Table des matières

Chapitre 1 : L'informatique embarquée 4
1. Introduction 4
2. Les Microcontrôleurs 4
3. La famille PIC 5
4. Architecture de PIC18F2620 5
5. Exemples de schémas bloc d'un système embarqué 5
Chapitre 2 : La base de la programmation 8
1. La variable 9
2. Les opérateurs 10
3. La structure de choix 11
4. La structure répétitive 11
Chapitre 3 : La séquence 12
1. Les entités de l'ordinogramme 12
2. La séquence en ordinogramme 12
3. La séquence en C 15
Chapitre 4 : La structure de choix simple 18
1. Si - Alors 18
2. Si – Alors - Sinon 19
Chapitre 5 : La « tant que » 20
1. Principe général 20
2. La boucle infinie 20
Chapitre 6 : Exemples de programmes 22
1. Le chronomètre 22
2. La carré d'un nombre 23
3. La moyenne de deux nombres 24
Chapitre 7 : La « répéter … tant que » 25
1. Principe général 25
2. Faire une « répéter…tant que » avec une « tant que » 26
3. Une « répéter…tant que » avec un bloc d'actions vide 27
Chapitre 8 : La boucle « pour » 28
1. Principe général 28
2. Faire une « pour » avec une « tant que » 29
Chapitre 9 : Les types en C 30
1. Précisions générales 30
2. Les différents types en C 31
Chapitre 10 : Les opérateurs logiques 32
1. L'algèbre de Boole 32
2. Les opérateurs logiques sur entiers 34
3. Les opérateurs logiques bit à bit 34
Chapitre 11 : Applications des opérateurs 35
1. Les masques binaires 35
2. Les opérateurs de décalage 35
Chapitre 12 : Les fonctions : notions de base 36
1. Introduction 36

2. Trois aspects importants liés à la fonction 36
3. Les transmissions d'informations liées à la fonction 37
Chapitre 13 : Les bits indicateurs d'état. 40
Chapitre 14 : Les entiers indicateurs d'état 41
1. Généralités 41
2. La structure de choix multiple 42
3. Gestion des menus à l'aide de la structure de choix multiple 44
Chapitre 15 : LDA 46
1. Langage de description algorithmique 46
2. Transformation ordinogramme en LDA 47
3. Exemple de transformation ordinogramme en LDA 48
Chapitre 16 : Eléments supplémentaires 49
1. Opérateurs supplémentaires 49
2. L'instruction break 49
3. Les tableaux et les chaînes de caractères 50
4. Les pointeurs 51
5. Notions supplémentaires sur la fonction 53
6. Les interruptions 56
Chapitre 17 : Qualité d'un programme 57
1. Les commentaires 57
2. Les indentations 58
Chapitre 18 : Programmer sa carte info 60
1. Le registre TRIS 60
2 Le registre ADCON1 60
3. Le PORTB 60
4. La directive #include 61
5. La directive #define 61
6. La boucle infinie 61
7. Construction d'un projet 62
Annexe 1 : Structure simplifiée d'un microcontrôleur 65
Annexe 2 : Brochage du PIC18F2620 65
Annexe 3 : Mon 1er programme 66
Annexe 4 : Schéma de principe de la carte info 67
Annexe 5 : PCB de la carte info (version 2009) 68
Bibliographie et liens internet 69
Table des figures 69
Questions de révision 70

Chapitre 1 : L'informatique embarquée

1. Introduction

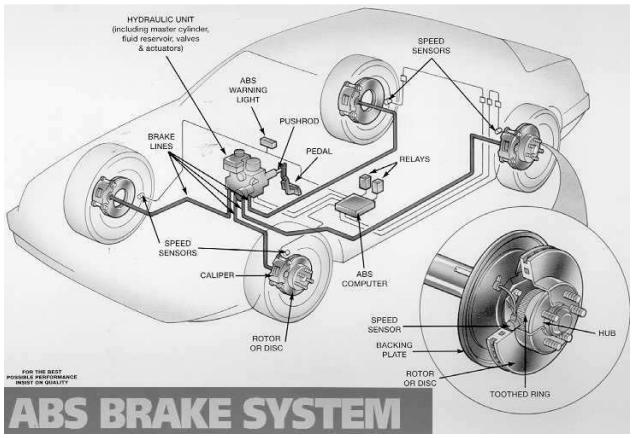
L'informatique embarquée est l'informatique destinée à un dispositif électronique.
On peut avoir de l'informatique embarquée dans un lecteur dvd, dans un réveil,…

Un système embarqué est destiné à prendre en compte des informations en provenance du monde réel et, en fonction de ces informations, à agir sur le monde réel.

Il doit donc disposer des éléments suivants :

- Interfaçage avec des capteurs
- Moyen de calcul et de prise de décision
- Interfaçage avec des actionneurs

Figure 1 : Exemple de système embarqué, l'ABS



Des capteurs de vitesse sur les roues renseignent au calculateur sur un éventuel blocage.
Le calculateur, après traitement, agit sur les cylindres de frein pour moduler la pression et éviter le blocage des roues.

2. Les Microcontrôleurs

Nous pouvons faire une distinction entre le microprocesseur (ou processeur) et le microcontrôleur :

- Le microprocesseur est le calculateur inclus au sein d'un microcontrôleur, c'est lui qui va exécuter le programme qu'on lui fournit. Il est donc capable de réaliser certaines opérations (mathématiques ou autres).
  Pour donner la cadence à laquelle le processeur exécute ses instructions, celui-ci est souvent accompagné d'un oscillateur à quartz.
- Le microcontrôleur est le composant complet, il possède un processeur interne mais est également accompagné d'une mémoire Programme (ROM) qui contient les instructions, une mémoire Données (RAM) qui contient des données utiles à l'exécution du programme et des ports d'entrées/sorties qui permettent au microcontrôleur d'interagir avec le monde extérieur (voir Annexe 1, p.65).

3. La famille PIC

PIC signifie « Programmable Interrupt Controler ».
Arizona Microchip propose des centaines de modèles différents regroupés en 5 familles : PIC10, PIC12, PIC16, PIC17, PIC18.

4. Architecture de PIC18F2620

Le PIC18F2620 (dont le brochage est donné Annexe 2, p.65) est celui qui correspond à un boîtier type DIL avec 28 broches. En gros il dispose de :

- broches nécessaires à son alimentation (5V DC)
- 2 broches pour l'oscillateur à quartz
- 3 ports d'entrées/sorties (8 bits par port)
- La possibilité de le programmer et de faire un reset du programme
- La possibilité d'utiliser un bus I^2C
- La possibilité d'utiliser 10 entrées analogiques
- 2 sorties PWM
- Une mémoire EEPROM pour la sauvegarde en cas de coupure d'alimentation

En se référant à la documentation du constructeur, il est possible d'accéder à l'architecture interne de ce PIC. On peut voir apparaître les 3 ports du PIC, l'ALU (l'unité arithmétique et logique) nécessaire aux opérations et différents blocs qui gèrent les données, la configuration et l'exécution des instructions.

5. Exemples de schémas bloc d'un système embarqué

Nous allons mettre en évidence le principe général de dispositifs électroniques qui utilisent un microcontrôleur. On commence souvent par exposer le schéma bloc du dispositif pour faire apparaître les entrées et les sorties au système, par convention les entrées se placent à gauche et les sorties à droite.

Le microcontrôleur est l'élément central du schéma bloc, son rôle est d'agir correctement[*] sur les sorties en prenant en compte les entrées du système !

Pour établir un schéma bloc, il est nécessaire de déterminer les entrées et les sorties du système, pour cela on peut tenter de répondre à deux questions :

Que doit faire le système ?
La réponse à cette question informe sur l'ensemble des sorties du point de vue du microcontrôleur.
De quoi le microcontrôleur a-t-il besoin pour faire ce qu'il doit faire ?
La réponse à cette question informe sur l'ensemble des entrées du point de vue du microcontrôleur.

[*] Conformément au comportement désiré à l'aide du programme informatique destiné au microcontrôleur.

Exemple 1 : Système d'affichage de la température et de la pression

Imaginons un système qui doit pouvoir afficher la température ambiante ainsi que la pression, une première chose à faire pour représenter le système est de tenter de répondre aux questions suivantes :

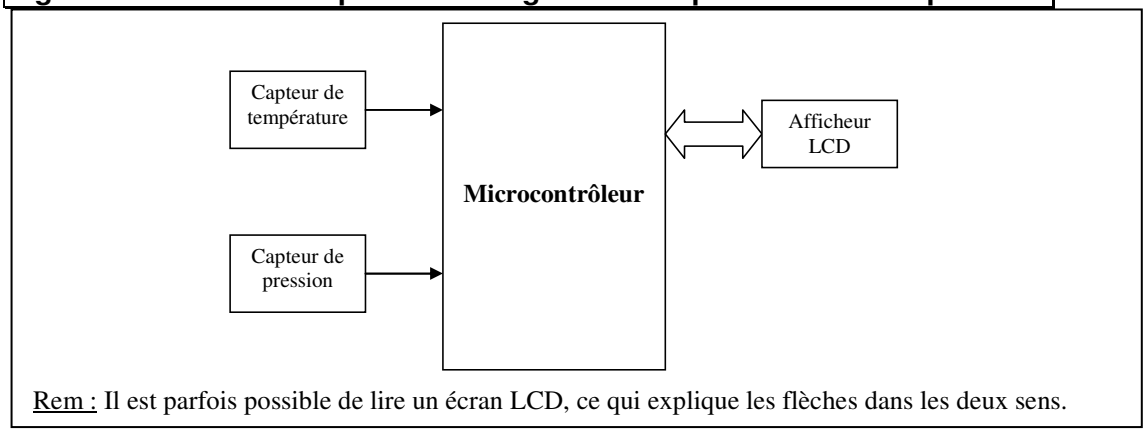
Que doit faire le système ?

Il doit pouvoir afficher le résultat, il faut donc un écran.
Il y a donc une sortie, l'afficheur.

De quoi le microcontrôleur a-t-il besoin pour faire ce qu'il doit faire ?

Il a besoin de connaître la température et la pression, il faut donc des capteurs.
Il y a donc deux entrées (analogiques).

Figure 2 : Schéma bloc pour l'affichage de la température et de la pression



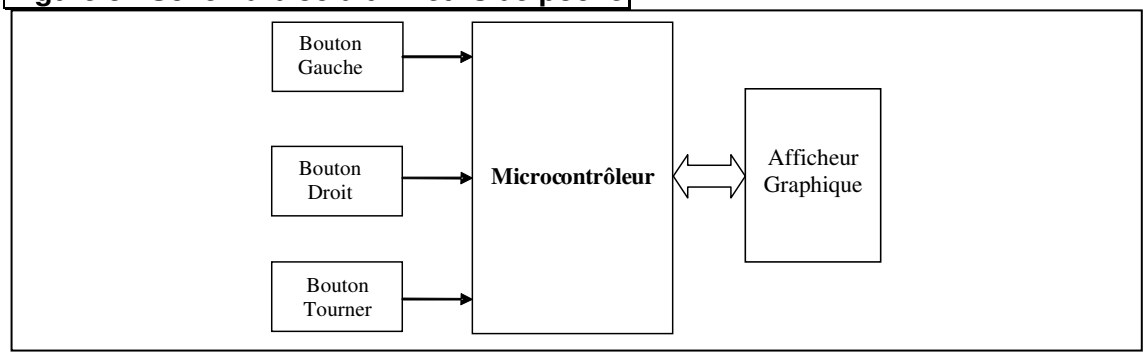
Rem : Il est parfois possible de lire un écran LCD, ce qui explique les flèches dans les deux sens.

Ici le programme qu'il faudra fournir au microcontrôleur devra simplement se charger d'acquérir les nouvelles valeurs de température et de pression et de rafraîchir l'écran.

Exemple 2 : Réalisation d'un Tétris de poche

Imaginons le Tétris le plus simple possible avec simplement 3 boutons (gauche, droite et tourner) et un écran graphique.

Figure 3 : Schéma bloc d'un Tétris de poche



Le schéma bloc ci-dessus fait apparaître les entrées et les sorties du système.
Ces entrées/sorties ont pu être déterminées en répondant à nos deux questions :

Que doit faire le système ?

Il doit afficher les pièces qui descendent, qui tournent,… tout se rapporte à l'affichage. Il n'y a donc qu'une sortie, l'afficheur graphique.

De quoi le microcontrôleur a-t-il besoin pour faire ce qu'il doit faire ?

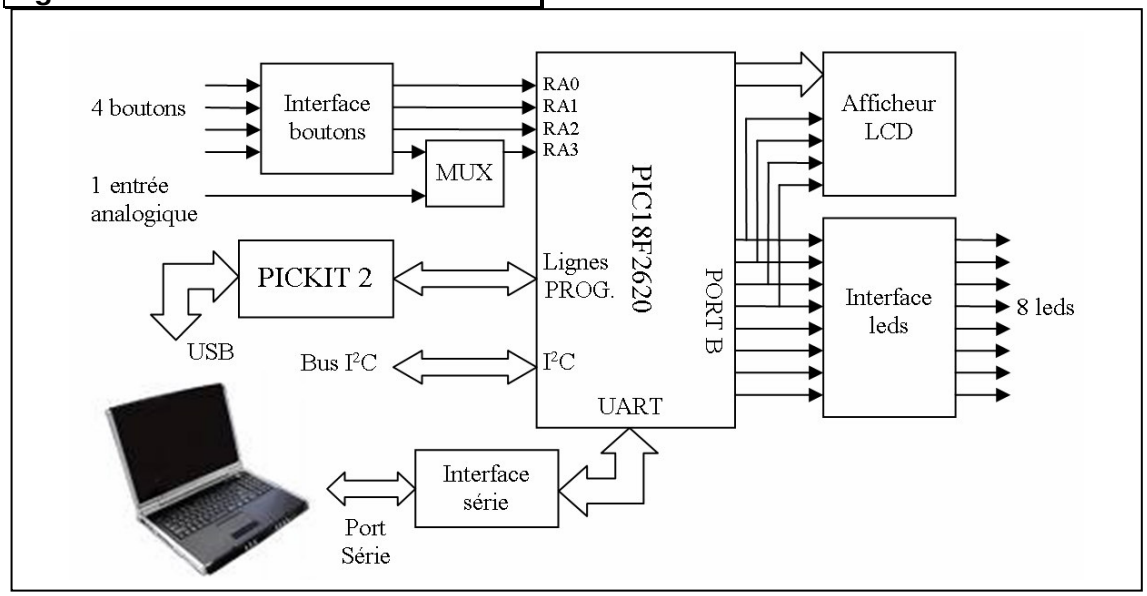
Il a besoin de "savoir " s'il faut déplacer la pièce à gauche, à droite ou la tourner. Il y a donc 3 entrées : les 3 boutons.

Si l'on compare ce schéma bloc avec celui de l'exemple 1 (Fig.2), on peut constater que la difficulté du schéma bloc semble plus ou moins équivalente (quelques entrées et un écran en sortie dans les deux cas). Cependant le programme du tétris de poche est plus difficile à développer car l'état du système ne dépend pas uniquement de l'état d'enfoncement des boutons mais également de la position actuelle des pièces (qui proviennent néanmoins des enfoncements déjà effectués des boutons).
Il apparaît la nécessité d'utiliser des éléments supplémentaires pour qualifier l'état du système, ce sont des variables. Nous verrons qu'il est possible de stocker des informations en RAM par l'intermédiaire de variables, on pourrait par exemple mémoriser le type de pièce en cours de manipulation, la position de cette pièce,…

Exemple 3 : Projet utilisant la carte info

Imaginons que l'on veuille utiliser la carte info pour gérer un système embarqué, nous pouvons soit l'utiliser telle quelle (et par exemple faire une application comme une calculatrice), soit ajouter des entrées/sorties supplémentaires. Nous pourrions alors faire, par exemple, un robot suiveur de ligne qui incorpore la carte info.

Figure 4: Schéma bloc de la carte info



Nous avons vu précédemment, en comparant les exemples 1 et 2, que ce n'est pas uniquement le nombre d'entrées/sorties qui détermine la difficulté du programme, c'est surtout le type d'application. Autrement dit, avec une carte de développement comme celle-ci, il est possible d'imaginer des applications à tout niveau de difficulté.

Chapitre 2 : La base de la programmation

L'objectif de ce chapitre est de faire ressortir tout ce qui est ESSENTIEL afin d'apporter une vision globale de la programmation, de commencer par bien fixer le contexte général.

La programmation est essentiellement basée sur les notions ci-dessous :

- La variable

  La variable est indispensable en informatique car elle est destinée à contenir une information. Une bonne compréhension de tout ce qui est lié à la notion de variable est une étape clé en programmation.
  Une variable a un nom, un rôle et est associée à un type.
  Une variable peut être affectée*, peut apparaître au sein d'opérations arithmétiques, au sein d'une comparaison,…
  La variable est étroitement liée à la notion d'opérateur.

- Le vrai ou faux

  Il est fréquent, au sein d'un programme, d'arriver à une situation qui nécessite d'évaluer si une condition est vérifiée ou non, si c'est vrai ou faux.
  Lors de l'évaluation d'une condition, la valeur '0' sera interprétée comme FAUX et toute autre valeur comme VRAI.

- Les opérateurs

  Les opérateurs sont étroitement liés à la notion de variable. Il y a essentiellement :

  - L'opérateur d'affectation (dés qu'on veut modifier la valeur d'une variable)
  - Les opérateurs arithmétiques (pour un calcul arithmétique)
  - Les opérateurs de comparaisons (souvent liés aux conditions)
  - Les opérateurs logiques (souvent liés aux conditions)
  - Les opérateurs modificateurs de bits

- La structure de choix

  Cette structure permet de prendre certains éléments en considération et de réaliser des actions différentes suivant les différents cas de figure rencontrés.

- La structure répétitive

  Cette structure permet de répéter (en tenant compte ou non de certaines conditions) une action ou un bloc d'actions.

* Signifie que l'on assigne une valeur à la variable.

1. La variable

- **Notion de variable**

Pour le programmeur la variable est associée à un nom qui représente une valeur significative. La notion de variable est fondamentale, toutes les données manipulées par un programme se font par l'intermédiaire de variables.

- **Type d'une variable**

Toute variable est associée à un type, celui-ci informe sur la valeur minimale et maximale que peut contenir une variable, sur la possibilité d'être négative, de correspondre à un nombre à virgule,... On peut classer les différents types de variables en trois catégories :

• La « famille des entiers »

Lorsqu'une variable est associée à un type de la famille des entiers, cette variable ne pourra contenir que des valeurs entières (…, -2, -1, 0, 1, 2, …).

Exemples :

Nom choisi pour la variable	Rôle de la variable	Remarques
nb_jours_par_mois	contenir le nombre de jours qu'il y a dans un mois.	La variable contiendra toujours des valeurs entières positives.
temperature_degre	La température en degré (au degré près).	La variable pourra contenir des valeurs entières positives et négatives.

• La « famille des réels »

Lorsqu'une variable est associée à un type de cette famille, elle est susceptible de contenir un nombre à virgule.

Exemple :

Nom choisi pour la variable	Rôle de la variable	Remarque
vitesse_voiture_kmh	contenir la valeur en km/h (avec précision) de la vitesse d'une voiture.	La variable peut contenir une valeur à virgule.

• Le type booléen (ou bit)

Une variable associée au type booléen ne peut contenir que la valeur 1 (VRAI) ou la valeur 0 (FAUX).

Exemple :

Nom choisi pour la variable	Rôle de la variable	Remarque
est_en_mode_veille	Contenir une valeur binaire qui indique si l'appareil est en mode veille ou pas.	Lorsque la variable contient 1, ça signifie qu'il est VRAI qu'on est en mode veille.

- **Nom et rôle d'une variable**

Lorsqu'on manipule des variables il faut savoir exactement de quoi il s'agit, il faut que le nom de la variable soit évocateur, qu'il évoque sans ambiguïté ce qu'il représente.

- **L'affectation d'une variable**

La valeur d'une variable peut changer régulièrement, pour modifier sa valeur on utilise l'affectation. L'affectation peut être représentée à l'aide de « ← » au sein de LDA ou d'ordinogrammes. En C, c'est le symbole « = » qui est utilisé.

Exemples :

• En LDA et ordinogrammes			• En C
est_en_mode_veille	←	1	*est_en_mode_veille* = 1 ;
nb_jours_par_mois	←	30	*nb_jours_par_mois* = 30 ;

2. Les opérateurs

Les opérateurs sont étroitement liés à la notion de variable, nous nous contenterons ici d'introduire brièvement :

- Les opérateurs arithmétiques (qui permettent d'obtenir le résultat d'un calcul).
- Les opérateurs de comparaisons (qui permettent d'obtenir un résultat booléen).
- Les opérateurs logiques (qui manipulent directement des valeurs booléennes).

- **Les opérateurs arithmétiques**

Les opérateurs arithmétiques sont +, -, *, / et permettent de fournir le résultat d'une opération. Il faut bien sûr que l'opérateur soit accompagné de deux éléments pour que l'opération puisse être effectuée, ces éléments sont appelés des opérandes.
Le résultat de l'opération peut ensuite, par exemple, être affecté à une variable.

Exemple 1 :

• En LDA et ordinogrammes	• En C
nombre1 ← 5+3	*nombre1* = 5+3 ;

Ici nous avons une variable (*nombre1*) qui est affectée du résultat de l'opération 5+3.
L'opération contient un opérateur (+) et deux opérandes (la constante 5 et la constante 3).

Exemple 2 :

• En LDA et ordinogrammes	• En C
nb_impulsions ← *nb_impulsions* +1	*nb_impulsions* = *nb_impulsions* +1 ;

Ici on met dans la variable *nb_impulsions* son ancienne valeur augmentée de la valeur 1.
On a incrémenté de 1 la valeur de la variable *nb_impulsions*.
L'opération contient un opérateur (+) et deux opérandes (la variable *nb_impulsions* et la constante 1).

- **Les opérateurs de comparaison**

Les opérateurs de comparaison sont par exemple <, >, … Ils permettent de fournir un résultat de type booléen (donc VRAI ou FAUX) suivant le résultat de la comparaison.

Exemple : *vitesse* > 30

Le résultat de la comparaison sera alors VRAI dans la mesure où le contenu de la variable *vitesse* est strictement supérieur à la valeur 30, dans tous les autres cas le résultat de la comparaison donne FAUX.
Les opérateurs de comparaison sont couramment utilisés pour constituer une condition ; les conditions sont par exemple utilisées au sein de structures alternatives.

Exemple en C : if(*vitesse*> 30)
{*consigne_moteur* = 10 ;}

- **Les opérateurs logiques**

Les opérateurs logiques sont par exemple ET, OU,… Ils permettent de prendre en compte une ou plusieurs données booléennes pour fournir un résultat booléen.

Exemple : • En LDA et ordinogrammes • En C

(*vitesse* > 30) ET (*pression* > 50) (*vitesse* > 30) && (*pression* > 50)

Ici le résultat global ne sera VRAI que si la condition (*vitesse* > 30) est vérifiée ET que la condition (*pression* > 50) est également vérifiée.
Comme le résultat global est booléen, il peut constituer une condition à part entière ; on peut alors trouver cette condition au sein d'une structure alternative.

Exemple en C : if ((*vitesse* > 30) && (*pression* > 50))
{*consigne_moteur* = 0 ;}

3. La structure de choix

La structure de choix est également appelée structure alternative.
Même s'il y a deux types de structures de choix (simple et multiple), nous insistons particulièrement sur l'importance de la structure de choix simple. Celle-ci permet d'examiner si une condition est vérifiée afin d'exécuter l'action ou le bloc d'actions qui convient.

4. La structure répétitive

La structure répétitive permet de répéter une action ou un bloc d'actions, soit de manière conditionnelle (en examinant si une condition est vérifiée), soit de
manière inconditionnelle (lorsqu'on connaît à l'avance le nombre de répétitions).
Une structure répétitive inconditionnelle contient néanmoins des conditions internes.

Chapitre 3: La séquence

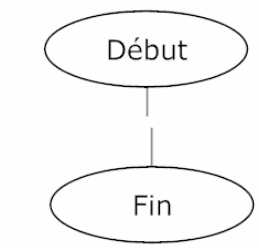
1. Les entités de l'ordinogramme

L'idée de l'ordinogramme est de faire apparaître, de manière visuelle, la structure du programme tout en restant indépendant du langage de programmation.

Figure 5 : Les différentes entités de l'ordinogramme

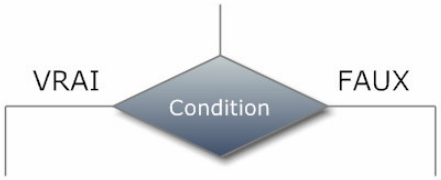
Un ordinogramme commence toujours par l'entité ici à droite

Et se termine toujours par



Toutes les formes d'action sont représentées comme ici à droite par un rectangle. Il peut s'agir d'une déclaration, d'une lecture, d'une écriture, d'une instruction ou d'un ensemble d'instructions.

Tous les tests sont représentés comme ici par un losange à l'intérieur duquel se trouve le prédicat (la condition) dont le résultat sera soit vrai soit faux selon que la condition est vérifiée ou non. Suivant le cas, la suite du programme s'exécutera soit dans un sens soit dans l'autre.



2. La séquence en ordinogramme

La séquence fait apparaître un ensemble d'actions qui sont réalisées les unes après les autres. Ces actions sont réalisées séquentiellement dans le temps.
Pour un ordinogramme, une action est représentée au sein d'un rectangle.

Nous allons retrouver les cinq piliers de la séquence, à savoir :

- La notion de déclaration de variables.
- La notion d'entrée dans l'ordinateur (lire)
- La notion d'affectation, symbolisée par le signe « ← »
- La notion d'opération
- La notion de sortie de l'ordinateur (écrire)

2.1 La partie déclaration de variables

Comme déjà précisé au chapitre précédent, une variable est destinée à contenir une information (une valeur significative).

Par ailleurs une variable a un nom et est associée à un type.
Pour pouvoir manipuler une variable, il faut l'avoir préalablement déclarée.
La déclaration est simplement la ligne qui permet d'introduire une nouvelle variable au sein du programme en précisant son nom et son type. Il y a trois catégories de types :

1°) La « famille des entiers » → Entiers
2°) La « famille des réels » → Réels
3°) Le type booléen → Bit

Pour déclarer une variable, il suffit de préciser le type suivi de son nom.

Exemples :	Syntaxe au sein d'ordinogrammes	Signification
	Réel *vitesse*	On déclare une variable nommée *vitesse* de type réel.
	Entier nb_enfoncements	On déclare une variable nommée *nb_enfoncements* de type entier.

Comme les déclarations sont dans la catégorie des actions, les lignes de syntaxes montrées ci-dessus doivent se trouver au sein d'un rectangle.

2.2 L'instruction de lecture

L'instruction de lecture correspond à l'acquisition d'une donnée pour pouvoir l'utiliser au sein du programme. Si on veut par exemple un programme qui puisse calculer la moyenne de trois nombres, il faut commencer par acquérir les valeurs de ces trois nombres (les entrées). L'acquisition d'une donnée au sein du programme correspond à une lecture. Au sein de l'ordinogramme, l'instruction de lecture utilise le mot clé Lire qui doit être souligné.

Une fois la lecture effectuée, la donnée se trouvera au sein d'une variable. Il faut bien sûr s'assurer de préciser la variable qui recevra la donnée.

Exemple :	Syntaxe au sein d'ordinogrammes	Signification
	Lire *nb_1*	L'acquisition de la donnée est alors réalisée et stockée au sein de la variable *nb_1*.

Il faut bien sûr que la variable *nb_1* ait été préalablement déclarée.

Comme une lecture est dans la catégorie des actions, la ligne de syntaxe montrée ci-dessus doit se trouver au sein d'un rectangle.

2.3 L'instruction d'affectation

L'affectation est une action qui permet de donner à une variable une certaine valeur.
Elle est représentée par le symbole clé « ← » au sein d'ordinogrammes.

L'instruction d'affectation sera sous la forme : *nom_de_variable* ← valeur[*]
Il faut également s'assurer de placer chaque affectation au sein d'un rectangle.

2.4 Les opérations

Une opération est une expression qui fournit un certain résultat. Une opération contient des opérateurs et des opérandes.
Les opérandes sont les éléments qui permettront à l'opération de fournir un résultat.
Ils apparaîtront comme des paramètres des opérations. Ils peuvent être soit une constante, soit une variable.
Les opérateurs sont principalement divisés en trois catégories :

- Les opérateurs arithmétiques (+,-,*,/)
- Les opérateurs de comparaison (>,≥,<,…)
- Les opérateurs logiques (ET, OU, NON)

2.5 L'instruction d'écriture

L'écriture indique que l'on renvoie un résultat.
Si l'on prend l'exemple du calcul de la moyenne de trois nombres, il faut pouvoir, après l'acquisition des entrées et le traitement, renvoyer le résultat.

L'instruction d'écriture sera sous la forme : Ecrire *nom_de_variable*
Comme une écriture est dans la catégorie des actions, la ligne de syntaxes montrée ci-dessus doit se trouver au sein d'un rectangle. On utilise le mot clé Ecrire (qui doit être souligné) pour préciser que l'on fournit un résultat.
Le contenu de la variable mentionnée constitue alors le résultat.

2.6 Exemple de séquence *Calcul de la moyenne de 3 nombres*

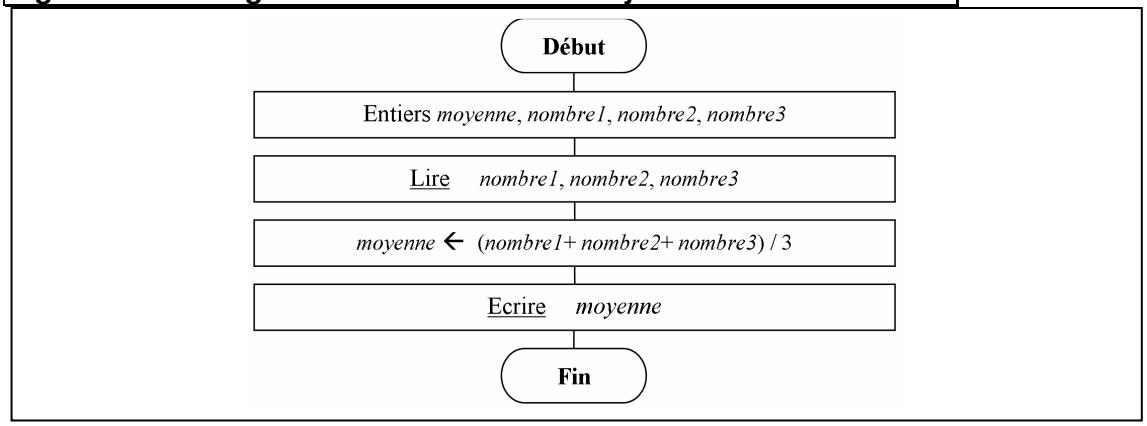
Il nous faut une variable qui permette de sortir le résultat, il s'agit de la variable de sortie.
On choisit alors un nom approprié, par exemple : *moyenne*

Il nous faut des variables qui permettront de contenir les nombres dont il faut calculer la moyenne, on choisira des noms de variable tels que : *nombre1, nombre2, nombre3*
Ce sont les variables d'entrées.

Nous allons maintenant exposer l'ordinogramme de la séquence qui répond à notre problème informatique. Il ne faut pas oublier de déclarer au début toutes les variables qui seront manipulées (et les associer à un type). Les variables d'entrées doivent être lues avant de prendre en compte les valeurs correspondantes. La variable de sortie doit permettre de fournir le résultat en sortie avec une instruction d'écriture.

[*] Peut être le résultat d'une opération

Figure 6 : Ordinogramme du calcul de la moyenne de trois nombres



Remarques :
- *moyenne* étant un entier, le résultat risque d'être approximatif.
- Nous aurions pu déclarer la variable *moyenne* comme Réel.
- Nous aurions pu calculer la somme de trois réels au lieu d'entiers.

3. La séquence en C

De la même manière que pour les ordinogrammes, on retrouve en C :
- La notion de déclaration de variables
- La notion d'entrée dans l'ordinateur (lire)
- La notion d'affectation, symbolisée par l'opérateur « = »
- La notion d'opération
- La notion de sortie de l'ordinateur (écrire)

3.1 La partie déclaration de variables

En C, le principe est le même sauf qu'à la fin de chaque déclaration il faut mettre « ; » et que les types sont plus nombreux.

Exemples de déclaration en C :

float *vitesse* ; où *vitesse* est le nom de la variable et **float** son type
int *nb_enfoncements* ; où *nb_enfoncements* est le nom de la variable et **int** son type

Pour simplifier nous commencerons par utiliser un nombre limité de types en C.
Nous utiliserons les types :

int	pour la « famille des entiers »
float	pour la « famille des réels »
bit ou **char**	pour le type booléen

Nous verrons plus tard qu'il y a des limites d'utilisation de ces différents types et que l'on peut par exemple être amené à utiliser le type **long** plutôt que le type **int** pour déclarer une variable de la « famille des entiers ».

3.2 L'instruction de lecture

En C, contrairement aux ordinogrammes, il n'existe pas un mot clé réservé à la lecture. La manière de lire une information dépendra de l'application et du système embarqué lui-même.
On peut néanmoins mettre en évidence le fait qu'il faille effectuer une lecture à l'aide de commentaires. Il suffit de préciser qu'on effectue une lecture après les 2 caractères « // ».

Exemple : //je lis *nombre1*

3.3 L'instruction d'affectation

En C le principe est le même sauf qu'à la fin de l'affectation il faut mettre « ; » et que le symbole d'affectation est « = » (au lieu de « ← »).

Exemples d'affectations en C :

est_en_mode_veille = 1 ; où *est_en_mode_veille* est la variable qui reçoit la valeur 1
nb_jours_par_mois = 30 ; où *nb_jours_par_mois* est la variable qui reçoit la valeur 30

Il faut bien sûr que la variable sujette à l'affectation ait été préalablement déclarée.

Remarque : Il est possible d'affecter autre chose qu'une constante à une variable, on peut par exemple lui affecter le résultat d'une opération.

3.4 Les opérations

Les opérations sont entre autres constituées d'opérateurs. Les opérateurs en C doivent respecter la syntaxe prévue, nous nous contenterons pour l'instant de décrire les opérateurs propres aux catégories suivantes :

- Les opérateurs arithmétiques (qui permettent d'obtenir le résultat d'un calcul).
- Les opérateurs de comparaisons (qui permettent d'obtenir un résultat booléen).
- Les opérateurs logiques (qui manipulent directement des valeurs booléennes).

- **Opérateurs arithmétiques**

Figure 7 : Les opérateurs arithmétiques en C

Opérateurs arithmétiques sur les réels :	Opérateurs arithmétiques sur les entiers :
+ - * / avec la hiérarchie habituelle.	+ - * / (quotient de la division) % (reste de la division) avec la hiérarchie habituelle.

- **Opérateurs de comparaison**

Figure 8 : Les opérateurs de comparaison en C

En LDA :	>	>=	<	<=	=	≠
En C :	>	>=	<	<=	==	!=

Les opérateurs de comparaison sont typiquement utilisés au sein de structures qui encapsulent des conditions (comme la **if( )** ou la **while( )**).
Le résultat des opérations qui utilisent des opérateurs de comparaison sera de type booléen. Une valeur de 0 sera alors interprétée comme FAUX alors que toute autre valeur sera interprétée comme VRAI.

- **Opérateurs logiques sur les entiers**

Figure 9 : Les opérateurs logiques en C

En LDA :	ET	OU	NON
En C :	&&	||	!

Les opérations qui manipulent des opérateurs logiques sont également constituées d'opérandes de type booléen. Nous verrons que l'on utilise fréquemment des variables de la « famille des entiers » pour contenir un booléen.
Le résultat d'une opération utilisant un opérateur de comparaison sera de type booléen.

3.5 L'instruction d'écriture

En C, contrairement à ce qu'on a vu avec les ordinogrammes, il n'existe pas un mot clé réservé à l'écriture. Tout comme pour la lecture, la manière de sortir un résultat dépendra de l'application et du système embarqué lui-même. On pourrait imaginer afficher le résultat sur écran, l'envoyer par I^2C, activer directement une sortie,…

On peut néanmoins mettre en évidence le fait qu'il faille effectuer une écriture à l'aide de commentaires. Il suffit de préciser qu'on effectue une écriture après les caractères « // ».

Exemple : //j'écris *moyenne*

3.6 Exemple de séquence en C

Nous allons juste nous contenter d'exposer l'équivalent en C de l'exemple de séquence vu en ordinogramme (Fig.6)

Figure 10 : Séquence en C - Calcul de la moyenne de trois nombres

```
int          moyenne, nombre1, nombre2, nombre3;
// je lis      nombre1, nombre2, nombre3
moyenne  =   (nombre1+ nombre2+nombre3)/3 ;
// j'écris     moyenne
```

Remarques :
- *moyenne* étant un entier, le résultat risque d'être approximatif.
- Nous aurions pu déclarer la variable *moyenne* comme Réel.
- Nous aurions pu calculer la somme de trois réels au lieu d'entiers.

Chapitre 4: La structure de choix simple

Il apparaît souvent le besoin de réaliser une action uniquement dans certains cas.
Pour ce faire, le microcontrôleur devra examiner une condition afin de préciser si celle-ci est vraie ou fausse. Suivant le résultat de la condition, il s'agira d'effectuer un bloc d'action ou un autre (ou éventuellement aucune action).

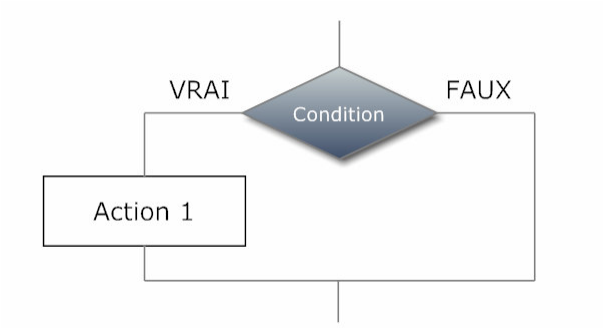
Au niveau des ordinogrammes, la structure de choix simple nécessite l'utilisation d'une entité particulière, le losange (voir Fig.5). Il ne peut il y avoir qu'une entrée et deux sorties ; les sorties doivent être accompagnées des mots clés « VRAI » et « FAUX ».
La condition doit se trouver à l'intérieur du losange.

En C, la structure de choix simple est reconnaissable à l'aide du mot clé « if » qui doit être accompagné de parenthèses pour contenir la condition associée à la structure.

1. Si - Alors

Dans la mesure où il faut faire une action ou un bloc d'actions lorsqu'une condition est vérifiée, et qu'il ne faut rien faire lorsque cette même condition n'est pas vérifiée, on utilise la structure de choix simple de la forme « Si – Alors ».

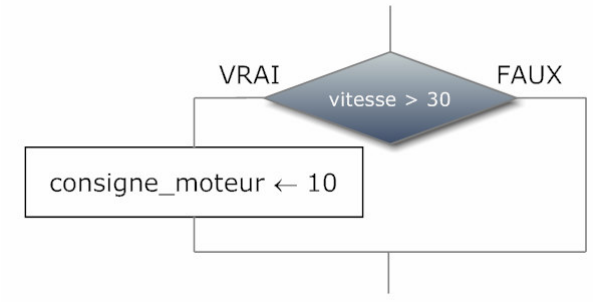
Structure visuelle (ordinogramme):



Syntaxe en C :

```
if (condition)
        {
        //bloc d'actions
        //délimité par '{' et '}'
        }
```

Exemple de morceau d'ordinogramme:



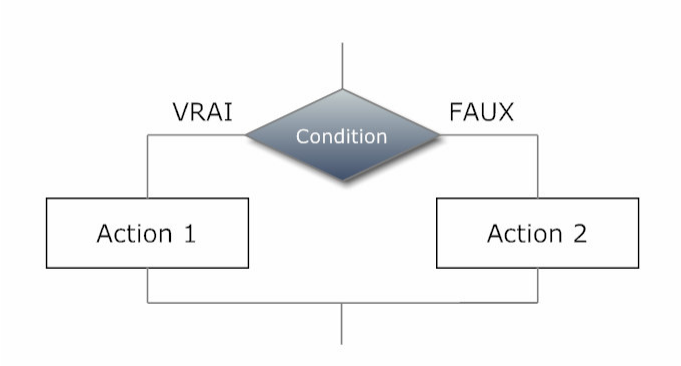
Equivalent en C :

```
if ( vitesse > 30 )
        {
        consigne_moteur =10;
        }
```

2. Si – Alors - Sinon

S'il s'agit de faire une action lorsqu'une condition est vérifiée et une autre action lorsqu'elle n'est pas vérifiée, on utilise la structure de choix simple de la forme « Si – Alors - Sinon ».

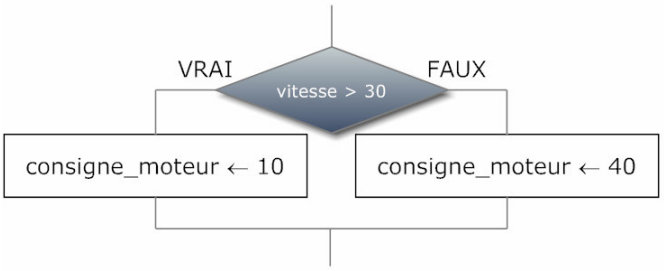
Structure visuelle (ordinogramme):



Syntaxe en C :

```
if (condition)
        {
        //bloc d'actions 1
        //délimité par '{' et '}'
        }
else
        {
        //bloc d'actions 2
        //délimité par '{' et '}'
        }
```

Exemple de morceau d'ordinogramme:



Equivalent en C :

```
if ( vitesse > 30 )
        {
        consigne_moteur =10;
        }
else
        {
        consigne_moteur =40;
        }
```

Chapitre 5: La « tant que »

1. Principe général

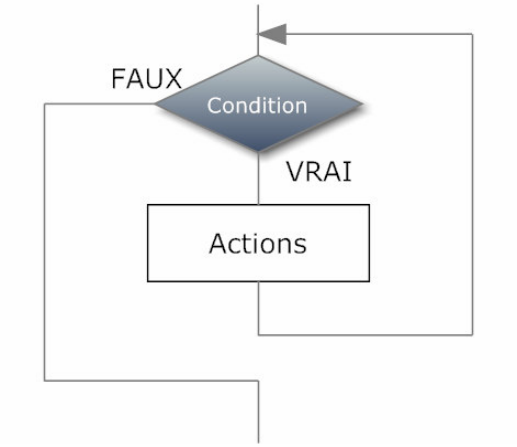
Ci-dessous une phrase qui décrit le principe général de la boucle « tant que » :

Tant que condition répéter action.

Cela signifie que tant qu'une condition est vérifiée, on répète un bloc d'actions.
Si nous représentons la structure d'une boucle « tant que » à l'aide d'un ordinogramme, nous obtenons la figure ci-dessous :

Figure 11 : Structure de la « tant que »

Ordinogramme



C

```
while (condition)
        {

        //actions

        }
```

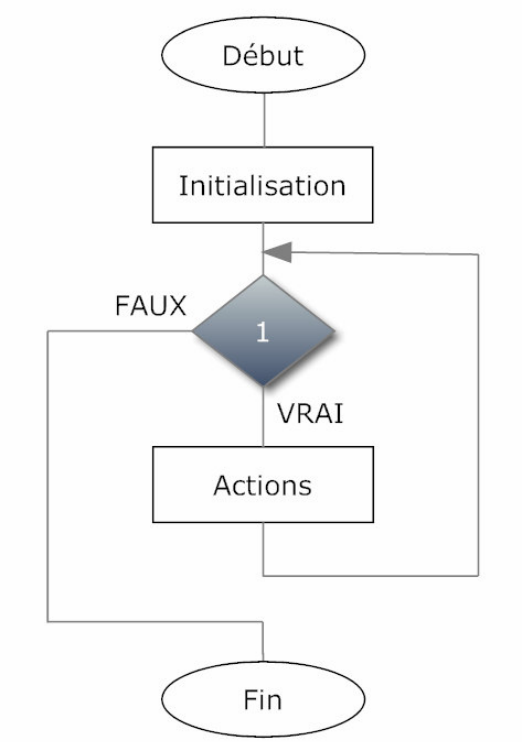
2. La boucle infinie

Typiquement un programme contient une boucle infinie (ou while(1)) de manière à ce que le microcontrôleur n'ait jamais fini de travailler.
Si le programme ne contient pas de boucle infinie, lorsque la dernière instruction a été exécutée, le microcontrôleur a fini de travailler. Dans ce cas, plus aucune entrée ne sera prise en compte (un enfoncement de bouton-poussoir par exemple).
Pour s'assurer que le microcontrôleur n'ait jamais fini de travailler on utilise une structure « tant que » qui contient une condition toujours vraie. On met alors la valeur 1 pour constituer la condition (la valeur 1 sera interprétée comme vrai) de manière à ne jamais sortir de la boucle.

La figure suivante met en évidence la structure de base qui est préconisée pour tout programme.

Figure 12 : Ordinogramme et programme en C manipulant une boucle infinie

Ordinogramme



C

```
void main(void)
{

//initialisation

while (1)
        {

        //actions


        }

}
```

Ci-dessus nous avons la structure typique d'un programme.
Le début du programme correspond au début de la fonction main() alors que la fin du programme correspond à la fin de la fonction main(). Le mot clé *main* signifie principal en anglais, il ne peut il y en avoir qu'un seul par projet et indique l'endroit où le microcontrôleur commencera à exécuter ses instructions.
La première accolade ouverte ″{″ après le main() peut-être associée à l'entité début, la dernière ″}″ à l'entité fin.

On met ensuite avant le while(1) tout ce qui n'est pas susceptible de se répéter, il peut il y avoir les déclarations, l'initialisation de la carte,…

Au sein de la boucle infinie on mettre toutes les actions qui sont susceptibles de se répéter. On voit d'ailleurs apparaître le début du bloc d'actions ″{″ et la fin du bloc d'actions ″}″ .

On peut également remarquer que l'on ne passera jamais par l'entité fin car la condition de la boucle infinie est toujours vraie.

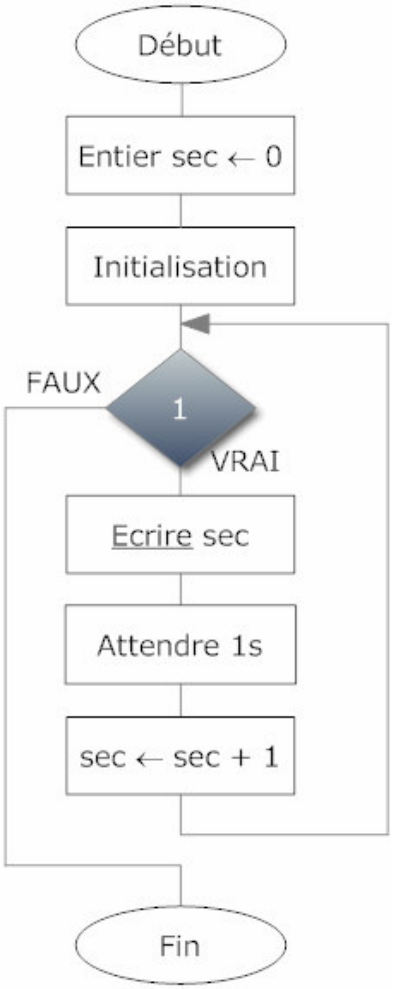
Chapitre 6: Exemples de programmes

1. Le chronomètre

Nous avons ci-dessous le programme le plus basique qui permet la gestion d'un chronomètre. Le nombre de secondes écoulées ne fait que s'incrémenter en continu.
A gauche nous avons la structure du programme représentée visuellement (ordinogramme) et à droite nous avons un programme en C basé sur cette structure.
Le programme en C se base sur l'environnement que l'on utilise en classe (les fonctions init(), delay_ms() et les fonctions d'affichage sont déjà implémentées).

Figure 13 : Exemple du chronomètre

Ordinogramme



C

```
char Ligne1[16];
void main(void)
{

int sec=0;

init();



while(1)
        {
        //AFFICHAGE DES SECONDES
        lcd_goto(0x00);
        sprintf(Ligne1,"chrono : %3d s",sec);
        lcd_puts(Ligne1);

        //DELAI D'UNE SECONDE
        delay_ms(1000);

        //INCREMENTATION DE sec
        sec = sec+1;

        }//fin du while(1)


}//fin du main
```

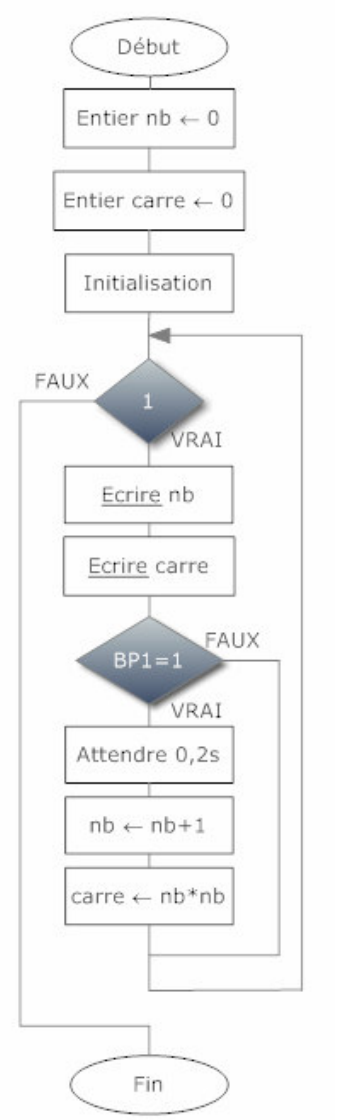
Comme déjà précisé, l'instruction d'écriture dépend de l'application en C. Ici nous combinons la fonction sprintf() aux routines d'affichage pour afficher les secondes écoulées.

2. La carré d'un nombre

Nous avons ci-dessous un programme qui permet d'incrémenter un nombre à l'aide du bouton-poussoir 1 pour afficher la valeur d'un nombre et de son carré.

Figure 14 : Exemple du carré d'un nombre

Ordinogramme



C

```
char Ligne1[16]; char Ligne2[16];
void main(void)
{
int nb=0;


int carre=0;

init();

while(1)
        {
        //AFFICHAGE DE nb
        lcd_goto(0x00);
        sprintf(Ligne1,"nb vaut : %2d",nb);
        lcd_puts(Ligne1);

        //AFFICHAGE DE carre
        lcd_goto(0x40);
        sprintf(Ligne2,"carre : %4d",carre);
        lcd_puts(Ligne2);

        if(BP1==1)
                {
                delay_ms(200) ;

                nb = nb+1 ;

                carre = nb*nb ;
                }


        }//fin du while(1)

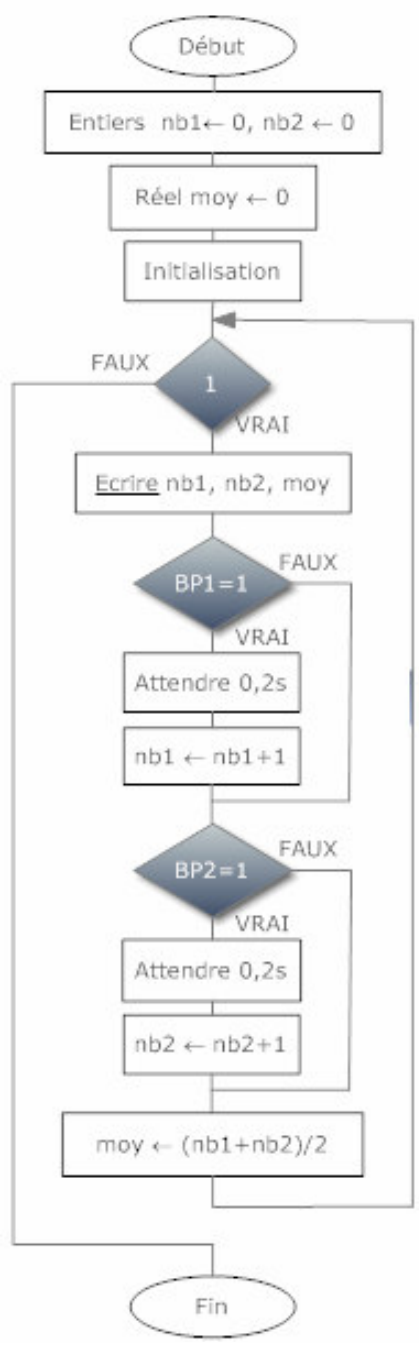

}//fin du main
```

On peut voir ici que tout ce qui ne devra être fait qu'une seule fois se trouvera avant la boucle infinie (les déclarations et l'initialisation).
Au sein de la boucle il y a d'abord un affichage du nombre et de son carré puis une structure de choix pour vérifier si le bouton-poussoir 1 a été enfoncé par l'utilisateur.
Nous avons une structure type Si-alors qui permet, lorsque le bouton est enfoncé, d'attendre un petit délai avant de mettre à jour la valeur du nombre et de son carré.
Les nouvelles valeurs du nombre et de son carré seront affichées au début du prochain tour de boucle (grâce à la boucle infinie).

3. La moyenne de deux nombres

Figure 15 : Exemple de la moyenne de deux nombres

Ordinogramme



C

```
char Ligne1[16]; char Ligne2[16];
void main(void)
{
int nb1=0, nb2=0;


float moy=0;

init();
while(1)
        {
        //AFFICHAGE DE nb1, nb2
        lcd_goto(0x00);
        sprintf(Ligne1,"nb1: %2d  nb2 : %2d ",nb1,nb2);
        lcd_puts(Ligne1);

        //AFFICHAGE DE moy
        lcd_goto(0x40);
        sprintf(Ligne2,"moy : %8f ",moy);
        lcd_puts(Ligne2);

        if(BP1==1)
                {
                delay_ms(200) ;
                nb1 = nb1+1 ;
                }


        if(BP2==1)
                {
                delay_ms(200) ;
                nb2 = nb2+1 ;
                }




        moy = (nb1 + nb2)/2 ;

        }//fin du while(1)


}//fin du main
```

Si on examine ce qui se répète au sein de la boucle infinie, il y a :

- Rafraîchissement de l'affichage.
- Mise à jour éventuelle de la valeur de nb1 (dépend si BP1 a été enfoncé ou non).
- Mise à jour éventuelle de la valeur de nb2 (dépend si BP2 a été enfoncé ou non).
- Calcul de la moyenne des deux nombres.

Chapitre 7: La « répéter … tant que »

Même si la structure répétitive « tant que » est la structure générale qui permet de gérer toutes les formes de répétitions, il existe d'autres formes de structures répétitives.
Il y a par exemple la « répéter … tant que » qui permet de mettre en évidence que le bloc d'actions sera au moins effectué une fois.

1. Principe général

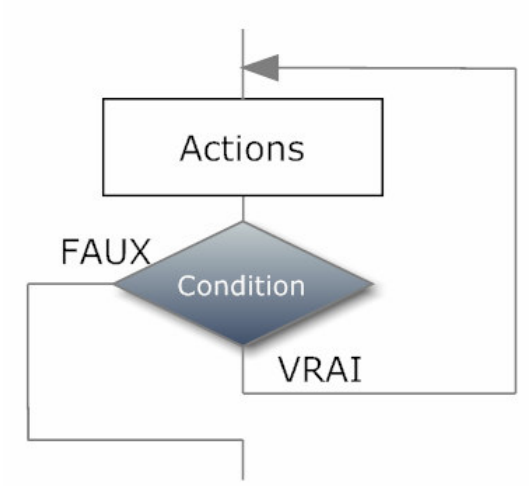
Ci-dessous une phrase qui décrit le principe général de la boucle « répéter… tant que » :

Répéter action tant que condition.

Ici la description commence par « répéter action… », ce qui signifie que l'action sera au moins exécutée une fois.
On peut d'ailleurs constater ci-dessous que l'ordinogramme de la « répéter …tant que » commence par un bloc d'actions (un rectangle).

Figure 16 : Structure de la «répéter… tant que »

Ordinogramme



C

```
do
        {

        //actions

        }
while (condition);
```

On peut remarquer qu'il y a un point-virgule à la fin de la ligne qui contient le while, ceci est propre de la « répéter … tant que ».

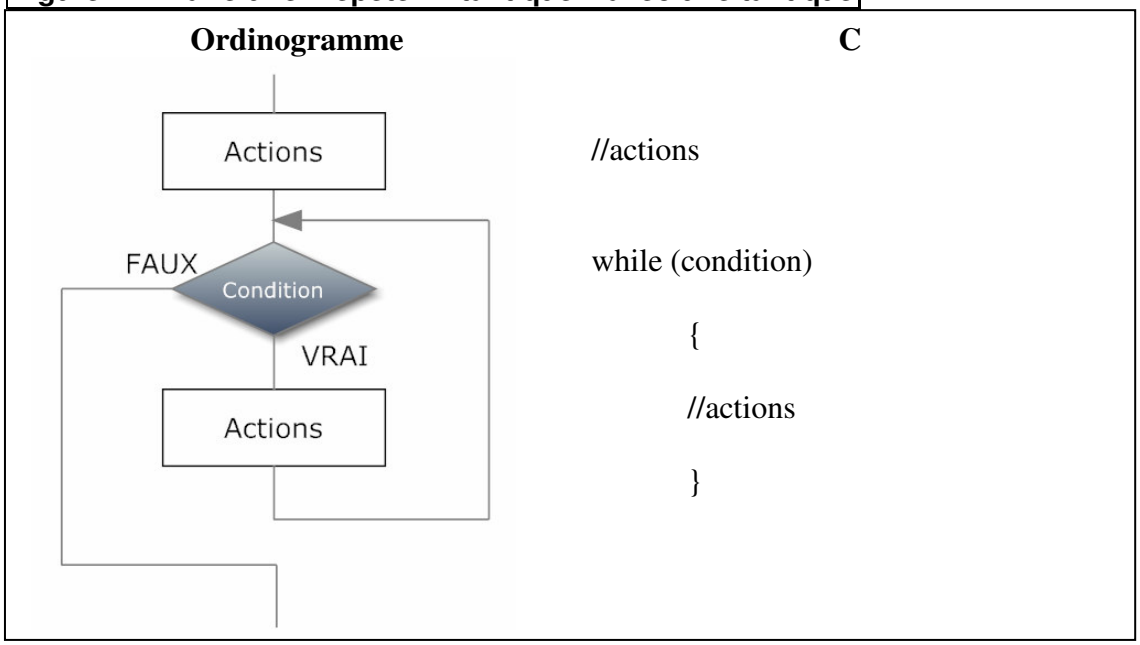
2. Faire une « répéter…tant que » avec une « tant que »

Il est possible de faire la même chose qu'une « répéter…tant que » avec une « tant que ». En effet, comme déjà précisé, la « tant que » est la structure répétitive la plus générale qui permet de faire toutes les formes de répétitions.
Il est cependant conseillé d'utiliser la structure adéquate afin de garder un maximum de lisibilité. Ce qui signifie que s'il y a un bloc d'actions qui est susceptible d'être répété mais qu'il faut de toute façon l'exécuter une fois, on choisira la « répéter…tant que ». Ce choix apporte bien sûr plus de lisibilité car il met directement en évidence, grâce à la structure, que le bloc d'actions sera au moins exécuté une fois.

Il est cependant intéressant de comprendre comment on pourrait faire une « répéter…tant que » avec une « tant que » ; on montre ainsi l'aspect général de la « tant que » et on illustre le fonctionnement de la « répéter … tant que ».

On peut remarquer ci-dessous qu'en plaçant le bloc d'actions d'abord une fois on est sûr qu'il sera exécuté au moins une fois. On a donc créé une « répéter…tant que » avec une « tant que », on voit d'ailleurs apparaître la structure de la « tant que » ci-dessous (telle que présentée Fig.11).

Figure 17 : Faire une «répéter… tant que » avec une tant que



Je tiens à rappeler qu'il s'agit d'un exemple didactique et qu'il faut bien sûr utiliser la structure répétitive la mieux adaptée. Non seulement elle apporte plus de lisibilité mais elle fait apparaître un code plus concis.

3. Une « répéter…tant que » avec un bloc d'actions vide

Dans certains cas on souhaite attendre qu'une condition soit remplie.
Un exemple souvent utilisé au cours est l'attente d'un relâchement d'un bouton-poussoir. Dans ce cas on va s'arranger pour rester bloqué dans une boucle tant que le bouton-poussoir est enfoncé.
Il est alors possible d'utiliser une « tant que » ou une « répéter tant … que » associée à un bloc d'actions vide.

Figure 18 : Exemple de «tant que » avec un bloc d'actions vide

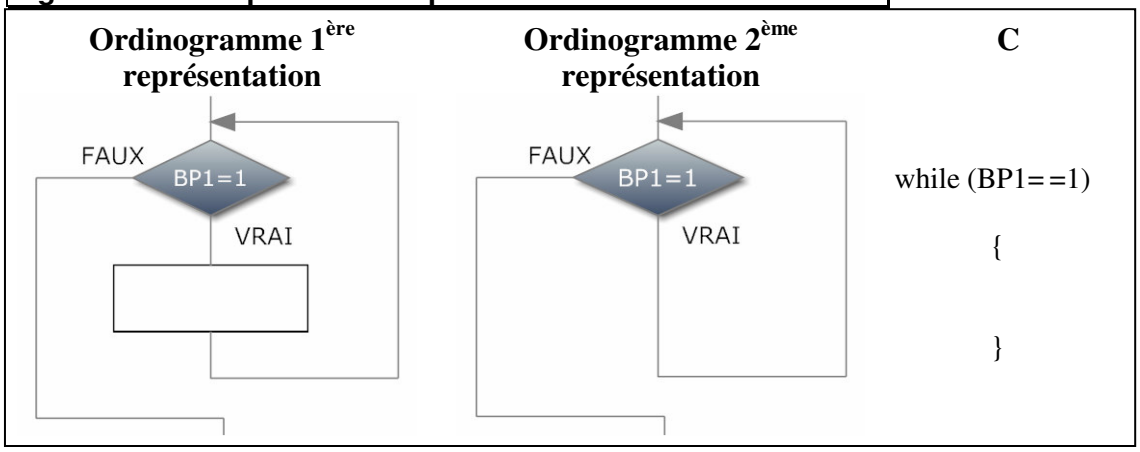
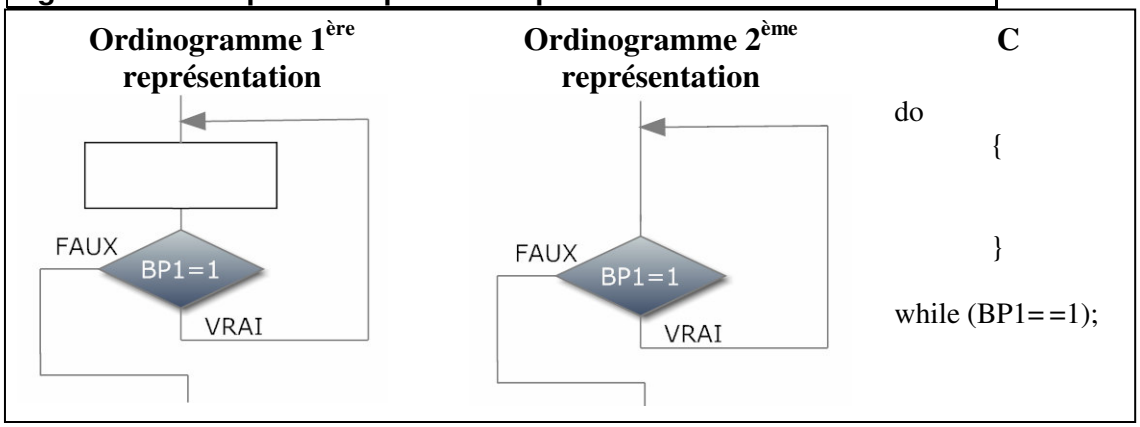


Figure 19 : Exemple de «répéter tant que » avec un bloc d'actions vide



On peut constater, en comparant les ordinogrammes des figures 18 et 19, que le fait d'utiliser une « tant que » ou une « répéter… tant que » revient au même. En effet, comme ici le bloc d'actions est vide, la nuance comme quoi le bloc d'actions sera exécuté au moins une fois dans la « répéter…tant que » n'a plus d'intérêt.
Pour cette raison on peut utiliser indifféremment la « tant que » ou la « répéter…tant que » lorsque le bloc d'actions est vide. Cependant, c'est souvent la « répéter…tant que » qui est choisie car en C il y a une écriture abrégée :

```
do
        {                    PEUT          while(condition) ;
        }                  EGALEMENT
while(condition) ;          S'ECRIRE
```

Chapitre 8: La boucle « pour »

Même si la structure répétitive « tant que » est la structure générale qui permet de gérer toutes les formes de répétitions, il existe une structure répétitive qui est très utile lorsqu'on connaît à l'avance le nombre de répétitions : la boucle « pour ».

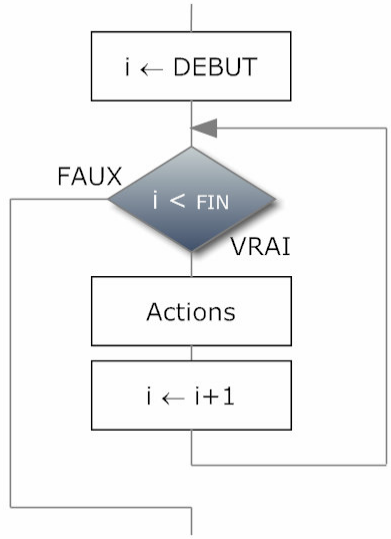
1. Principe général

La boucle « pour » utilise un compteur interne (une variable) qui permet de faire en sorte que le nombre de fois que la répétition aura lieu correspond bien à ce qui a été spécifié. La variable qui est utilisée pour le compteur est en général nommée « *i* ».

La boucle « pour » gère la variable compteur de manière interne, typiquement il y a une incrémentation de cette variable à la fin de chaque répétition du bloc d'actions.
Il y a également un test propre à la boucle qui est effectué régulièrement de manière à déterminer s'il faut continuer à exécuter le bloc d'actions ou si le nombre de répétitions qui ont déjà été effectuées a atteint ce qui a été spécifié (dans ce cas on sort de la boucle).

Figure 20 : Structure de la «pour » et syntaxe en C

Ordinogramme



C

```
for(i=DEBUT ; i<FIN ; i++)
        {
        //Actions
        }
```

Exemple particulier, on veut 10 clignotements de la LED1 :

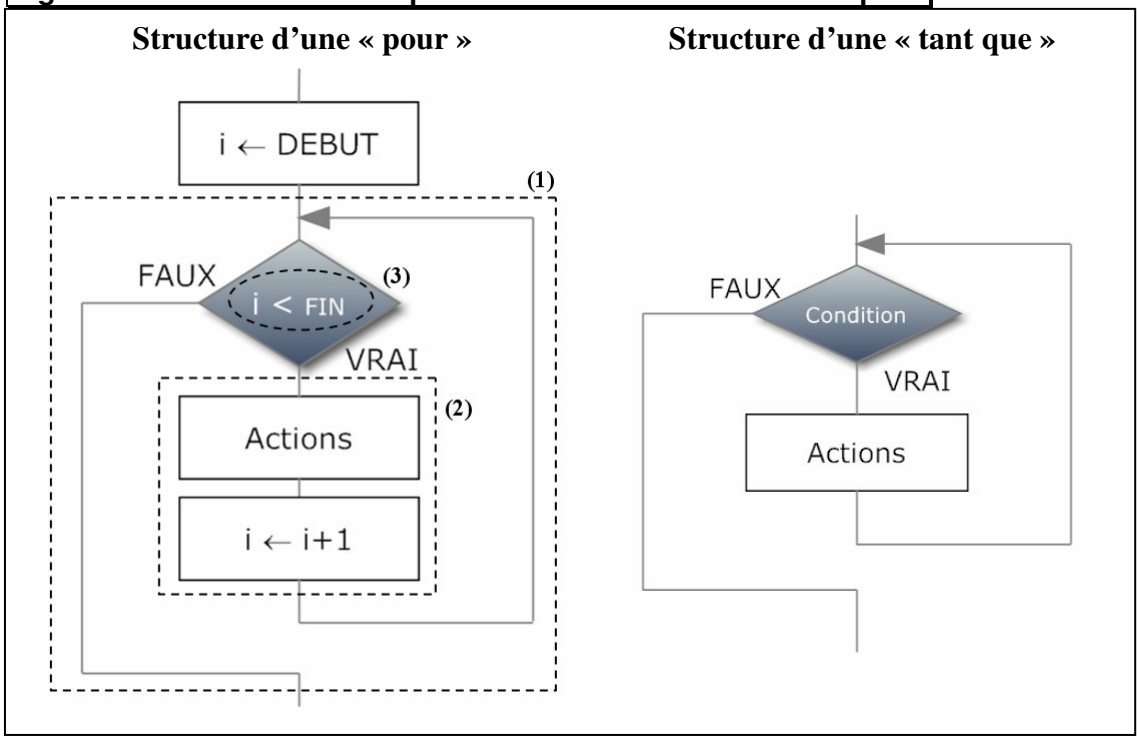
```
for(i=0; i<10 ; i++)
        {
        //Un clignotement de LED1
        }
```

La figure ci-dessus illustre, de manière visuelle, le principe de la « pour ». On peut voir qu'il y a d'abord une initialisation de la variable compteur « i » puis un test au sein d'une boucle qui permettra au processeur d'exécuter le bloc d'actions autant de fois que désiré. Bien sûr, après chaque bloc d'actions, il y a incrémentation de la variable compteur « i ». On peut remarquer que la syntaxe en C est très concise, il suffit d'écrire par exemple for(i=0; i<10 ; i++) suivi du bloc d'actions délimité par ″{″ et ″}″ pour répéter 10 fois ce bloc d'actions.

2. Faire une « pour » avec une « tant que »

Si on analyse la structure de la « pour », on peut remarquer qu'elle permet de mettre en évidence une « tant que » ; la figure ci-dessous le montre bien.
A gauche nous avons la structure d'une « pour » telle que donnée Fig.20 et à droite la structure d'une « tant que « telle que donnée Fig.11.

Figure 21 : Structure d'une «pour » et structure d'une « tant que »



On voit bien qu'au niveau de la structure, la « pour » se rapproche d'une « tant que ».
La zone (1) fait bien apparaître une « tant que » qui englobe un bloc d'actions (2) et une condition (3).

Il est donc possible de faire une « pour » à l'aide d'une « tant que ».
Il faut pour cela déclarer la variable compteur, l'affecter de la valeur de départ puis la comparer à la valeur de fin au sein de la condition de la « tant que ». Il faut également prendre soin que le bloc d'actions de la « tant que » contienne les actions à répéter mais également l'incrémentation de la variable compteur.

Chapitre 9: Les types en C

1. Précisions générales

Toute variable est associée à un type qui fait référence, entre autres, à la place mémoire qui sera réservée. Chaque variable est codée sur un certain nombre de bits et chaque combinaison binaire correspond à une valeur de cette variable.
Déterminons le lien entre le nombre de bits utilisés pour le codage et le nombre de combinaisons binaires différentes possibles que cette variable peut contenir :

On voit ci-dessous (Fig.22) qu'avec 1 bit il n'y a que 2 combinaisons possibles alors qu'à chaque fois qu'on ajoute 1 bit ça double le nombre de combinaisons possibles. En effet, pour 2 bits il y a 4 possibilités ('0' suivi de chaque possibilité pour 1 bit → 2 + '1' suivi de chaque possibilité pour 1 bit → 2). Avec 3 bits il y a 8 valeurs possibles ('0' suivi de chaque possibilité pour 2 bits → 4 + '1' suivi de chaque possibilité pour 2 bits → 4).

Figure 22 : Lien entre le nombre de bits et les nombres positifs représentables

Nombre de bits		Codes différents	Nombre de codes différents		Nombres positifs représentables
1	□	0 1	2	2^1	0 à 1
2	□□	0 0 0 1 1 0 1 1	4	2^2	0 à 3
3	□□□	non donnés	8	2^3	0 à 7
...					
8	□□□□□□□□	non donnés	256	2^8	0 à 255
16	□□□□□□□□ □□□□□□□□	non donnés	65 536	2^{16}	0 à 65 535

Avec n bits il y a 2^n possibilités, ce qui permet de représenter des nombres de 0 à 2^n-1. Nous comprenons maintenant le lien entre le nombre de bits réservés en mémoire pour une variable et le nombre de valeurs différentes que pourra contenir cette variable. Avec n bits il y a 2^n valeurs différentes. Nous avons vu dans le tableau précédent qu'avec 8 bits on peut représenter tous les nombres positifs de 0 à 255.

Mais comment faire si on veut qu'une variable puisse contenir un nombre négatif ?

On utilise un type particulier qui va permettre d'interpréter certains codes binaires comme des nombres négatifs. La convention utilisée est généralement le complément à 2, l'interprétation se fait alors (pour un octet) comme à la page suivante.

En résumé nous dirons que le type influence :
- Le nombre de valeurs possibles que pourra prendre une variable
- La manière d'interpréter la valeur provenant du code binaire de la variable

Figure 23 : Interprétation des nombres positifs et négatifs

Motif binaire	Valeur interprétée avec un type pour valeurs positives	Valeur interprétée avec un type pour valeurs négatives
00000000	0	0
00000001	1	1
...	...	...
01111111	127	127
10000000	128	-128
10000001	129	-127
...	...	...
11111110	254	-2
11111111	255	-1

2. Les différents types en C

Nous avions mentionné 3 catégories de types : La « famille des entiers »
La « famille des réels »
Le type booléen

En C, ces catégories de types sont encore affinées pour permettre plus de souplesse.
On pourra par exemple choisir une variable de type **char** pour contenir des valeurs entières de faibles valeurs ou de type **long** pour des valeurs entières élevées.
Les types **char** et **long** font tous les deux partie de la « famille des entiers » mais le type **char** n'occupe qu'un octet en mémoire alors que le type **long** en occupe quatre.
Il convient bien sûr d'utiliser le type qui occupe le moins de place mémoire tout en restant adapté à son application. On évite ainsi de gaspiller de l'espace mémoire en RAM et de ralentir le temps d'exécution du programme (manipuler une variable de 4 octets prend plus de temps que manipuler une variable de 1 octet).

Figure 24 : Les différents types en C

Syntaxe pour la référence du type en C	Description du type	Place occupée en mémoire	Description résumée	
long	entier long signé	4 octets	entier	
unsigned long	entier long non signé	4 octets	entier	
int	entier standard signé	2 octets	entier	*
unsigned int	entier positif	2 octets	entier	*
short	entier court signé	2 octets	entier	
unsigned short	entier court non signé	2 octets	entier	
char	caractère signé	1 octet	caractère	*
unsigned char	caractère non signé	1 octet	caractère	*
bit	booléen	1/8 d'octet	booléen	
float	réel standard	3 octets	réel	*
double	réel double précision	3 ou 4 octets	réel	

Cette description des types de variables s'applique au compilateur PICC18
Les types complétées de * sont ceux que nous utiliserons principalement.

Chapitre 10: Les opérateurs logiques

Nous allons commencer par étudier l'algèbre de Boole avant d'expliquer les opérateurs logiques. Nous verrons plus loin qu'il y a en C des opérateurs logiques sur entiers et des opérateurs logiques bit à bit.

1. L'algèbre de Boole

1.1 Généralités

L'algèbre de Boole est un outil mathématique qui permet de mettre en équation des fonctions combinatoires élémentaires (ET, OU,…) ou des fonctions combinatoires plus complexes (imbrication de fonctions combinatoires élémentaires).
Toute fonction combinatoire correspond à une expression mathématique appelée expression booléenne.

De la même manière qu'au cours de mathématique, les expressions booléennes sont constituées de variables, de constantes et d'opérateurs mathématiques. La grande différence réside dans le fait qu'avec l'algèbre de Boole les variables et les constantes ne peuvent prendre que 2 valeurs possibles : 0 et 1.

Souvent, pour représenter une fonction combinatoire, on utilise une équation qui contient la sortie dans le membre de gauche et l'expression booléenne correspondante dans le membre de droite. On trouvera par exemple : S = a + b

a, b et S sont des variables (ne peuvent donc prendre que les valeurs 0 et 1).
a et b sont les entrées.
S est la sortie, elle dépend de l'état des entrées et de l'expression booléenne.
a + b est l'expression booléenne de la fonction combinatoire de S.

1.2 Quelques particularités de l'algèbre de Boole

Non seulement les variables ne peuvent prendre que les valeurs 0 et 1 mais il en est de même pour les expressions. Ce qui signifie qu'une expression telle que (a+b) ne pourra contenir que les valeurs 0 et 1, même si a et b valent tous les deux 1.

$\boxed{1+1=1}$ Car le 2 n'existe pas et le résultat ne peut valoir que 0 ou 1.

$\boxed{a+a=a}$ Soit a vaut 0 et a + a = 0, soit a vaut 1 et a + a =1.

$\boxed{a+\overline{a}=1}$ $\boxed{a\times\overline{a}=0}$ Le trait horizontal au dessus signifie « complément de ».
On comprend alors que si $a=0$, $\overline{a}=1$ et si $a=1$, $\overline{a}=0$.

En examinant tous les cas, on peut montrer que les formules ci-dessus sont toujours vérifiées.

1.3 Les fonctions combinatoires élémentaires

Il y a trois fonctions combinatoires élémentaires : les fonctions ET, OU et NON.
Nous verrons que ces fonctions correspondent à des opérateurs particuliers au niveau de l'algèbre de Boole.

- La fonction ET correspond à une multiplication

L'opération $a \times b$ ne vaudra 1 que si $a = 1$ et $b = 1$
La table de vérité de la fonction ET est donnée ci-dessous :

a	b	a ET b	$S = a \times b$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

- La fonction OU correspond à une addition

L'opération $a + b$ vaudra 1 dés que $a = 1$ ou $b = 1$ (ou non exclusif)
La table de vérité de la fonction OU est donnée ci-dessous :

a	b	a OU b	$S = a + b$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

- La fonction NON correspond au trait horizontal au dessus

L'opération $\overline{a}$ vaudra 1 dés que $a = 0$ et inversement
La table de vérité de la fonction NON est donnée ci-dessous :

a	NON a	$S = \overline{a}$
0	1	1
1	0	0

1.4 Les théorèmes de De Morgan

Le complément d'une somme est égal au produit des compléments → $\boxed{\overline{a+b} = \overline{a} \times \overline{b}}$

Le complément d'un produit est égal à la somme des compléments → $\boxed{\overline{a \times b} = \overline{a} + \overline{b}}$

Ces lois traduisent le fait que :
- Le résultat d'une somme vaudra 1 dés qu'un des éléments de la somme vaut 1
- Le résultat d'un produit vaudra 0 dés qu'un des éléments du produit vaut 0

2. Les opérateurs logiques sur entiers

Nous avons vu qu'en logique la notion de VRAI et de FAUX peut être assimilée aux valeurs respectives 1 et 0. On comprend alors qu'un seul bit permet de contenir une information de type VRAI / FAUX.
Il faut néanmoins savoir que le microprocesseur est conçu pour agir directement sur un ensemble de bits (8 dans notre cas) et non sur un seul bit. C'est la raison pour laquelle un entier peut également contenir une information de type VRAI / FAUX.
Dans ce cas :

0	sera considéré comme FAUX
toute autre valeur	sera considérée comme VRAI

Une opération telle que $nb < 10$ peut alors renvoyer un entier qui sera interprété comme VRAI ou FAUX.
On peut alors trouver par exemple :

```
if(nb < 10)
        {
        //actions
        }
```

Ça signifie que si *nb* est strictement inférieur à 10, l'opération $nb < 10$ renverra un entier différent de 0 qui sera considéré comme vrai et le bloc d'actions sera exécuté.

Il est bien sûr possible d'utiliser les opérateurs logiques sur entiers, en C la syntaxe est :

ET ↔ && OU ↔ || NON ↔ !

On peut alors trouver par exemple :

```
if( !(BP1==1) && (nb < 10) )
        {
        //actions
        }
```

Ça signifie que le bloc d'actions ne sera exécuté que si BP1 n'est pas à 1 et que *nb* est strictement inférieur à 10.

3. Les opérateurs logiques bit à bit

Il existe en C des opérateurs logiques tels que ET, OU,… qui agissent directement sur les bits qui constituent une variable. La syntaxe en C est :

ET bit à bit ↔ & OU bit à bit ↔ |

Ces opérateurs sont illustrés à l'aide d'un exemple :

```
char    op1=0b00110011, op2=0b11110000, op3 ;
op3 = op1&op2 ;       //op3 contient alors le motif binaire 0b00110000
op3 = op1 | op2 ;     //op3 contient alors le motif binaire 0b11110011
```

Chapitre 11: Applications des opérateurs

1. Les masques binaires

Dans certains cas il est utile de pouvoir vérifier (ou tester) si un bit particulier d'une variable est à '0' ou à '1'. Dans ce cas on ne peut pas simplement tester la valeur de cette variable car sa valeur ne dépend pas uniquement de l'état du bit qui nous intéresse mais également de l'état des autres bits qui constituent cette variable. La démarche est alors d'utiliser un masque bien choisi afin d'obtenir un motif binaire qui ne dépendra que du bit dont on veut vérifier l'état.

Exemple 1 :

```
if ( (motif & 0b00010000) == 0b00010000)
        { //le bit d'indice 4 de motif est à 1}
else
        { //le bit d'indice 4 de motif est à 0}
```

Exemple 2 :

```
if ( (motif | 0b11111101) == 0b11111101)
        { //le bit d'indice 1 de motif est à 0}
else
        { //le bit d'indice 1 de motif est à 1}
```

Il peut également être utile de modifier l'état de certains bits d'une variable ou d'un registre de configuration sans modifier l'état des autres bits.
Dans ce cas on peut utiliser un masque binaire qui permet de sélectionner les bits à affecter. Pour affecter une valeur 1 il faut utiliser un OU bit à bit, pour affecter une valeur 0 il faut utiliser un ET bit à bit. Exemple :

```
TRISA = TRISA | 0b00000011 ;      //on affecte la valeur 1 aux bits 0 et 1 de TRISA.
                                  //Les autres bits restent inchangés.
```

2. Les opérateurs de décalage

Les opérateurs de décalage font partie des opérateurs manipulateurs de bits au même titre que les opérateurs logiques bit à bit ; il permettent d'agir directement sur le motif binaire.

Ces opérateurs sont illustrés à l'aide d'un exemple :

```
char    op1=0b00110011, op2=0b11110000, op3 ;
op3 = op1>>1 ;    //op3 contient alors le motif binaire 0b00011001, op1 reste inchangé
op3 = op2 <<2 ;   //op3 contient alors le motif binaire 0b11000000, op2 reste inchangé
```

Les opérateurs permettent par exemple de gérer certains jeux de lumière ou de faciliter l'implémentation des routines LCD en mode 4 bits.

Chapitre 12 : Les fonctions : notions de base

1. Introduction

Une fonction permet de contenir un ensemble d'instructions à exécuter.
Une fonction a un nom et permet d'effectuer une tâche.

Imaginons que l'on veuille faire clignoter une led à 1 Hz, il faudra alors répéter :

allumer la led – attendre 500 ms– éteindre la led – attendre 500 ms

Pour allumer ou éteindre une led c'est très simple, il faudra en général une seule instruction. Il n'est donc pas très utile d'avoir une fonction pour ce genre de tâche.
Par contre pour attendre 500 ms il faut beaucoup d'instructions, il est donc préférable d'utiliser une fonction. On peut alors utiliser une fonction qui s'appelle par exemple « delai500ms » et qui contient toutes les instructions nécessaires à provoquer un délai de 500 ms. Il suffira alors d'écrire « delai500ms() ; » dans le programme qui gère le clignotement de la led pour provoquer un délai de 500 millisecondes.
Cette manière de faire rend le programme plus lisible, il met bien en évidence la tâche à effectuer. Il faut bien sûr s'assurer de donner un nom évocateur à la fonction.

2. Trois aspects importants liés à la fonction

- **La déclaration d'une fonction (ou prototype d'une fonction)**

  Une fonction doit souvent être déclarée.
  La déclaration se trouve en général dans le « .h » de la librairie qui la contient.
  Déclarer une fonction consiste à préciser son nom, les arguments éventuels[*], et des informations concernant la valeur de retour[†].
  La manière de déclarer une fonction est montrée ci-dessous :

```
Type_valeur_de_retour   nom_de_la_fonction  (type_argument1, type_argument 2,...) ;
```

  Exemple :

```
float moyenne (int, int) ;
```

- **La définition d'une fonction (ou implémentation d'une fonction)**

  On appelle la définition d'une fonction l'ensemble des instructions qui précisent les actions à exécuter lors de son appel.

[*] Voir plus tard, il s'agit d'éléments supplémentaires nécessaires à la fonction.
[†] Une fonction peut avoir une valeur de retour, ça signifie qu'elle peut fournir un résultat (le résultat d'un calcul par exemple).

Cet ensemble est constitué de deux parties :

- un en-tête (header),
- un corps formé d'un bloc d'instructions (situé entre « { » et « }»).

Exemple :

```
float moyenne (int data1, int data2)
        { /*corps de la fonction*/
        }
```

- **L'appel d'une fonction**

Une fonction est destinée à pouvoir être appelée afin d'exécuter la tâche qu'elle peut remplir. Imaginons que nous sommes dans le programme principal et que nous voulons qu'il se produise un délai de 500 ms, nous pouvons écrire « delai500ms() ; » qui aura pour effet d'appeler la fonction « delai500ms » au moment de l'exécution de cette ligne du programme. Il faut bien sûr que la fonction « delai500ms » ait été préalablement implémentée !
Lors de l'appel de la fonction, l'exécution du programme se poursuivra à la 1ère ligne du corps de la fonction. Une fois que les instructions de la fonction sont terminées (à la fin du corps de la fonction qui se termine par « }» ou lorsqu'un mot clé *return* est rencontré), on revient à la ligne qui suit l'appel de la fonction pour poursuivre l'exécution du programme.
Dans la mesure où une fonction possède un ou plusieurs arguments, ceux-ci* seront transmis au corps de la fonction au moment de l'appel. Cette transmission est aussi appelée le passage de paramètres.

3. Les transmissions d'informations liées à la fonction

Dans certains cas il est nécessaire de donner des informations supplémentaires à la fonction pour qu'elle puisse effectuer sa tâche. Imaginons une fonction dont la tâche est de positionner le curseur sur un écran LCD, on comprend tout de suite que si on ne précise pas à la fonction où le curseur doit être positionné, la tâche ne pourra pas être réalisée. Il est donc nécessaire de préciser la position du curseur afin que cette information (donnée) soit apportée à la fonction au moment de l'appel.

- **Les données apportées à la fonction au moment de l'appel**

Nous allons montrer ci-dessous deux exemples de fonctions qui n'ont pas de valeur de retour†, le 1er exemple nécessite un argument, le 2ème n'en a pas besoin.

* En réalité il s'agit d'une copie des arguments qui sont passés en paramètre. Cependant si on passe une copie de l'adresse d'une variable en paramètre, cette copie permet de modifier la valeur de cette variable (passage par adresse, par pointeur).

† Voir un peu plus loin.

Exemple 1, fonction permettant de positionner le curseur sur un écran LCD :

```
void pos_lcd (char);    // prototype de la fonction

Une manière de faire appel à une telle fonction est par exemple :
char pos = 5 ;
pos_lcd (pos);          // place le curseur à la position n° 5  de l'afficheur lcd
```

Exemple 2, fonction permettant d'effacer un écran LCD :

```
void clear_lcd (void); // prototype de la fonction

Pour appeler cette fonction il suffit d'écrire :
clear_lcd ( );
```

L'exemple 1 nécessite d'apporter une donnée de type char à la fonction.
Lors de l'appel, le contenu de la variable *pos* sera apporté à la fonction.
L'apport de donnée(s) se trouve entre parenthèses : pos_lcd (pos);

L'exemple 2 ne nécessite pas de données supplémentaires pour que la fonction puisse réaliser sa tâche. Ceci est mis en évidence par le mot clé *void* entre parenthèses dans le prototypes et par des parenthèses vides : clear_lcd( ) ;

Ce qui est commun aux deux fonctions, c'est que lorsque la fonction est finie (retour de la fonction), la fonction ne renvoie pas de valeur de retour. Ceci est mis en évidence par le mot clé *void* au tout début des prototypes des deux fonctions. *void* signifie « vide » et permet de préciser qu'il n'y a pas d'argument et/ou valeur de retour.

- **Un résultat apporté par la fonction au moment du retour**

Dans certains cas on veut qu'une fonction puisse fournir un résultat ; par exemple calculer la moyenne de deux nombres entiers ou générer un nombre entier aléatoire. Dans ces conditions, nous dirons qu'il s'agit de fonctions avec valeur de retour.

Le type de la valeur de retour est déjà mis en évidence au niveau du prototype de la fonction, c'est le premier mot clé qui apparaît. Dans la mesure où ce mot clé n'est pas « void », cela signifie que la fonction a une valeur de retour, qu'elle permet de fournir un résultat. Le fait que la fonction a une valeur de retour est également mis en évidence au sein de la définition de la fonction à l'aide du mot clé *return* (voir exemple 2 ci-dessous).

Exemple 1, fonction permettant de générer un nombre aléatoire :

```
int random_int (void);   // prototype de la fonction
                         // pas d'argument mais une valeur de retour de type « int »

Une manière de faire appel à une telle fonction est par exemple :
```

```
int a ;
a= random_int ();
```

Exemple 2, calcul de la moyenne de deux nombres entiers :

```
float moyenne (int, int) ;        // prototype de la fonction
                                  // deux arguments de type « int »
                                  // une valeur de retour de type « float »
```

Une manière de faire appel à une telle fonction est par exemple :

```
int a = 6, b = 11 ;
float c ;
c = moyenne (a,b);      //la variable c contiendra la valeur (6+11)/2, soit 8,5
```

Montrons ci-dessous une manière d'implémenter cette fonction.

```
float  moyenne (int nb1, int nb2)
        {
        float moy ;
        int somme ;
        somme =  nb1+ nb2 ;
        moy = somme / 2 ;
        return moy ;
        }
```

Essayons de comprendre ce qu'il se passe lorsqu'on appelle la fonction, c'est-à-dire lorsqu'on rencontre la ligne « c = moyenne (a,b); » :

Comme il s'agit d'une affectation (symbole « = »), il faut mettre dans la variable *c* le résultat de la fonction *moyenne*. Pour cela il faut faire appel à la fonction (c'est-à-dire exécuter les lignes d'instructions qui se trouvent dans le corps de la définition de la fonction). Il faut ensuite pouvoir, à la fin de la fonction, recevoir une valeur de retour.

L'appel se produit donc avec copie des valeurs des variables *a* et *b* au sein des variables *nb1* et *nb2*. Les deux variables locales[*] *nb1* et *nb2* peuvent donc être utilisées au sein de la fonction.

Le corps de la fonction commence par la partie déclaration de variables qui doit toujours se trouver au début. Dans notre cas il y a deux variables locales à la fonction (variables temporaires) qui sont déclarées (*somme* et *moy*).
Une fois que la variable *moy* contient le résultat de la moyenne de *nb1* et de *nb2* (et donc de *a* et de *b* juste avant l'appel), la fonction peut se terminer grâce à l'instruction « return ». La ligne « return moy ; » signifie que l'exécution de la fonction doit se terminer, que la valeur de retour de la fonction est le contenu de la variable *moy* et qu'il faut poursuivre l'exécution du programme à l'endroit où l'on se trouvait au moment de l'appel de la fonction. Dans notre cas, la suite est de mettre dans la variable *c*, la valeur de retour de la fonction. Pour rappel, la ligne en cours était : « c = moyenne (a,b); ».

[*] Voir la différence entre variables locales et globales un peu plus loin.

Chapitre 13 : Les bits indicateurs d'état.

Un bit indicateur d'état est également appelé « flag » (ou drapeau en français).
Une variable de type « flag » est une variable qui contient une valeur 0 ou 1 suivant que l'on soit dans une situation ou une autre.

Il faut savoir qu'un « flag » est un outil pratique pour le programmeur, c'est lui qui choisit d'utiliser un « flag ». Souvent un « flag » bien utilisé va aider à écrire le programme mais aussi à le rendre plus lisible.

Exemple 1:

Imaginons que l'on programme un appareil qui a la possibilité d'être en mode veille.
Il est logique qu'en mode veille toutes les fonctionnalités habituelles ne sont plus disponibles (une action sur un bouton n'aura pas forcément le même effet).
Dans ce cas le programmeur pourra choisir d'utiliser une variable telle que *est_en_mode_veille* dont la valeur informe sur l'état de veille de l'appareil.
Si la variable *est_en_mode_veille* vaut 0 (FAUX) ça signifie qu'il est faux qu'on est en mode veille, sinon on est en mode veille.
On pourrait alors trouver au sein du programme une structure telle que :

```
if (est_en_mode_veille==0)
        {
        //gestion du mode normal
        }
else
        {
        //gestion du mode veille
        }
```

Il faut bien sûr que le programmeur s'arrange pour qu'à tout moment la variable *est_en_mode_veille* contienne une valeur représentative de la réalité. Ce qui signifie qu'il faut affecter la valeur 1 lorsqu'on sort du mode normal (et que l'on passe en mode veille) et affecter la valeur 0 lorsqu'on sort du mode veille (et que l'on passe en mode normal).

Exemple 2:

Imaginons par exemple un robot qui trie des objets mais qui ne peut pas avancer pendant qu'il est occupé à trier. Nous pouvons alors utiliser une variable de type « flag » qui contient à tout moment l'information sur le fait que le robot est en train de trier ou non. Dans ce cas, pour savoir s'il a le droit d'avancer, il doit aller vérifier l'état du « flag » pour savoir si le tri est en cours. Ce qui est délicat avec ce type de variable c'est qu'il faut constamment que la valeur de la variable « flag » reflète l'état réel de ce qu'il indique. Autrement dit, dans l'exemple du robot trieur, si un moment au sein du programme le robot démarre un tri, il faut mettre à jour notre variable ; même chose évidemment s'il s'arrête de trier.

Chapitre 14 : Les entiers indicateurs d'état.

1. Généralités

L'idée ici est la même que pour le bit d'indicateur d'état sauf qu'il se rapporte à un élément qui peut se retrouver dans plus de deux situations différentes.
On peut imaginer par exemple une variable nommée *menu_en_cours* qui contient le numéro du menu en cours. On a alors une variable qui contient un entier indiquant le menu en cours d'utilisation.
De la même manière que pour les « flags », les entiers indicateurs d'état doivent refléter à tout moment la réalité. Dans notre exemple, la variable *menu_en_cours* doit à tout moment contenir le numéro correct du menu qui est en cours.
C'est au programmeur d'assurer une bonne gestion des variables d'états.

Exemples :

L'élément dont les états sont représentés par la variable	Nom donné à la variable
L'affichage de l'écran qui peut correspondre à plusieurs menus	*menu_en_cours*
Un jeu qui peut avoir différents degrés de difficulté. La difficulté peut être paramétrée en début de partie ou même varier au sein d'une même partie.	*difficulte_jeu*
Un jeu de tétris peut avoir un type de pièce spécifique en cours de descente, il peut être utile de connaître le type de pièce qui est en train de descendre pour savoir l'action à faire lors d'un déplacement à gauche ou à droite.	*piece_en_descente*

Remarques :

Dans tous les exemples que l'on vient de voir, les noms donnés aux variables ne sont pas les uniques possibilités, nous aurions pu par exemple choisir de nommer la variable *menu_en_cours* du nom de *menu_actuel*. Il y a donc plusieurs possibilités pour le nom d'une variable mais il y en a des meilleurs que d'autres. Il faut essayer de faire en sorte que le nom de la variable évoque réellement son utilité. On devrait même pouvoir deviner dans laquelle des catégories de variable on se trouve juste avec le nom de la variable :

Une variable telle que *nb_objets_stockes* évoque un nombre entier positif.
La variable *vitesse_m_par_s* évoque une grandeur physique, c'est probablement un réel.
La variable *jeu_a_demarre* donne une information sur l'état du système, on devine qu'avec ce nom de variable, si *jeu_a_demarre* contient la valeur '1' c'est que le jeu a démarré. Enfin, la variable *menu_en_cours* donne une information sur le menu qui est en cours, avec un tel nom, on peut se douter qu'il y a de nombreux menus possibles et que la variable indique lequel est d'application à l'emplacement concerné du programme.

Les deux dernières catégories de variable (« bit indicateur d'état » et « entier indicateur d'état ») sont des éléments difficiles à cerner au début. Une telle variable survient lorsque le programmeur ressent de besoin de connaître une information concernant l'état du système afin de gérer un élément extérieur qui apparaît.

2. La structure de choix multiple

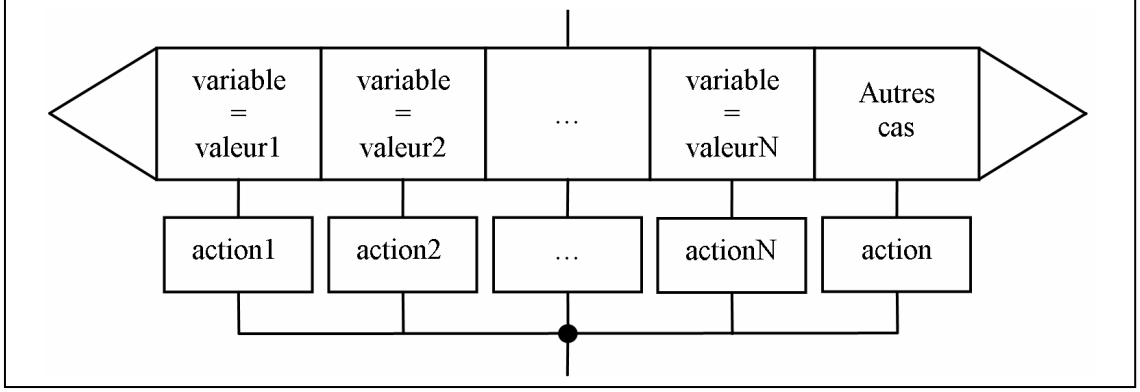
Il faut manipuler la structure de choix multiple avec extrême précaution ! Il doit s'agir d'une même variable susceptible de prendre différentes valeurs.

La structure de choix multiple permet de faire une action selon que la valeur d'une variable corresponde à une certaine valeur. On peut alors décrire l'action qui doit être effectuée pour chaque valeur qu'une même variable est susceptible de rencontrer.

Choix multiple en Ordinogramme[*] :

Ci-dessous la manière de représenter un choix multiple en ordinogramme lorsqu'un cas par défaut est présent (le mot clé « autrement » figure au sein du LDA) :

Figure 25 : Ordinogramme de la structure de choix multiple avec action par défaut



Une seule action sera réalisée suivant la valeur de la variable. Soit la variable contient une des N valeurs spécifiées et l'action correspondante est réalisée, soit la variable a une valeur différente de celles spécifiées et c'est l'action par défaut qui sera réalisée.

Par contre, si on se trouve dans l'autre cas de figure (sans action par défaut), l'ordinogramme aura une allure similaire à ci-dessus mais le bloc « Autres cas » et l'action correspondante ne seront pas représentés.

[*] Il n'est pas certain que l'ordinogramme d'un choix multiple soit normalisé mais celui décrit ici est celui que l'on retrouve fréquemment en pratique.

Choix multiple en C :

En C la structure du choix multiple est souvent appelée « switch…case » car ce sont les mots clés principaux. « Switch » signifie changer en anglais et « case » signifie cas.
La structure « switch…case » se charge d'encapsuler les blocs d'actions des différents cas, chaque cas apparaît lorsque la variable (de la catégorie des entiers) correspond à une valeur spécifiée. Lors de l'exécution du programme, le microprocesseur va alors changer les valeurs de comparaison de la variable de manière à déterminer le bloc d'actions à exécuter.

Comme déjà précisé, il ne faut utiliser la structure de choix que lorsqu'il s'agit d'une même variable susceptible de prendre différentes valeurs. Voici la syntaxe en C :

```
switch (variable de type char ou int)
        {
        case valeur1:           //si variable = valeur1 : exécuter ce bloc d'instructions
                  .......;
                 break;
         case valeur2:          //si variable = valeur2 :  exécuter ce bloc d'instructions
                  .......;
                 break;
        ……                      //...

        default:                //aucune des valeurs précédentes : exécuter ce bloc
                  .......;      //d'instructions, pas de "break" ici
        }
```

Le mot clé *default* ici dessus correspond au mot clé *autrement* du LDA, il se peut donc qu'il ne soit pas présent.

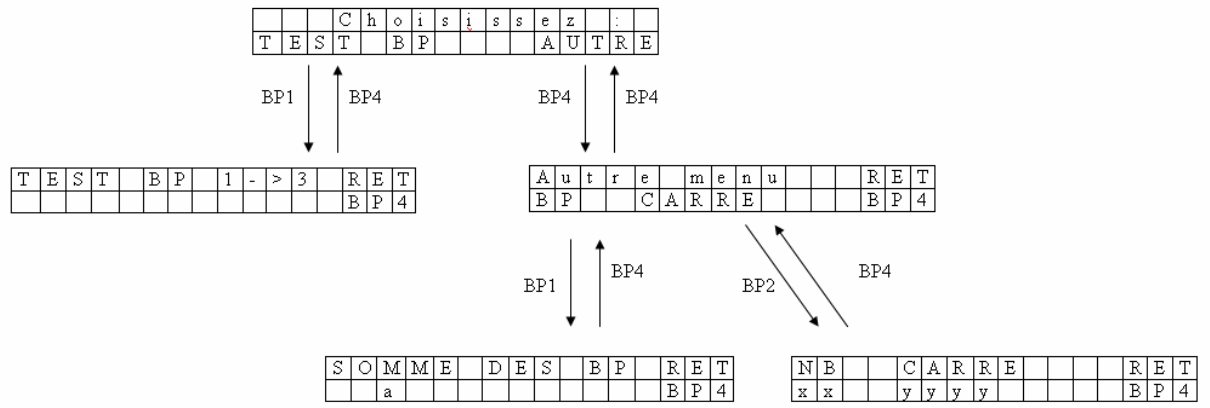
Le mot clé *break* signifie sortir de la structure* courante. Dans notre cas, ça veut dire sortir de la structure de choix multiple. Il s'agit de continuer le programme juste après la fermeture de l'accolade.

Les accolades '{' et '}' permettent de délimiter la structure de choix multiple.
De la même manière, les accolades au sein d'une structure de choix permettent de délimiter la (ou les) ligne(s) d'action à exécuter, dépendant de la condition.

* La structure de choix simple n'est pas concernée.

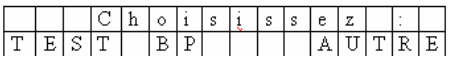
3. Gestion des menus à l'aide de la structure de choix multiple

La structure de choix multiple convient très bien pour gérer une arborescence de menus. On a bien affaire à une même variable (par exemple *menu_en_cours*) susceptible de prendre de nombreuses valeurs différentes. Chaque valeur correspond alors à un numéro de menu. Imaginons l'arborescence de menus ci-dessous :

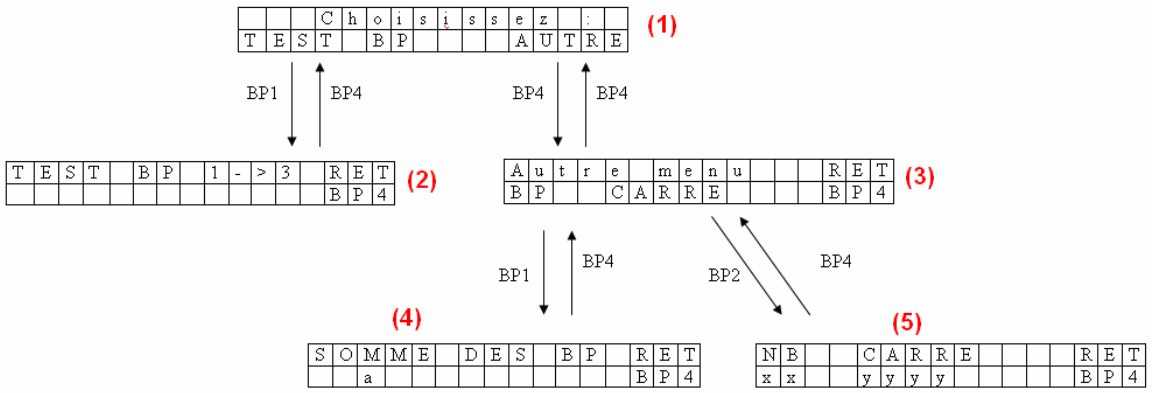


On va alors commencer par choisir une valeur à notre variable *menu_en_cours* pour chaque état d'affichage de menu. Il faut bien sûr s'assurer que des menus différents aient des numéros de menus différents. On décidera par exemple :

Lorsque *menu_en_cours* vaut 1 ↔



En continuant la numérotation des menus, on aura par exemple :



Le programmeur devra alors prendre soin de s'arranger pour que la variable *menu_en_cours* ait toujours la valeur correcte du menu qui est d'application.
On doit par exemple s'assurer que si on est dans le menu numéro (3) et que l'on détecte un enfoncement de BP4, on mette à jour la variable *menu_en_cours* en lui donnant la valeur 1. En effet, l'arborescence ci-dessus illustre bien qu'à partir du menu (3), un enfoncement de BP4 nous amène au menu (1).

Nous exposons ci-dessous la structure d'un programme en C qui gère l'arborescence présentée. Tous les détails du programme ne sont pas présentés (l'ensemble des déclaration de variables, l'implémentation des fonctions,…) mais la gestion de la variable *menu_en_cours* y est illustrée.

```
void main(void)
{
char menu_en_cours=1;
//…
init();

while(1)
        {
        switch(menu_en_cours)
                {
                case 1 :
                        affiche_menu_1();
                        if(BP1)
                                menu_en_cours=2;
                        else if(BP4)
                                menu_en_cours=3;
                        break;

                case 2 :
                        affiche_menu_2();
                        LED1=BP1;  LED2=BP2;  LED3=BP3;
                        if(BP4)
                                menu_en_cours=1;
                        break;

                case 3 :
                        affiche_menu_3();
                        if(BP1)
                                menu_en_cours=4;
                        else if(BP2)
                                menu_en_cours=5;
                        else if(BP4)
                                menu_en_cours=1;
                        break;
                case 4 :
                        affiche_menu_4();
                        if(BP4)
                                menu_en_cours=3;

                        break;
                case 5 :
                        affiche_menu_5();
                        if(BP4)
                                menu_en_cours=3;

                        break;
                }
        }//fin du while(1)

}//fin du main
```

Chapitre 15: LDA

1. Langage de description algorithmique

LDA signifie Langage de Description Algorithmique, il permet de mettre en évidence la structure du programme en restant indépendant du langage de programmation.
Le LDA est écrit en français, il utilise un certain nombre de mots clés qui doivent être soulignés. Il y a des mots clés tels que : Lire, Ecrire, Entiers, Réels… et des mots clés qui permettent de mettre en évidence les structures comme montré ci-dessous.

Structure de choix en LDA

```
si expression alors
        //bloc d'action 1
sinon
        //bloc d'action 2
fsi
```

Structure de choix en C

```
if(expression)
        {//bloc d'action 1
        }
else
        {//bloc d'action 2
        }
```

Choix multiple en LDA

```
selon que
        i=1 faire
                //séquence 1 d'instructions
oq      i=2 faire
                //séquence 2 d'instructions
oq      i=3 faire
                //séquence 3 d'instructions
autrement
        //séquence 4 d'instructions
fselon
```

Choix multiple en C

```
switch (i)
        {case 1 : //séquence 1 d'instructions
                  break;
         case 2 : //séquence 2 d'instructions
                  break;
         case 3 : //séquence 3 d'instructions
                  break;
         default :
                 //séquence 4 d'instructions
        }
```

« tant que » en LDA

```
tant que expression faire
        //bloc d'actions
ftant
```

« tant que » en C

```
while (expression)
        {//bloc d'actions
        }
```

«répéter… tant que » en LDA

```
répéter
        //bloc d'actions
tant que  expression
```

«répéter…tant que » en C

```
do      {//bloc d'actions
        }
while (expression) ;
```

«boucle pour » en LDA

```
pour i de 1 à 10 faire
        //bloc d'actions
fpour
```

«boucle pour » en C

```
for (i=1 ; i<=10 ; i++)
        {//bloc d'actions
        }
```

2. Transformation ordinogramme en LDA

Pour pouvoir transformer un ordinogramme en LDA, il faut pouvoir ressortir les éléments qui participent à la structure générale du programme, c'est-à-dire :

- La structure de choix (avec ou sans « sinon »)
- La structure répétitive type « tant que »
- La structure répétitive type « répéter … tant que »
- La structure répétitive type « pour »

Pour que la transformation ordinogramme → LDA soit possible[*], l'ordinogramme doit être structuré ; c'est-à-dire organisé conformément aux structures citées ci-dessus.

Pistes pour transformer un ordinogramme en LDA :

- Le LDA ne contient pas d'informations particulières pour signaler le début et la fin, il faut donc enlever les entités « début » et « fin » de l'ordinogramme.

- Tout ce qui se trouve au sein d'un rectangle dans l'ordinogramme correspond à une action, il faut donc, pour le LDA, simplement recopier à la suite le contenu des différents rectangles.

- Lorsque vous avez un losange au niveau de l'ordinogramme, vous devez commencer par déterminer s'il s'agit d'un test ou d'une boucle.
  Une manière de faire est de suivre le cheminement de chaque branche de sortie et de vérifier qu'elles se rejoignent bien avant de poursuivre la suite du programme. Si c'est le cas, il s'agit d'une structure de choix.

- Dans la mesure où vous décelez qu'il s'agit d'une structure de choix vous devez commencer par écrire le mot clé « si » accompagné de la condition indiquée dans le losange. Ensuite vous devez regarder la branche « FAUX » et voir s'il y a des lignes d'actions associées ; s'il n'y en a pas, la structure alternative en LDA ne contiendra pas de « sinon ». Il faut ensuite que toutes les actions de la branche « VRAI » se retrouvent dans le LDA après le mot clé « alors ». Pour toutes les actions de la branche « FAUX », c'est après le mot clé « sinon ». Vous devez terminer la structure de choix par un « fsi ».

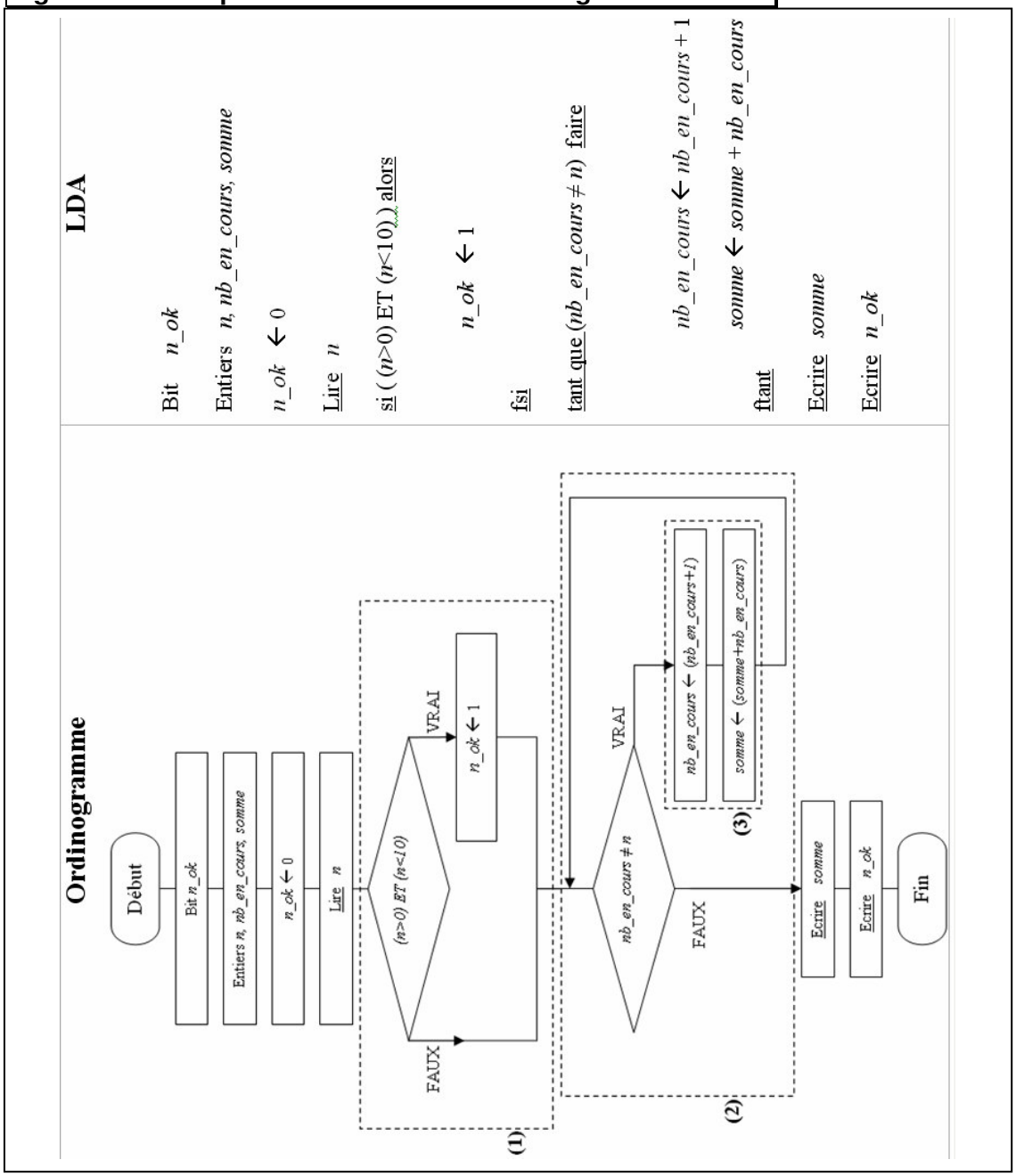
- Dans le cas où vous avez une structure répétitive vous devez vous référer aux structures des différentes boucles ont été vues et essayer d'identifier le cas rencontré à l'une d'entre elles.

[*] Sans faire appel à des artifices particuliers

3. Exemple de transformation ordinogramme en LDA

Imaginons un programme qui doit prendre en compte un nombre *n* introduit par l'utilisateur et apporter comme résultat la somme des *n* 1[ers] nombres. Il faut aussi que le programme apporte à l'utilisateur une indication par rapport à la validité du nombre introduit (le nombre *n* doit être strictement positif et strictement inférieur à 10).

Figure 26 : Exemple de transformation ordinogramme → LDA



(1) Correspond à la structure de choix (type Si-Alors)
(2) Correspond à la structure répétitive « tant que »
(3) Correspond au bloc d'actions de la « tant que », c'est le bloc «Actions » Fig.11.

Chapitre 16 : Eléments supplémentaires

1. Opérateurs supplémentaires

- **Opérateurs d'incrémentation et de décrémentation**

```
i= i + 1 ;      est équivalent à      i++ ;
i= i - 1 ;      est équivalent à      i— ;
i= i + 10 ;     est équivalent à      i += 10 ;
i= i – 30 ;     est équivalent à      i -= 30 ;
```

A noter une priorité dans la syntaxe :

```
c= i++ / b;     // c= i / b et i est incrémenté après l'opération
c= ++i / b;     // i est d'abord incrémenté. Ce qui signifie que c sera affecté de
                //c= (i +1) / b   en considérant la valeur de i avant incrémentation.
```

- **Opérateurs de manipulation de bits Xor et complément à 1**

Opérande1 (op1) 00110011		
Opérande2 (op2) 11110000		
op1 ^ op2	11000011	// ^ est l'opérateur Xor
~op1	11001100	// ~ est l'opérateur pour le complément à 1

2. L'instruction break

L'instruction « break ; » permet de sortir prématurément d'une boucle. Elle a pour effet de forcer la suite du programme juste après l'accolade fermée « } » de la boucle en cours. Elle met donc fin à la boucle en cours de portée la plus proche (que ce soit une *for*, *while* ou d*o…while*). L'instruction « break ; » est également utilisée pour sortir de la structure « *switch case* ».

On peut l'utiliser par exemple lorsqu'on veut faire un maximum de 50 clignotements d'une LED en gardant néanmoins la possibilité d'interrompre les clignotements à l'aide d'un bouton-poussoir.

```
for (i=0 ;i<50 ;i++)
        {
        if(BP1==1)
                break ;
        LED1=1 ;
        delay_ms(200) ;
        LED1=0 ;
        delay_ms(200) ;
        }
```

3. Les tableaux et les chaînes de caractères

Le langage C permet d'utiliser des tableaux. On nomme ainsi un ensemble d'éléments de même type désignés par un identificateur unique, chaque élément est repéré par un indice précisant sa position au sein de l'ensemble.

La déclaration d'un tableau se présente comme suit : type nom [dim];

On précise d'abord le type puis le nom* du tableau accompagné d'une constante entre [ ] pour préciser la dimension.

Exemples : int compteur[10]; // le compilateur réserve des places pour 10 entiers
float nombre[20]; // le compilateur réserve des places pour 20 réels

Dans certains cas, on peut être amené à définir un tableau à deux dimensions, la dimension prend la forme :

type nom[dim1][dim2];

On peut aussi initialiser les tableaux au moment de leur déclaration :

Exemples : int liste[10] = {1,2,4,8,16,32,64,128,256,512};
float nombre[4] = {2.67,5.98,-8,0.09};
int x[2][3] = {{1,5,7},{8,4,3}}; // 2 lignes et 3 colonnes //

Pour utiliser un élément du tableau il suffit de l'appeler par son nom et son indice.

Exemple : var = liste[2] ;

La ligne ci-dessus va affecter la variable *var*† de l'entier qui se trouve à l'indice 2 du tableau nommé *liste*. La variable *var* va alors recevoir le 3ème élément du tableau *liste* car on compte les indices à partir de 0. A la fin de l'instruction, la variable *var* contiendra alors la valeur 4 (en se référant au tableau *liste* déclaré juste au dessus).

En langage C, les chaînes de caractères sont des tableaux de caractères. Leur manipulation est donc analogue à celle d'un tableau à une dimension :

En déclarant un tableau, on définit automatiquement un pointeur. Un pointeur est destiné à contenir l'adresse d'un autre objet.

---

* Le nom du tableau est un pointeur (voir plus loin) sur le type défini au moment de la déclaration du tableau. Ce pointeur contient l'adresse du 1er élément du tableau.

† Il est bien sûr nécessaire que la variable *var* ait été préalablement déclarée et que son type soit le même que celui des éléments du tableau (sans quoi la conséquence de l'affectation risque de ne pas apporter le résultat souhaité).

4. Les pointeurs

● L'opérateur d'adresse &

Expliquons d'abord ce qu'est l'adresse d'une variable.

Lorsqu'on déclare une variable, on réserve en réalité de la place en RAM qui permettra de placer une valeur dans cette variable. Imaginons que l'on déclare successivement quatre variables et que la 1ère place disponible en RAM soit la case 0 (à l'adresse 0). On peut illustrer l'influence des déclarations de variables sur la RAM comme ci-dessous :

1°) On déclare

char *variable1* ;
char *variable2* = 0b11100011 ;
int *nombre1* ;
int *nombre2* = 1023 ;

2°) Après la déclaration, nous avons en RAM :

adresses	Cases en RAM								Nom de la variable
0	?	?	?	?	?	?	?	?	*variable1*
1	1	1	1	0	0	0	1	1	*variable2*
2	?	?	?	?	?	?	?	?	*nombre1*
3	?	?	?	?	?	?	?	?	
4	0	0	0	0	0	0	1	1	*nombre2*
5	1	1	1	1	1	1	1	1	
…									

La déclaration d'une variable de type char réserve la place nécessaire en RAM (un octet). Comme *variable1* n'a pas été initialisée, son contenu n'est pas connu.
La prochaine case mémoire disponible servira à contenir la valeur de *variable2*.
Pour les entiers c'est le même principe sauf qu'il faut réserver 2 octets par entier.
La page précédente nous a mis en évidence le fait que chaque variable est destinée à contenir une donnée mais aussi que chaque variable a une adresse ; on peut remarquer par exemple que *variable2* est à l'adresse 1.
On peut alors comprendre l'opérateur d'adresse & qui signifie « l'adresse de ». On peut dire que dans notre exemple, &*variable2* est la même chose que l'adresse 1.

● L'opérateur *

Nous avons mis en évidence qu'une variable contient une donnée et a une adresse (se trouve quelque part en RAM). On peut alors accéder au contenu de cette variable (la donnée) de manière directe. Lorsqu'on écrit par exemple « i = 30 ; » il s'agit d'un adressage direct car on manipule directement la variable dont l'adresse a été attribuée au moment de la déclaration.
Il est cependant possible de manipuler une variable de manière indirecte en n'accédant plus directement au contenu de la variable mais en se servant de l'adresse de la variable. Il s'agit alors d'un adressage indirect.

Pour manipuler une adresse on n'utilise plus une variable classique, on utilise une variable pointeur (ou simplement pointeur). Pour résumer, nous avons :

Une variable classique :	Se trouve à une adresse	|	Contient une donnée
Une variable pointeur :	Se trouve à une adresse	|	Contient une adresse

De la même manière qu'une variable classique, il est possible d'apporter des précisions supplémentaires concernant le contenu d'une variable pointeur. En effet :

> Pour une variable classique on peut, au moment de sa déclaration, préciser qu'il s'agit par exemple d'un entier. Nous avons dans ce cas :
> Une variable classique de type entier :
> - Se trouve à une adresse
> - Contient une donnée de type « entier »
>
> Pour une variable pointeur on peut, au moment de sa déclaration, préciser par exemple qu'elle est destinée à contenir l'adresse d'un entier. Nous avons alors :
> Une variable pointeur sur entier :
> - Se trouve à une adresse
> - Contient une adresse d'un entier.

Il faut bien comprendre que lorsqu'on déclare un pointeur sur un type particulier, ce type aura une influence lors de chaque manipulation de ce pointeur.

Pour déclarer une variable pointeur on utilise l'opérateur * qui doit se trouver juste avant le nom de la variable pointeur. Exemple :

```
int     *ptr_int_1, *ptr-int2 ;          //on déclare deux pointeurs sur entier
```

Au moment de la déclaration de ces deux pointeurs sur entier, il va il y avoir attribution des adresses de ces deux pointeurs mais leurs contenus sont indéterminés. Cela signifie que tant que l'on n'affecte pas ces pointeurs d'une adresse significative, ces pointeurs contiennent une adresse quelconque et ne permettent donc pas de manipuler indirectement le contenu d'une variable.
Une fois qu'un pointeur contient l'adresse d'une variable, on dit que ce pointeur pointe sur cette variable. On peut alors modifier le contenu de cette variable en utilisant l'opérateur *. Lors de la manipulation du pointeur on peut traduire « * » par « ce qui est pointé par ».

On peut alors s'inspirer de l'exemple ci-dessous :

```
int     a, b ;          // déclaration de 2 variables de type entier appelées « a » et « b »
int     *ptr_int ;      // déclaration d'un pointeur sur entier appelé « ptr_int »
a = 20 ;                // on met 20 dans « a »
ptr_int = &b ;          // on met l'adresse de « b » dans « ptr_int »
                        // on dit que « ptr_int » pointe sur « b »
*ptr_int = 15 ;         // on met dans ce qui est pointé par « ptr_int » la valeur 15
                        // on met donc indirectement 15 dans « b »
```

```
a= *ptr_int ;          // on met dans « a » ce qui est pointé par « ptr_int »
                       // on met donc dans « a » le contenu de « b », c'est-à-dire 15
ptr_int = &a ;         // « ptr_int » pointe maintenant sur a
*ptr_int = b+1 ;       // va mettre indirectement dans « a » la valeur 16
ptr_int ++ ;           // ATTENTION :  va modifier l'adresse contenue par « ptr_int »
                       // « ptr_int » contient son ancienne adresse (l'adresse de « a »)
                       // augmentée du nombre d'octets qu'occupe un entier.
```

5. Notions supplémentaires sur la fonction

Les variables globales et les variables locales :

Les fonctions manipulent des variables, il en existe de deux catégories différentes : les variables globales et les variables locales.

- **Les variables globales**

Il n'y a qu'un seul niveau de variables globales et elles sont accessibles par toutes les fonctions pour autant que la déclaration de ces variables globales précède les fonctions. Un programme bien construit possède peu de variables globales.
Une variable globale est déclarée en dehors de toute fonction ; elle pourra être reconnue par toutes les fonctions dont la définition apparaît plus loin dans le fichier source.
Si une fonction utilise une variable globale déclarée dans un autre fichier source, il faut déclarer cette variable en la faisant précéder du mot-clé « **extern** », afin de prévenir le compilateur que l'allocation de l'emplacement mémoire est pris en considération dans un autre fichier source.
Il est aussi possible de ne pas partager la variable globale avec d'autres fichiers sources en la faisant précéder du mot-clé « **static** ». Le même nom de variable pourra être utilisé sans problème dans d'autres fichiers sources.

- **Les variables locales**

Les variables locales ne sont connues que d'une seule fonction, celle dans laquelle la variable est déclarée. La variable locale a la même durée de vie que la fonction qui l'utilise, son emplacement mémoire et sa valeur ne sont pas rémanents.
Il est parfois nécessaire de fixer l'emplacement d'une variable locale et ainsi fixer sa valeur entre deux appels de la fonction qui l'utilise. Il suffit de la déclarer à l'aide du mot-clé « **static** ». Elle ressemble alors à une variable globale à la portée près, car celle-ci est limitée à la fonction qui l'utilise.

Les passages de paramètres :

Lorsque les fonctions possèdent des arguments, il y aura des passages de paramètres aux moments des appels des fonctions. Autrement dit, il y aura des données supplémentaires apportées à la fonction pour qu'elle puisse effectuer sa tâche.

- **Passage par valeur**

Le passage par valeur illustre tout ce que nous avons vu jusqu'à présent en terme de passage de paramètre. Il s'agit de recopier la valeur des paramètres effectifs (ceux introduits au moment de l'appel de la fonction) au sein des variables locales à la fonction. Dans notre exemple du calcul de la moyenne (p.39), le passage des paramètres consiste à copier les valeurs des variables effectives (*a* et *b*) au sein des variables *nb1* et *nb2*. Tout ça fonctionne très bien lorsqu'on veut calculer la moyenne mais ne permet en aucun cas de modifier la valeur de *a* ou de *b*.

Imaginons une fonction dont la tâche est de multiplier une variable entière par 10. Essayons avec un passage par valeur :

Prototype de la fonction : `void multiplie_par_10 (int) ;`

Définition de la fonction :
```
void multiplie_par_10 (int nb)
        {nb = nb *10 ;}
```

Au sein de la fonction principale :
```
int b, a = 5 ;
multiplie_par_10(a) ;
b = a ;  //on affecte la valeur 5 à b car a n'a
        //pas été modifié avec le passage par valeur
```

Le problème ici est que ce n'est que la valeur de la variable introduite au moment de l'appel qui est transmise à la fonction. On a alors la valeur de *a* qui sera communiquée à la fonction via la variable locale et temporaire *nb*. On aura alors *nb* qui contiendra la valeur 5. Après la ligne «nb = nb *10 ; », *nb* contiendra la valeur 50, mais lorsque la fonction est terminée, la variable *a* n'a pas changé. En effet, ce n'est que la copie de la variable *a* qui a été utilisée.
Il est cependant possible d'utiliser un passage par valeur pour modifier le contenu d'une variable dans la mesure où la modification se fait en dehors de la fonction (avec l'affectation). Exemple :

Prototype de la fonction : `int renvoie_10_fois_plus (int) ;`

Définition de la fonction :
```
int renvoie_10_fois_plus (int nb)
        {
        int nb_fois_10 ;
        nb_fois_10 = nb * 10 ;
        return nb_fois_10 ;
        }
```

Au sein de la fonction principale :
```
int b, a = 5 ;
a= renvoie_10_fois_plus(a) ;
b = a ;  //on affecte la valeur 50 à b car a a été
        //modifié en dehors de la fonction
```

- **Passage par adresse, par pointeur**

En C, tout ce qui sera transmis à une fonction sera copié, c'est ce qu'on appelle le passage par valeur. On peut alors se demander comment une fonction peut-elle modifier le contenu d'une variable qui est déclarée en dehors de la fonction.

Une 1ère solution est de manipuler une variable globale, dans ce cas la variable est directement modifiable à l'intérieur de la fonction. Il n'est alors pas nécessaire de faire passer cette variable globale comme paramètre de la fonction car elle y est directement accessible. Il faut cependant faire attention avec les variables globales car elles peuvent être modifiées au sein du programme principal mais aussi au sein d'autres fonctions, ce qui nécessite une grande prudence du programmeur.
C'est une des raisons pour laquelle il faut éviter d'utiliser trop de variables globales.

Une 2ème solution est le passage par adresse. L'idée est de transmettre, au moment de l'appel d'une fonction, une copie de l'adresse d'une variable.
Grâce à l'adresse d'une variable (même une copie de l'adresse), il est possible de modifier le contenu de cette variable.
Mais qu'est-ce qui permet de contenir l'adresse d'une variable ? Réponse : un pointeur.
Nous allons donc utiliser une fonction qui utilise un pointeur pour le passage de paramètre. Grâce à cette façon de faire, au moment de l'appel de la fonction, l'adresse de la variable sera transmise à la fonction par l'intermédiaire d'un pointeur.

Exemple :

Prototype de la fonction : `void multiplie_par_10 (int *) ;`

Définition de la fonction :
```
void multiplie_par_10 (int *nb)
        {*nb = *nb *10 ;}
```

Au sein de la fonction principale :
```
int b, a = 5 ;
multiplie_par_10(&a) ;
b = a ;  //on affecte la valeur 50 à b car a a été
         //modifié grâce à son adresse
```

Le prototype de la fonction nous montre qu'il faut passer un pointeur sur entier comme paramètre à la fonction. La définition de la fonction précise que c'est *nb* qui servira de pointeur sur entier au sein de la fonction et qui contiendra l'adresse d'un entier au moment de l'appel de la fonction.
Au moment de l'appel de la fonction (lorsque la ligne « multiplie_par_10(&a) ; » est rencontrée), l'adresse de la variable *a* sera recopiée au sein du pointeur sur entier *nb*.
Ensuite, le corps de la fonction est exécuté, c'est-à-dire :

« *nb = *nb *10 ; »

On affecte à ‘ce qui est pointé par *nb*’ (donc à *a*) ce qui est à droite du symbole d’affectation « = ». On met donc dans *a* ce qui est pointé par *nb* (donc *a*) multiplié par 10. En résumé, on place dans *a* son ancienne valeur multipliée par 10.

6. Les interruptions

Les interruptions sont très utiles en informatique embarquée.
Si on veut par exemple faire clignoter une led en permanence, avec une gestion classique le µC va passer tout son temps à la gestion du clignotement ; il va d’ailleurs passer la majorité de son temps à attendre.
L’idée est que sur une période de clignotement, le µC n’a réellement que deux choses à faire :

allumer la led et éteindre la led

On va alors utiliser les interruptions pour gérer le clignotement de la led.
De cette manière, on va pouvoir utiliser le programme principal pour autre chose et gérer le clignotement de la led par interruption. Le programme principal va alors s’interrompre régulièrement pour aller changer l’état de la led.

Il peut il y avoir plusieurs sources d’interruptions (interne, timer, PWM, UART,…).

Une bonne gestion des interruptions est indispensable afin de pouvoir profiter de toute la puissance que peut apporter un microcontrôleur.

Nous étudierons les interruptions plus en détail à l’aide d’exercices et de manipulations.

Chapitre 17 : Qualité d'un programme

Pour concevoir un bon programme il faut qu'il soit concis (nécessite une bonne maîtrise des outils), qu'il soit efficace (bon choix des types de variables,…) et qu'il soit lisible. La lisibilité d'un programme est un élément très important ! En effet, le fait de concevoir un programme lisible (donc clair) apporte naturellement moins d'erreurs.
Pour augmenter la lisibilité d'un programme il faut bien choisir ses noms de variables et de fonctions mais aussi enrichir le programme de commentaires et mettre en évidence la structure du programme à l'aide d'indentations.

1. Les commentaires

Les commentaires sont généralement des explications destinées à mieux présenter le programme pour le rendre plus compréhensible. Mais les commentaires peuvent apporter d'autres informations, on pourrait imaginer une en-tête avec le nom du programme, la date, la version du programme,…

En C, les commentaires peuvent se trouver sur une seule ligne, ils sont alors précédés de « // ». Le compilateur ne prend alors pas en compte tout ce qui suit « // ».

Exemples de commentaires sur une ligne :

- *nb_en_cours* = (*nb_en_cours+1*) ; //incrémentation de *nb_en_cours*
- delay_ms(200); //millisecondes
- RB0 = 1; //allume la 1ère led
- while (BP1) ; //attend le relâchement du bouton-poussoir 1
- //initialisation
  init() ;
  menu=menu_principal ;
- while(1)
  {
  …
  } //fin de la boucle principale

Parfois il est utile de mettre des commentaires sur plusieurs lignes, dans ce cas l'ensemble des lignes de commentaires se trouvent entre « /* » et « */ ».

Exemples de commentaires sur plusieurs lignes :

- /*--*
  * *Ceci est l'entête* *
  * *programme 1 - 3/1/2008* *
  * *version 1.0* *
  --/
  void main(void) {...

- ```
  /*    ci-dessous affichage du menu principal à l'écran
        initialisation écran
        effacement écran
        positionnement curseur
        initialisation du tableau de caractère du menu en cours à afficher
  */
  lcd_init() ;
  …
  ```

Les commentaires sont très utiles pour permettre à quelqu'un de se plonger dans votre programme ou alors à vous-même de vous y replonger longtemps après (et que vous avez oublié le fonctionnement). De plus, les commentaires facilitent la réutilisation de parties de programmes (comme le programme est plus clair la recherche est plus facile).

2. Les indentations

Indenter signifie décaler vers la droite, on fait cela en programmation pour faire ressortir la structure, pour rendre le programme plus clair.

Une bonne indentation rend le programme plus clair, plus lisible. Il est donc indispensable d'utiliser une indentation pour faire ressortir les différents blocs d'actions qui se rapportent aux différentes structures (structure de choix simple, structure de choix multiple et structures répétitives).

Les décalages à droites se font en général avec la touche « tab ».

Exemples d'indentations :

- **Exemple 1 : indentation pour un choix simple**

Nous avons vu par exemple la structure « si condition alors faire bloc d'actions » mais le bloc d'actions (qui contient en général plusieurs lignes) doit pouvoir être facilement identifié ; pour ce faire on indente toutes les lignes du bloc d'actions comme ci-dessous :

```
if(expression)
        {
        //ligne d'instruction 1//
        //ligne d'instruction 2//
        }
//ligne d'instruction 3//
```

On peut alors facilement identifier les lignes qui se rapportent à la condition (les lignes d'instruction 1 et 2) car celles-ci sont endentées. On voit également que la ligne d'instruction 3 sera exécutée sans condition, elle est en dehors du bloc d'actions de la structure de choix.
```

- **Exemple 2 : indentation pour un choix simple (une seule ligne d'action)**

```
if(expression)
        //ligne d'instruction 1//
//ligne d'instruction 2//
```

Les « { } » ne sont pas nécessaires si le bloc d'actions ne contient qu'une seule ligne. Ci-dessus l'indentation met en évidence la ligne à exécuter lorsque l'expression est vraie.

- **Exemple 3 : indentation pour un choix simple (avec un « sinon »)**

```
if(expression)
        {
        //ligne d'instruction 1//
        //ligne d'instruction 2//
        }
else
        {
        //ligne d'instruction 3//
        //ligne d'instruction 4//
        }
//ligne d'instruction 5//
```

Ici on voit bien se détacher les deux blocs d'actions qui correspondent au *if()* et au *else*. Par contre, la ligne d'instruction 5 ne fait pas partie de ces deux blocs, son exécution ne dépend pas de la condition.

- **Exemple 4 : indentation pour une boucle infinie**

```
while (1)
        {
        //ligne d'instruction 1//
        //ligne d'instruction 2//
        …
        }
```

- **Exemple 5 : indentation pour une boucle POUR**

```
for (i=1 ;i<=n ;i++)
        {
        nb_en_cours   =       (nb_en_cours+1) ;
        somme         =       (somme + nb_en_cours) ;
        }
//ligne d'instruction 1000//
```

On voit directement le bloc qui sera répété n fois. La ligne d'instruction 1000 qui suit n'est par contre pas répétée, elle ne sera exécutée qu'après être sorti de la boucle.

Chapitre 18 : Programmer sa carte info

La programmation du PIC disposé sur la carte info doit tenir compte de la configuration matérielle et de l'usage que l'on a fait des PORTS du PIC.
Le PIC18F 2520/2620 dispose des PORTA, PORTB et PORTC. Chacun de ces PORTS dispose de 8 entrées /sorties.
Au début du programme, il faut indiquer au PIC comment on utilise chacune de ces entrées/sorties.

1. Le registre TRIS

L'affectation d'un registre TRIS permet de configurer les bits d'un port en entrée ou en sortie (« 0 » indique un usage en sortie et « 1 » indique un usage en entrée).

Lors de la conception de la carte info, un choix a été fait :
Les PORTB et PORTC en sortie et le PORTA en entrée (à l'exception de RA4 en sortie).

Il faut donc inclure au début de chacun de vos programmes les instructions :

```
TRISA=0b11101111 ;      //Tout le portA en entrée sauf RA4
TRISB=0b00000000 ;      //Tout le portB en sortie
TRISC=0b00000000 ;      //Tout le portC en sortie
```

2 Le registre ADCON1

Le PIC dispose de registres internes qui nous permettent de le configurer, le registre ADCON1[*] est lié à la configuration des entrées en mode analogique ou numérique.

Pour ne travailler, avec la carte info, qu'avec des entrées numériques, nous allons inclure au sein du programme l'instruction :

```
ADCON1=0b00001111 ;     ou      ADCON1= 0x0F ;
```

3. Le PORTB

Une LED a été connectée sur chaque sortie du PORTB.

On peut considérer le PORTB dans son ensemble ou chaque sortie de manière individuelle. On peut alors affecter directement « PORTB » pour agir sur les 8 leds d'un coup ou alors on peut agir individuellement sur chaque led. Pour allumer une led (led0 à led7), il suffit de mettre le bit du PORTB correspondant à 1 et pour l'éteindre à 0.

[*] Pour plus de détails sur ce registre de configuration, vous pouvez vous référer au « datasheet » du PIC.

Exemples :

Pour allumer la led 3 on écrira : RB3=1 ;

Pour éteindre tout le PORTB on écrira : PORTB=0b00000000 ; ou PORTB=0 ;

Pour allumer tout le PORTB on écrira : PORTB=0b11111111 ; ou PORTB=0xFF ;

Il est préférable d'inclure au début de votre programme l'instruction : PORTB=0 ;

4. La directive #include

La directive #include permet d'ajouter à votre programme des fichiers d'en-tête, il faudra par exemple inclure au début du programme la ligne : #include <pic18.h>

De cette manière votre programme peut accéder aux lignes qui se trouvent dans la librairie <pic18.h> et utiliser les informations qui s'y trouvent. Ces informations sont en rapport avec le PIC utilisé, il y a par exemple l'adresse du registre interne ADCON1, ce qui est indispensable pour la configuration des entrées analogiques.

5. La directive #define

La directive #define joue le rôle de traducteur ou de fonction copier/coller.
Nous utiliserons cette fonctionnalité pour rendre le programme plus lisible.

On décidera alors d'inclure au sein du programme des lignes telles que :

```
#define    BP1    RA0
#define    LED0   RB0
```

La 1ère ligne permet d'appeler le bouton poussoir 1 BP1 plutôt que RA0. Cette ligne indique au compilateur qu'à chaque fois que l'on écrit BP1 il doit le remplacer par RA0 ; RA0 correspond à la broche 0 du PORTA, celle où est connectée le bouton poussoir 1.

6. La boucle infinie

Pour éviter de recommencer à chaque fois tout le début du programme on utilise une astuce qui consiste à créer une boucle infinie.

```
while (1)
{

}
```

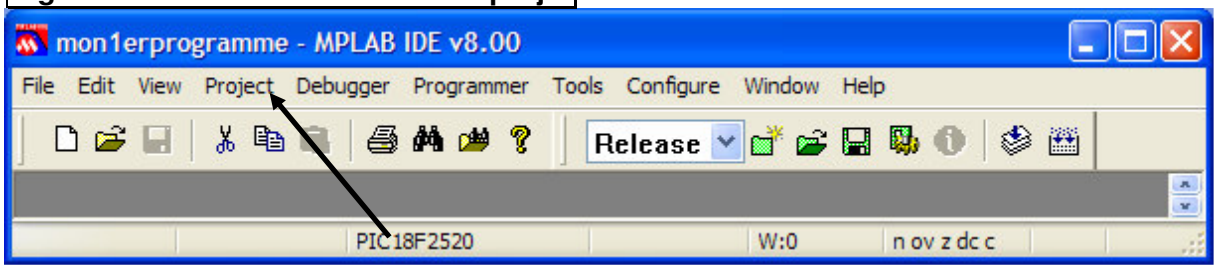
On met alors tout ce qui ne devra être fait qu'une seule fois en dehors de la boucle, juste avant la boucle (l'initialisation par exemple). Par contre tout ce qui est susceptible de se répéter se trouvera au sein de la boucle infinie.

7. Construction d'un projet

La construction d'un projet se fait au sein d'un environnement de développement (qui permet d'écrire le programme) qui a été rattaché à un compilateur (qui transforme votre programme pour être compris par le PIC).

Démarrons le programme MPLAB IDE. On voit que la barre d'outil comporte 10 possibilités. Sélectionnons ensuite, comme montré ci-dessous l'option « Project ».

Figure 27 : MPLAB - Démarrer un projet



Choisissons ensuite « Project Wizard ».

Un écran de bienvenue apparaît : **Welcome !** Sélectionnons « Suivant » :

L'écran **Step One** apparaît, il permet de sélectionner le PIC à utiliser.
A l'aide de l'ascenseur, sélectionnons **PIC18F2620**. Passons ensuite à l'étape suivante :

L'écran **Step Two** apparaît, il permet de sélectionner le langage de programmation.
A l'aide de l'ascenseur, sélectionnons « HI-TECH PICC-18 Toolsuite »
Que l'on trouve dans le répertoire C:\HTSOFT\PIC18\BIN\PICC18.EXE

Il faut valider les **toolsuite Contents** :
PICC-18 Global
PICC-18 Compiler (picc18.exe)
PICC-18 Assembler (picc18.exe)
PICC-18 Linker (picc18.exe)

Passons ensuite à l'étape suivante :

L'écran **Step Three** apparaît, il permet de créer le nouveau projet.
A l'aide de l'étiquette **Browse** il est possible de choisir le répertoire.
Créons ou choisissons un répertoire intitulé **mesprogramme-nom,** et dans la case nom de fichier indiquons **mon1erprogramme**.
Ensuite **enregistrer** et passons à l'étape **suivante**.

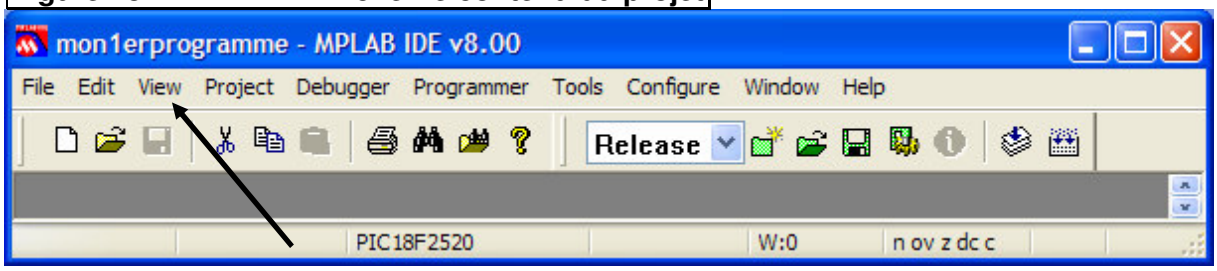
L'écran **Step Four** apparaît, passons à l'étape **suivante**.

L'écran **Summary** apparaît, il donne un résumé des choix effectués.
Sélectionnons **Terminer**
L'écran **Save Workspace As** apparaît en indiquant comme nom de fichier mon1erprogramme.mcw, il faut ensuite **Enregistrer.**

Normalement une nouvelle fenêtre doit apparaître à gauche, il s'agit de l'affichage des éléments constituant le projet, intitulée mon1erprogramme.

Si celle-ci n'est pas apparue il faut faire comme indiqué ci-dessous :

Figure 28 : MPLAB – Afficher le contenu du projet



Dans la barre d'outils sélectionnez « View » et ensuite « Project ».
Nous pouvons maintenant commencer un premier programme.

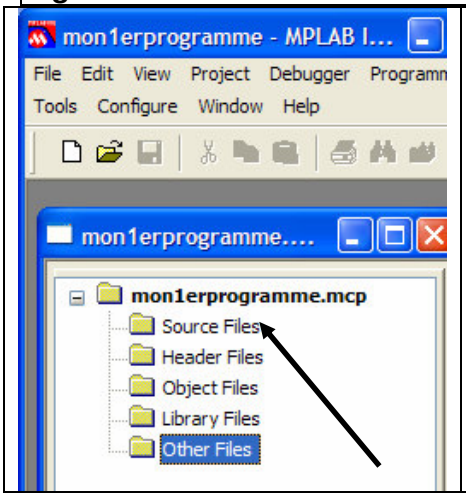
Sélectionnons File → New :
Une nouvelle fenêtre de travail MPLAB IDE Editor apparaît dans laquelle on peut commencer à écrire le programme.

Tapez le titre : « mon 1er Programme » puis sélectionnez « Save ».

Choisissez le répertoire mon1erprogramme.
Donnez un nom au fichier « mon1erprogramme.c ».
Enregistrez.

Dans la fenêtre projet :

Figure 29 : MPLAB – Inclure votre programme dans votre projet



Sélectionnez :

« Source Files »

Cliquez sur le bouton droit de la souris
Sélectionnez « Add Files »

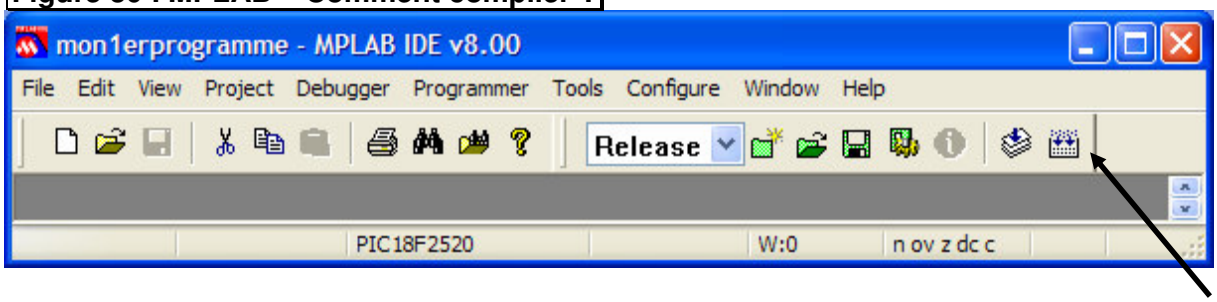
Choisissez mon1erprogramme.c

Ouvrez

mon1erprogramme.c apparaît dans la rubrique "Source Files"

Dans la fenêtre MPLAB IDE Editor, vous pouvez taper votre 1er programme, un exemple de programme de base est donné en annexe (Annexe 3, p.66).

Figure 30 : MPLAB – Comment compiler ?



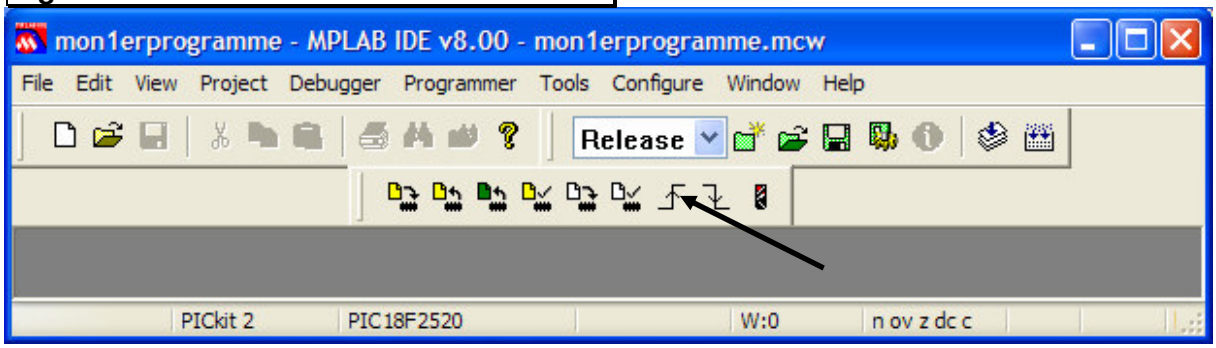
Sélectionnez l'icône qui correspond à « Build All », vous compilez ainsi pour la première fois votre programme.

Vous devez voir apparaître la fenêtre intitulée « output », et obtenir le message final BUILD SUCCEEDED.

Après avoir connecté le programmateur PICKIT2 et votre carte info à celui-ci, choisissez Programmer → Select Programme → PICKIT2.

Envoyez ensuite le programme en choisissant Programmer → Program.
Il vous reste enfin à relâcher le reset de votre carte info en sélectionnant, comme ci-dessous, « Bring Target MCLR to VDD ».

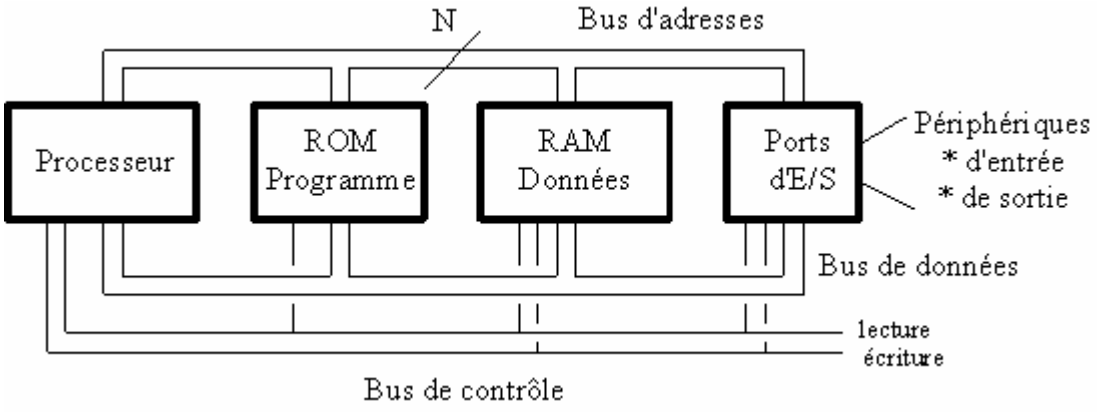
Figure 31 : MPLAB – Relâchement du reset



Quelques instants après, vos 8 leds doivent s'allumer ☺ …

Annexe 1 : Structure simplifiée d'un microcontrôleur

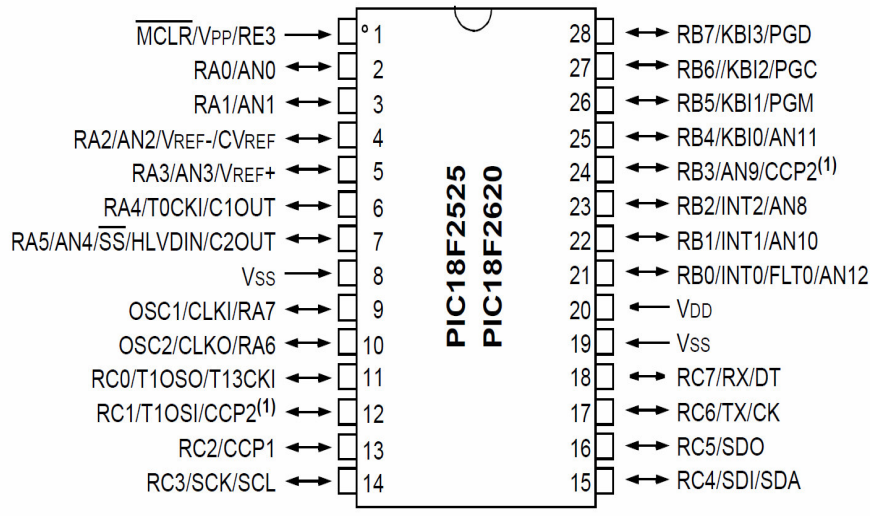
Structure simplifiée d'un microcontrôleur



http://fr.wikipedia.org

Annexe 2 : Brochage du PIC18F2620

Brochage du PIC18F2620



Annexe 3 : Mon 1er programme

MPLAB – Mon 1er programme

```
// mon 1er programme

#include <pic18.h>

__CONFIG(1, HS&IESODIS&FCMDIS);
__CONFIG(2, BORDIS&PWRTEN&WDTDIS );
__CONFIG(3, PBANDIS);              //mettre en commentaire si PIC18F2520
__CONFIG(4, LVPDIS&XINSTDIS&STVRDIS&LPT1DIS);
__CONFIG(5, UNPROTECT);
__CONFIG(6, WRTEN);                //mettre en commentaire si PIC18F2520

#define BP1 RA0
#define BP2 RA1
#define BP3 RA2
#define BP4 RA3

#define LED0 RB0
#define LED1 RB1
#define LED2 RB2
#define LED3 RB3
#define LED4 RB4
#define LED5 RB5
#define LED6 RB6
#define LED7 RB7

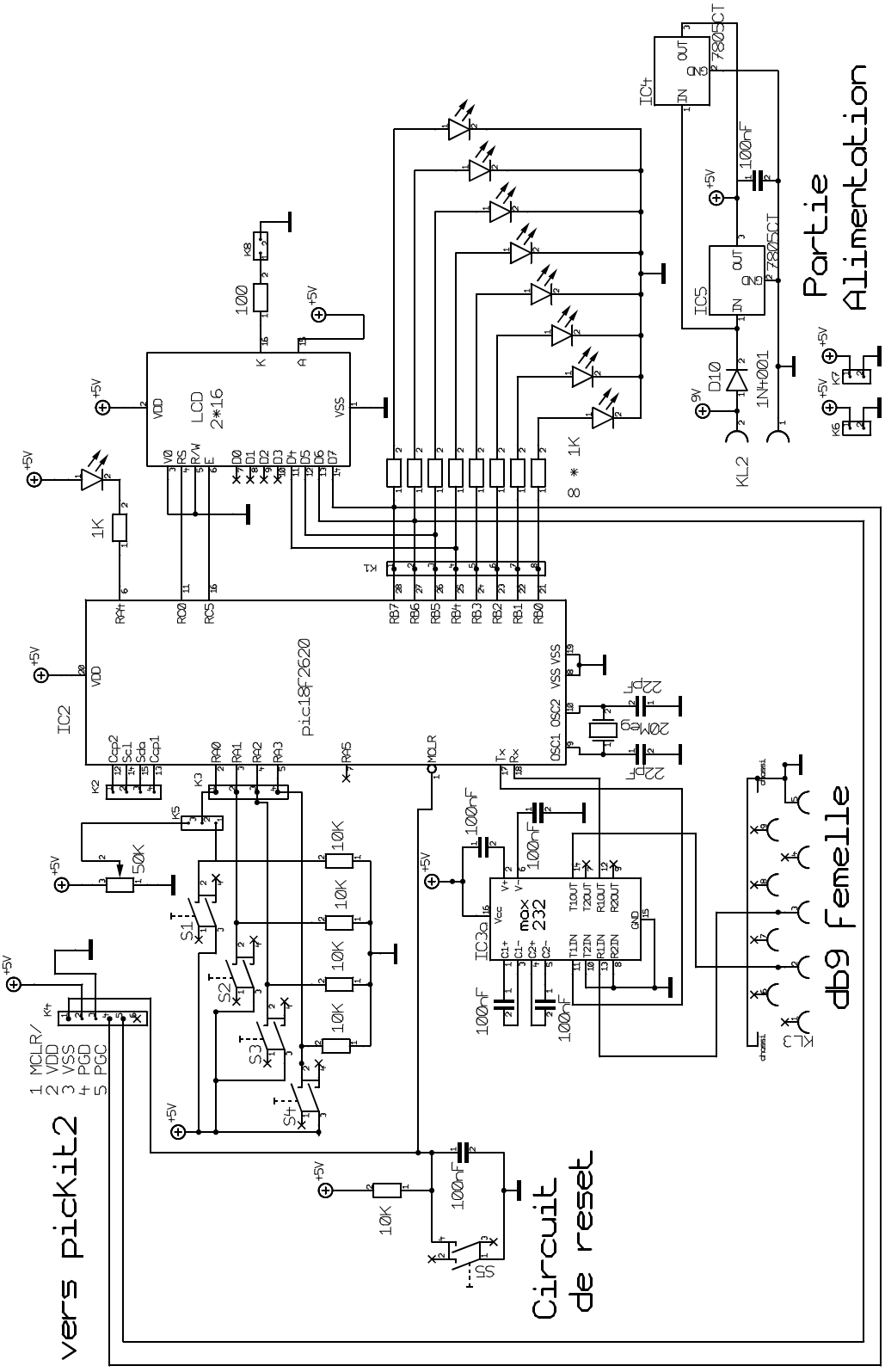
void main ()

{
// Configuration des registres du PIC en fonction de la carte info
 ADCON1=0x0F;
 TRISA=0b11101111;
 TRISB=0b00000000;
 TRISC=0b00000000;

        while(1) //boucle infinie
        {
        PORTB =255 ;
        }
}
```

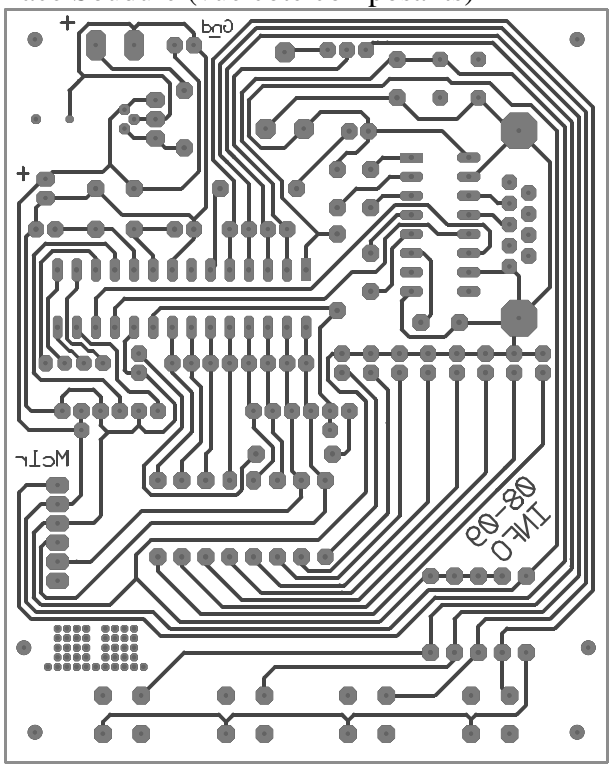
Ce programme commence par l'initialisation puis on va entrer dans une boucle infinie au sein de laquelle il va il y avoir répétition d'une instruction : PORTB =255 ;

Annexe 4 : Schéma de principe de la carte info

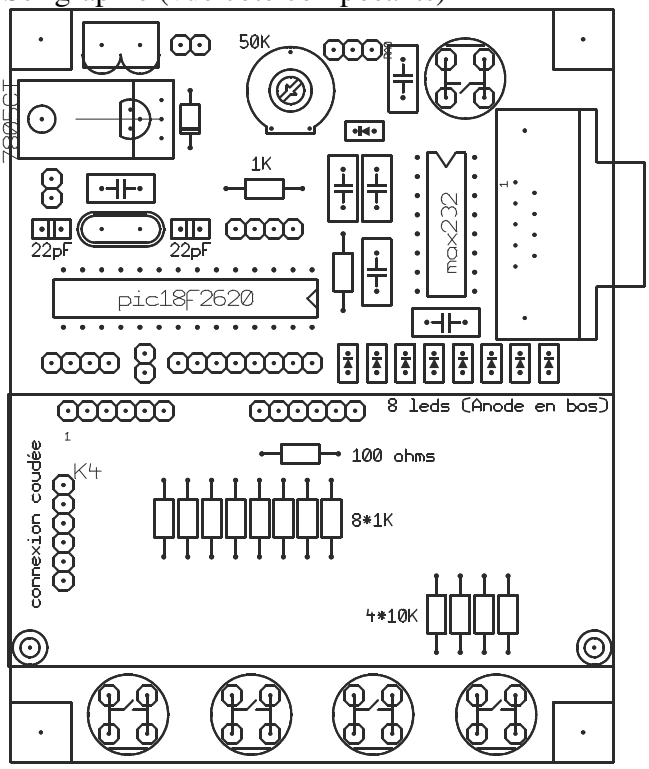


Annexe 5 : PCB de la carte info (version 2009)

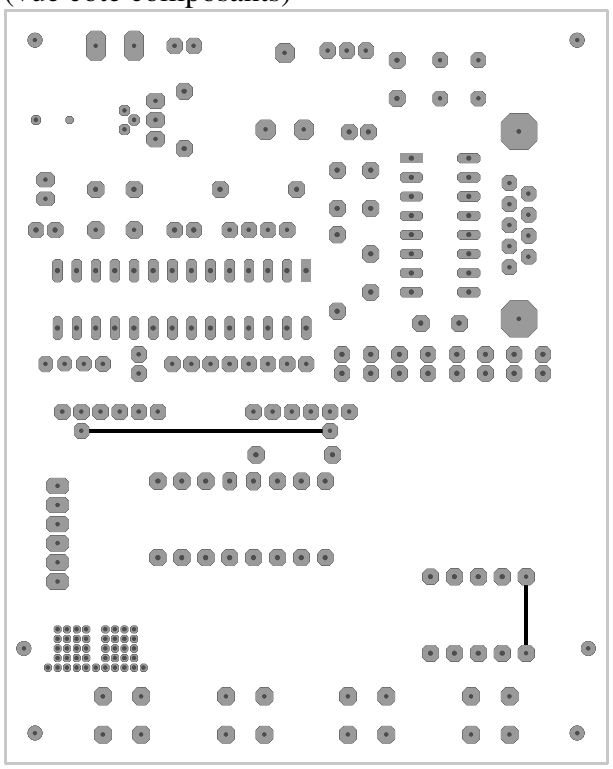
Face Soudure (vue coté composants)



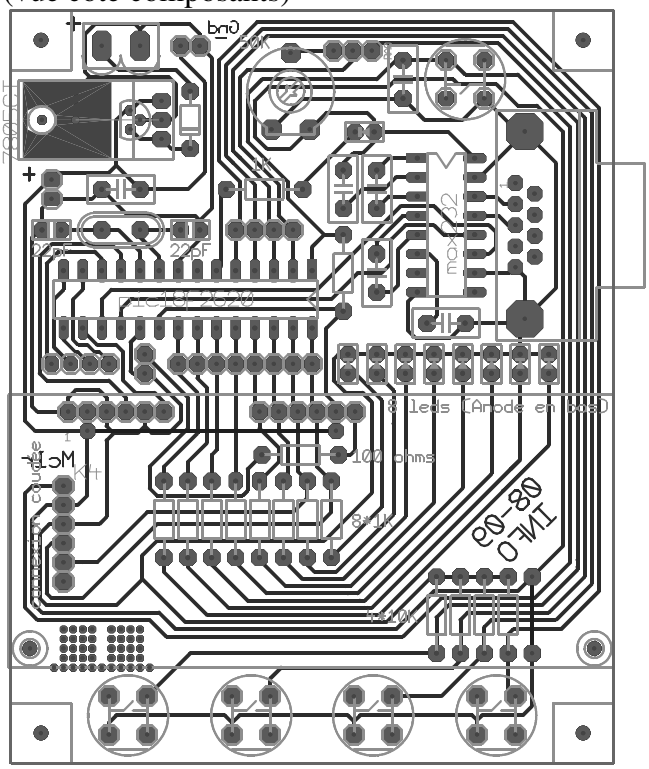
Sérigraphie (vue coté composants)



Emplacement des « straps »
(vue coté composants)



Face Soudure + sérigraphie + « straps »
(vue coté composants)



Bibliographie et liens internet

M. Deldime, Syllabus d'informatique appliquée, INRACI, 2005-2006
M. Mus et M. Pochet, Syllabus d'informatique appliquée, INRACI, 2008-2009
M. Mazzeo, M. Mus et M. Pochet, Syllabus d'informatique embarquée, INRACI, 2009-2010
Claude Delannoy, *Langage C*, Paris, Eyrolles, 1999
Claude Delannoy, Programmer en l*angage C*, Paris, Eyrolles, 2007

http://clubelek.insa-lyon.fr/joomla/fr/base_de_connaissances/electronique
http://ducry.info-compta.ch/Ordinogramme.pdf
http://fr.wikipedia.org/wiki/Algèbre_de_Boole_(logique)

Table des figures

Figure 1 : Exemple de système embarqué, l'ABS 4
Figure 2 : Schéma bloc pour l'affichage de la température et de la pression 6
Figure 3 : Schéma bloc d'un Tétris de poche 6
Figure 4: Schéma bloc de la carte info..... 7
Figure 5 : Les différentes entités de l'ordinogramme..... 12
Figure 6 : Ordinogramme du calcul de la moyenne de trois nombres 15
Figure 7 : Les opérateurs arithmétiques en C 16
Figure 8 : Les opérateurs de comparaison en C..... 17
Figure 9 : Les opérateurs logiques en C..... 17
Figure 10 : Séquence en C - Calcul de la moyenne de trois nombres 17
Figure 11 : Structure de la « tant que »..... 20
Figure 12 : Ordinogramme et programme en C manipulant une boucle infinie..... 21
Figure 13 : Exemple du chronomètre..... 22
Figure 14 : Exemple du carré d'un nombre 23
Figure 15 : Exemple de la moyenne de deux nombres 24
Figure 16 : Structure de la «répéter… tant que »..... 25
Figure 17 : Faire une «répéter… tant que » avec une tant que 26
Figure 18 : Exemple de «tant que » avec un bloc d'actions vide 27
Figure 19 : Exemple de «répéter tant que » avec un bloc d'actions vide 27
Figure 20 : Structure de la «pour » et syntaxe en C..... 28
Figure 21 : Structure d'une «pour » et structure d'une « tant que » 29
Figure 22 : Lien entre le nombre de bits et les nombres positifs représentables 30
Figure 23 : Interprétation des nombres positifs et négatifs..... 31
Figure 24 : Les différents types en C 31
Figure 25 : Ordinogramme de la structure de choix multiple avec action par défaut..... 42
Figure 26 : Exemple de transformation ordinogramme → LDA..... 48
Figure 27 : MPLAB - Démarrer un projet 62
Figure 28 : MPLAB – Afficher le contenu du projet..... 63
Figure 29 : MPLAB – Inclure votre programme dans votre projet 63
Figure 30 : MPLAB – Comment compiler ? 64
Figure 31 : MPLAB – Relâchement du reset..... 64

Questions de révision

Questions sur le chapitre 1 : L'informatique embarquée

1.1) L'informatique embarquée est l'informatique destinée à……

1.2) Citez des exemples de systèmes embarqués.

1.3) Citez les éléments qui doivent se trouver au sein d'un système embarqué.

1.4) Qu'est-ce qui différencie un microprocesseur d'un microcontrôleur ?

1.5) Que signifie PIC ?

1.6) De quoi est constitué le PIC18F2520/2620 ?

1.7) Expliquez le rôle général du microcontrôleur représenté au sein du schéma bloc d'un système embarqué.

1.8) Pourquoi se pose-t-on les questions :
Que doit faire le système ?
De quoi le microcontrôleur a-t-il besoin pour faire ce qu'il doit faire ?

1.9) Exposez les schémas bloc des dispositifs suivants :
- Un système d'affichage de la température et de la pression
- Un tétris de poche
- Un projet utilisant la carte info

1.10) Qu'est-ce qui influence la complexité du programme ?

Questions sur le chapitre 2 : La base de la programmation

2.1) Qu'est-ce qu'une variable ?

2.2) Quelles sont les trois grandes familles de variables ? Expliquez et donnez un exemple.

2.3) Quelles sont les valeurs que l'on associe à la logique booléenne (au vrai et au faux) ?

2.4) Quelles sont les différentes catégories d'opérateurs ? Expliquez.

2.5) A quoi sert la structure de choix ?

2.6) A quoi sert la structure répétitive ? Détaillez un peu.

2.7) Expliquez brièvement ce qu'est le type d'une variable.

2.8) Qu'est-ce qu'une affectation ?

2.9) Quel est le symbole d'affectation en LDA (qui est le même pour les ordinogrammes) ?

2.10) Quel est le symbole de l'affectation en C ?

2.11) Mis à part le symbole d'affectation, quelle est la petite différence entre une affectation en C et en LDA ?

2.12) Quels sont les opérateurs arithmétiques ? Donnez un exemple d'utilisation.

2.13) Donnez un exemple d'opérateur de comparaison.

2.14) De quel type est le résultat d'une opération qui contient un opérateur de comparaison ?

2.15) Donnez un exemple d'opérateur logique (en LDA et en C).

2.16) De quel type est le résultat d'une opération qui contient un opérateur logique ?

Questions sur le chapitre 3 : La séquence

3.1) Quels sont les cinq piliers de la séquence ?

3.2) Quelles sont les différentes entités de l'ordinogramme ?

3.3) Par quelle entité un ordinogramme doit toujours commencer ?

3.4) Par quelle entité un ordinogramme doit toujours terminer ?

3.5) A quoi ressemble l'entité de l'ordinogramme qui permet de faire un test ? Expliquez.

3.6) Citez différentes formes d'actions qui pourront être placées au sein d'un rectangle.

3.7) Qu'est-ce qu'une variable est destinée à contenir ?

3.8) On dit qu'une variable a …… et est associée à……

3.9) Quelles sont les 3 catégories de types et les mots clés associés ?

3.10) Qu'est-ce que la déclaration d'une variable ?

3.11) Quelle est la syntaxe pour la déclaration d'une variable au sein d'ordinogramme ?

3.12) Donnez un exemple de déclaration pour chaque catégorie de type.

3.13) Qu'est ce que l'instruction de lecture ? Quel mot clé utiliser ?

3.14) Qu'est-ce qu'un affectation ? Quel symbole utilise-t-on ?

3.15) Qu'est-ce qu'une opération ?

3.16) Quelles sont les différentes catégories d'opérateurs ?

3.17) Qu'est ce que l'instruction d'écriture? Quel mot clé utiliser ?

3.18) Quelle est la syntaxe en C pour déclarer une variable de la « famille des entiers » ?

3.19) Quelle est la syntaxe en C pour déclarer une variable de la « famille des réels » ?

3.20) Quelle est la syntaxe en C pour déclarer une variable de type booléen ?

3.21) Expliquez ce qui différencie la lecture en LDA (ou ordinogrammes) et la lecture en C ?

3.22) Quelle est la syntaxe de l'affectation en C ? Donnez un exemple.

3.23) Quels sont les opérateurs arithmétiques en C et en LDA ? Expliquez le principe.

3.24) Quels sont les opérateurs de comparaison en C et en LDA ? Expliquez le principe.

3.25) Quels sont les opérateurs logiques en C et en LDA ? Expliquez le principe.

3.26) Expliquez ce qui différencie l'écriture en LDA (ou ordinogrammes) et la l'écriture en C ?

3.27) Donnez un exemple de séquence en C et son équivalent en ordinogramme.

Questions sur le chapitre 4 : La structure de choix simple

4.1) Quel est le principe général de la structure de choix simple ?

4.2) Qu'est-ce qu'une condition?

4.3) Comment faire apparaître une structure de choix simple à l'aide d'entités d'ordinogramme ?

4.4) Quel est le principe général du « Si-Alors » ?

4.5) Comment représenter le « Si-Alors » à l'aide d'entités d'ordinogramme ?

4.6) Quelle est la syntaxe du « Si-Alors » en C ?

4.7) Citez un exemple d'utilisation de « Si-Alors » en ordinogramme et son équivalent en C.

4.8) Quel est le principe général du « Si-Alors-Sinon » ?

4.9) Comment représenter le « Si-Alors-Sinon » à l'aide d'entités d'ordinogramme ?

4.10) Quelle est la syntaxe du « Si-Alors-Sinon » en C ?

4.11) Citez un exemple d'utilisation de « Si-Alors-Sinon » en ordinogramme et son équivalent en C.

Questions sur le chapitre 5 : La "tant que"

5.1) Quelle est la phrase qui décrit le principe général de la « tant que » ?

5.2) Donnez la structure de la « tant que » à l'aide des entités propres aux ordinogrammes.

5.3) Donnez la syntaxe en C de la « tant que ».

5.4) Qu'est-ce qu'une boucle infinie ?

5.5) Pourquoi utilise-t-on une boucle infinie au sein du programme principal ?

5.6) Donnez la structure d'une boucle infinie à l'aide des entités propres aux ordinogrammes.

5.7) Donnez la syntaxe en C d'une boucle infinie.

5.8) Exposez la structure typique d'un programme à l'aide des entités propres aux ordinogrammes.

5.9) Donnez la structure typique d'un programme en C.

5.10) Que signifie le mot clé « main » en anglais ?

5.11) Que nous indique le mot clé « main » par rapport à l'exécution des instructions ?

5.12) A quoi peut-on rattacher l'entité « début » au sein du programme en C ?

5.13) A quoi peut-on rattacher l'entité « fin » au sein du programme en C ?

5.14) Que met-on avant le while(1) ?

5.15) Que met-on dans le while(1) ?

5.16) Quelle est la particularité de l'entité « fin » lorsqu'on utilise un while(1) ?

Questions sur le chapitre 6 : Exemples de programmes

6.1) Essayez d'exposer par vous-même l'ordinogramme d'un chrono de base.

6.2) Sachant que la syntaxe en C utilisée en classe est :

```
init() ;                    //pour initialiser la carte

lcd_goto(0x00) ;
sprintf(Ligne1, ″chrono : %3d s″, sec) ;
lcd_puts(Ligne1) ;          //pour afficher la valeur contenue par la variable sec

delay_ms(1000) ; //pour attendre une seconde
```

Basez-vous sur votre ordinogramme du chrono de base pour donner le programme en C correspondant.

6.3) Essayez d'exposer par vous-même l'ordinogramme du carré d'un nombre

6.4) En vous basant sur la syntaxe en C utilisée en classe et sur l'ordinogramme exposé à la question précédente, donnez le programme en C du carré d'un nombre.

6.5) Essayez d'exposer par vous-même l'ordinogramme du calcul de la moyenne de 2 nombres.

6.6) En vous basant sur la syntaxe en C utilisée en classe et sur l'ordinogramme exposé à la question précédente, donnez le programme en C de la moyenne de 2 nombres.

Questions sur le chapitre 7 : La "repeater…tant que"

7.1) Quelle est la phrase qui décrit le principe général de la « répéter…tant que » ?

7.2) Donnez la structure de la « répéter…tant que » à l'aide des entités propres aux ordinogrammes.

7.3) Donnez la syntaxe en C de la « répéter…tant que ».

7.4) Qu'est-ce qui différencie la « répéter…tant que » de la « tant que » ?

7.5) Pourquoi préfère-t-on utiliser une « répéter…tant que » plutôt qu'une « tant que » lorsque le bloc d'actions doit être exécuté au moins une fois (alors qu'il est possible d'utiliser une « tant que ») ?

7.6) Exposez à l'aide des entités propres aux ordinogrammes la manière de réaliser une « répéter...tant que » avec une « tant que ».

7.7) Exposez la syntaxe en C qui permet de faire une « répéter...tant que » à l'aide d'une « tant que ».

7.8) Exposez les deux formes de représentations (à l'aide des entités propres aux ordinogrammes) d'une « tant que » avec le bloc d'actions qui est vide.

7.9) Exposez la syntaxe en C, utilisant la « tant que », qui permet d'attendre le relâchement de BP1.

7.10) Exposez les deux formes de représentations (à l'aide des entités propres aux ordinogrammes) d'une « répéter…tant que » avec le bloc d'actions qui est vide.

7.11) Exposez la syntaxe en C, utilisant la « répéter…tant que », qui permet d'attendre le relâchement de BP1.

7.12) Que peut-on remarquer lorsqu'on compare la « tant que » et la « répéter...tant que » lorsqu'on a un bloc d'actions qui est vide ?

7.13) Dans le cas de l'attente qu'une condition soit validée on préfère souvent utiliser la « répéter...tant que » plutôt que la « tant que ». Pourquoi ?

7.14) Quelle est la syntaxe en C de l'écriture abrégée de la « répéter...tant que » (avec un bloc d'actions vide)?

Questions sur le chapitre 8 : La boucle "pour"

8.1) Dans quel cas préfère-t-on la boucle « pour » aux autres structures répétitives ?

8.2) Que doit utiliser (en interne) la « pour » afin de gérer le nombre de répétitions ?

8.3) Quel nom de variable utilise-t-on habituellement comme compteur au sein de la boucle « pour » ?

8.4) Expliquez le principe général de la boucle « pour ».

8.5) Donnez la structure de la « pour » à l'aide des entités propres aux ordinogrammes.

8.6) Donnez la syntaxe en C de la « pour ».

8.7) Donnez la syntaxe en C qui permet de gérer 10 clignotements de la LED1.

8.8) Pourquoi dans certains cas on préfère utiliser une « pour » plutôt qu'une « tant que » (alors qu'il est possible d'utiliser une « tant que ») ?

8.9) Exposez la structure de la « pour » puis montrez qu'elle permet de mettre en évidence la structure d'une « tant que »

Questions sur le chapitre 9 : Les types en C

9.1) A quoi rattache-t-on un type en C ?

9.2) Combien il y a-t-il de combinaisons binaires différentes avec 1 bit ? Lesquelles ?

9.3) Combien il y a-t-il de combinaisons binaires différentes avec 2 bits ? Lesquelles ?

9.4) Combien il y a-t-il de combinaisons binaires différentes avec 3 bits ? Lesquelles ?

9.5) Combien il y a-t-il de combinaisons binaires différentes avec n bits ?

9.6) Lorsqu'on ajoute 1 bit on dit ……le nombre de combinaisons possibles. Expliquez.

9.7) Lorsqu'on représente les nombres positifs, quel est l'ensemble des nombres représentables avec n bits ?

9.8) On dit que le type influence l'interprétation des motifs binaires, expliquez.

9.9) Quelle est la convention habituellement utilisée pour la représentation des nombres négatifs ?

9.10) Exposez un tableau qui permet de mettre en évidence l'interprétation des nombres positifs et négatifs pour des motifs binaires de 8 bits (des octets).

9.11) En résumé on pourra dire que le type influence : ……

9.12) Quelles sont les trois catégories de types qui ont été mentionnées au cours ?

9.13) Donnez deux types différents en C qui appartiennent pourtant à la même catégorie de type telle que mentionné au début du cours. Qu'est-ce qui différencie ces types ?

9.14) On dit qu'il convient d'utiliser le type approprié. Expliquez.

9.15) Exposez un tableau qui reprend les différents types en C, le tableau est composé de quatre colonnes : La syntaxe, la description, la place occupée en RAM (compilateur PICC18) et la description résumée.

Questions sur le chapitre 10 : Les opérateurs logiques

10.1) Qu'est-ce que l'algèbre de Boole ?

10.2) On dit que toute fonction combinatoire correspond à…… appelée……

10.3) De quoi est constituée toute expression booléenne ?

10.4) Contrairement à ce qu'on voit au cours de mathématique, les variables et les constantes ont une particularité, laquelle ?

10.5) De quelle manière représente-t-on souvent une fonction combinatoire ? Expliquez.

10.6) Citez les particularités vues au cours sur l'algèbre de Boole.

10.7) Quelles sont les fonctions combinatoires élémentaires ?

10.8) A quelle opération correspond l'opérateur logique ET ?

10.9) A quelle opération correspond l'opérateur logique OU ?

10.10) Comment représente-t-on la fonction NON ?

10.11) Donnez la table de vérité de la fonction ET.

10.12) Donnez la table de vérité de la fonction OU.

10.13) Donnez la table de vérité de la fonction NON.

10.14) Donnez les théorèmes de De Morgan.

10.15) On peut dire que les théorèmes de De Morgan traduisent : ……

10.16) Un entier peut également contenir une information de type VRAI/FAUX, détaillez.

10.17) Donnez la syntaxe en C du ET logique.

10.18) Donnez la syntaxe en C du OU logique.

10.19) Donnez la syntaxe en C du NON logique.

10.20) Donnez un exemple en C d'utilisation d'opérateurs logiques sur entiers.

10.21) Donnez la syntaxe en C du ET bit à bit.

10.22) Donnez la syntaxe en C du OU bit à bit.

10.23) Illustrez le ET bit à bit à l'aide d'un exemple.

10.24) Illustrez le OU bit à bit à l'aide d'un exemple.

Questions sur le chapitre 11 : Applications des opérateurs

11.1) Quelle méthode utilise-t-on pour tester si un bit particulier d'une variable est à 0 ou à 1 ?

11.2) Exposez une structure qui permet de mettre en évidence le fait de faire une action lorsque le bit d'indice 4 de *motif* est à 1 et une autre action lorsque le bit d'indice 4 de *motif* est à 0.

11.3) Exposez une structure qui permet de mettre en évidence le fait de faire une action lorsque le bit d'indice 1 de *motif* est à 0 et une autre action lorsque le bit d'indice 1 de *motif* est à 1.

11.4) Quel opérateur doit-on utiliser pour forcer à la valeur 1 certains bits d'une variable ?

11.5) Quel opérateur doit-on utiliser pour forcer à la valeur 0 certains bits d'une variable ?

11.6) Donnez la syntaxe en C qui permet d'affecter la valeur 1 aux bits 0 et 1 de TRISA en prenant soin de ne pas modifier l'état des autres bits.

11.7) Les opérateurs de décalage font partie d'une catégorie d'opérateurs, laquelle ?

11.8) Sachant que op1 correspond à 0b00110011, que contiendra op3 si on écrit : op3=op1>>1 ;

11.9) Sachant que op2 correspond à 0b11110000, que contiendra op3 si on écrit : op3=op2<<2 ;

11.10) Citez deux exemples d'application qui utilisent les opérateurs de décalage.

Questions sur le chapitre 12 : Les fonctions : notions de base

12.1) Que permet de contenir une fonction ?

12.2) Une fonction a …… et permet d'effectuer ……

12.3) Dans certains cas il est préférable d'utiliser une fonction plutôt que d'écrire les lignes correspondantes dans le programme principal, pourquoi ?

12.4) Citez les trois aspects liés à la fonction.

12.5) Donnez une autre appellation du prototype d'une fonction.

12.6) Donnez une autre appellation de l'implémentation d'une fonction.

12.7) En quoi consiste la déclaration d'une fonction ?

12.8) Dans quel fichier se trouve en général la déclaration d'une fonction ?

12.9) Donnez la syntaxe générale de la déclaration d'une fonction.

12.10) Donnez un exemple de déclaration d'une fonction.

12.11) Qu'est-ce que la définition d'une fonction ?

12.12) Quelles sont les deux parties qui constituent la définition d'une fonction ?

12.13) Illustrez à l'aide d'un exemple l'allure d'une définition de fonction.

12.14) Qu'entend-on par l'appel d'une fonction ?

12.15) Pour qu'une fonction puisse être appelée, il faut qu'elle ait été préalablement ……

12.16) Où se poursuivra l'exécution du programme au moment de l'appel d'une fonction ?

12.17) Quelles sont les deux manières de terminer les instructions propres à la fonction ?

12.18) Comment appelle-t-on la transmission des arguments au corps de la fonction lors de l'appel de cette fonction ?

12.19) Citez un exemple de fonction qui a besoin de recevoir une (des) information(s) supplémentaire(s) lors de son appel.

12.20) Comment appelle-t-on les données apportées à la fonction au moment de l'appel ?

12.21) Donnez un exemple de fonction qui n'a pas besoin de donnée supplémentaire au moment de l'appel.

12.22) Exposez un exemple de prototype de fonction avec argument.

12.23) Exposez un exemple de prototype de fonction sans argument.

12.24) Donnez un exemple d'appel de fonction avec argument.

12.25) Donnez un exemple d'appel de fonction sans argument.

12.26) Comment met-on en évidence, au niveau du prototype, qu'une fonction n'a pas de valeur de retour.

12.27) À quel endroit du prototype d'une fonction le type de la valeur de retour apparaît-il ?

12.28) Quel est le mot clé à utiliser au sein du corps de la définition de la fonction pour permettre à la fonction de fournir un résultat ?

12.29) Montrez à quoi ressemble le prototype d'une fonction destinée à renvoyer un nombre entier aléatoire.

12.30) Montrez à quoi ressemble l'appel d'une fonction destinée à renvoyer un nombre entier aléatoire.

12.31) Donnez le prototype, l'implémentation et une manière de faire appel à une fonction qui calcule la moyenne de deux nombres entiers.

12.32) Expliquez de quelle manière les données apportées à la fonction au moment de l'appel pourront être manipulées au sein du corps de la fonction.

12.33) Expliquez le principe de fonctionnement d'une affectation qui utilise la valeur de retour d'une fonction.

Questions sur le chapitre 13 : Les bits indicateurs d'état

13.1) Quelle est l'autre appellation (dans le jargon) d'un bit indicateur d'état ?

13.2) Qu'est ce qu'un bit indicateur d'état ?

13.3) Quelle est la traduction en français du mot « flag » ?

13.4) Que peut apporter l'utilisation d'un flag ?

13.5) Citez un exemple de « flag ».

13.6) Donnez un exemple de structure de programme qui manipule un flag.

13.7) L'utilisation d'un flag nécessite une bonne gestion, expliquez.

Questions sur le chapitre 14 : Les entiers indicateurs d'état

14.1) Quel est le point commun entre un bit indicateur d'état et un entier indicateur d'état ?

14.2) Qu'est ce qui différencie un bit indicateur d'état d'un entier indicateur d'état ?

14.3) Donnez un exemple d'entier indicateur d'état.

14.4) De la même manière que les « flags », les entiers indicateurs d'état doivent refléter à tout moment……

14.5) C'est au programmeur d'assurer une bonne …… des variables d'état.

14.6) Donnez trois exemples de noms de variables qui correspondent à des entiers indicateurs d'état.

14.7) Pour les trois exemples donnés ci-dessus, détaillez le rôle de chaque variable.

14.8) Que doit-on prendre en considération lorsqu'on choisit le nom d'une variable ?

14.9) Une variable telle que *nb_objets_stockes* évoque……

14.10) À quelle catégorie la variable *jeu_a_demarre* semble appartenir ? Expliquez.

14.11) À quelle catégorie la variable *menu_en_cours* semble appartenir ? Expliquez.

14.12) On dit que les variables d'état apparaissent lorsque le programmeur en ressent le besoin, expliquez.

14.13) Pourquoi faut-il manipuler la structure de choix multiple avec précaution ?

14.14) Quel est le principe général de la structure de choix multiple ?

14.15) Montrez comment représenter la structure de choix multiple de manière visuelle (ordinogramme).

14.16) En C la structure de choix multiple est souvent appelée ……

14.17) Donnez la syntaxe en C de la structure de choix multiple.

14.18) Que signifie « switch » en anglais ?

14.19) Que signifie « case » en anglais ?

14.20) A quoi correspond le mot « default » de la syntaxe en C au niveau du LDA (ou de l'ordinogramme) ?

14.21) A quoi sert le mot « break » ?

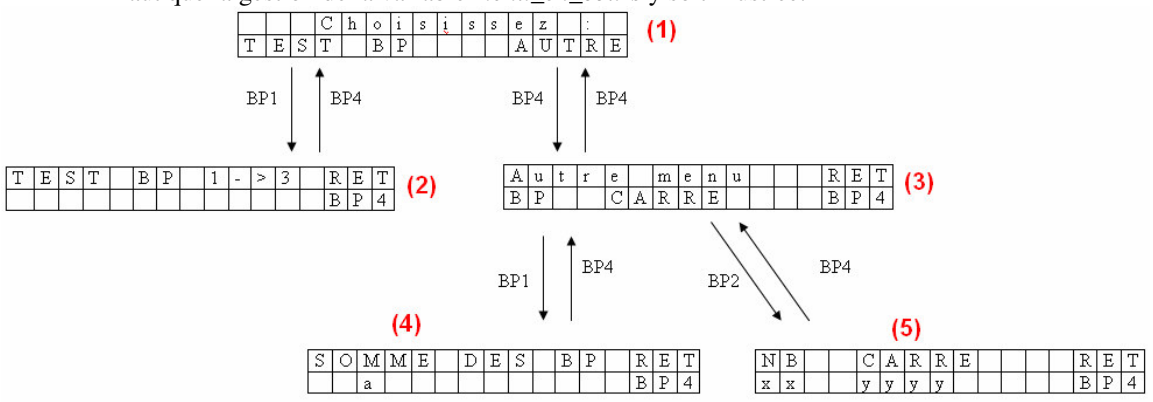
14.22) Quelle est la structure qui n'est pas concernée par le mot « break » ?

14.23) A quoi servent les accolades '{' et '}' au sein de la structure de choix multiple ?

14.24) La structure de choix multiple convient très bien pour une arborescence de menu. Pourquoi ?

14.25) Quelle est la première chose qu'il y a lieu de faire pour gérer une arborescence de menu à l'aide de la structure de choix multiple ?

14.26) Essayez de donner la structure d'un programme en C qui gère l'arborescence présentée ci-dessous. Il faut que la gestion de la variable *menu_en_cours* y soit illustrée.



Questions sur le chapitre 15 : LDA

15.1) Que signifie LDA ?

15.2) A quoi sert le LDA ?

15.3) Citez quatre mots clés du LDA (qui n'appartiennent pas à une structure).

15.4) Les mots clés doivent être ……

15.5) Exposez le LDA (et la syntaxe en C en vis-à-vis) de la structure de choix.

15.6) Exposez le LDA (et la syntaxe en C en vis-à-vis) de la structure de choix multiple.

15.7) Exposez le LDA (et la syntaxe en C en vis-à-vis) de la « tant que ».

15.8) Exposez le LDA (et la syntaxe en C en vis-à-vis) de la « répéter…tant que ».

15.9) Exposez le LDA (et la syntaxe en C en vis-à-vis) de la « pour ».

15.10) Pour que la transformation ordinogramme → LDA soit possible (sans faire appel à des artifices particuliers), l'ordinogramme doit être……

15.11) Citez les éléments qui participent à la structure générale d'un programme.

15.12) Que signifie ordinogramme structuré ?

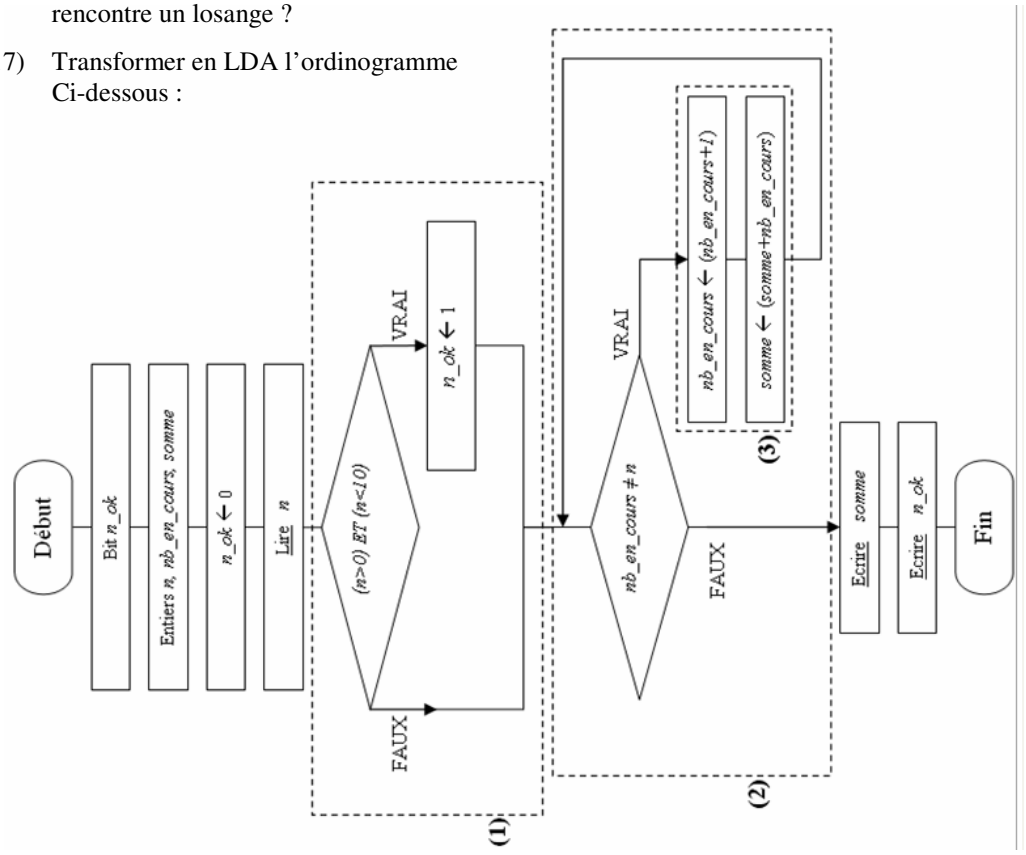
15.13) Citez un ensemble de pistes qui aident à transformer un ordinogramme en LDA.

15.14) Que doit-on faire avec les entités d'ordinogramme « début » et « fin » en LDA ?

15.15) Lorsqu'on veut transformer un ordinogramme → LDA, que doit-on faire avec le contenu des rectangles ?

15.16) Lorsqu'on veut transformer un ordinogramme → LDA, que doit-on commencer à faire lorsqu'on rencontre un losange ?

15.17) Transformer en LDA l'ordinogramme Ci-dessous :



Questions sur le chapitre 16 : Eléments supplémentaires

16.1) Ecrivez deux instructions différentes (mais équivalentes) qui permettent d'ajouter 1 à la variable i.

16.2) Ecrivez deux instructions différentes (mais équivalentes) qui permettent d'enlever 1 à la variable i.

16.3) Ecrivez deux instructions différentes (mais équivalentes) qui permettent d'ajouter 10 à la variable i.

16.4) Ecrivez deux instructions différentes (mais équivalentes) qui permettent d'enlever 30 à la variable i.

16.5) Expliquez ce qu'il se passe lorsqu'on écrit : c=i++/b ;

16.6) Expliquez ce qu'il se passe lorsqu'on écrit : c=++i/b ;

16.7) Quel est l'opérateur XOR en C ?

16.8) Quel est l'opérateur pour le complément à 1 en C ?

16.9) Si op1 vaut 0b00110011 et op2 vaut 0b11110000, que vaut (op1 ^ op2) ?

16.10) Si op1 vaut 0b00110011, que vaut (~op1) ?

16.11) A quoi sert l'instruction « break ; » ?

16.12) Ecrivez les lignes en C qui permettent de faire un maximum de 50 clignotements de la LED1.
Il faut prévoir la possibilité d'interrompre les clignotements à l'aide d'un bouton poussoir.

16.13) En C il est possible d'utiliser des tableaux, Expliquez.

16.14) Donnez la syntaxe de la déclaration d'un tableau en C.

16.15) Donnez deux exemples de déclarations de tableaux en C.

16.16) Montrez comment se présente la déclaration en C d'un tableau à deux dimensions.

16.17) Donnez un exemple en C de déclaration et d'initialisation d'un tableau de 10 entiers nommé *liste*.

16.18) Donnez un exemple en C de déclaration et d'initialisation d'un tableau de 4 réels nommé *nombre*.

16.19) Donnez un exemple en C de déclaration et d'initialisation d'un tableau à deux dimensions.

16.20) Donnez un exemple d'instruction en C où la variable *var* est affectée d'un élément d'un tableau.

16.21) En C, de quelle manière gère-t-on les chaînes de caractères ?

16.22) En déclarant un tableau, on définit automatiquement un ……

16.23) Un pointeur est destiné à……

16.24) On dit que le nom du tableau est un pointeur sur……

16.25) Quel est l'opérateur d'adresse en C ?

16.26) Lorsqu'on déclare une variable, on réserve en réalité de la place ……

16.27) L'opérateur d'adresse & signifie ……

16.28) Qu'est-ce qu'un adressage direct ? Donnez un exemple.

16.29) Qu'est-ce qu'un adressage indirect ?

16.30) Pour manipuler une adresse on n'utilise plus une variable classique, on utilise ……

16.31) Une variable classique se trouve…… et contient……

16.32) Une variable pointeur se trouve……et contient……

16.33) De la même manière qu'une variable classique, il est possible d'apporter des précisions supplémentaires concernant le contenu d'une variable pointeur, expliquez.

16.34) Au niveau de la déclaration d'un pointeur, qu'est-ce qui apportera une influence sur la manipulation du pointeur ?

16.35) Donnez un exemple de déclaration de deux pointeurs sur entiers.

16.36) Une fois qu'un pointeur contient l'adresse d'une variable, on dit que ……

16.37) Lors de la manipulation du pointeur on peut traduire « * » par……

16.38) Commentez les lignes ci-dessous :

```
int a, b ;
int *ptr_int ;
a = 20 ;
ptr_int = &b ;
*ptr_int = 15 ;
a= *ptr_int ;
ptr_int = &a ;
*ptr_int = b+1 ;
ptr_int ++ ;
```

16.39) Quelles sont les deux catégories (concernant la portée) de variables ?

16.40) Dans quelle mesure une variable globale est-elle accessible par une fonction ?

16.41) Un programme bien construit possède…… de variables globales.

16.42) Dans quel cas fait-on précéder du mot clé « extern » la déclaration d'une variable globale.

16.43) Dans quel cas fait-on précéder du mot clé « static » la déclaration d'une variable globale, qu'est ce que ça permet ?

16.44) Les variables locales ne sont connues que d'une seule fonction, laquelle ?

16.45) La variable locale a la même durée de vie que ……, son emplacement mémoire et sa valeur ne sont pas ……

16.47) Dans quel cas fait-on précéder du mot clé « static » la déclaration d'une variable locale, qu'est ce que ça permet ?

16.48) Qu'est-ce qu'implique le fait qu'il y ait passage de paramètres au moment de l'appel d'une fonction ?

16.49) Que signifie passage par valeur ? Détaillez.

16.50) Comment appelle-t-on les paramètres introduits au moment de l'appel d'une fonction ?

16.51) Montrez par un exemple qu'une variable (non globale) passée en paramètre (par valeur) ne pourra être modifiée par l'intermédiaire d'une fonction.

16.52) Lorsqu'on utilise une affectation (en dehors de la fonction), il est possible de modifier la valeur d'une variable en utilisant une fonction qui reçoit la valeur de cette variable, donnez un exemple.

16.53) Avec quelle sorte de variable une fonction peut elle modifier sa valeur sans utiliser de passage par adresse ?

16.54) Que signifie passage par adresse, par pointeur ? Détaillez.

16.55) Qu'est-ce qui permet de contenir l'adresse d'une variable ?

16.56) Donnez un exemple complet (déclaration, définition et appel) d'une fonction qui peut multiplier la valeur d'une variable par 10 sans utiliser de variables globales ni d'affectation en dehors de la fonction.

16.57) Donnez un exemple d'utilisation des interruptions en programmation.

16.58) Citez plusieurs sources d'interruptions.

16.59) Une bonne gestion des interruptions est indispensable pour ……

Questions sur le chapitre 17 : Qualité d'un programme

17.1) Pour concevoir un bon programme il faut qu'il soit …… (nécessite une bonne maîtrise des outils), qu'il soit …… (bon choix des types de variables,…) et qu'il soit ……

17.2) Un programme …… apporte naturellement moins d'erreurs.

17.3) Pour augmenter la lisibilité d'un programme il faut bien choisir ses noms de …… et de …… mais aussi enrichir le programme de …… et mettre en évidence la structure du programme à l'aide …….

17.4) Que sont des « commentaires » en programmation ?

17.5) Quelle est la syntaxe en C pour mettre un commentaire sur une seule ligne ? Donnez un exemple.

17.6) Quelle est la syntaxe en C pour mettre un commentaire sur plusieurs lignes ? Donnez un exemple.

17.7) Donnez plusieurs exemples de commentaires sur une ligne et de commentaires sur plusieurs lignes.

17.8) Que sont des « indentations » en programmation ?

17.9) Pourquoi faut-il indenter en programmation ?

17.10) Quelle est la touche du clavier qu'on utilise généralement pour les indentations ?

17.11) Donnez un exemple d'utilisation des indentations pour une structure de choix simple type « Si-Alors ».

17.12) Pour un choix simple, les { } ne sont pas nécessaires si le bloc d'actions ……

17.13) Donnez un exemple d'utilisation des indentations pour un « Si-Alors-Sinon ».

17.14) Faites apparaître une boucle infinie en utilisant les indentations.

17.15) Donnez un exemple d'utilisation des indentations pour une structure répétitive « pour ».

Questions sur le chapitre 18 : Programmer sa carte info

18.1) La programmation du PIC disposé sur la carte info doit tenir compte de la configuration ……

18.2) Le PIC18F 2520 dispose des ….., ….. et ….. Chacun de ces PORTS dispose de ….. entrées /sorties.

18.3) Quel est le rôle du registre TRIS ? Détaillez.

18.4) Il faut inclure les lignes ci-dessous au début de chacun de vos programmes, commentez ces lignes:
TRISA=0b11101111 ; TRISB=0b00000000 ; TRISC=0b00000000 ;

18.5) Quel est le rôle du registre ADCON1 ? Détaillez.

18.6) Sur la carte info, quel port est utilisé pour les LEDS ?

18.7) Sur la carte info, quel port est utilisé pour les boutons ?

18.8) Pour allumer la led 3 on écrira : ……

18.9) Pour éteindre tout le PORTB on écrira : …… ou ……

18.10) Pour allumer tout le PORTB on écrira : …… ou ……

18.11) Il est préférable d'inclure au début de votre programme l'instruction : ……

18.12) Que sont les directives #include et #define ? Détaillez.

18.13) Donnez deux exemples d'utilisation de la directive #define avec la carte info.

18.14) Quelle astuce utilise-t-on pour éviter de recommencer à chaque fois tout le début du programme ?

18.15) La construction d'un projet se fait au sein d'un……qui est rattaché à un ……

18.16) Ouvrez MPLAB et faites toutes les opérations de création d'un projet. Ecrivez un programme de base, compilez-le et testez-le.