CPE LYON

LIVE AND DISCOVER

Bases des systèmes embarqués

2 - Langage C et 8051

Version 2022

09/11/2022 12:02

UNIVERSITÉ DE LYON
établissement associé

CPE LYON

LIVE AND DISCOVER

Architecture générique 8051
Rappels

établissement associé UNIVERSITE DE LYON

Caractéristiques générales de l'architecture 8051 (souvenirs du module ELN3….)

8051 = une famille de microcontrôleurs

- Un processeur 8 bits
- Une architecture Harvard (espaces mémoire Code et Données séparé)…
- Un jeu d'instruction CISC
- Plusieurs espaces mémoire accessibles (CODE, XDATA, DATA, SFR, IDATA, BIT….)
- Plusieurs périphériques d'interfaçage

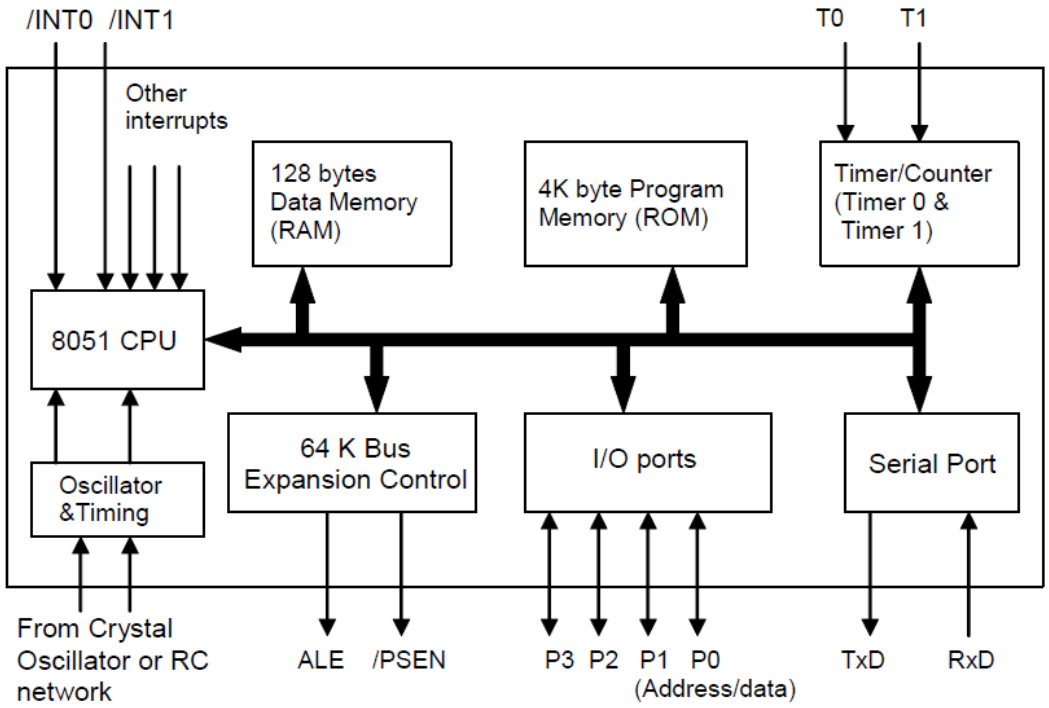
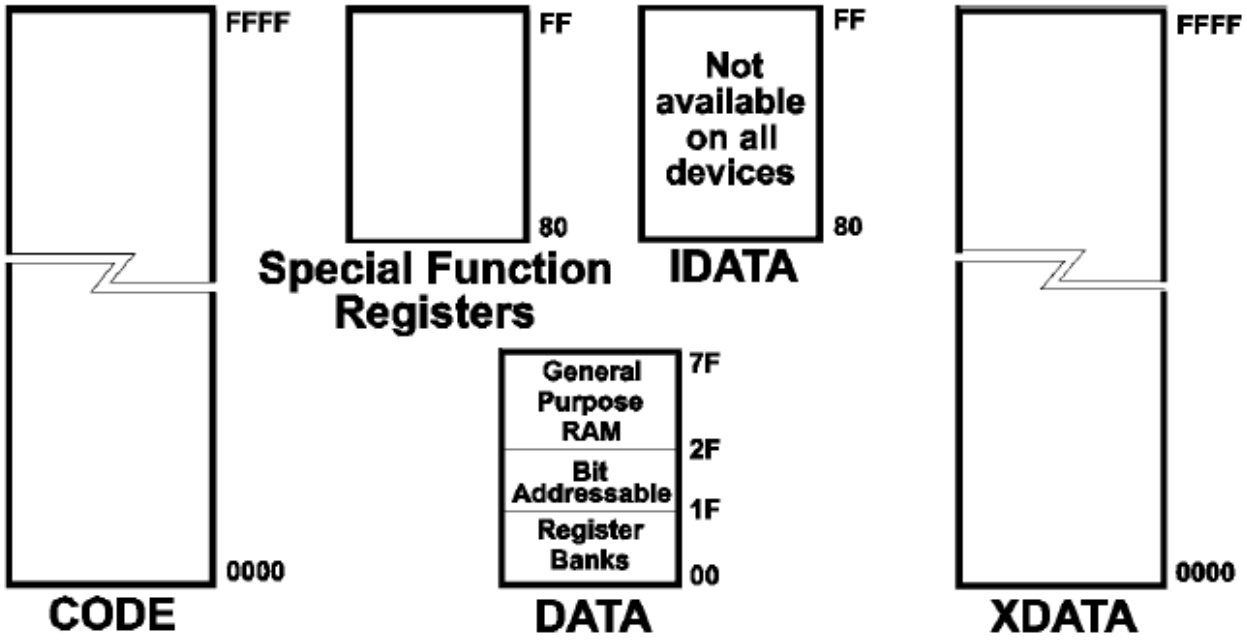


Diagramme bloc d'un 8051 générique

Espaces mémoire du 8051

- Espace CODE: accès lecture seul - contient le code à exécuter (instructions + opérandes)
- Espace XDATA: External (historiquement) DATA – accès R/W – stockage des données - temps d'accès « long »
- Espace DATA: accès R/W – stockage des données – temps d'accès « court »
- Espace IDATA: accès R/W – stockage des données – temps d'accès « moyen »
- Espace SFR (Special Fonction Registers) – Accès aux périphériques d'entrées-sorties



Architecture mémoire 8051

Source: The final Word on the 8051 – Mathew chapman - 1994

Vue détaillée de l'espace mémoire DATA du 8051
Partie basse - adresses 00 à 7FH

- Une zone réservée aux bancs de registre R0-R7
- Une zone accessible bit à bit par l'intermédiaire d'instructions assembleur spécifiques

Espace Data du 8051

Byte Address	Bit Address							
7F	General Purpose RAM							
30								
2F	7F	7E	7D	7C	7B	7A	79	78
2E	77	76	75	74	73	72	71	70
2D	6F	6E	6D	6C	6B	6A	69	68
2C	67	66	65	64	63	62	61	60
2B	5F	5E	5D	5C	5B	5A	59	58
2A	57	56	55	54	53	52	51	50
29	4F	4E	4D	4C	4B	4A	49	48
28	47	46	45	44	43	42	41	40
27	3F	3E	3D	3C	3B	3A	39	38
26	37	36	35	34	33	32	31	30
25	2F	2E	2D	2C	2B	2A	29	28
24	27	26	25	24	23	22	21	20
23	1F	1E	1D	1C	1B	1A	19	18
22	17	16	15	14	13	12	11	10
21	0F	0E	0D	0C	0B	0A	09	08
20	07	06	05	04	03	02	01	00
1F	Bank 3							
18								
17	Bank 2							
10								
0F	Bank 1							
08								
07	Default Register Bank for R0 – R7							
00								

Bit Address

Espace BIT Accessibilité bit à bit

On dispose d'instructions assembleur spécifiques pour manipuler ces bits

Bancs de registre R0-R7

SILICON LABS

CPE LYON LIVE AND DISCOVER

Vue détaillée de l'espace mémoire DATA du 8051
Partie haute - adresses 80H à FFH

- Pour l'accès aux périphériques du 8051, via des registres

Espace SFR du 8051 générique

Byte Address	Bit Address								
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	-	D0	PSW
B8	-	-	-	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	-	-	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	Not bit-addressable								SBUF
98	9F	96	95	94	93	92	91	90	SCON
90	97	96	95	94	93	92	91	90	P1
8D	Not bit-addressable								TH1
8C	Not bit-addressable								TH0
8B	Not bit-addressable								TL1
8A	Not bit-addressable								TL0
89	Not bit-addressable								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	Not bit-addressable								PCON
83	Not bit-addressable								DPH
82	Not bit-addressable								DPL
81	Not bit-addressable								SP
80	87	86	85	84	83	82	81	80	P0

Figure 1.6b Summary of 8051 on-chip data Memory
(Special Function Registers)

SILICON LABS

CPE LYON LIVE AND DISCOVER

Jeu d'instructions assembleur du 8051

La multitude de mode d'adressage pour (par exemple) l'instruction MOV illustre le caractère CISC du jeu d'instruction du 8051

RRC	A	Rotate accumulator right through carry	1	1
SWAP	A	Swap nibbles within the accumulator	1	1
Logic operations				
ANL	A,Rn	AND register to accumulator	1	1
ANL	A,direct	AND direct byte to accumulator	2	2
ANL	A,@Ri	AND indirect RAM to accumulator	1	2
ANL	A,#data	AND immediate data to accumulator	2	2
ANL	direct,A	AND accumulator to direct byte	2	2
ANL	direct,#data	AND immediate data to direct byte	3	3
ORL	A,Rn	OR register to accumulator	1	1
ORL	A,direct	OR direct byte to accumulator	2	2
ORL	A,@Ri	OR indirect RAM to accumulator	1	2
ORL	A,#data	OR immediate data to accumulator	2	2
ORL	direct,A	OR accumulator to direct byte	2	2
ORL	direct,#data	OR immediate data to direct byte	3	3
XRL	A,Rn	Exclusive OR register to accumulator	1	1
XRL	A,direct	Exclusive OR direct byte to accumulator	2	2
XRL	A,@Ri	Exclusive OR indirect RAM to accumulator	1	2
XRL	A,#data	Exclusive OR immediate data to accumulator	2	2
XRL	direct,A	Exclusive OR accumulator to direct byte	2	2
XRL	direct,#data	Exclusive OR immediate data to direct byte	3	3
Boolean variable manipulation				
CLR	C	Clear carry flag	1	1
CLR	bit	Clear direct bit	2	2
SETB	C	Set carry flag	1	1

DJNZ	direct,rel	Decrement direct byte and jump in not zero	3
NOP		No operation	1
Data transfer			
MOV	A,Rn	Move register to accumulator	1
MOV	A,direct*)	Move direct byte to accumulator	2
MOV	A,@Ri	Move indirect RAM to accumulator	1
MOV	A,#data	Move immediate data to accumulator	2
MOV	Rn,A	Move accumulator to register	1
MOV	Rn,direct	Move direct byte to register	2
MOV	Rn,#data	Move immediate data to register	2
MOV	direct,A	Move accumulator to direct byte	2
MOV	direct,Rn	Move register to direct byte	2
MOV	direct,direct	Move direct byte to direct byte	3
MOV	direct,@Ri	Move indirect RAM to direct byte	2
MOV	direct,#data	Move immediate data to direct byte	3
MOV	@Ri,A	Move accumulator to indirect RAM	1
MOV	@Ri,direct	Move direct byte to indirect RAM	2
MOV	@Ri,#data	Move immediate data to indirect RAM	2
MOV	DPTR,#data16	Load data pointer with a 16-bit constant	3
MOVC	A,@A+DPTR	Move code byte relative to DPTR to accumulator	1
MOVC	A,@A+PC	Move code byte relative to PC to accumulator	1
MOVX	A,@Ri	Move external RAM (8-bit addr.) to A	1
MOVX	A,@DPTR	Move external RAM (16-bit addr.) to A	1
MOVX	@Ri,A	Move A to external RAM (8-bit addr.)	1
MOVX	@DPTR,A	Move A to external RAM (16-bit addr.)	1

Extrait du jeu d'instructions du 8051…

SILICON LABS

CPE LYON LIVE AND DISCOVER

Exemple de code assembleur 8051 (durant le module ELN3)

- Beaucoup de manipulation de registres de base tels que A (accumulateur) R0, … R7, DPTR, PSW etc….
- Mise en place de fonctionnalités d'entrées-sorties basiques à l'aide de dispositifs matériels (mise en oeuvres de registres, interfacés dans l'espace mémoire XDATA du 8051)

```
        push AR4
        push AR5
        mov A,R7      ; Récupérateur de l'adresse de début de tableau
        mov R0,A      ; R0 pointeur case de référence
        mov R1,A
        inc R1         ; R1 pointeur de parcours de table = R0+1
        add A,R5
        mov R4,A      ; R4 "pointe" sur la premiere case hors table
BCL1:   mov A,@R0
        clr C
        subb A,@R1    ; Comparaison valeur pointée par R0 avec valeur pointée par R1
        jc  No_exchange ; si [R0] < [R1] alors pas d'échange
        mov A,@R0       ; si [R0] >= [R1]  alors échange
        xch A,@R1       ; échange effectué
        mov @R0,A       ; suite échange
No_exchange:
        inc R1          ; incrément du pointeur parcours de table
        mov A,R4
        cjne A,AR1,BCL1 ; test pointeur parcours de table hors table?
                        ; si non: retour à BCL1
        inc R0          ; incrément du pointeur case de référence
        mov A,R0
        mov R1,A
        inc R1          ; R1 pointeur de parcours de table = R0+1
        mov A,R4
        cjne A,AR1,BCL1 ; test pointeur parcours de table hors table?
                        ; si non: retour à BCL1
        pop AR5
        pop AR4
```

Bilan ELN3

Même si vous ne maitrisez pas toutes les finesses de ce domaine, on peut néanmoins admettre que vous aviez, en fin de module, des compétences de base sur les points suivants:

- Je connais l'architecture interne d'un (petit) processeur.
- J'ai assimilé l'architecture globale d'un système à microprocesseur.
- J'ai assimilé quelques mécanismes de base d'un processeur tels que interruptions, sauts, sous-programmes, etc…
- Je sais coder en assembleur des programmes simples qui permettent de faire des manipulations simples de données.
- Je sais donner la possibilité au processeur de communiquer avec le monde extérieur en fabriquant des ports d'entrée-sorties placés dans un espace mémoire du processeur.

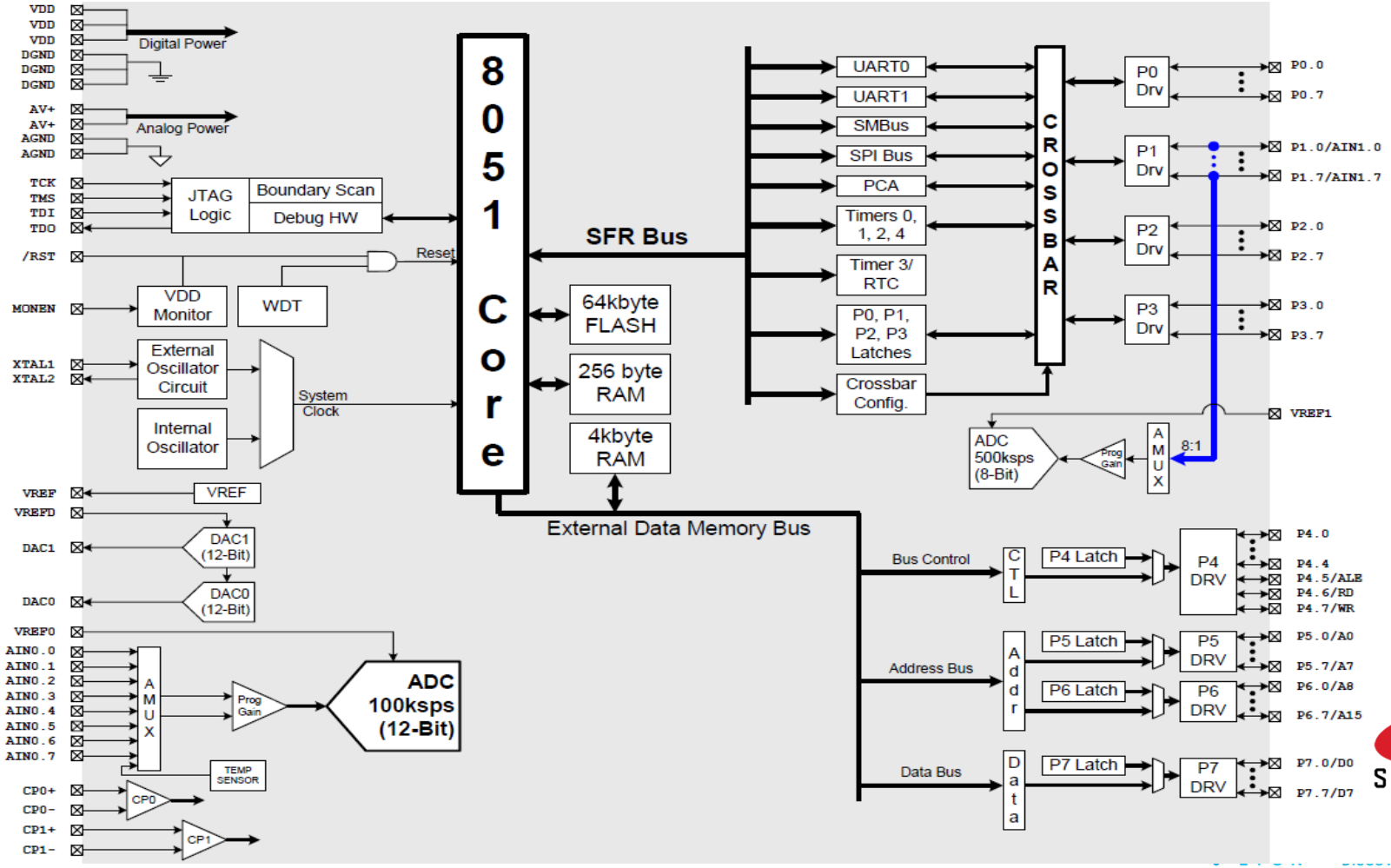
LIVE AND DISCOVER

Le microcontrôleur 8051F020 utilisé en TP ELN3 et BSE

établissement associé

Architecture du microcontrôleur 8051F020

Son architecture est dérivée de la famille 8051 « historique »(espaces mémoire, jeu d'instruction, périphériques de base), mais son cœur, de conception plus récente, est plus rapide et il est doté de nombreux périphériques



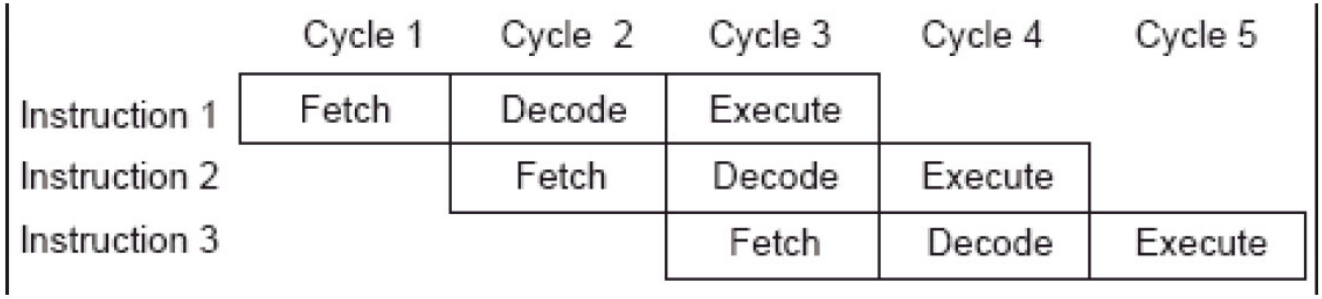
Caractéristiques générales du cœur « Processeur » CIP-51 équipant le 8051F020

Un cœur processeur hérité de l'architecture 8051

- Processeur 8 bits
- Jeu d'instruction compatible 8051
- Capable de gérer 2 espaces mémoire distincts CODE et XDATA (Archi Harvard?)

MAIS en plus:

- Puissance max 25 Mips à 25 MHz (1 Mips @12MHz pour la version historique)
- 256 octets de mémoire RAM interne
- Architecture Pipeline à 3 étages pour l'optimiser les temps d'exécution

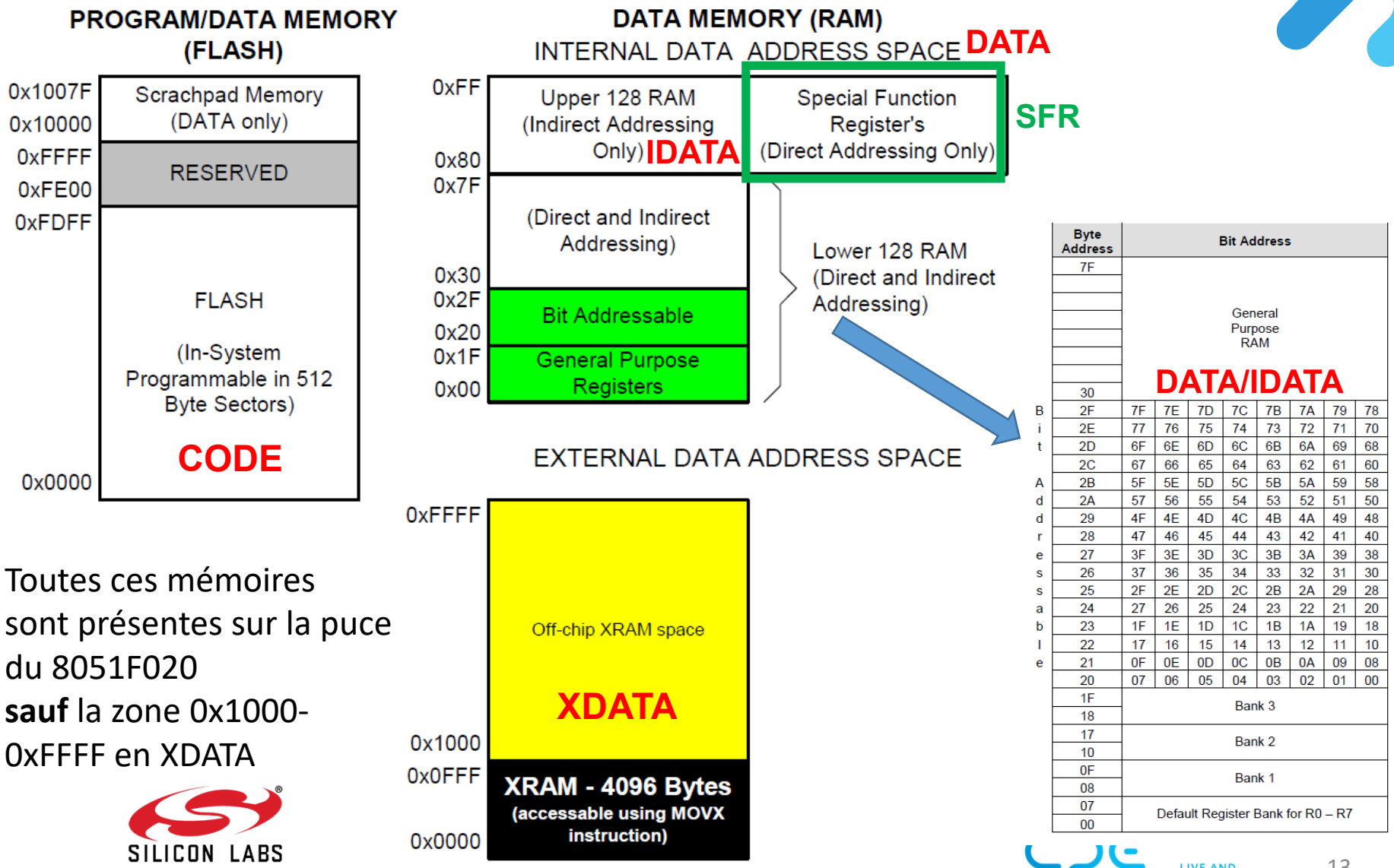


Clocks to Execute	1	2	2/3	3	3/4	4	4/5	5	8
Number of Instructions	26	50	5	14	7	3	1	2	1

SILICON LABS

AND OVER

Mémoires CODE, XDATA, DATA et IDATA



Toutes ces mémoires sont présentes sur la puce du 8051F020
sauf la zone 0x1000-0xFFFF en XDATA

SILICON LABS

Mémoire SFR – Special Function Registers

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	PCA0M0	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)

Pour lire ce tableau:
Le registre P3IF est accessible à l'adresse A8H + 5 = ADH (dans l'espace SFR bien sûr)

SILICON LABS

CPE LYON LIVE AND DISCOVER

CPE LYON

LIVE AND DISCOVER

Langage C et microcontrôleur
Compilateur C51

Famille 8051

Exemple élémentaire:
Addition de 2 nombres 16-bits en C et en assembleur

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

La version **C**

Avec declarations de variables et lignes de code

```
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
```

La version **assembleur**,

avec directives de segmentation et code assembleur

Macro Assembleur 8051

Exemple de fichier source assembleur

- **2 lignes d'instructions assembleur seulement!!**
- Tout le reste est constitué de directives d'assemblage
- Découpage en segments de CODE, DATA, IDATA, XDATA et BIT
- Création de zone de stockage pour des « variables »

©Homer Simpson
Extrait de la série TV
« Les Simpson »

Pratiquer l'assembleur requiert:

- La connaissance du langage assembleur
- La connaissance de l'architecture interne du processeur

C'est lourd!!

```
1  ;------------------------------------------------------------------------------
2  ;  FILE NAME   :  BASE_SM2.ASM
3  ;  TARGET MCU  :  C8051F020
4  ;------------------------------------------------------------------------------
5  $include (c8051f020.inc)              ; Include register definition file.
6  ;------------------------------------------------------------------------------
7  ; EQUATES
8  ;------------------------------------------------------------------------------
9  GREEN_LED      equ   P1.6             ; Port I/O pin connected to Green LED.
10 ;------------------------------------------------------------------------------
11 ; DATA Segment
12 ;------------------------------------------------------------------------------
13 data_test     segment  DATA
14              rseg     data_test
15 tempo_data:       DS   10
16 ;------------------------------------------------------------------------------
17 ; Put the STACK segment in the main module.
18 ;------------------------------------------------------------------------------
19 ?STACK       SEGMENT IDATA           ; ?STACK goes into IDATA RAM.
20              RSEG    ?STACK          ; switch to ?STACK segment.
21              DS      20  ; reserve your stack space - 20 bytes in this example.
22 ;------------------------------------------------------------------------------
23 ; XDATA SEGMENT
24 ;------------------------------------------------------------------------------
25 Ram_externe  SEGMENT XDATA     ; Reservation de 50 octets en XRAM
26              RSEG    Ram_externe
27 cp_bcl:          DS      1
28 table_dest0:     DS      20
29 table_dest:      DS      200
30 ;------------------------------------------------------------------------------
31 ; RESET and INTERRUPT VECTORS
32 ;------------------------------------------------------------------------------
33        ; Reset Vector
34        cseg AT 0           ; SEGMENT Absolu
35        ljmp Start_pgm ; Locate a jump to the start of code at the reset vector
36 ;------------------------------------------------------------------------------
37 ; CODE SEGMENT
38 ;------------------------------------------------------------------------------
39 Prog_base   segment  CODE
40             rseg     Prog_base      ; Switch to this code segment.
41             using    0              ; Specify register bank for the following
42                                       ; program code.
43 ;*****************************************************************************
44 ;Initialisations de la mémoire code - Stockade de constante
45 ;*****************************************************************************
46 alphabet:  DB 'abcdefghijklmnopqrstuvwxyz'
47 chiffres:  DB '0123456789'
48 ;*****************************************************************************
49 ;Initialisations de périphériques - Fonctionnalités Microcontroleur
50 ;*****************************************************************************
51 Start_pgm:
52
53         mov WDTCN, #0DEh    ; Disable WDT
```

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
- Un code en C aisément réutilisable
- Un langage qui reste proche de la machine

MAIS:

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

Spécificité du code C sur des microcontrôleurs 8 bits

Il faut tenir compte:

- Faible mémoire de code et **surtout de données**
- Faible puissance de calcul (architecture 8 bits)
- Haut niveau de réactivité temporelle attendu
- Accès direct aux mémoire physiques et aux **périphériques**

Rappel sur les divers éléments d'un code C…

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

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 pré-processeur (inclusion, macros, compilations conditionnelles)

Pris en compte par le compilateur

Compilateur et librairies dépendant de la cible visée….

Géré par des librairies de fonctions (librairies standard du C)

La base de la compilation

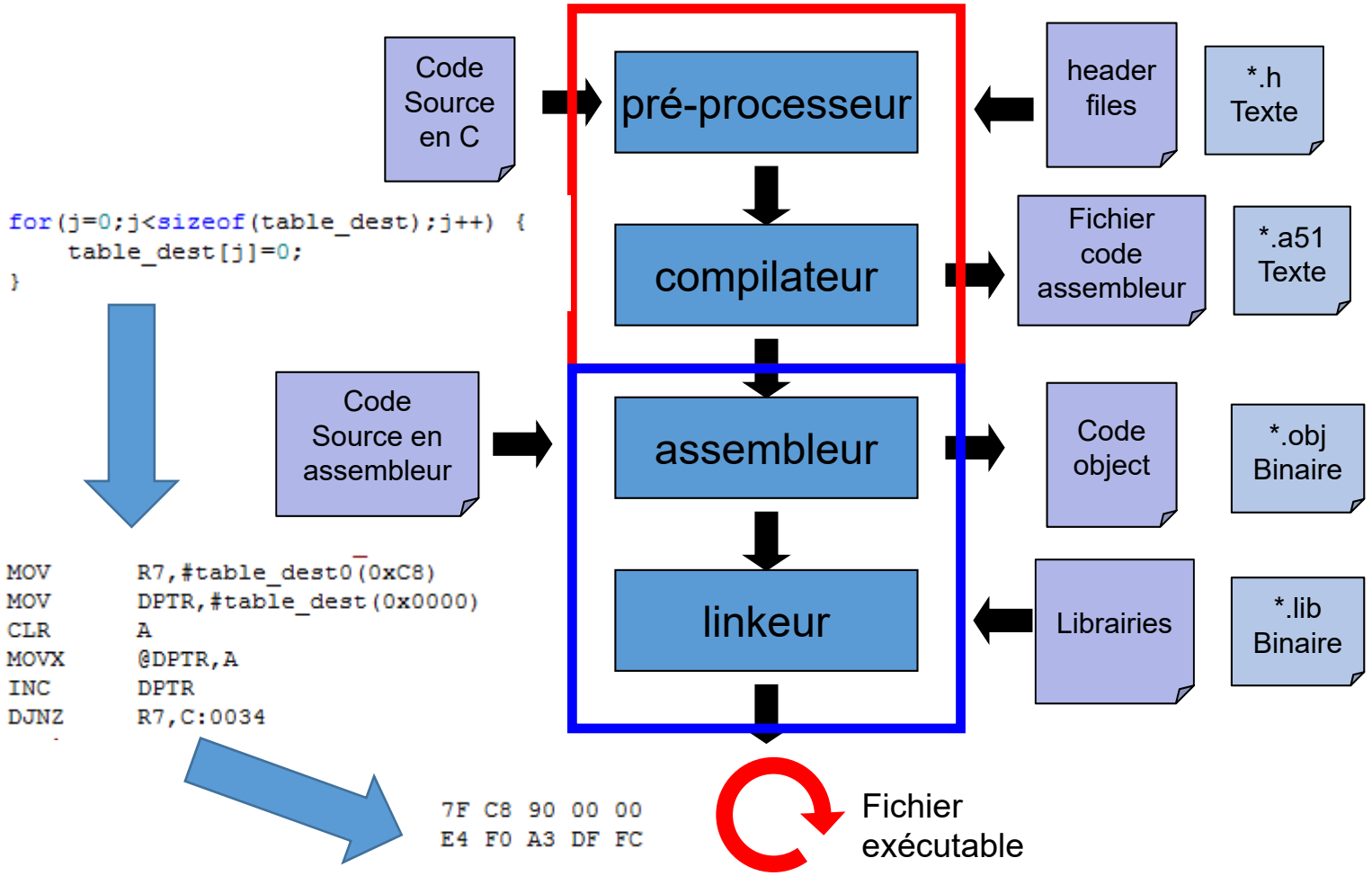
- Le pre-processeur gère
  - Les Macros (#define)
  - Les Inclusions (#include)
  - Les Inclusions conditionnelles (#ifdef, #if)
  - Les extensions du langage (en partie) (#pragma).

- 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" (editeur de lien) procéde à l'édition de lien des codes objets (qui se référencient). Il crée l'exécutable (éventuellement en lien avec un format d'exécutable)

Les étapes de la compilation du code



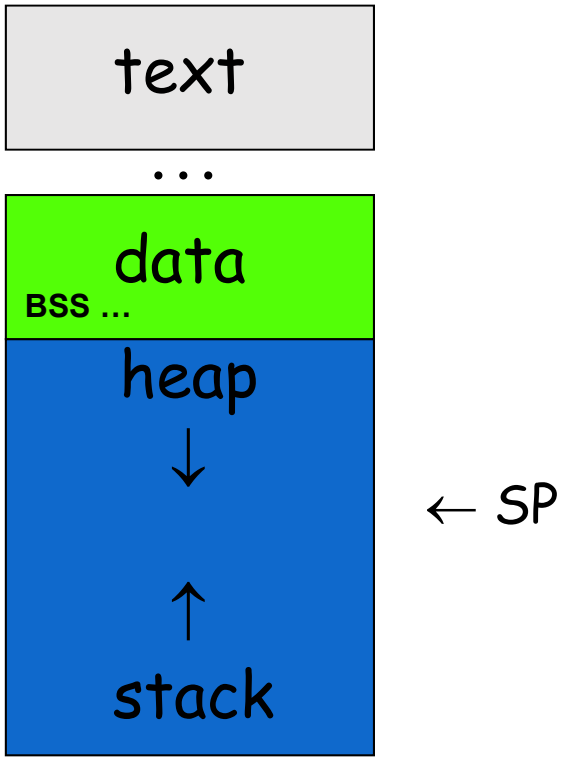
Organisation classique de la mémoire gérée par un compilateur C

Zones mémoires:

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

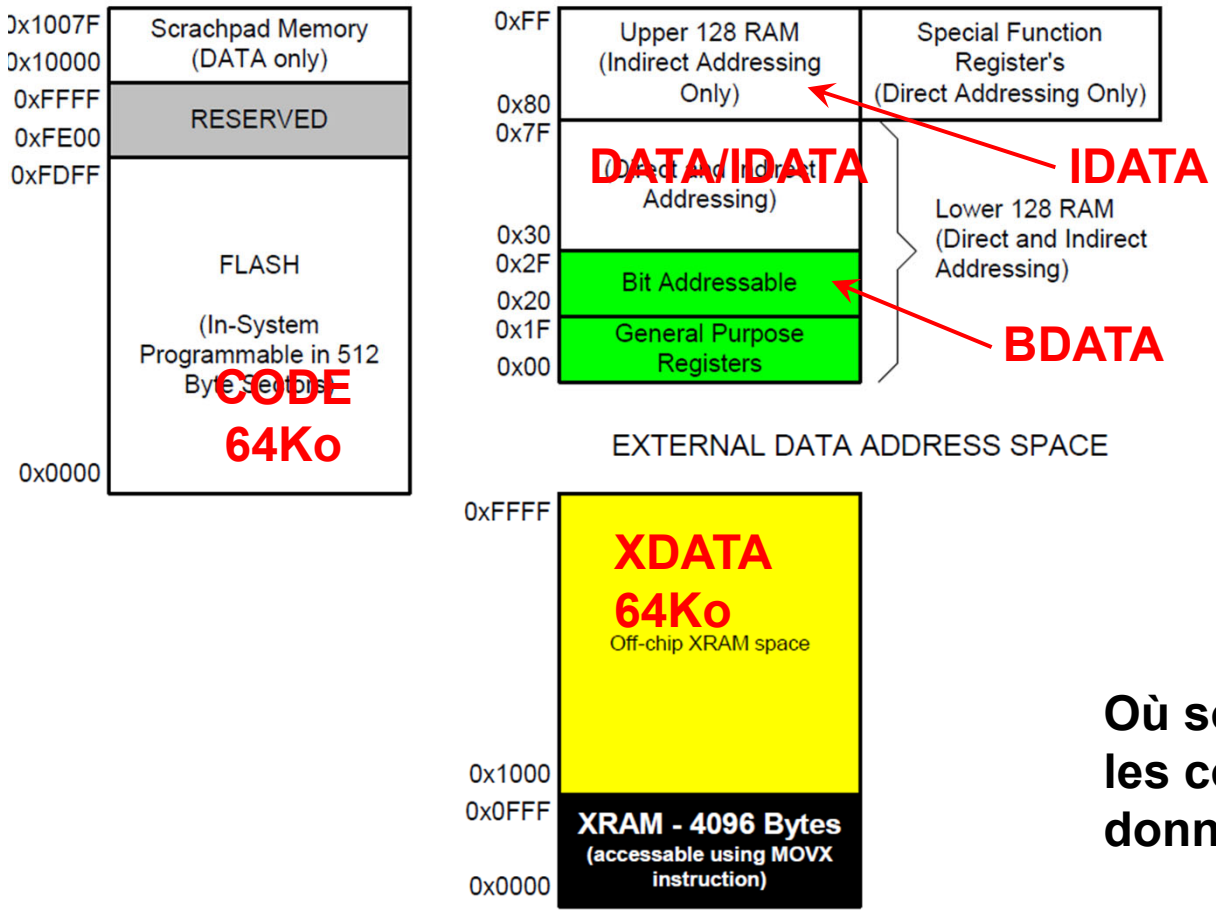
```
char strng="hello";
int count, this, that;

main()
{
int i, j, k;
char *sp;
……..
for (i=0;i != 100;i++)
……
sp= (char*) malloc(sizeof(i));
```

???



Organisation de la mémoire en C dans un environnement Microcontrôleur 8051

Point de vue 8051



Point de vue Compilateur C

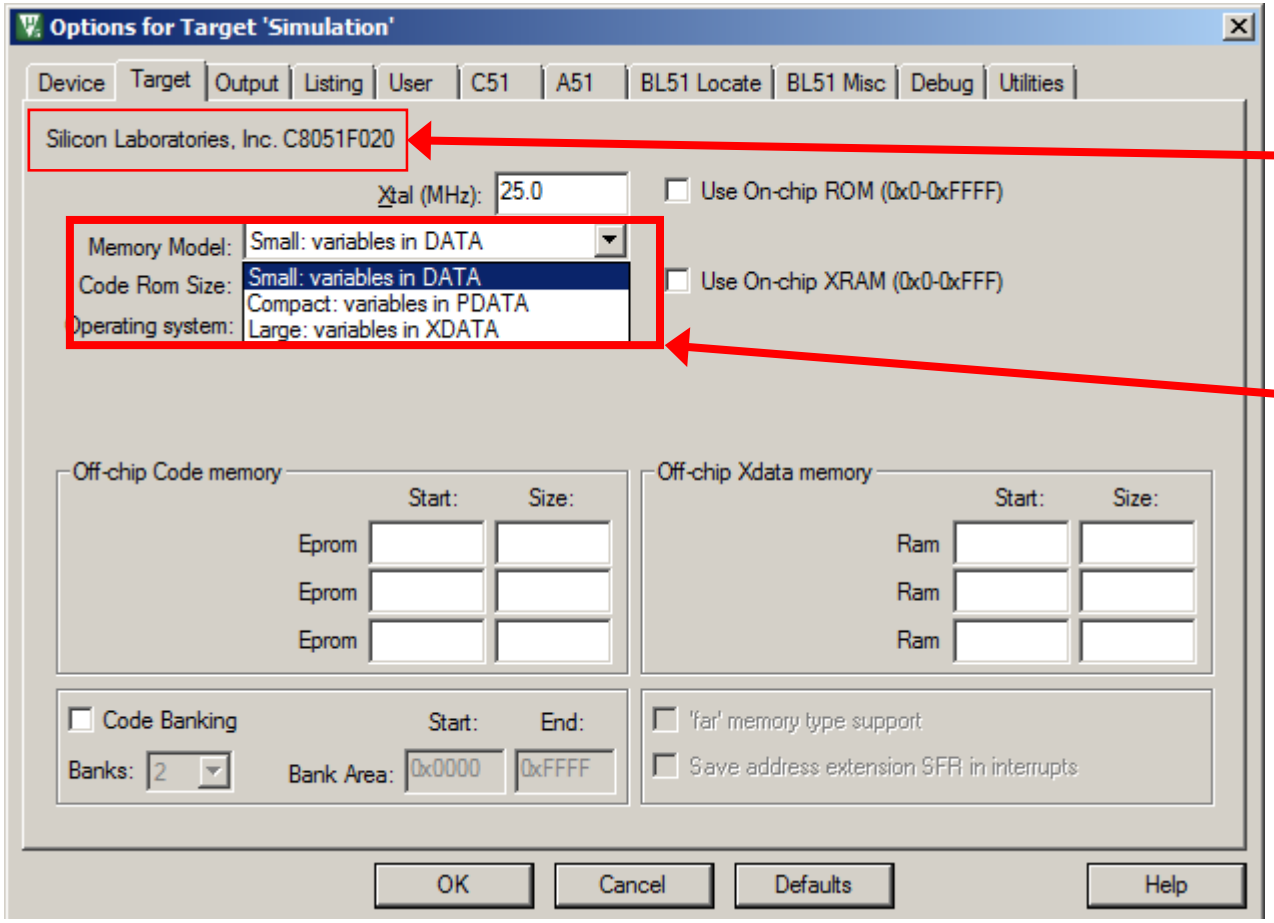
- Text (Code)
- Data
- Heap (à éviter)
- Stack

Solution: production par le compilateur d'un code assembleur manipulant des segments data – idata – xdata – bdata - code

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

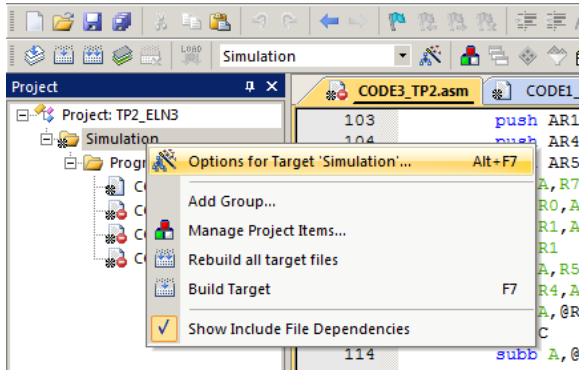
Gestion de la mémoire via l'environnement de développement Microvision

Sous Microvision, il est possible d'indiquer au compilateur (et à l'éditeur de liens) où devront être stockés les diverses variables gérées par le compilateur C. Cette configuration sera réalisée par modification de la configuration globale du projet



Le processeur a été sélectionné au moment de la création du projet

Le « modèle » mémoire est spécifié dans un menu de configuration



Les modèles de gestion mémoire dans Microvision

Modèle SMALL

- Le 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 (2 cycles)

Modèle Compact (Mode intermédiaire cité pour information)

Modèle LARGE

- **Par défaut**, toutes les variables sont placées dans la mémoire externe (xdata). La pile reste toutefois en idata.
- Mémoire plus importante, mais relative lenteur des temps d'accès (3 + 3 cycles)

Exemple de compilation selon les modèles mémoire

Code compilé:

```
int var1; //2 octets

int var2; //2 octets

char tab[12]; // 12 octets

float tabf[24]; // 24 * 4
octets

long Test1,test2; // 2 * 4
octets

                    = 120 octets
```

Les fichiers avec l'extension M51 sont générés à l'issue de l'édition de liens; ils contiennent les informations relatives à l'utilisation de la mémoire:

- *Quantité de mémoire utilisée*
- *Localisation du code*
- *Localisation des variables*

En Mode SMALL: (Extrait fichier M51)

```
TYPE   BASE     LENGTH   RELOCATION   SEGMENT NAME
-----------------------------------------------------

******* DATA MEMORY *******

REG    0000H    0008H    ABSOLUTE     "REG BANK 0"
DATA   0008H    0078H    UNIT         ?DT?BASE_TP3
IDATA  0080H    0001H    UNIT         ?STACK
```

(Length = 78H = 120D)

En mode LARGE:

```
TYPE   BASE     LENGTH   RELOCATION   SEGMENT NAME
-----------------------------------------------------

******* DATA MEMORY *******

REG    0000H    0008H    ABSOLUTE     "REG BANK 0"
IDATA  0008H    0001H    UNIT         ?STACK


******* XDATA MEMORY *******

XDATA  0000H    0078H    UNIT         ?XD?BASE_TP3
```

Types de mémoire explicitement définis

Ajout de spécificateurs de mémoire au moment de la déclaration des variables

Il est possible au moment de la déclaration d'une variable, de spécifier l'espace mémoire dans lequel elle devra être placée.

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.

Exemples de déclarations explicites de type mémoire:

```
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;
```

Sans spécificateur de mémoire, la variable est placée dans l'espace mémoire spécifié lors du choix de modèle mémoire.
Si la variable a un spécificateur de mémoire, alors celle-ci est placée dans l'espace mémoire correspondant au spécificateur quel que soit le modèle mémoire sélectionné.

Codage en assembleur – Incidence de l'espace mémoire utilisé sur la taille du code produit

Tous ces exemples de code ne font que lire 1 octet dans un espace mémoire donné

Extension	Memory Type	Related ASM
data	Directly-addressable data memory (data memory addresses 0x00-0x7F)	MOV A, 07Fh (2 clk cycle)
idata	Indirectly-addressable data memory (data memory addresses 0x00-0xFF)	MOV R0, #080h (4 clk cycle) MOV A, @R0
xdata	External data memory	MOV DPTR,#1234H (6 clk cycle) MOVX @DPTR,A
code	Program memory	MOV DPTR,#1234H (7 clk cycle) CLR A MOVC A,@A+DPTR

Temps d'exécution

X2

X3

X3,5

La « complexité » du code (taille et temps d'exécution) varie beaucoup selon les espaces mémoire utilisés.
L'accès à la zone DATA est le plus rapide.

Gestion mémoire recommandée

- Modèle SMALL (par défaut les variables sont placées dans l'espace DATA)
- Déclarations explicites des constantes en CODE et des variables de grande taille ou rarement utilisées en XDATA

IMPORTANT

Types spécifiques de données gérés par le compilateur C51

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	±1.175494E-38 — ±3.402823E+38
double	32	4	±1.175494E-38 — ±3.402823E+38
sbit	1		0 or 1
sfr	8	1	0 — 255
sfr16	16	2	0 — 65535

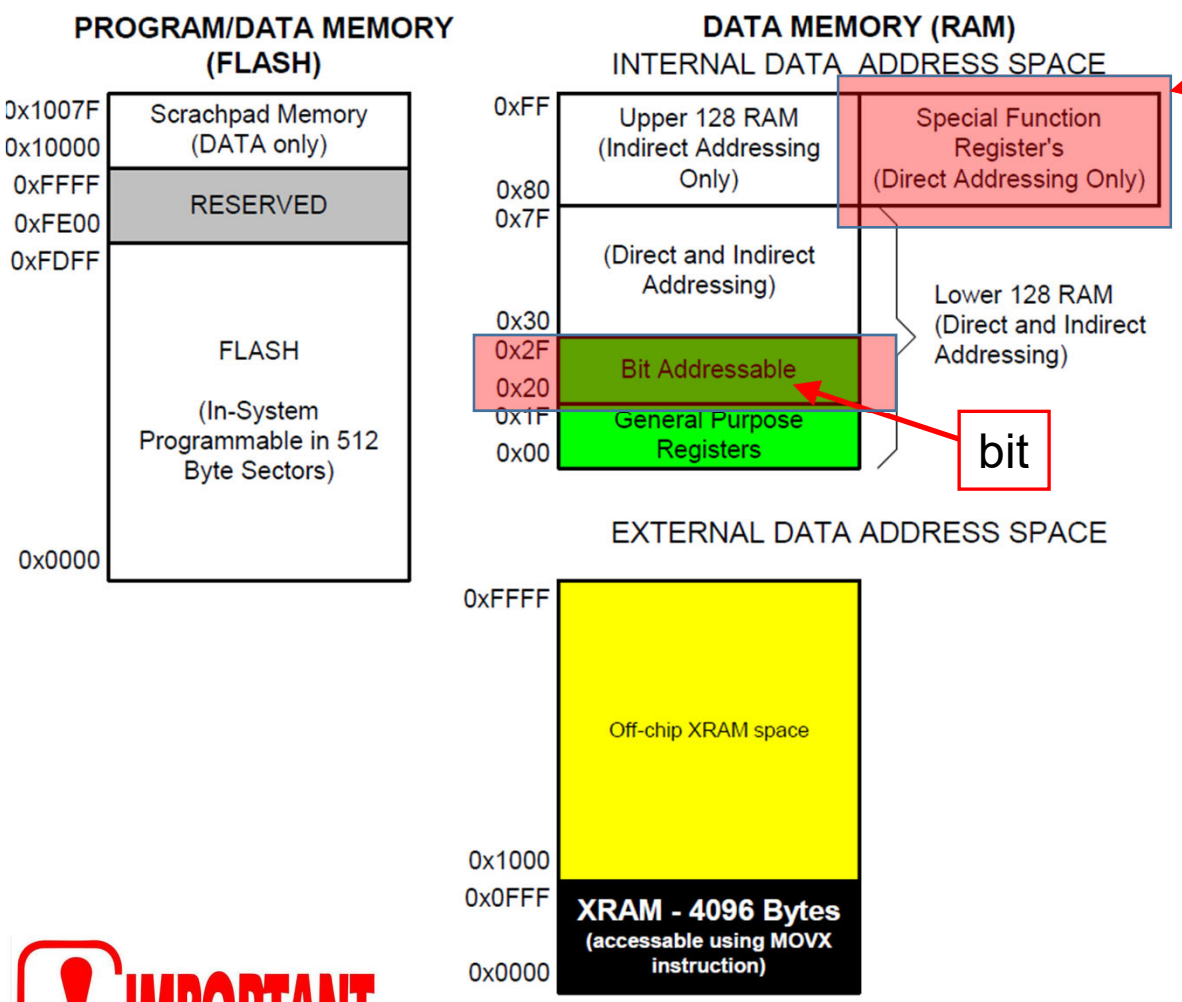
Ces types ne sont pas ANSI-C et sont spécifiques au compilateur

Type bit pour la manipulation de variables booléennes (grâce à l'espace mémoire accessible bit à bit et aux instructions assembleur de manipulation de booléen

Types sbit, sfr, and sf16: pour l'accès à la zone SFR
Doivent être déclarés en global (ne permettent pas les indirections)

IMPORTANT

Localisation, bit, sbit, sfr, sfr16….



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. Permet de manipuler un bit donné, dans un registre donné **accessible bit à bit**

bit: déclaration d'une variable booléenne – cette variable sera logée dans la zone mémoire DATA accessible bit à bit (20H à 2FH) -

IMPORTANT

Les déclarations SBIT, SFR et SFR16

Ces types permettent de manipuler bits et registres de 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)	1(9)	2(A)	3(B)	4(C)	5(D)	6(E)	7(F)
	bit addressable							

Déclarations sfr possibles

Déclarations sbit possibles dans ces registres

IMPORTANT

Exemples de déclarations sfr16 sensées

Exemples de manipulation de registre et de bit

sfr:

Déclaration:

```
sfr P4 = 0x84;
```

Exemple d'utilisation:

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

Signification de l'exemple: Le registre nommé « P4 » correspondant à l'adresse 0x84 dans l'espace mémoire SFR est initialisé avec la valeur 0x22

sbit:

Déclaration:

```
sbit LED = P1^6;
```

compilable si cette déclaration est précédée de `sfr P1 = 0x90;` car il faut que P1 soit défini

Exemple d'utilisation:

```
LED = 1;  (ASM: setb 96H ou setb P1.6)
```

Signification de l'exemple: Le bit 6 du registre d'adresse 0x90 dans l'espace mémoire SFR est mis à 1. Ce bit est nommé « LED »

bit:

Déclaration:

```
Bit Flag0;
```

Exemple d'utilisation:

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

Signification de l'exemple: La variable booléenne Flag0 est mise à zéro (sa localisation dans la mémoire est gérée par le compilateur, elle est « transparente » pour l'utilisateur)

IMPORTANT

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
```

```
/*  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  */
```

Ce fichier est fourni par le constructeur.

Ce fichier contient toutes les déclarations sfr et sbit du 8051F020, mais aucune déclaration SFR16

Ce fichier ne doit pas être modifié!!

Les déclarations SFR16 du 8051F020

- Elles ne sont pas faites dans le fichier c8051F020.h
- Ces SFR16 permettent de directement manipuler des grandeurs 16 bits contenues dans 2 registres 8 bits (exemple: registres de Timers, registres de sorties de CAN, registres d'entrée de CAN)
- Les 2 registres 8 bits concaténés doivent se trouver à 2 adresses consécutives de l'espace SFR
- Dans la déclaration SFR16, on indique l'adresse du premier registre. Ce dernier sera considéré comme le poids faible de la valeur 16 bits

Exemple de déclaration:

```
sfr16 RCAP2 = 0xCA;  //Concaténation de 2 registres RCAP2L (0XCA),
                     //RCAP2H (0xCB) – adresse n: pds faible et
                     // adresse n+1 : pds fort qui seront manipulés sous le nom de RCAP2
```

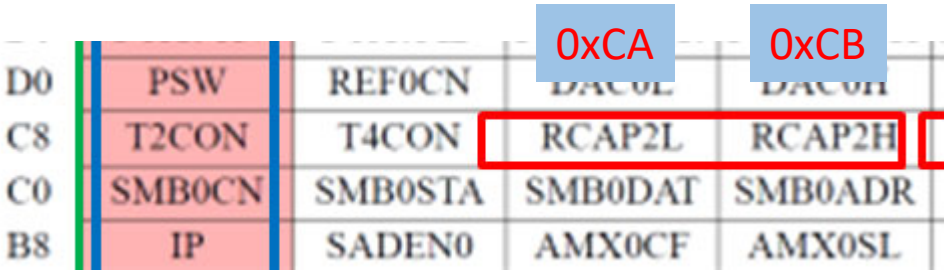
Exemple d'utilisation:

```
RCAP2 = 1234H;  (ASM: MOV 0CAH,#34H et MOV 0CBH,#12H)
```

Signification de l'exemple: Le registre d'adresse 0xCA (espace SFR) est initialisé avec la valeur 0x34 et le registre d'adresse 0xCB (espace SFR) est initialisé avec la valeur 0x12

Les SFR16 potentiels (les SFR16 qui ont du sens):

```
sfr16 DP       = 0x82;    // data pointer
sfr16 TMR3RL   = 0x92;    // Timer3 reload value
sfr16 TMR3     = 0x94;    // Timer3 counter
sfr16 ADC0     = 0xbe;    // ADC0 data
sfr16 ADC0GT   = 0xc4;    // ADC0 greater than
sfr16 ADC0LT   = 0xc6;    // ADC0 less than wir
sfr16 RCAP2    = 0xca;    // Timer2 capture/rel
sfr16 T2       = 0xcc;    // Timer2
sfr16 RCAP4    = 0xe4;    // Timer4 capture/rel
sfr16 T4       = 0xf4;    // Timer4
sfr16 DAC0     = 0xd2;    // DAC0 data
sfr16 DAC1     = 0xd5;    // DAC1 data
```



IMPORTANT

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

```
//Déclarations
sbit LED = P1^6;/* now the functions may be written to use
this location */
sfr16 T2 = 0xCC /* Timer 2: T2L 0CCh  T2H 0CDH */

//Exemples d'utilisation
void main (void)
{
  Config_Timer2_and_Run();
 /* forever loop, toggling pin 0 of port 1 */
 while (1==1)
    {      LED = !LED;
              T2 = 0;
              while ( T2 != 0x1234);
    }
}
```

Les données déclarées avec des type sfr, sfr16 et sbit sont manipulées comme des variables classiques. La seule différence est qu'elles correspondent à des positions mémoire bien définies

Remarques sur les variables de type int

Une variable « int » fait seulement 16 bits (ainsi que signed short int, unsigned short int, unsigned int, signed int).

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

Exemple: int ***var*** = 70000; //pas d'erreur à la compilation (et pas de warning)

// mais ***var*** contient en fait 4464 car 70000 =65536+4464

Remarques sur les variables de type char

Une variable « char » permet de stocker des informations sur 8 bits (ainsi que signed char, unsigned char).
Cette information 8 bits peut être interprétée comme un code ASCII mais aussi comme une valeur numérique (tout dépend du code)

signed char	8	1	-128 — +127
unsigned char	8	1	0 — 255

Exemple: char value = 0x41; // Hexa
char value = 0101; // Octal (ce n'est pas du binaire!!)
char value = 65; // décimal
char value = 'A'; // codage ASCII

Les constantes octales doivent commencer par un zéro et ne comporter que des chiffres octaux

Remarque: Le C ne reconnait pas l'écriture en binaire
Toutes ces déclarations produisent le même résultat.
Ainsi dans la mémoire, la case mémoire représentative de cette variable contient: 0100 0001 (binaire)
4 1 H

Exemple:

```
char i;                 // i variable sur 8 bits – ne « consomme » qu'un octet
for(i=0;i <10;i++)  { ……..}   // i est une variable utilisée comme compteur de boucle,
```

donc comme valeur numérique. Là où un « int » aurait consommé 2 octets, le « char » n'en consomme qu'un seul, largement suffisant pour stocker une valeur qui évolue de 0 à 10. De plus le code assembleur se trouve grandement simplifié.

Données binaires et données texte

Commençons par une évidence: quel que soit le format « données binaires » ou « données textes », les informations stockées dans la mémoire sont stockées sous forme d'une suite de '0' ou de '1'

En mode « données binaires » les informations sont stockées brutes sans un quelconque encodage.

Exemple:

- une température de 18° sera stockée dans un octet. L'octet contiendra la valeur 18 (12H ou 00010010B)

```
char temperature = 18;
```

- L'année 2021 sera stockée dans 2 octets (un entier int). L'octet de poids fort contiendra 07H et l'octet de poids faible E5H (2021 = 7E5H)

```
int annee = 2021;
```

En mode « données texte » (ou données ASCII), les informations sont codées en respectant un encodage (ASCII par exemple)

- une température de 18° sera stockée dans un tableau de caractères.

```
char temperature[2] = {'1','8'}; // 2 octets nécessaires pour encoder
```

ou (mieux encore) dans une chaine de caractères

```
char temperature[] = « 18 »; // 3 octets nécessaires pour encoder
```

- L'année 2021 sera stockée dans 1 tableau de 4 caractères

```
int annee[4] = {'2','0','2','1'}; // 4 octets nécessaires pour encoder
```

ou (mieux encore) dans une chaine de caractères

```
int annee[] = « 2021 »; // 5 octets nécessaires pour encoder
```

Exercices SFR, SFR16, SBIT, BIT

Dans ces exercices, on considère que le fichier C8051F020.H n'est pas inséré dans le fichier source (omission de la ligne **`#include <c8051f020.h>`**

1 - Initialiser le registre PCON avec la valeur 22H

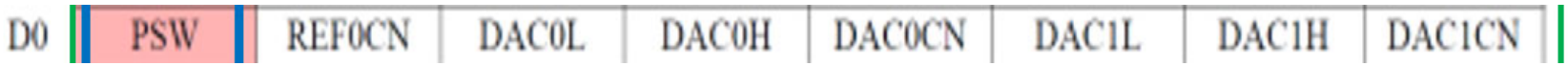
Adresse SFR de PCON: 0x87

2 – Mettre à 1, le bit EX0 du registre IE

Le bit EX0 est le bit 0 du registre IE – Adresse SFR de IE: 0xA8

3 – Le code suivant vous semble-t-il correct? Justifier.

```
sfr16 DAC0 = 0xD3;
DAC0 = 0x1234;
```



4- Quelle est l'erreur?

```
bit flag_tst;
flag_tst = 2;
```

Les limitations pour les fonctions mathématiques

L'utilisation de calcul en réel (double, float …) est très gourmande en puissance de calcul en l'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 et rappel: le type char peut être utilisé pour désigner un entier 8 bits

Les limitations pour les fonctions mathématiques - Exemple

Exemple de stockage d'informations de durée dans des variables

Unité: Seconde

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

Variable sur 4 octets
Utilisation d'une librairie de calculs en flottant
Limites: 1,175 X 10-38s à 3,402 X 10+38s

Unité: MilliSeconde

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

Variable sur 2 octets
Calcul en multiprécision
Limites: 0 à 65536ms – Résolution 1 ms

Unité: CentiemeSeconde

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

Variable sur 1 octets
Calcul en 1 instruction ou presque (8 bits)
Limites: 0 à 2550ms – Résolution 10 ms

Attention aux valeurs limites de la variable!!

Votre choix du type de variable dépendra de la plage de valeur et de la résolution.

Exemple: pour stocker un temps de 0 à 2s avec une résolution de 10ms, une variable char (octet) suffira pour peu que l'on change d'unité….

Opérateurs logiques bit à bit

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

Ces opérateurs sont constamment utilisés en programmation bas-niveau dans l'embarqué

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

L'objectif est de modifier un ou plusieurs bits d'un registre sans altérer la valeur des autres bits.

```
EIE2 = EIE2 & 0xBF;  // Mise à zéro de EIE2.4 - Disable UART1 Interrupts
EIE2    &= ~0x40;    // Ecriture plus concise

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

IMPORTANT

Mise à 1 de bits dans un registre par masquage

Exemple:

Mise à 1 des bits 6 et 5 d'un registre nommé **Reg_ctrl**:

- Utilisation de l'opérateur OU (|)

```
Reg_ctrl                          b7 b6 b5 b4 b3 b2 b1 b0
Masque                             0  1  1  0  0  0  0  0
                                  ------------------------
Reg_ctrl = Reg_ctrl | Masque      b7  1  1 b4 b3 b2 b1 b0
```

Ecritures possibles en C:

```
Reg_ctrl = Reg_ctrl | 0x60;
Reg_ctrl|= 0x60

Reg_ctrl = Reg_ctrl |( (1<<6)|(1<<5) ); //écriture plus explicite??
Reg_ctrl |= ( (1<<6)|(1<<5) );
```

Valeur invariable « 1 »

Numéro du bit à décaler 0 à 7

Positionner un bit à 1 dans un registre: opérateur OU |

IMPORTANT

Mise à 0 de bits dans un registre par masquage

Exemple:

Mise à 0 des bits 6 et 5 d'un registre nommé Reg_ctrl:

- Utilisation de l'opérateur ET (&)

```
Reg_ctrl                          b7 b6 b5 b4 b3 b2 b1 b0
Masque                             1  0  0  1  1  1  1  1
                                  ----------------------
Reg_ctrl = Reg_ctrl & Masque      b7  0  0 b4 b3 b2 b1 b0
```

Ecritures possibles en C:

```
Reg_ctrl = Reg_ctrl & 0x9F;
Reg_ctrl &= 0x9F;
Reg_ctrl = Reg_ctrl & ~0x60;
Reg_ctrl &= ~0x60;

Reg_ctrl = Reg_ctrl & ~( (1<<6)|(1<<5) );
Reg_ctrl &= ~( (1<<6)|(1<<5) );
```

Positionner un bit à 0 dans un registre: opérateur ET &

IMPORTANT

Inversion de bits (complémenter) dans un registre par masquage

Exemple:

Inversion des bits 6 et 5 d'un registre **Reg_ctrl**:

- Utilisation de l'opérateur OU Exclusif (^)

	b7	b6	b5	b4	b3	b2	b1	b0
Reg_ctrl								
Masque	0	1	1	0	0	0	0	0
Reg_ctrl = Reg_ctrl ^ Masque	b7	/b6	/b5	b4	b3	b2	b1	b0

Ecritures possibles en C:

```
Reg_ctrl = Reg_ctrl ^ 0x60;
Reg_ctrl^= 0x60
Reg_ctrl = Reg_ctrl ^( (1<<6)|(1<<5) );
Reg_ctrl ^= ( (1<<6)|(1<<5) );
```

Complémenter un bit dans un registre: opérateur OU exclusif ^

IMPORTANT

Limites d'utilisation d'opérateurs logiques pour les masquages

Attention certains registres ne peuvent pas être manipulés par des opérations de masquage car ils ne sont pas « Read-Modify-Write » c'est-à-dire qu'il y a au moins un bit dans ce registre dont le rôle est différent en lecture et en écriture.
Une opération de masquage sur ce type de registre risque de provoquer quelques effets de bord….

Figure 13.4. RSTSRC: Reset Source Register

R	R/W	R/W	R/W	R	R	R/W	R	Reset Value
-	CNVRSEF	C0RSEF	SWRSEF	WDTRSF	MCDRSF	PORSF	PINRSF	Variable
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xEF

(Note: Do not use read-modify-write operations on this register.)

L'information est donnée par le constructeur dans le tableau récapitulatif
Décrivant la fonctionnalité du registre

Bit4: SWRSF: Software Reset Force and Flag
Write: 0: No Effect.
1: Forces an internal reset. /RST pin is not affected.
Read: 0: Prior reset source was not a write to the SWRSF bit.
1: Prior reset source was a write to the SWRSF bit.
Bit3: WDTRSF: Watchdog Timer Reset Flag

Le bit4 de ce registre a un rôle différent en lecture et en écriture ce qui rend l'opération de masquage hasardeuse

Résumé: manipulation de bits dans des registres

Cas1: le registre est accessible bit à bit

- Utiliser les déclarations sbit dans le registre C8051F020.H et manipuler directement le bit concerné.

Exemple: on souhaite mettre à 0 le bit 7 du registre IE

```
EA = 1;  //EA, bit 7 du registre IE, accessible bit à bit
```

Cas2: le registre n'est pas accessible bit à bit

- Utiliser les opérations de masquage.

Exemple: on souhaite mettre à 0 le bit 0 du registre IE2

```
IE2 &= ~(1<<0);   //ET3, bit0 du registre IE2 mis à zéro
```

Remarque:

Tous les registres de l'espace SFR accessibles bit à bit sont des registres de contrôle; aussi dans ces registres, la manipulation bit à bit a du sens.
*Les registres de l'espace SFR, non accessibles bit à bit sont soit des registre de contrôle et de configuration, soit des registres de données. En règle générale (il y a toujours des cas particuliers), la manipulation bit à bit de ces registres est utile pour les registres de contrôle et de configuration, mais **n'a aucun sens** pour les registres de données.*

Exercices de manipulation de bits dans l'espace SFR

Pour coder ces exemples, on considère que l'on a à notre disposition le fichier 8051F020.h qui définit tous les bits et registres de l'espace SFR

OBJECTIF du masquage: modifier l'état d'un bit dans un registre sans modifier l'état des autres bits

1 – Mettre à 0 le bit 4 du registre PCON

2 – Mettre à 1 le bit 7 du registre PCON

3 – Mettre à 1, le bit 0 du registre IE et à 0, le bit 7 du registre IE

4 – Ecrire une boucle d'attente: tant que le bit 1 du registre SCON1 est à zéro, alors attendre…

Ordre de stockage en mémoire - Définition de l'Endianness

On définit ici l'ordre de rangement des octets dans la mémoire lorsque l'on souhaite stocker des nombres d'une taille supérieure à l'octet.

- Little Endian (architecture Intel X86) – dans la mémoire on stocke l'octet de poids faible en premier.

  Exemple: soit le nombre 32 bits 0xA0B70708

Adresses	N-1	N	N+1	N+2	N+3	N+4
Contenu	....	08	07	B7	A0	....

- Big Endian (architecture Motorola 68000) – dans la mémoire on stocke l'octet de poids fort en premier.

  Exemple: soit le nombre 32 bits 0xA0B70708

Adresses	N-1	N	N+1	N+2	N+3	N+4
Contenu	....	A0	B7	07	08	....

➢ **Le compilateur C51 stocke ses variables dans le format Big Endian**

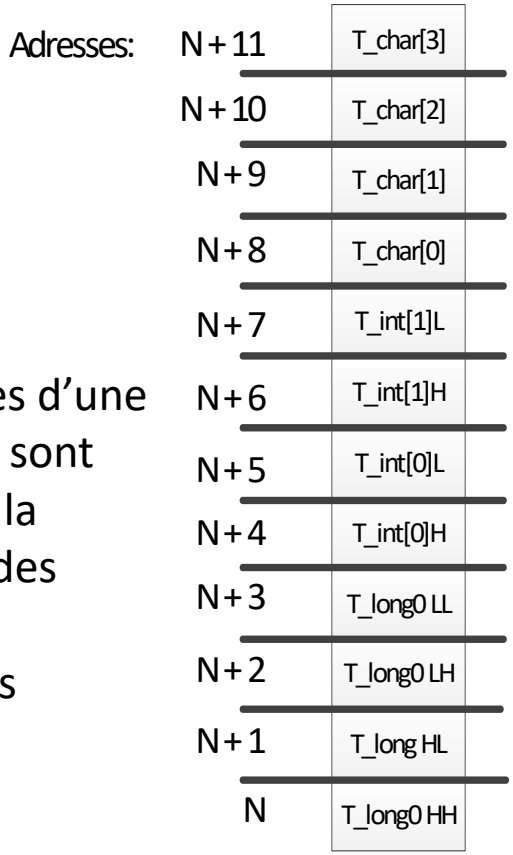
Accès « octet » à l'intérieur d'un entier « Int » ou « long »

Il est souvent nécessaire d'accèder individuellement aux différents octets à l'intérieur d'une variable de type INT ou LONG

Le langage 'C' permet de le faire **efficacement** en utilisant une construction de type UNION

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

Contenu Mémoire de la structure
Stockage en Big Endian



Les membres d'une STRUCTURE sont placés dans la mémoire à des adresses consécutives

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

Contenu Mémoire de l'union
Stockage en Big Endian

Adresses:			
N + 3	T_long0 LL	T_int[1] L	T_char[3]
N + 2	T_long0 LH	T_int[1] H	T_char[2]
N + 1	T_long0 HL	T_int[0] L	T_char[1]
N	T_long0 HH	T_int[0] H	T_char[0]

Tous les membres d'une UNION résident dans la même zone mémoire. Il y a donc plusieurs moyens d'accéder à un même octet

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

Rappel: Les membres de l'union occupent la même zone mémoire

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

int val1,val2;
char pds_faible;
tst_ok  to_convert;

to_convert.t_long=0x12345678;
val1= to_convert.t_int[1];  // val1 = 0x5678
val2= to_convert.t_int[0];  // val2 = 0x1234
pds_faible=to_convert.t_char[3]; // pds_faible = 0x78
```

Contenu Mémoire

Stockage en Big Endian

Adresses:				
N + 3	0x78	T_long0 LL	T_int[1] L	T_char[3]
N + 2	0x56	T_long0 LH	T_int[1] H	T_char[2]
N + 1	0x34	T_long0 HL	T_int[0] L	T_char[1]
N	0x12	T_long0 HH	T_int[0] H	T_char[0]

Exemple de l'utilisation d'un type union

Méthode 1: extraction de l'octet ML fait par le processeur par programme (modulo et division)
Méthode 2: extraction de l'octet ML à la compilation

Contenu Mémoire

Stockage en Big Endian

Adresses:				
N + 3	0x78	T_long0 LL	T_int[1] L	T_char[3]
N + 2	0x56	T_long0 LH	T_int[1] H	T_char[2]
N + 1	0x34	T_long0 HL	T_int[0] L	T_char[1]
N	0x12	T_long0 HH	T_int[0] H	T_char[0]

```
46 typedef union {
47   long t_long;
48   int  t_int[2];
49   char t_char[4];
50 }tst_ok;
51 //-----------------------------------------------------------------
52 // MAIN Routine
53 //-----------------------------------------------------------------
54 void main (void) {
55
56 //int val1,val2;
57 tst_ok  to_convert;
58 long test_value = 0x12345678;
59 long temp;
60 char ML_byte;
61 // Soit un entier long composé des octets HH MH ML LL (du poids fort au faible)
62 // extraction de l'octet ML par le code
63     temp =  test_value % (256L*256);
64     ML_byte = temp / 256;
65 // extraction de l'octet ML au moment de la compilation
66     to_convert.t_long = test_value;
67     ML_byte=to_convert.t_char[2]; // Récupération ML octet ML
68 // dans les 2 cas, on récupère bien la valeur 0x56
```

Testé sur 8051F020

Méthode 1: 1289 cycles – 258uS @20Mhz

Méthode 2: 19 cycles – 3,8uS @20Mhz

68X plus rapide!!

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 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 (interruption par exemple). 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.

type	sémantique
`const char c;`	caractère constant
`const char *p;`	pointeur vers caractère constant
`char * const p;`	pointeur constant vers caractère
`const char * const p;`	pointeur constant vers caractère constant

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 – type1 – Les pointeurs génériques

L'architecture du 8051 à plusieurs espaces mémoire impose une gestion particulière des pointeurs.

Objectif: s'affranchir des divers espaces mémoires du 8051

-> Solution de type 1: les pointeurs génériques – on déclare un pointeur sans spécifier l'espace mémoire sur lequel il pointe.

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

int *numptr; /* int ptr */
```

L'information relative au pointeur est codée sur 3 octets: 1 octet pour specifier le type de la mémoire sur lequel on pointe, 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

Remarque: rien n'empêche de déclarer des pointeurs génériques avec un spécificateur de mémoire.

```
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 – type2 – Les pointeurs à mémoire spécifique

Les pointeurs à mémoire spécifique permettent de palier la lenteur des pointeurs génériques

Ils sont déclarés pour gérer un seul type de mémoire

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

L'information relative au pointeur est codée sur 1 ou 2 octets selon l'espace mémoire pointé (addresses sur 1 ou 2 octets).

Remarque: rien n'empêche de déclarer des pointeurs à mémoire spécifique avec un spécificateur de mémoire.

```
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 */
```

Démonstration pointeur générique versus pointeur spécifique

```
char i;
char tab[12];//par défaut en DATA
char *ptr_generique;
char data *ptr_specifique;
```

```
ptr_generique = tab;
ptr_generique++;
i = *ptr_generique;

ptr_specifique = tab;
ptr_specifique++;
i = *ptr_specifique;
```

```
    73:     ptr_generique = tab;
C:0x012C    750800    MOV       ptr_generique(0x08),#0x00
C:0x012F    750900    MOV       0x09,#0x00
C:0x0132    750A0B    MOV       0x0A,#tab(0x0B)
    74:     ptr_generique++;
C:0x0135    7401      MOV       A,#0x01
C:0x0137    250A      ADD       A,0x0A
C:0x0139    F50A      MOV       0x0A,A
C:0x013B    E4        CLR       A
C:0x013C    3509      ADDC      A,0x09
C:0x013E    F509      MOV       0x09,A
    75:           i = *ptr_generique;
    76:
C:0x0140    AB08      MOV       R3,ptr_generique(0x08)
C:0x0142    FA        MOV       R2,A
C:0x0143    A90A      MOV       R1,0x0A
C:0x0145    1201AD    LCALL     C?CLDPTR(C:01AD)
C:0x0148    F524      MOV       0x24,A
```

Pointeur générique stocké en
08 (type espace)
09 (adr pds fort)
0A(adr pds faible)
Tab est à l'adresse 0B

Paramètres passés à CLDPTR:
R3: type mémoire
R2: Adr MSB
R1: Adr LSB

Appel d'une fonction de librairie

```
    77:           ptr_specifique = tab;
C:0x014A    751B0B    MOV       ptr_specifique(0x1B),#tab(0x0B)
    78:     ptr_specifique++;
C:0x014D    051B      INC       ptr_specifique(0x1B)
    79:           i = *ptr_specifique;
    80:
C:0x014F    A81B      MOV       R0,ptr_specifique(0x1B)
C:0x0151    E6        MOV       A,@R0
C:0x0152    F524      MOV       0x24,A
```

Pointeur spécifique stocké en 1B
Tab est à l'adresse 0B

Résultat test:
48 cycles en générique
15 cycles en spécifique

Fonction de gestion de pointeur générique CLDPTR

```
                C?CLDPTR:
C:0x01AD    BB0106    CJNE    R3,#0x01,C:01B6
C:0x01B0    8982      MOV     DPL(0x82),R1
C:0x01B2    8A83      MOV     DPH(0x83),R2
C:0x01B4    E0        MOVX    A,@DPTR
C:0x01B5    22        RET
C:0x01B6    5002      JNC     C:01BA
C:0x01B8    E7        MOV     A,@R1
C:0x01B9    22        RET
C:0x01BA    BBFE02    CJNE    R3,#PCA0CPH4(0xFE),C:01BF
C:0x01BD    E3        MOVX    A,@R1
C:0x01BE    22        RET
C:0x01BF    8982      MOV     DPL(0x82),R1
C:0x01C1    8A83      MOV     DPH(0x83),R2
C:0x01C3    E4        CLR     A
C:0x01C4    93        MOVC    A,@A+DPTR
C:0x01C5    22        RET
```

1

2

3

4

Rappel:
Paramètres passés à CLDPTR:
R3: type mémoire
R2: Adr MSB
R1: Adr LSB

- **1** - R3 = 01H → gestion XDATA
- **2** - R3 = 0H → gestion IDATA
- **3** - R3 = 0FEH → gestion PDATA (paged external data)
- **4** - Autre valeur de R3: gestion CODE

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 Code Size Execution Time	1 byte 4 bytes 4 cycles	2 bytes 9 bytes 7 cycles	3 bytes 11 bytes + library call 13 cycles

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

#pragma asm
Ajouter votre code ici
#pragma endasm

Attention à la manipulation de registres et de variables utilisés par ailleurs 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
}
```

Sous Microvision, il faut forcer à la compilation la génération d'un fichier Source Assembleur qui sera ensuite assemblé pour produire un fichier objet.

Appel de code Assembleur depuis le C
Coexistence de sources en C et en assembleur

Il n'est pas rare de devoir appeler des programmes assembleur depuis un code source en C. La réciproque est aussi possible mais est rarement utilisée de nos jours.

Dans le cas d'un code assembleur appelé depuis un source C, le cas le plus fréquent est l'appel d'un sous programme assembleur depuis un appel de fonction dans le source en C.

- Condition impérative: respecter les règles de passage de paramètres imposées par le compilateur.
  - De 0 à 3 paramètres - Passage de paramètres par registres (R1-R7)
  - Au-delà de 3 paramètres: Passage de paramètre par mémoires en utilisant des segments mémoires spécifiques

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 retournée

- La valeur retournée est toujours retournée 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

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"

```
<[>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 réentrantes

Keil C51 ne respecte pas la spécification du langage C,

les fonctions réentrantes 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();    }}
```

Entrées-sorties de base

Un grand nombre de fonctions d'entrées-sorties s'appuient sur les 2 fonctions ci-dessous. Elle existent par défaut et gèrent de manière basique le périphérique UART0.
Il est possible de les ré-écrire afin d'utiliser les fonctions de gestion d'I/O sur un autre périphérique.

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: User's Guide Cx51 Compiler 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.

Source: User's Guide Cx51 Compiler Keil

Exécution du code au démarrage du processeur (plus généralement, après un évènement « Reset »)

Sur le 8051F020:

- Un seul espace mémoire destiné au stockage de code: l'espace « code »
- Cette mémoire est une mémoire de techno « Flash », programmée via le dispositif de débogage.
- Après un évènement Reset, le processeur exécute le code placé à partir de l'adresse 0

Sur certains microcontrôleurs:

- Multiboots – plusieurs modes de démarrage à partir de divers périphériques
- Code spécifique de boot – Copie de mémoire – exploitation de mémoires externes (mémoires série)

Démarrage d'une application codée en C dans le microcontrôleur 8051F020

Dans un code bas niveau (pas d'OS qui supervise l'exécution du code) le point de départ correspond au **Reset du processeur**.

- Rappel: Point de vue processeur – l'exécution du code démarre à partir de l'adresse 0000
- Point de vue développeur d'application en C: c'est la fonction **Main** qui est exécutée en premier.
- Réellement sous Microvision: exécution du programme assembleur **startup.a51**, initialisation des variables globales, puis exécution du **Main**

Le fichier `Startup.a51`

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

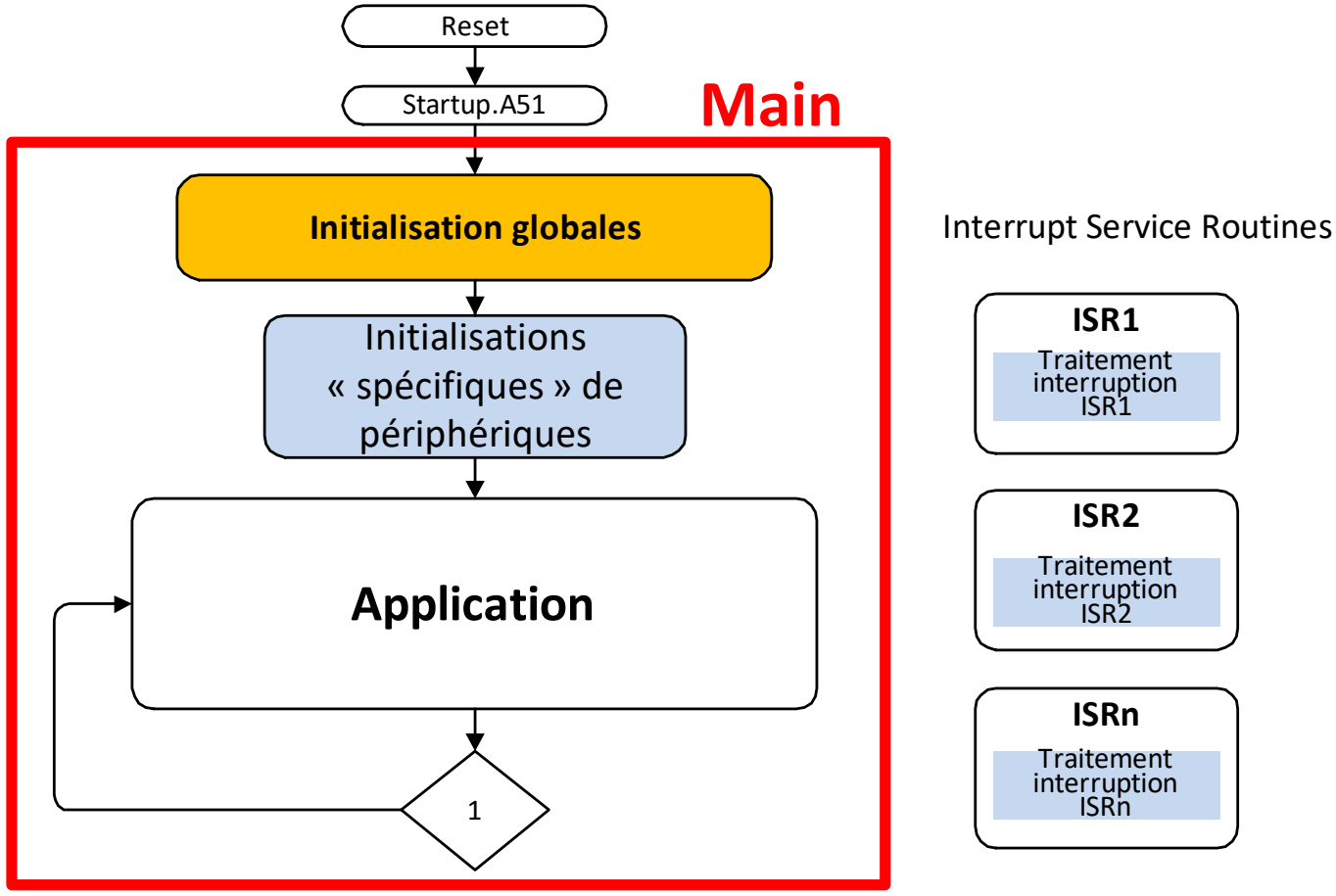
Startup Code

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

Ce fichier est paramétrable
- Zones mémoire à effacer
- Position des pointeurs
- - etc…

Algorigramme global d'une application



Sources documentaires

- **Datasheet 8051F020**
- **Documentation du compilateur C51 Keil (Aide en ligne)**
- **Documentation générique sur le langage C**

Source des illustrations dans ce cours:

- **Datasheet 8051F020 de Silicon Laboratories**
- **« Cx51 User's Guide » de Keil**
- **La famille SIMPSON**

LIVE AND DISCOVER

Contact

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

- CPE Lyon – Sciences du Numérique
- Campus Lyon Tech la Doua
- 43, bd du 11 novembre 1918 – Bâtiment Hubert Curien
- B.P. 2077 – 69616 Villeurbanne cedex – France