INITIATION AU LANGAGE C SUR PIC DE MICROSHIP

I. Historique du langage C

- 1972 : naissance du C dans les laboratoires BELL par Dennis Ritchie et développement du système d'exploitation UNIX.
- 1978 : vulgarisation du langage et sortie du livre de B. Kernighan et D. Ritchie "The C programming language".
- 1983 : normalisation A.N.S.I du langage C. Ce travail commence en 83 et dure 5 ans.
- 1988 : existence d'une norme : " C-ANSI ". La 2ème édition du livre de Kernighan et Ritchie devient la référence.

II. Les qualités du langage C

Le langage C est :

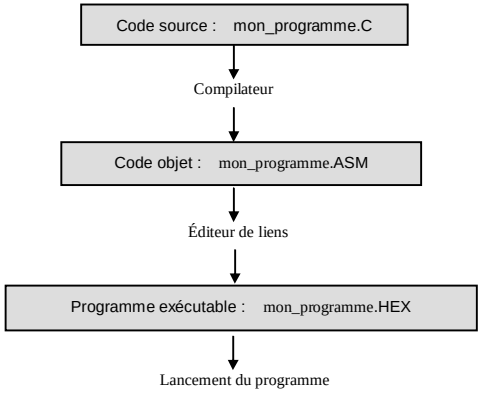
- PORTABLE : Les modifications d'un programme pour passer d'un système à un autre sont minimes.
- COMPLET : Un texte C peut contenir des séquences de bas niveau (proches du matériel) en assembleur.
- SOUPLE : Tout est possible en C mais une grande rigueur s'impose.
- EFFICACE : On réfléchit (devant une feuille de papier) et on écrit (peu).

III. Notion de filière de développement

On désigne ainsi l'ensemble des outils qui interviennent entre le texte source C et le code objet téléchargé dans le microcontrôleur PIC.

Les étapes de génération d'un programme écrit en langage C sont :

- **L'édition** du fichier source ***mon_programme.C*** avec un éditeur de texte (simple sans mise en forme du texte).
- **La compilation** du fichier source pour obtenir un fichier objet : ***mon_programme.ASM***. La compilation est la transformation des instructions C en instructions assembleur pour microcontrôleur PIC.
- **L'édition de liens** permet d'intégrer des fonctions prédéfinies. Le programme auxiliaire *Éditeur de liens (linker ou binder)* génère à partir du fichier *mon_programme.ASM* un fichier exécutable ***mon_programme*.HEX** compatible avec le PIC.
- Remarque : le cours suivant repose sur le compilateur PICC de HI-TECH



IV. Structure d'un programme en C

```
/* Tout ce qui se trouve entre ces symboles
est du commentaire */
// Ce qui est à droite de ces symboles est également du commentaire

void main()           // Un programme en C comporte au moins une fonction
                      // principale appele main.
                      // Nous verrons plus loin le terme void.

{                     // Les accolades définissent l'intérieur de la
                      // fonction.
      fonction_1() ; // Les espaces doivent être remplacés par _
      fonction_2() ; // Les fonctions sont séparée par un ;
      …
      fonction_n() ; // Attention à la casse car le C fait la différence
                      // entre minuscules et majuscules.
}                     // On aligne les accolades et on décale le corps de la
                      // fonction par souci de lisibilité.
```

Le programme doit également contenir la définition des différentes fonctions créées par le programmeur (voir § IX. Les Fonctions). Les fonctions doivent être définies (au moins leur prototype) avant d'être appelées par une autre fonction.

Le langage C comporte des bibliothèques de fonctions prédéfinies sous forme de fichiers comportant l'extension .h. Ces fonctions seront ajoutées au programme exécutable lors de *l'édition de liens*. Pour incorporer dans un programme un fichier .h, on utilise la commande #include <*fichier*.h> placée habituellement en début de fichier. Exemples :

```
#include <htc.h>            // htc.h contient des fonctions réalisant des
                            // temporisations logiciels.
#include <pic16f877a.h>     // Grâce à ce fichier .h, le compilateur connaît
                            // l'adresse de chaque registre et port
                            // Ex : PORTB correspond à l'adresse 06, etc.
```

V. Composition d'un programme en C

Un programme en C utilise 2 zones mémoire principales :

- La zone des VARIABLES est un bloc RAM où sont stockées des données manipulées par le programme.
- La zone des FONCTIONS et CONSTANTES est un bloc ROM qui recevra le code exécutable du programme.

Avant d'utiliser une variable, une fonction ou une constante, il faut la déclarer afin d'informer le compilateur de son existence.

Leur nom que l'on utilise est un *identificateur*. Leur écriture doit :

- Utiliser les lettres de l'alphabet, de a à z, et de A à Z, les chiffres de 0 à 9 (sauf pour le premier caractère), le souligné (_).
- Ne contenir ni espace, ni caractère accentué.
- Être représentative de leur rôle dans le programme.

VI. Les différents types de valeur du langage C

Toutes les valeurs (constantes et variables) utilisée en C sont classées selon des types. Un type décide de l'occupation mémoire de la donnée. Pour déclarer correctement une variable ou une constante, il faut donc savoir auparavant ce qu'elle va contenir. On distingue les types suivants :

```
char a ;              // Déclare un entier signé 8 bits [-128 à +127 ]

unsigned char b ;     // Déclare un caractère non signé 8 bits [0 à 255]

int c ;               // Déclare un entier signé 16 bits [-32768 à +32767 ]

unsigned int d ;      // Déclare un entier non signé 16 bits [0 à 65535]

long e ;              // Déclare un entier signé 32 bits
                      // [-2147483648 à +2147483647]

unsigned long f ;     // Déclare un entier non signé 32 bits
                      // [0 à 4292967295]

float g;              // Déclare un réel signé 32 bits dont la valeur
                      // absolue est comprise entre 3,4.10-38 et 3,4.10+38

double h;             // Déclare un réel signé 64 bits dont la valeur
                      // absolue est comprise entre 1,7.10-308 et 1,7.10+308

long double i;        // Déclare un réel signé 80 bits dont la valeur
                      // absolue est comprise entre 3,4.10-4932 et 3,4.10+4932
```

VII. Représentation des différentes bases et des codes ASCII

```
int a = 4 ;           // Un nombre seul représente un nombre décimal.

int b = 0b1010 ;      // Un nombre précédé de 0b est un nombre binaire.

int p = 0x00FF ;      // Un nombre précédé de 0x est un nombre hexadécimal.

char c = 'x' ;        // Un caractère entre '' représente son code ASCII.
```

VIII. Les opérateurs

A. *L'opérateur d'affectation*

Cet opérateur a déjà été utilisé dans les exemples précédents. Il permet, entre autres, d'initialiser une variable.

```
=    Exemple : a = 5 ;              // Range 5 dans a.
               PORTB = 0 ;          // le PORTB est mis à 0
```

Attention : Le transfert de la valeur va toujours de la droite vers la gauche du signe égal.

B. *Les opérateurs arithmétiques*

```
+    Exemple : a = 5 ; b = 4 ; x = a+b ;  // rend la somme de a et b.
                                          // x vaut 9

-    Exemple : a = 5 ; b = 4 ; x = a-b ;  // rend la soustraction de a et b.
                                          // x vaut 1

*    Exemple : a = 5 ; b = 4 ; x = a*b ;  // rend la multiplication de a et b.
                                          // x vaut 20

/    Exemple : a = 10 ; b = 3 ; x = a/b ; // rend le quotient de la division
                                          // entière de a et b.
                                          // x vaut 3

%    Exemple : a = 10 ; b = 3 ; x = a%b ; // rend le reste de la division
                                          // entière de a et b.
                                          // x vaut 1
                                          // % se prononce « modulo »
```

C. *Les opérateurs de manipulation de bits*

```
~    Exemple : a = 0b0110 ; x = ~a ;      // rend le complément de a.
                                          // x vaut 0b1001

&    Exemple : a = 2 ; b = 3 ; x = a&b ;  // rend le ET bit à bit de a et b.
                                          // x vaut 0b10

|    Exemple : a = 2 ; b = 5 ; x = a|b ;  // rend le OU bit à bit de a et b.
                                          // x vaut 0b111

^    Exemple : a = 2 ; b = 7 ; x = a^b ;  // rend le OU EXCLUSIF bit à
                                          // bit de a et b.
                                          // x vaut 0b101

>>   Exemple : a = 2 ; b = 1 ; x = a>>b ; // rend la valeur de a décalée à
                                          // droite de b bits.
                                          // x vaut 0b01
<<   Exemple : a = 2 ; b = 3 ; x = a<<b ; // rend la valeur de a décalée à
                                          // gauche de b bits.
                                          // x vaut 0b10000
```

D. *Les opérateurs de tests*

Remarque : En C, **FAUX** est la valeur **0**, **VRAI** est tout ce qui est **≠ 0**.

```
>    Exemple : a = 6 ; x = a>4 ;          // rend VRAI si a est supérieure
                                          // à 4. FAUX sinon.
                                          // x vaut VRAI

>=   Exemple : a = 2 ; x = a>= 2 ;        // rend VRAI si a est supérieure ou
                                          // égale à 2. FAUX sinon.
                                          // x vaut VRAI
```

```
<    Exemple : a = 6 ; x = a<3 ;          // rend VRAI si a est inférieure
                                          // à 3. FAUX sinon.
                                          // x vaut FAUX

<=   Exemple : a = 3 ; x = a<= 6 ;        // rend VRAI si a est inférieure ou
                                          // égale à 6. FAUX sinon.
                                          // x vaut VRAI

==   Exemple : a = 6 ; x = a == 5 ;       // rend VRAI si a est égale à 5.
                                          // FAUX sinon.
                                          // x vaut FAUX

!=   Exemple : a = 4 ; x = a != 2 ;       // rend VRAI si a est différente
                                          // de 2. FAUX sinon.
                                          // x vaut VRAI

&&   Exemple : a = 9 ; b = 1 ;            // ET LOGIQUE : rend VRAI si les
               x = (a == 9) && (b != 8);  // deux tests sont VRAIS.
                                          // FAUX sinon.
                                          // x vaut VRAI

||   Exemple : a = 6 ; b = 3 ;            // OU LOGIQUE : rend VRAI si au
               x = (a == 5) || (b != 3);  // moins un des deux tests
                                          // est VRAI. FAUX sinon.
                                          // x vaut FAUX

!    Exemple : a = 1 ; x = !a ;           // NEGATION LOGIQUE : rend VRAI si
                                          // a est FAUX. FAUX sinon.
               y = !(a == 5) ;            // x vaut FAUX et y vaut VRAI
```

Remarque :
Les opérateurs ont une priorité. Cette priorité n'est pas forcément celle des mathématiques et varie d'un langage informatique à un autre. Il vaut donc mieux utiliser les parenthèses pour éviter tous problèmes.

IX. Les Fonctions

A. *Présentation*

Un programme en C est un ensemble de fonctions :

- La fonction principale *main* qui est la première fonction appelée lors de l'exécution du programme.
- Les fonctions écrites par le programmeur qui doivent être déclarées avant leur appel.
- Fonctions prédéfinies issues des bibliothèques standards du compilateur (dont le code n'est pas écrit par le programmeur mais inséré dans le programme par l'éditeur de liens grâce au fichier.h).

Pour qu'un programme soit structuré, chaque fonction doit effectuer une tâche bien spécifique.

Exemple :

Programme en C appelant des fonctions prédéfinies et des fonctions écrites par le programmeur.

```
#include <pic16f877a.h>       // Fichier de définition des adresses des
                              // registres du PIC 16F877A

#include <delay.h>            // Fichier de fonctions prédéfinies pour
                              // temporisations logicielles
                      // Note prof : <delay.h> le fichier se trouve dans le
                      // répertoire des fichiers d'entête du compilateur
                      // "delay.h" le fichier se trouve dans le même
                      // répertoire que le fichier source

// Directives d'assemblage




// Définitions des fonctions écrite par le programmeur

void PORTB_en_sortie(void)
{
                TRISB=0;      // PORTB en sortie si TRISB=0
}

void Allumer_LED_PORTB(void)
{
                PORTB=0xFF;   // RB0 à RB7 mis à 1
}

void Eteindre_LED_PORTB(void)
{
                PORTB=0x00;   // RB0 à RB7 mis à 0
}


// Fonction principale

void main(void)
{
    PORTB_en_sortie();        // fonction écrite par le programmeur
    while(VRAI)               // Répéter toujours
                              // voir § La structure alternative ou sélection page 5
    {
       Allumer_LED_PORTB();   // fonction écrite par le programmeur
       __delay_ms(250) ;      // fonction prédéfinie
                              // Note prof : Le fichier htc.c doit être
                              // ajouté au projet MPLAB en plus du htc.h
       Eteindre_LED_PORTB() ;// fonction écrite par le programmeur
       __delay_ms (250) ;     // fonction prédéfinie
    }
}
```

B. *Syntaxe d'écriture d'une fonction*

<type de la valeur de retour> **nom_fonction(***<liste des paramètres reçus par la fonction>***)**
{

définitions des variables locales ;

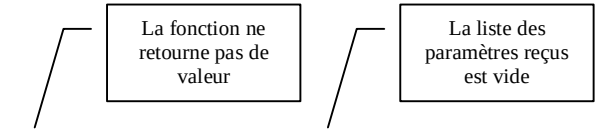
instructions ;

}

Remarques :

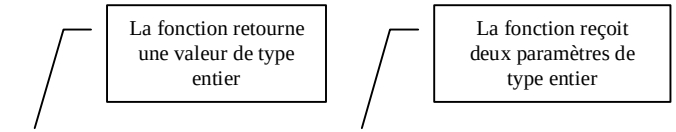
- La liste des paramètres reçus peut-être vide.
- La valeur de retour peut être de n'importe quel type :
  int, **float**…
  void si la fonction ne retourne pas de valeur,
- Une variable définie dans une fonction n'existe que dans celle-ci (variable locale).

- Exemple de fonction dont la liste des paramètres reçus est vide et ne retournant pas de valeur :




```
void Allumer_LED_PORTB(void)
{
                PORTB=0xFF;    // RB0 à RB7 mis à 1
}
```

- Exemple de fonction recevant 2 entiers et retournant la somme de ces derniers :




```
int Ma_fonction_somme(int a, int b)
{
        int resultat ;          // déclaration d'une variable locale appelée
                                // resultat

        resultat = a+b ;        // Les variables a et b ont été déclarées et
                                // initialisées lors de l'appel de la fonction

        return resultat ;       // l'instruction return permet de retourner
                                // une valeur à la fonction appelante.
}
```

C. *Appel de fonction avec passage de paramètres par valeur*

La valeur du paramètre passé est recopiée dans une variable locale à la fonction appelée. Une modification de cette variable n'a aucun effet sur la variable la fonction appelante.

Exemple de fonction *main* appelant *Ma_fonction_somme* :

```
void main()
{
        int a,x,y,z ;           // déclaration de quatre entiers a, x, y et z;
        x = 1 ;                 // initialisation des variables locales
        y = 2 ;
        a = 5 ;

        z = Ma_fonction_somme(x,y) ;  // z vaut la somme de x et de y.
}
```

Dans la fonction *main*, la variable locale **a** vaut **5**.

Dans la fonction *Ma_fonction_somme*, la variable locale **a** vaut **1** et la variable locale **b** vaut **2**.

Après appel de la fonction *Ma_fonction_somme*, la variable **a** de la fonction *main* vaut **toujours 5**.

Bien sûr, z vaut **3**.

X. Les variables Globales (Permanentes) et Locales (Temporaires)

Une ***variable globale*** (ou permanente) est **déclarée en en-tête du programme**. Elle est valide pendant toute la durée d'exécution du programme car elle fait l'objet d'une *réservation mémoire permanente en RAM*. Elle peut être utilisée et modifiée par toutes les fonctions du programme.

Une ***variable locale*** (ou temporaire) est déclarée **à l'intérieur d'une fonction**. Son existence est limitée à la durée d'exécution de cette fonction. Elle est donc ignorée par les autres fonctions. Elle peut (bien que cette façon de procéder soit déconseillée), porter le même nom qu'une variable globale ou qu'une autre variable locale se trouvant dans une autre fonction.
Par défaut, une variable locale est *rangée en pile LIFO* mais elle peut être *rangée en RAM à une adresse fixe (statique)*. Elle n'en demeure pas moins visible qu'à l'intérieur de la fonction où elle a été déclarée.

Attention : la pile et la mémoire RAM des PIC sont très limitées.

Exemple :

```
Void fonction_exemple()
{
        int a=12 ;      // variable locale a en pile.
        static char b ; // variable locale b en RAM à une adresse fixe.
}
```

XI. Algorithme

A. *Définition*

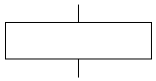
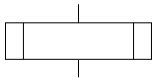
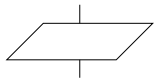
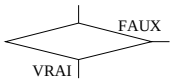
Un algorithme est un ensemble de règles opératoires rigoureuses, ordonnant à un processeur d'exécuter dans un ordre déterminé une succession d'opérations élémentaires, pour résoudre un problème donné. C'est un outil méthodologique général qui ne doit pas être confondu avec le programme proprement dit.

Un algorithme peut être :

- Représenté graphiquement par un **organigramme (ou ordinogramme).**
- Écrit sous forme littérale avec un **langage algorithmique.**

B. *Organigramme*

L'organigramme est une **représentation graphique normalisée** utilisée pour analyser ou décoder un problème. Il utilise des symboles représentant des traitements, des liaisons, des données…
Un organigramme bien représenté doit être fléché et fermé, compris entre un début et une fin.

SYMBOLE	DÉSIGNATION
	**Traitement** Opération ou groupe d'opération sur des données, instructions, etc.
	**Sous-programme** Portion de programme considérée comme une simple opération.
	**Entrée-Sortie** Mise à disposition d'une information à traiter ou enregistrement d'une information traitée.
	**Embranchement** Test, exploitation de conditions variables impliquant le choix d'une parmi deux. Symbole utilisé pour représenter une décision.
	**Début, fin ou interruption** Début, fin ou interruption d'un organigramme
Sens conventionnel des liaisons. Le sens général de liaison doit être : ▪ De haut en bas, ▪ De gauche à droite.	

C. *Structures algorithmiques fondamentales*

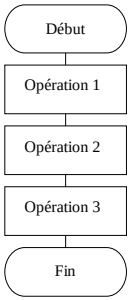
Les opérations relatives à la résolution d'un problème peuvent en fonction de leur enchaînement, être organisées selon trois familles de structures :

- Structures linéaires,
- Structures alternatives,
- Structures répétitives.

1. La structure linéaire ou séquence

La structure linéaire se caractérise par une **suite d'actions à exécuter successivement** dans l'ordre de leur énoncé.

Organigramme	Langage algorithmique	Exemple en langage C

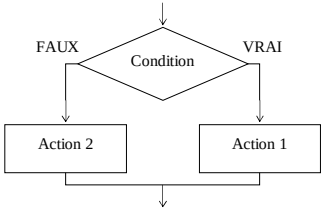
	Début algorithme :
	FAIRE opération 1
	FAIRE opération 2
	FAIRE opération 3
	Fin algorithme.
		void main( )
		{
		fonction_1( ) ;
		fonction_2( ) ;
		fonction_3( ) ;
		}

2. La structure alternative ou sélection

La structure alternative n'offre que deux issues possibles s'excluant mutuellement. Les structures alternatives définissent une **fonction de choix** ou de **sélection** entre l'exécution de l'un ou de l'autre des deux traitements. Également désignées par **structures conditionnelles**, elles sont représentatives de **saut** ou rupture de séquence.

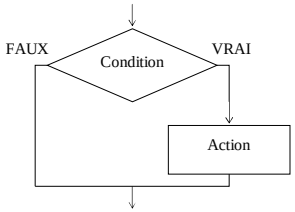
a) La structure alternative complète

Organigramme	Langage algorithmique	Exemple en langage C

	**SI** condition VRAIE
	ALORS action 1
	SINON action 2
	FIN SI
		if(condition = = VRAI)
		{
		fonction_1( ) ;
		}
		else
		{
		fonction_2( ) ;
		}

b) La structure alternative réduite

Organigramme	Langage algorithmique	Exemple en langage C

	**SI** condition VRAIE
	ALORS action
	FIN SI
		if(condition = = VRAI)
		{
		fonction( ) ;
		}

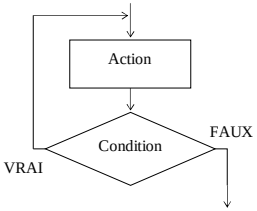
3. Les structures répétitives

Une structure répétitive ou itérative répète l'exécution d'un traitement.

a) Faire… Tant que…

Dans cette structure, le traitement est exécuté une première fois puis sa répétition se poursuit jusqu'à ce que la condition soit vérifiée.

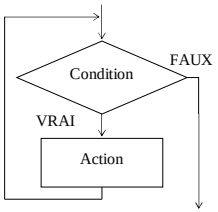
Organigramme	Langage algorithmique	Exemple en langage C

	**FAIRE**
	action
	TANT QUE condition VRAIE
		do
		{
		fonction( ) ;
		}
		while(condition = = VRAI) ;

b) Tant que… Faire…

Dans cette structure on commence par tester la condition, si elle est vraie alors le traitement est exécuté.

Organigramme	Langage algorithmique	Exemple en langage C

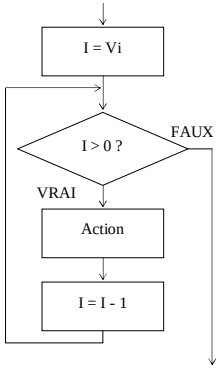
	**TANT QUE** condition VRAIE
	FAIRE action
	FIN TANT QUE
		while(condition = = VRAI)
		{
		fonction( ) ;
		}

4. La structure répétition contrôlée

Dans cette structure la sortie de la boucle d'itération s'effectue lorsque le nombre souhaité de répétition est atteint.
D'où l'emploi d'une variable de boucle (indice I) caractérisée par :

- Sa valeur initiale.
- Sa valeur finale.
- Son pas de variation.

Organigramme	Langage algorithmique	Exemple en langage C

	**POUR** I = Vi à I = 0 et par pas de 1
	FAIRE action
	FIN POUR
		for(I = Vi ; I > 0 ; I = I – 1)
		{
		fonction( ) ;
		}
		Équivalent à :
		I=Vi;
		while(I>0)
		{
		fonction( );
		I=I-1;
		}

Remarque : En C, la séquence est exécutée tant que la condition est vraie.

XII. Les masques

Les masques sont utilisés en informatique pour tester l'état d'un ou plusieurs bits d'un mot binaire. Ils permettent également de forcer un ou plusieurs bits à un état désiré (0 ou 1). On utilise, pour effectuer des masques, les propriétés de la logique de Boole à savoir : l'élément absorbant et l'élément neutre du OU, du ET et du OU EXCLUSIF.

A. *Un masque pour forcer un bit à 1*

Soit l'état du PORTB = 0b10010111. On veut forcer le bit 3 à '1' sans modifier les autres bits.

Propriété utilisée : Élément absorbant et élément neutre du OU logique.

		Équivalent en C :
PORTB :	1 0 0 1 0 1 1 1	
Masque : +	0 0 0 0 1 0 0 0	`PORTB = PORTB | 0b00001000 ;`
Résultat :	0 0 0 0 **1** 0 0 0	

B. *Un masque pour forcer un bit à 0*

Soit l'état du PORTB = 0b10010111. On veut forcer le bit 1 à '0' sans modifier les autres bits.

Propriété utilisée : Élément absorbant et élément neutre du ET logique.

		Équivalent en C :
PORTB :	1 0 0 1 0 1 1 1	
Masque : .	1 1 1 1 1 1 0 1	`PORTB = PORTB & 0b11111101 ;`
Résultat :	1 0 0 1 1 1 **0** 1	

Remarque : Le fichier pic16f877a.h contient la définition de chaque bit des registres et ports de notre PIC. Ceci permet de les forcer directement sans passer par les masques :

Pour forcer le bit 3 du PORTB à 1 : `RB3=1 ;`

Pour forcer le bit 1 du PORTB à 0 : `RB1=0 ;`

Il faut prendre connaissance de la syntaxe et de la casse des registres, des ports et des bits en éditant le fichier pic16f877a.h. Les masques restent très utiles pour la manipulation des bits non définis dans ce fichier

C. *Un masque pour faire basculer l'état d'un bit*

Soit l'état du PORTB = 0b100X0111. On veut faire basculer l'état de RB4 sans modifier les autres bits.

		Équivalent en C :
PORTB :	1 0 0 X 0 1 1 1	
Masque : ⊕	0 0 0 1 0 0 0 0	`PORTB = PORTB ^ 0b00010000 ;`
Résultat :	1 0 0 $\overline{X}$ 0 1 1 1	

Équivalent à : RB4 = !RB4 ;

D. *Un masque pour tester l'état d'un bit*

Rappel : En C, **FAUX** est la valeur **0**, **VRAI** est tout ce qui est **≠ 0.**

On veut tester l'état du bit 5 du PORTB. Si RB5 = 1 alors on appelle la fonction action_1(), sinon, on appelle la fonction action_2() ;

Il y a deux solutions :

Masque en ET :	Masque en OU :

`if(PORTB & 0b00100000)`
`{`
`action_1() ; // alors RB5 = 1`
`}`
`else`
`{`
`action_2() ; // sinon RB5 = 0`
`}`
	`if(PORTB | 0b11011111 == 0xFF)`
	`{`
	`action_1() ; // alors RB5 = 1`
	`}`
	`else`
	`{`
	`action_2() ; // sinon RB5 = 0`
	`}`

Remarque : grâce au contenu du fichier pic16f877a.h, on peut également écrire :

```
if(RB5)
{
        action_1() ;   // alors RB5 = 1
}
else
{
        action_2() ;   // sinon RB5 = 0
}
```