INFO2 : INFORMATIQUE EMBARQUEE
TD2 : Types de données et opérateurs logiques

1 Les types de données

Le microprocesseur, compte-tenu de sa technologie, ne sait représenter et traiter que des informations exprimées sous forme binaire. Quelle que soit sa nature initiale, toute information devra être codée sous cette forme. Connaître le contenu d'un emplacement de la mémoire (un ou plusieurs octets) ne suffit pas pour lui attribuer une signification (nombre entier, nombre réel, caractère, … ?). La définition d'un *type* pour cette information codée en binaire permet de lui attribuer une signification. La taille des types n'est que partiellement standardisée. Cette souplesse permet au langage d'être efficacement adapté à des processeurs très variés, mais elle complique la portabilité des programmes écrits en C. Nous décrirons ici les types de données supportés par le compilateur C MPLAB C18 de Microchip que nous utiliserons en TP.

1.1 Types entiers

La documentation du compilateur C MPLAB C18 de Microchip (**`MPLAB_C18_User_Guide_51288j.pdf`**) nous indique les types de données supportés suivant:

Type	Size	Minimum	Maximum
`char`(1,2)	8 bits	-128	127
`signed char`	8 bits	-128	127
`unsigned char`	8 bits	0	255
`int`	16 bits	-32,768	32,767
`unsigned int`	16 bits	0	65,535
`short`	16 bits	-32,768	32,767
`unsigned short`	16 bits	0	65,535
`short long`	24 bits	-8,388,608	8,388,607
`unsigned short long`	24 bits	0	16,777,215
`long`	32 bits	-2,147,483,648	2,147,483,647
`unsigned long`	32 bits	0	4,294,967,295

Note 1: A plain *`char`* is signed by default.
2: A plain *`char`* may be unsigned by default via the `-k` command-line option.

1.2 Types flottants

Cf. norme IEEE754.

Type	Size	Minimum Exponent	Maximum Exponent	Minimum Normalized	Maximum Normalized
`float`	32 bits	-126	128	$2^{-126} \approx 1.17549435e-38$	$2^{128} * (2-2^{-15}) \approx 6.80564693e+38$
`double`	32 bits	-126	128	$2^{-126} \approx 1.17549435e-38$	$2^{128} * (2-2^{-15}) \approx 6.80564693e+38$

1.3 Représentation des différentes bases et des codes ASCII

Voici quelques exemples illustrant les notations des différentes bases décimale, binaire ou hexadécimale ou ASCII :

```
char a = 50;                  // Un nombre seul représente un nombre décimal.

char a = 0b00110010;          // Un nombre précédé de 0b est un nombre binaire.

char a = 0x32;                // Un nombre précédé de 0x est un nombre hexadécimal.

char a = '2';                 // Un caractère entre ' et ' représente son code ASCII.
```

Remarques :

- **//** signifie que ce qui est à droite de ces symboles est du commentaire.
- **/*…** ***/** Tout ce qui se trouve entre ces symboles est du commentaire.

1.4 Exercice

Compléter le tableau suivant :

Type	Notation binaire	Notation Hexa	Valeur décimale
unsigned char	0b00110011		
signed char	0b00110011		
unsigned char	0b10110001		
char	0b10110001		
int		0x9FAB	
unsigned int		0x9FAB	
short		0xFFFF	
unsigned short		0xFFFF	

Que fait le programme suivant ?

```
void main(void)
{
        unsigned char a = 255;
        a = a + 1;
        printf("%u", a);
}
```

2 Les opérateurs de manipulation de bits

2.1 Opérations de base

Dans les exemples ci-dessous, les variables sont toutes de type **`unsigned char`** ou toutes de type **`char`** :

Opérateur	Exemple	Commentaire
~	`a = 0b00000110; x = ~a;`	Complément à 1 de **a**. **x** vaut **0b11111001**.
&	`a = 2; b = 3; x = a & b;`	ET bit à bit de **a** et **b**. **x** vaut **0b00000010**.
|	`a = 2; b = 5; x = a | b;`	OU bit à bit de **a** et **b**. **x** vaut **0b00000111**.
^	`a = 2; b = 7; x = a ^ b;`	OU Exclusif bit à bit de **a** et **b**. **x** vaut **0b00000101**.
>>	`a = 2; b = 1; x = a >> b;`	Valeur de a décalée à droite de **b** bits. **x** vaut **0b00000001**. Si **a** est non signé, insère **b 0** à gauche. Si **a** est signé, le bit de signe est propagé **b** fois. Correspond à une division ENTIERE de **a** par 2^b.
<<	`a = 2; b = 3; x = a << b;`	Valeur de a décalée à gauche de **b** bits. **x** vaut **0b00010000**. Insère **b 0** à droite. Correspond à une multiplication de **a** par 2^b.

2.2 Exercices sur les opérations de base

2.2.1 Variante 1

Donner la valeur de *x* après les opérations suivantes en tenant compte du type des variables déclarées :

```
// Déclaration des variables
unsigned char a;
unsigned char b;
unsigned char x;
```

Opération	Valeur de **a** en binaire	Valeur de **b** en binaire	Valeur de **x** en binaire	Valeur de **x** en hexadécimal
`a = 7; b = 5; x = a & b;`				
`a = 0xAA; b = 0x75; x = a ^ b;`				
`a = 0xD2; b = 1; x = a << b;` `x = x >> b;`				
`a = 64; x = ~a;`				
`a = 0xFE; b = 0xC9; x = a | b;`				

2.2.2 Variante 2

```
// Déclaration des variables
char a, b, x;
```

Opération	Valeur de **a** en binaire	Valeur de **b** en binaire	Valeur de **x** en binaire	Valeur de **x** en hexadécimal
`a = 7; b = 5; x = a ^ b;`				
`a = 0xAA; b = 0x75; x = a & b;`				
`a = 0x52; b = 1; x = a << b;` `x = x >> b;`				
`a = 0x55; x = ~a;`				
`a = 0xE; b = 0xC0; x = a | b;`				

2.3 Masquages

Dans les tableaux ci-dessous, **a** et **masque** sont des octets (type : **unsigned char**). **op** représente un opérateur bit à bit à choisir parmi ET, OU et OU EXCLUSIF. Il faut donner l'**op**érateur et le **masque** à appliquer à **a** afin d'obtenir la nouvelle valeur de **a** souhaitée (repérée par **a(après)**). Il faut également donner l'instruction correspondante en langage C.

2.3.1 Masque pour forcer un bit à 1

But : Forcer à 1 le bit de poids 3 sans modifier les autres bits.

a op. masque a(après)	0b b7 b6 b5 b4 b3 b2 b1 b0 (bi = 0 ou 1) ---------- = 0b b7 b6 b5 b4 1 b2 b1 b0	Instruction en C :

2.3.2 Masque pour forcer un bit à 0

But : Forcer à 0 le bit de poids 5 sans modifier les autres bits.

a op. masque a(après)	0b b7 b6 b5 b4 b3 b2 b1 b0 (bi = 0 ou 1) ---------- = 0b b7 b6 0 b4 b3 b2 b1 b0	Instruction en C :

2.3.3 Masque pour faire basculer l'état d'un bit

But : Faire basculer l'état du bit de poids 0 sans modifier les autres bits.

a op. masque a(après)	0b b7 b6 b5 b4 b3 b2 b1 b0 (bi = 0 ou 1) ---------- = 0b b7 b6 b5 b4 b3 b2 b1 /b0	Instruction en C :

2.4 Ecriture de fonctions de rotation

Ecrire une fonction de rotation à droite et une fonction de rotation à gauche qui permettent respectivement la rotation à droite et la rotation à gauche de *b* bits sur l'octet *a*.

Les prototypes de ces fonctions sont :

```
unsigned char RotD(unsigned char a, unsigned char b);
unsigned char RotG(unsigned char a, unsigned char b);
```