCU-s-low-power-modes-in-foreground-background-systems>

Remarques sur la Gestion du temps



Besoins récurrents:

- Récupérer une information temporelle nécessaire au traitement
- Synchroniser un traitement avec le temps
- Attendre un certain temps ou un certain instant... Synchroniser ou attendre...

La gestion du temps:

- Fonctions Temporisation logicielles - *Temporisations < qq µS*
- Fonctions Temporisation pilotées par Timer – *Temporisations précises de longue durée mais limitation liée à la résolution du timer*
- Interruption pilotées par Timer – *Temporisations de longue durée ne requérant pas une résolution temporelle importante*

Structuration du code avec des fonctions

- Intérêts:
 - Divide and conquer ..
 - Limiter la responsabilité et la complexité d'une portion de code
 - Limite les adaptations/modifications à une fonction et pas à l'ensemble du code
 - Simplifier la lecture du code
 - Expliciter les entrées/sorties
 - Eviter le copier/coller dans le code
 - Faciliter la maintenabilité
- Problèmes et solutions :
 - Augmentation de l'utilisation de la « pile » et du temps d'exécution du code.
 - Utilisation possible des macros et des fonctions inline.



Rappel: séparation du code applicatif et du code de configuration et d'utilisation des périphériques (Application indépendante du microcontrôleur)

Structuration du code

Séparation code bas niveau – Code applicatif



Règle impérative: séparation du code applicatif et du code de configuration et d'utilisation des périphériques (Application indépendante du microcontrôleur)

Séparation des codes

- Codes dépendant du processeur:
 - la configuration du microcontrôleur
 - la configuration des périphériques
 - l'utilisation des périphériques
- Codes dépendant de l'application

Utiliser des fichiers source différents.

Certaines familles de microcontrôleurs proposent des bibliothèques logicielles de gestion de périphériques

Conséquence: dans un code applicatif, on ne manipule jamais directement des registres.

Ex: `TF2 = 0;` devient `CLR_Flag_INT_Timer2();`

Structuration du code avec des macros



Utilisation des macros sans paramètre

```
#define nom reste_de_la_ligne
```

Utile aussi bien dans les codes de bas niveau

```
#define sysclk 22118400
```

Que dans les codes applicatifs

```
#define Dist_DCT_PP1 600 // en mm
#define Dist_DCT_PP1_Colis1 (Dist_DCT_PP1 - (Taille_Colis_Type1/2))
```

Objectifs: faciliter les changements de configuration, de paramètres, à agissant à un seul endroit

Foreground / Background Avantages



- Coût logiciel faible
- Besoin en mémoire limité
- Un seul espace mémoire pour les activités
- Peu de problème de parallélisme
- Coût temporel minimal pour les interruptions
- Pas de coût de licence pour un RTOS

Foreground / Background

Désavantages



Background

- Le temps de réponse des activités en background est donné par le temps d'exécution de la boucle
 - Non déterministe, car lié aux conditionnels if for, while
 - Pas très réactif
 - Change avec le code
- Toutes les activités ont la même priorité
 - Un évènement important est traité avec la même priorité qu'un autre.

Foreground

- Beaucoup de limitations sur le code exécutable en ISR (=> une interaction avec le background)
- Le code est plus difficile à maintenir

Foreground / Background de « compromis »



- **Background:** gestion des tâches de priorité basse – Contraintes temporelles nulles
- **Foreground**
 - Interruption Timer Base de temps: Exécution des tâches de priorité élevée – Contraintes temporelles importantes
 - Autres interruptions sur des évènements - pas de traitement – stockage pour traitement dans l'interruption base de temps ou le background.

Sources documentaires



- Datasheet 8051F020
- Documentation du compilateur C51 Keil (Aide en ligne)
- Documentation générique sur le langage C
- Cours « Introduction aux systèmes embarqués » Fabrice JUMEL – CPE Lyon



CONTACT

François JOLY
Domaine Scientifique de la Doua
43, bd du 11 novembre 1918 – Bâtiment Hubert
Curien
B.P. 2077 – 69616 Villeurbanne cedex – France

Tél. : 04 72 43 13 36
francois.joly@cpe.fr www.cpe.fr

membre de UNIVERSITÉ DE LYON

