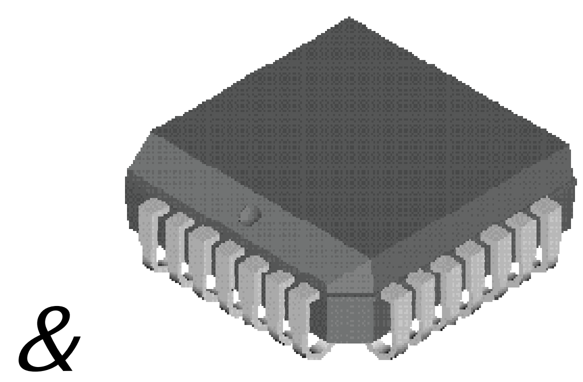
UNIVERSITE BORDEAUX I

Institut de Maintenance Aéronautique

Notions de Langage C



&

Introduction au cours « temps réel »

iUP3 - MASTER 1 ISI Mesures & Essais

Denis Michaud

2006-2007

V1. 7 version Professeur

Sommaire :

A. Généralités : langages de programmation 4
A.1. Différents langages : 4
A.1.1. Introduction : 4
A.1.2. Généralités : Langage de programmation : 4
A.1.3. Assembleur 5
- pour micro-processeur, microcontrôleur, ...5
- pour DSP, PC, ...5
- fabricant: Intel, Motorola, Thomson, Atmel, Texas I, NS, ...5
A.1.4. Langage Structuré....5
- BASIC, QuickBasic, ...5
- PASCAL, Turbo Pascal....5
- Orienté objet : Visual C++, C++, Borland C++, VisualBasic, VBA 5
- C-ANSI,....5
A.2. C, la naissance d'un langage de programmation portable 6
A.2.1. Historique : le C : 6
A.2.2. Avantages....6
A.2.3. Désavantages....7
A.2.3.a. (1) efficience et compréhensibilité : 7
A.2.3.b. (2) portabilité et bibliothèques de fonctions 9
A.2.3.c. (3) discipline de programmation....9
A.3. Domaines d'utilisation :....9
- Testeur Automatique d'équipement 9
- Calculateur aéronef, Electronique embarquée....9
- Automobile, téléphone,… spatial....9

B. Les bases de la programmation en C : 10
B.1. La filière C....10
B.2. Exemples C: 11
B.3. Description de la syntaxe en C : 12
B.3.1. Caractéristiques du C....12
B.3.2. Structure d'un programme C 12
B.3.3. Connaissances de base 13
B.3.4. Fonctions d'entrées/sorties les plus utilisées 14
B.3.4.a. Dans CONIO.H....14
B.3.4.b. Dans STDIO.H, ...15
B.3.5. les fonctions....15
B.3.5.a. Généralités 15
B.3.5.b. Définition d'une fonction utilisateur en C 15
B.3.5.c. La déclaration....16
- Type d'une fonction....16
- Nombre et type des paramètres d'entrée....16
- L'appel à la fonction 17
- Définition de la fonction 17
B.3.5.d. Les fonctions standards 17
B.3.5.e. Structure d'une fonction....18
B.3.6. Discussion de l'exemple 'Hello_World' 19
B.3.6.a. HELLO_WORLD en C....19
B.3.6.b. Discussion 20
B.3.6.c. Exercice 2.3 20
B.3.6.d. Exercice 2.4 20
B.3.6.e. Exercice 2.5 21
B.3.7. Règles d'écriture des programmes C 21
B.3.8. Exemple de programme 21
B.3.9. Identificateurs 22
B.3.10. Variables / identificateurs / adresse / pointeurs 23

B.3.11. Expressions / opérateurs 24
B.3.12. Les VARIABLES 25
B.3.13. LES OPERATEURS : 32
B.3.14. LES STRUCTURES DE CONTROLE : 34
B.3.15. POINTEUR 39

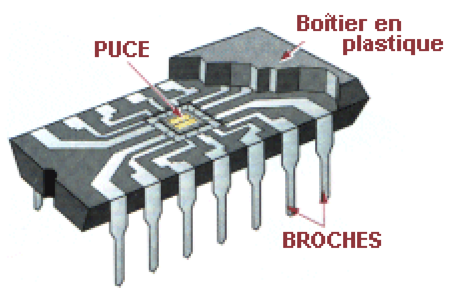
C. Généralités : les microprocesseurs 42
C.1. Le microprocesseur : c'est quoi ? 42
C.2. Qu'est-ce qu'une Unité de Traitement ? 43
C.3. Composants d'un microprocesseur 43
C.4. Les BUS 49
C.5. L'interruption 51
C.5.1. Niveau Hard 51
C.5.2. Niveau Soft 51
C.6. Familles de processeurs 52
C.7. Architecture d'un micro-contrôleur 52

D. Le HC12 de Motorola 53
D.1. Avantages / inconvénients : Les 68HC12 et 68HC912 53
D.2. La famille HC12 56
D.3. Package HC12C12 57
D.4. Kit PK-HCS12xxx 58
D.4.1. PK-HCS12C32 Starter Kit pour Motorola MC9S12C32 58
D.4.2. HC9S12E128 : Structure interne 60
D.4.3. Pile et registre: memory map: 60
D.4.4. Mécanisme d'interruption 62
D.5. Soft: Metrowerks & CodeWarrior 64
D.6. Exemple de programme pour HCS12E128 65

E. Définitions : 68

F. Bibliographie : 70

G. Glossaire : sigles & acronymes. 72



A. Généralités : langages de programmation

A.1. Différents langages :

A.1.1. Introduction :

Aujourd'hui, l'informatique est présente dans tous les domaines de la vie courante, mais à des degrés différents. Il y a pour cela trois grandes raisons :

- les gains (en temps, argent, qualité) que l'informatique peut apporter,
- le prix abordable des matériels,
- la disponibilité de logiciels dans tous les domaines.

Deux domaines sont pleinement exploités :

- les ***logiciels généraux***, vendus en grande série, et donc relativement bon marché, (windows, word, ...)

- les ***logiciels spécifiques***, d'un coût total important et donc limités à des sujets très pointus, pour de très grosses industries.(WVOffice de VIEWlogic@, gestion de site ou de process industriel)

Le ***domaine intermédiaire***, qui peut encore se développer, concerne les programmes spécifiques, pour des applications de moindre importance. Pour cela, il est nécessaire de disposer de langages de programmation.

Les tableurs et bases de données par exemple disposent désormais de véritables langages de programmation (souvent orientés objets) qui vont plus loin que les précédents langages de macro-commandes.

A.1.2. Généralités : Langage de programmation :

Un ordinateur est une machine bête, ne sachant qu'obéir, et à très peu de choses :

- addition, soustraction, multiplication en binaire, uniquement sur des entiers,
- sortir un résultat ou lire une valeur binaire (dans une mémoire par exemple),
- comparer des nombres.

Sa puissance vient du fait qu'il peut être PROGRAMME, c'est à dire que l'on peut lui donner, à l'avance, la séquence (la suite ordonnée) des ordres à effectuer l'un après l'autre.

Le grand avantage de l'ordinateur est sa rapidité. Par contre, c'est le programmeur qui doit TOUT faire. L'ordinateur ne comprenant que des ordres codés en binaire (le langage machine), des langages dits "évolués" ont été mis au point pour faciliter la programmation, au début des années 60, en particulier **FORTRAN** (FORmula TRANslator) pour le calcul scientifique et **COBOL** pour les applications de gestion.

Puis, pour des besoins pédagogiques principalement, ont été créés le **BASIC**, pour une approche simple de la programmation, et **PASCAL** au début des années 70. Ce dernier (**comme le C**) favorise une approche méthodique et disciplinée (on dit ***"structurée"***).

A.1.3. Assembleur

- *pour micro-processeur, microcontrôleur,*
- *pour DSP, PC,*
- *fabricant: Intel, Motorola, Thomson, Atmel, Texas I, NS, ….*

A.1.4. Langage Structuré

- *BASIC, QuickBasic,*
- *PASCAL, Turbo Pascal*
- *Orienté objet : Visual C++, C++, Borland C++, VisualBasic, VBA …..*
- *C-ANSI,*

- Le **langage C est né en 1972** dans les laboratoires de la Bell Telephone (AT&T) des travaux de Brian Kernighan et Dennis Ritchie.
- Il a été conçu à l'origine pour l'écriture du système d'exploitation UNIX (90-95% du noyau est écrit en C) et s'est vite imposé comme le langage de programmation sous UNIX.
- Très inspiré des langages BCPL (Martin Richard) et B (Ken Thompson), il se présente comme un "super-assembleur" ou "assembleur portable".
  En fait c'est un compromis entre un langage de haut niveau (Pascal, Ada ...) et un langage de bas niveau (assembleur).
  Il a été par le comité X3J11 de **l'American National Standards Institute** (**ANSI**).

Le C est souvent le meilleur choix. En effet, c'est un langage structuré, avec toutes les possibilités des autres langages structurés. Mais il permet également (avec son extension C++) de gérer des objets. A l'inverse, il permet également une programmation proche du langage machine, ce qui est nécessaire pour accéder aux interfaces entre l'ordinateur et son extérieur. Mais son principal avantage est que ces trois types de programmation peuvent être combinés dans un même programme, tout en restant portable sur tous les ordinateurs existants.

Le langage C a néanmoins deux inconvénients majeurs, c'est d'être un peu plus complexe d'utilisation (mais uniquement du fait de ses nombreuses possibilités), et d'être séquentiel, ce qui ne lui permettra pas d'être le langage optimal pour les machines massivement parallèles (mais aujourd'hui il n'existe pas encore de langage universel pour ce type de machines qui puisse combiner efficacement des calculs procéduraux et du déclaratif).

A.2. C, la naissance d'un langage de programmation portable ...

- (1) efficience et compréhensibilité :
- (2) portabilité et bibliothèques de fonctions
- (3) discipline de programmation

A.2.1. Historique : le C :

Dans les dernières années, aucun langage de programmation n'a pu se vanter d'une croissance en popularité comparable à celle de C et de son jeune frère C++. L'étonnant dans ce fait est que le langage C n'est pas un nouveau-né dans le monde informatique, mais qu'il trouve ses sources en **1972** dans les 'Bell Laboratories': Pour développer une version portable du système d'exploitation UNIX, Dennis M. Ritchie a conçu ce langage de programmation structuré, mais très 'près' de la machine.

A.2.1.a. K&R-C

En 1978, le duo Brian W. Kernighan / Dennis M. Ritchie a publié la définition classique du langage C (connue sous le nom de *standard K&R-C*) dans un livre intitulé 'The C Programming Language'.

A.2.1.b. ANSI-C

Le succès des années qui suivaient et le développement de compilateurs C par d'autres maisons ont rendu nécessaire la définition d'un standard actualisé et plus précis. En 1983, le 'American National Standards Institute' (ANSI) chargeait une commission de mettre au point 'une définition explicite et indépendante de la machine pour le langage C', qui devrait quand même conserver l'esprit du langage. Le résultat était le *standard ANSI-C*. La seconde édition du livre 'The C Programming Language', parue en **1988**, respecte tout à fait le standard ANSI-C et elle est devenue par la suite, la 'bible' des programmeurs en C.

A.2.1.c. C++

En 1983 un groupe de développeurs de AT&T sous la direction de Bjarne Stroustrup a créé le langage C++. Le but était de développer un langage qui garderait les avantages de ANSI-C (portabilité, efficience) et qui permettrait en plus la programmation orientée objet. Depuis 1990 il existe une ébauche pour un *standard ANSI-C++*. Entre-temps AT&T a développé deux compilateurs C++ qui respectent les nouvelles déterminations de ANSI et qui sont considérés comme des quasi-standards (AT&T-C++ Version 2.1 [1990] et AT&T-C++ Version 3.0 [1992]).

A.2.2. Avantages

Le grand succès du langage C s'explique par les avantages suivants; C est un langage:

(1) universel :

C n'est pas orienté vers un domaine d'applications spéciales, comme par exemple FORTRAN (applications scientifiques et techniques) ou COBOL (applications commerciales ou traitant de grandes quantités de données).

(2) compact :

C est basé sur un noyau de fonctions et d'opérateurs limité, qui permet la formulation d'expressions simples, mais efficaces.

(3) moderne :

C est un langage structuré, déclaratif et récursif; il offre des structures de contrôle et de déclaration comparables à celles des autres grands langages de ce temps (FORTRAN, ALGOL68, PASCAL).

(4) près de la machine :

comme C a été développé en premier lieu pour programmer le système d'exploitation UNIX, il offre des opérateurs qui sont très proches de ceux du langage machine et des fonctions qui permettent un accès simple et direct aux fonctions internes de l'ordinateur (p.ex: la gestion de la mémoire).

(5) rapide :

comme C permet d'utiliser des expressions et des opérateurs qui sont très proches du langage machine, il est possible de développer des programmes efficients et rapides.

(6) indépendant de la machine :

bien que C soit un langage près de la machine, il peut être utilisé sur n'importe quel système en possession d'un compilateur C. Au début C était surtout le langage des systèmes travaillant sous UNIX, aujourd'hui C est devenu le langage de programmation standard dans le domaine des micro-ordinateurs.

(7) portable :

en respectant le standard ANSI-C, il est possible d'utiliser le même programme sur tout autre système (autre hardware, autre système d'exploitation), simplement en le recompilant.

(8) extensible :

C ne se compose pas seulement des fonctions standard; le langage est animé par des bibliothèques de fonctions privées ou livrées par de nombreuses maisons de développement.

A.2.3. Désavantages

Evidemment, rien n'est parfait. Jetons un petit coup d'oeil sur le revers de la médaille:

A.2.3.a. (1) efficience et compréhensibilité :

En C, nous avons la possibilité d'utiliser des expressions compactes et efficientes. D'autre part, nos programmes doivent rester compréhensibles pour nous-mêmes et pour d'autres. Comme nous allons le constater sur les exemples suivants, ces deux exigences peuvent se contredire réciproquement.

- ***Exemple 1***

Les deux lignes suivantes impriment les **N** premiers éléments d'un tableau **A[]**, en insérant un espace entre les éléments et en commençant une nouvelle ligne après chaque dixième chiffre:

```
for (i=0; i<n; i++)
    printf("%6d%c", a[i], (i%10==9)?'\n':' ');
```

Cette notation est très pratique, mais plutôt intimidante pour un débutant. L'autre variante, plus près de la notation en Pascal, est plus lisible, mais elle ne profite pas des avantages du langage C:

```
for (I=0; I<N; I=I+1)
    {
     printf("%6d", A[I]);
     if ((I%10) == 9)
         printf("\n");
     else
         printf(" ");
    }
```

- ***Exemple 2***

La fonction **copietab()** copie les éléments d'une chaîne de caractères **T[]** dans une autre chaîne de caractères **S[]**. Voici d'abord la version 'simili-Pascal' :

```
void copietab(char S[], char T[])
{
  int I;
  I=0;
  while (T[I] != '\0')
   {
     S[I] = T[I];
     I = I+1;
   }
  S[I] = '\0';
}
```

Cette définition de la fonction est valable en C, mais en pratique elle ne serait jamais programmée ainsi. En utilisant les possibilités de C, un programmeur expérimenté préfère la solution suivante:

```
void copietab(char *S, char *T)
{
  while (*S++ = *T++);
}
```

La deuxième formulation de cette fonction est élégante, compacte, efficace et la traduction en langage machine fournit un code très rapide...; mais bien que cette manière de résoudre les problèmes soit le cas normal en C, il n'est pas si évident de suivre le raisonnement.

- ***Conclusion***

Bien entendu, dans les deux exemples ci-dessus, les formulations 'courtes' représentent le bon style dans C et sont de loin préférables aux deux autres. Nous constatons donc que:

la programmation efficiente en C nécessite beaucoup d'expérience et n'est pas facilement accessible à des débutants.

Sans commentaires ou explications, les programmes peuvent devenir incompréhensibles, donc inutilisables.

A.2.3.b. (2) portabilité et bibliothèques de fonctions

- ***Les limites de la portabilité***

La portabilité est l'un des avantages les plus importants de C: en écrivant des programmes qui respectent le standard ANSI-C, nous pouvons les utiliser sur n'importe quelle machine possédant un compilateur ANSI-C. D'autre part, le répertoire des fonctions ANSI-C est assez limité. Si un programmeur désire faire appel à une fonction spécifique de la machine (p.ex: utiliser une carte graphique spéciale), il est assisté par une foule de fonctions 'préfabriquées', mais il doit être conscient qu'il risque de perdre la portabilité. Ainsi, il devient évident que les avantages d'un programme portable doivent être payés par la restriction des moyens de programmation.

A.2.3.c. (3) discipline de programmation

- ***Les dangers de C***

Nous voici arrivés à un point crucial: C est un langage près de la machine, donc dangereux et bien que C soit un langage de programmation structuré, il ne nous force pas à adopter un certain style de programmation (comme p.ex. Pascal). Dans un certain sens, tout est permis et la tentation de programmer du 'code spaghetti' est grande. (Même la commande 'goto', si redoutée par les puristes ne manque pas en C). Le programmeur a donc beaucoup de libertés, mais aussi des responsabilités: il doit veiller lui-même à adopter un style de programmation propre, solide et compréhensible.

- ***Remarque***

Au fil de l'introduction du langage C, ce manuel contiendra quelques recommandations au sujet de l'utilisation des différents moyens de programmation. Il est impossible de donner des règles universelles à ce sujet.

A.3. Domaines d'utilisation :.

- ***Testeur Automatique d'équipement***

- ***Calculateur aéronef, Electronique embarquée***

- ***Automobile, téléphone,… spatial.***

B. Les bases de la programmation en C :

La programmation en langage C implique la maîtrise des notions suivantes :

-les variables ;
-les opérateurs ;
-les structures de contrôle;
-les fonctions ;
-les pointeurs.

B.1. La filière C

Schéma: Bibliothèques de fonctions et compilation

Fichiers en-tête

graphics.h
/* declarations */

math.h
/* declarations */

. . .

Programme principal

```
#include . . .
#include . . .
#include <graphics.h>
#include <math.h>


main (void)
{
. . .
}
```

Compiler

code compilé
de la fonction 'main'

Bibliothèques précompilées

graphics.lib

maths.lib

. . .

Linker

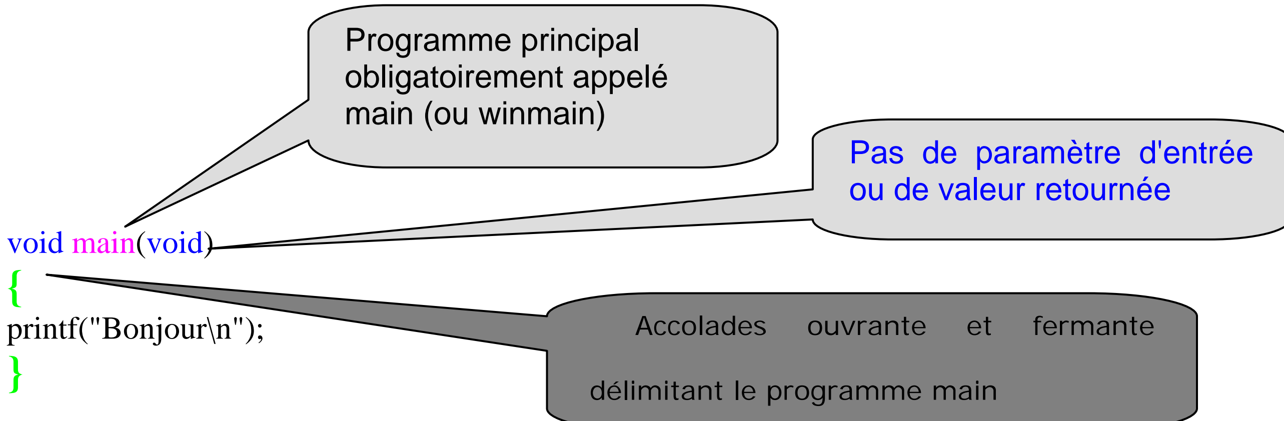
Programme exécutable

.EXE

Programmer en utilisant des bibliothèques de fonctions

B.2. Exemples C:

B.2.1. un programme en langage C



Fonction "printf" appelée par le programme main, affiche le message "Bonjour" à l'écran.

B.2.2. Exemple : Hello C !

Suivons la tradition et commençons la découverte de C par l'inévitable programme 'hello world'. Ce programme ne fait rien d'autre qu'imprimer les mots suivants sur l'écran:

Comparons d'abord la définition du programme en C avec celle en langage algorithmique.

HELLO_WORLD en langage algorithmique

```
programme HELLO_WORLD
|   (* Notre premier programme en C *)
|   écrire "hello, world"
fprogramme
```

HELLO_WORLD en C

```
#include <stdio.h>
main()
/* Notre premier programme en C */
{
   printf("hello, world\n");
   return 0;
}
```

- *Dans la suite du chapitre, nous allons discuter les détails de cette implémentation.*

B.3. Description de la syntaxe en C :

B.3.1. *Caractéristiques du C*

Langage structuré, conçu pour traiter les tâches d'un programme en les mettant dans des blocs.

Il produit des **programmes efficaces** : il possède les mêmes possibilités de contrôle de la machine que l'***assembleur*** et il génère un code **compact et rapide**.

Déclaratif : normalement, tout objet C doit être déclaré avant d'être utilisé. S'il ne l'est pas, il est considéré comme étant
du type entier.
Format libre : la mise en page des divers composants d'un programme est totalement libre.

Cette possibilité doit être exploitée pour rendre les programmes lisibles.

Modulaire : une application pourra être découpée en modules qui pourront être compilés séparément.

Un ensemble de programmes déjà opérationnels pourra être réuni dans une librairie. Cette aptitude permet au langage C de se développer de lui même.

Souple et permissivité : peu de vérifications et d'interdits, hormis la syntaxe. Il est important de remarquer que la tentation est grande d'utiliser cette caractéristique pour écrire le plus souvent des atrocités.

Transportable : les entrées/sorties sont réunies dans une librairie externe au langage.

B.3.2. *Structure d'un programme C*

Un programme C est composé de :

- **Directives du préprocesseur** : elles permettent d'effectuer des manipulations sur le texte du programme source avant la compilation :
  - inclusion de fichiers,
  - substitutions,
  - macros,
  - compilation conditionnelle.

Une directive du préprocesseur est une ligne de programme source commençant par le caractère dièse (#).

Le préprocesseur (*/lib/cpp*) est appelé automatiquement, en tout premier, par la commande */bin/cc*.

- **Déclarations/définitions** :
  - **Déclaration** : la déclaration d'un objet C donne simplement ses caractéristiques au compilateur et ne génère aucun code.
  - **Définition** : la définition d'un objet C déclare cet objet et crée effectivement cet objet.
- **Fonctions** : Ce sont des sous-programmes dont les instructions vont définir un traitement sur des variables.
- **Des commentaires** : éliminés par le préprocesseur, ce sont des textes compris entre **/*** et ***/**.

On ne doit pas les imbriquer et ils peuvent apparaître en tout point d'un programme (sauf dans une constante de type chaîne de caractères ou caractère).

Pour ignorer une partie de programme il est préférable d'utiliser une directive du préprocesseur (**#if 0 ... #endif**)

B.3.3. Connaissances de base

regardons ce petit programme :

```
#include <stdio.h>
#define TVA 18.6
void main(void)
 {
  float HT,TTC;
  puts ("veuillez entrer le prix H.T.");
  scanf("%f",&HT);
  TTC=HT*(1+(TVA/100));
  printf("prix T.T.C. %f\n",TTC);
 }
```

On trouve dans ce programme :

* des directives du pré processeur (commençant par #) **(pas de ;)**

#include : inclure le fichier définissant (on préfère dire déclarant) les fonctions standard d'entrées/sorties (en anglais STanDard In/Out), qui feront le lien entre le programme et la console (clavier/écran). Dans cet exemple il s'agit de puts, scanf et printf.
#define : définit une constante. A chaque fois que le compilateur rencontrera, dans sa traduction de la suite du fichier en langage machine, le mot TVA, ces trois lettres seront remplacées par 18.6. Ces transformation sont faites dans une première passe (appelée pré compilation), où l'on ne fait que du "traitement de texte", c'est à dire des remplacements d'un texte par un autre sans chercher à en comprendre la signification.

* une **entête de fonction**. Dans ce cas on ne possède qu'une seule fonction, la fonction principale (**main** function). Cette ligne est obligatoire en C, elle définit le "point d'entrée" du programme, c'est à dire l'endroit où débutera l'exécution du programme. Les fonctions sont écrites sous la forme : entête { corps }
L'entête est de la forme : ***type_résultat nom (arguments)*** . Le type_résultat n'était obligatoire (avant la norme ANSI) que s'il était différent de int (entier). Il doit désormais être void (rien) si la fonction ne renvoie rien (dans un autre langage on l'aurait alors appelé sous-programme, procédure, ou sous-routine). Les arguments, s'ils existent, sont passés par valeur. Si la fonction ne nécessite aucun argument, il faut indiquer (void) d'après la norme ANSI, ou **du moins (),**
Le corps est composé de déclarations de variables locales, et d'instructions, toutes terminées par un ;

* un "**bloc d'instructions**", délimité par des accolades { }, et comportant :

* des ***déclarations de variables***, sous la forme : type listevariables;

Une variable est un case mémoire de l'ordinateur, que l'on se réserve pour notre programme. On définit le nom que l'on choisit pour chaque variable, ainsi que son type, ici float, c'est à dire réel (type dit à virgule flottante, d'où ce nom). Les trois types scalaires de base du C sont l'entier (int), le réel (float) et le caractère (char). On ne peut jamais utiliser de variable sans l'avoir déclarée auparavant. Une faute de frappe devrait donc être facilement détectée, à condition d'avoir choisi des noms de variables suffisamment différents (et de plus d'une lettre).

* des ***instructions***, toutes terminées par un **;** **(point virgule)**. Une instruction est un ordre élémentaire que l'on donne à la machine, qui manipulera les données (variables) du programme, ici soit par appel de fonctions (puts, scanf, printf) soit par affectation (=).

L'instruction nulle est composée d'un **;** seul.
Il est recommandé, afin de faciliter la lecture et le "deboguage" de ne mettre qu'une seule instruction par ligne dans le source du programme.
Un bloc est une suite d'instructions élémentaires délimitées par des accolades { et }
Un bloc peut contenir un ou plusieurs blocs inclus
Un bloc peut commencer par des déclarations/définitions d'objets qui seront locaux à ce bloc. Ces objets ne pourront être utilisés que dans ce bloc et les éventuels blocs inclus à ce bloc.

Détaillons les 4 instructions de notre programme :

puts affiche à l'écran le texte qu'on lui donne (entre parenthèses, comme tout ce que l'on donne à une fonction, et entre guillemets, comme toute constante texte en C).

scanf attend que l'on entre une valeur au clavier, puis la met dans la mémoire (on préfère dire variable) HT, sous format réel (%f).

une **affectation** : on commence par diviser TVA par 100 (à cause des parenthèses), puis on y ajoute 1, puis on le multiplie par le contenu de la variable HT. Le résultat de ce calcul est stocké (affecté) dans la variable cible TTC. Une affectation se fait toujours dans le même sens : on détermine (évalue) tout d'abord la valeur à droite du signe =, en faisant tous les calculs nécessaires, puis elle est transférée dans la mémoire dont le nom est indiqué à gauche du =. On peut donc placer une expression complexe à droite du =, mais à sa gauche seul un nom de variable est possible, aucune opération.

printf affichera enfin le résultat stocké dans TTC.

B.3.4. Fonctions d'entrées/sorties les plus utilisées

Le **langage C se veut totalement indépendant du matériel sur lequel il est implanté**. Les entrées sorties, bien que nécessaires à tout programme (il faut lui donner les données de départ et connaître les résultats), ne font donc pas partie intégrante du langage. On a simplement prévu des **bibliothèques** de fonctions de base, qui sont néanmoins standardisées, c'est à dire que sur chaque compilateur on dispose de ces bibliothèques, contenant les même fonctions (du moins du même nom et faisant apparemment la même chose, mais programmées différemment en fonction du matériel). Ces bibliothèques ont été définies par la **norme ANSI**, on peut donc les utiliser tout en restant portable.

Nous détaillerons ici deux bibliothèques : **CONIO.H et STDIO.H**.

B.3.4.a. Dans CONIO.H

on trouve les fonctions de base de gestion de la console :

putch(char) : affiche sur l'écran (ou du moins stdout) le caractère fourni en argument (entre parenthèses). stdout est l'écran, ou un fichier si on a redirigé l'écran (en rajoutant >nomfichier derrière l'appel du programme, sous DOS ou UNIX). Si besoin est, cette fonction rend le caractère affiché ou EOF en cas d'erreur.

getch(void) : attend le prochain appui sur le clavier, et rend le caractère qui a été saisi. L'appui sur une touche se fait sans écho, c'est à dire que rien n'est affiché à l'écran. En cas de redirection du clavier, on prend le prochain caractère dans le fichier d'entrée.

getche(void) : idem getch mais avec écho

B.3.4.b. Dans STDIO.H,

on trouve des fonctions plus évoluées, pouvant traiter plusieurs caractères à la suite (par les fonctions de conio.h), et les transformer pour en faire une chaîne de caractères ou une valeur numérique, entière ou réelle par exemple. Les entrées sont dites "bufférisées", c'est à dire que le texte n'est pas transmis, et peut donc encore être modifié, avant le retour chariot.

On possède encore d'autres fonctions dans STDIO, en particulier pour gérer les fichiers.

B.3.5. les fonctions

B.3.5.a. Généralités

• Les fonctions en C correspondent à la notion de sous programme en Assembleur.
Comme en mathématique, elles peuvent accepter un ou plusieurs paramètres d'entrée et renvoyer un résultat ou aucun.
Elles peuvent renvoyer plusieurs résultats mais de façon plus compliquée qui ne sera pas abordée ici.

• En C, on trouve 2 types de fonctions :

-**les fonctions standards** : fournies avec le compilateur C, correspondant aux fonctions les plus couramment utilisées par le programmeur : saisie, affichage, …

-**les fonctions utilisateurs** : écrites par l'utilisateur pour répondre à un besoin spécifique.

Remarque : il est souvent judicieux de regrouper les fonctions utilisateurs dans une bibliothèque utilisateur ; ce qui permettra une réutilisation ultérieure des fonctions développées pour une application spécifique.

B.3.5.b. Définition d'une fonction utilisateur en C

La mise en œuvre des fonctions utilisateurs nécessite 3 étapes :
- la déclaration ;
- l'appel ;
- la définition.

```
#include <stdio.h>
#include <conio.h>

/* Déclaration des fonctions du programme */
void efface();
void affiche(int x);
int calcule(int x,inty);

void main(void)
{
int a,b,s;

efface();            /*Appel à la fonction */
```

```
scanf("%d %d",&a,&b);
s=calcule(a,b);  /* Appel de la fonction avec spécifications des arguments d'entréee */
affiche(s);
}

void efface(void)
{
clrscr();
}

void affiche(int x) /* Entête de la fonction (sans ";")
{/* Corps de la fonction */
printf("Le résultat vaut %d\n",x);
}/* Fin du corps de la fonction */

int calcule(int x,int y) /* Entête de la fonction avec arguments */
{
/* Corps de la fonction */
int w;
w=x+2*y;
return(w); /* Résultat de la fonction retourné au programme appelant */
}
```

Dans l'exemple, la fonction efface() est appelée dans la fonction main().
On dit que la fonction main() est la fonction **appelante**.

B.3.5.c. La déclaration

Comme toute variable, une fonction doit être déclarée **avant** son utilisation.
La déclaration permet de définir :

- le type de la fonction ;
- le nom de la fonction ;
- le nombre de paramètres d'entrée et leur type.

- ***Type d'une fonction***

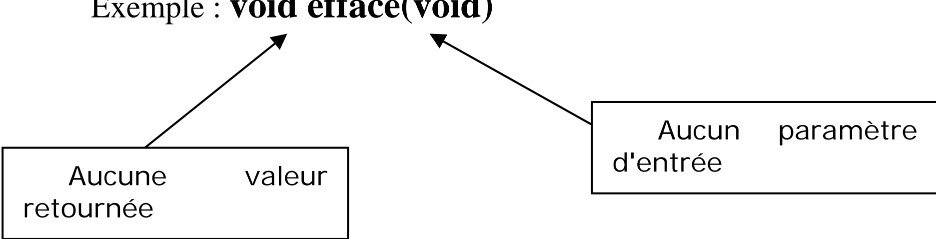
  • Il dépend du résultat renvoyé par la fonction à la fonction appelante.
  Pour une fonction, les types sont les mêmes que ceux définis pour les variables.

  • Si la fonction ne renvoie aucun résultat, elle est de type **void** (néant).

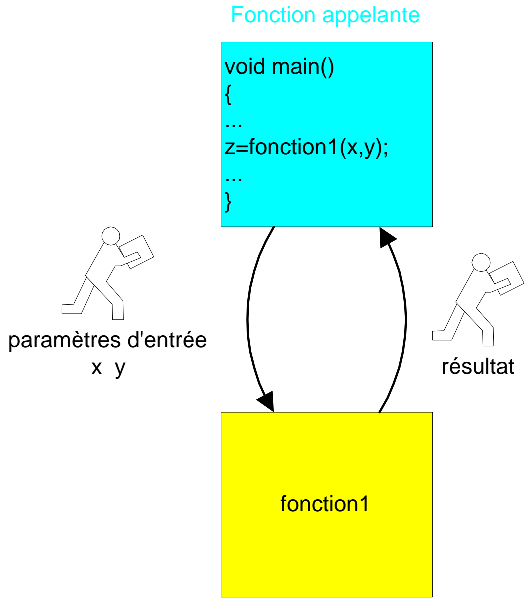
- ***Nombre et type des paramètres d'entrée***

  La liste des arguments d'entrée définit le type de chaque argument. Les types possibles sont ceux définis pour une variable.
  Elle définit un nom qui ne revêt pas une importance capitale pour le compilateur.
  Lorsque la fonction n'admet aucun paramètre, on utilise le terme **void** dans la déclaration.

  Exemple : **void efface(void)**



- ***L'appel à la fonction***



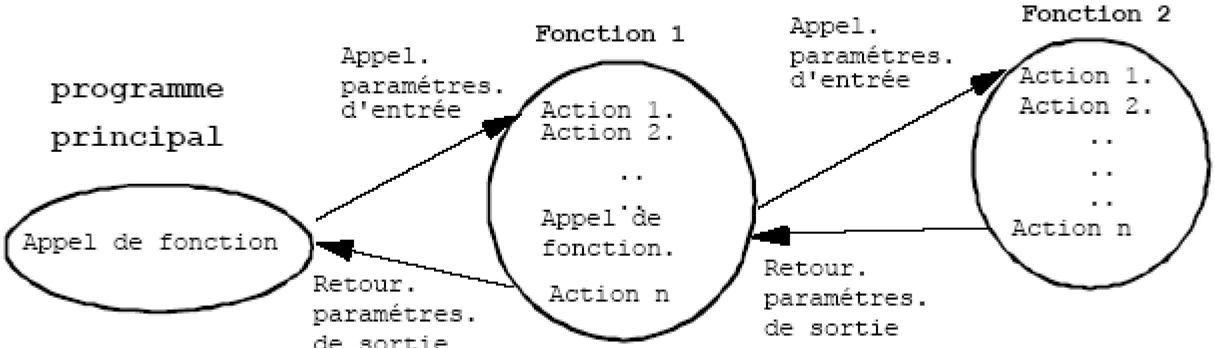
• Lors de l'appel à la fonction 1, les arguments éventuels sont placés dans la pile système.

• Le µP sauve l'adresse de retour dans la pile, quitte la fonction main() et commence à exécuter la fonction 1 après avoir récupéré la valeurs des arguments dans la pile.

• A la fin, il place le résultat dans la pile, puis revient dans la fonction main() à l'endroit spécifié par l'adresse de retour sauvée dans la pile.

• Le µP récupère le résultat dans la pile et le place dans z.

Le programme principal `main()` et les fonctions ont besoin de se communiquer des valeurs. Lors d'un appel d'une procédure les valeurs passées sont les paramètres d'entrée et à la fin d'une fonction les paramètres renvoyés sont les paramètres de sortie.



- ***Définition de la fonction***

  • La première ligne doit correspondre exactement à la déclaration avec le point virgule en moins.

  • Il ne reste plus qu'à écrire le corps de la fonction sous la forme d'un morceau de programme en langage C.

B.3.5.d. Les fonctions standards

• Le compilateur met à la disposition du programmeur outre les fonctions standard de base communes à tous les compilateurs C, un ensemble de fonctions qui recouvre un certain nombre de domaines et facilitent la programmation.
• Le compilateur fournit en même temps des fichiers d'entête (d'extension .h) qui contiennent la déclaration des fonctions standards et différentes définitions nécessaires à la mise en œuvre correcte de ces fonctions.
Ces fichiers d'entête sont spécifiques à chaque classe de fonctions standards et doivent insérés **avant** l'utilisation de la fonction standard.
Pour cela, on utilise la directive "#include <nom du fichier d'entête>" à placer au début du programme.

Attention "<" et ">" sont obligatoires.

Exemple :

```
#include <stdio.h>

void main(void)
{
printf("Bonjour\n");
}
```

Fichier d'entête pour la fonction printf

B.3.5.e. Structure d'une fonction

Une fonction est un bloc de code d'une ou plusieurs instructions qui peut renvoyer une valeur à l'expression qui l'utilise.

- Elle peut retourner une valeur à la fonction appelante.
- Le programme principal est une fonction dont le nom doit impérativement être main.
- Les fonctions ne peuvent pas être imbriquées.

La forme générale (en ANSI)d'une fonction est :

[classe] [type] nom([liste_de_parametres_formels])

bloc_de_la_fonction

Les éléments entre [] dans cette syntaxe, signifie que cet élément est facultatif, car une valeur par défaut existe.

En C, le programme principal et les sous-programmes sont définis comme fonctions. Il n'existe pas de structures spéciales pour le programme principal ni les procédures (comme en Pascal ou en langage algorithmique).

Le programme principal étant aussi une 'fonction', nous devons nous intéresser dès le début à la définition et aux caractéristiques des fonctions en C. Commençons par comparer la syntaxe de la définition d'une fonction en C avec celle d'une fonction en langage algorithmique:

Définition d'une fonction en langage algorithmique

```
fonction <NomFonct> (<NomPar1>, <NomPar2>, ...):<TypeRés>
   <déclarations des paramètres>
   <déclarations locales>
   <instructions>
ffonction
```

Définition d'une fonction en C

```
<TypeRés> <NomFonct> (<TypePar1> <NomPar1>,
                      <TypePar2> <NomPar2>, ... )
{
   <déclarations locales>
   <instructions>
}
```

En C, une fonction est définie par:

* une ligne déclarative qui contient:

<TypeRés> - le type du résultat de la fonction

<NomFonct> - le nom de la fonction

<TypePar1> <NomPar1>, <TypePar2> <NomPar2>, ...

les types et les noms des paramètres de la fonction

* un bloc d'instructions délimité par des accolades { }, contenant:

<déclarations locales> - les déclarations des données locales (c.-à-d.: des données qui sont uniquement connues à l'intérieur de la fonction)

<instructions> - la liste des instructions qui définit l'action qui doit être exécutée

Résultat d'une fonction

Par définition, *toute fonction en C fournit un résultat* dont le type doit être défini. Si aucun type n'est défini explicitement, C suppose par défaut que le type du résultat est **int** (integer).

Le retour du résultat se fait en général à la fin de la fonction par l'instruction **return**.

Le type d'une fonction qui ne fournit pas de résultat (comme les procédures en langage algorithmique ou en Pascal), est déclaré comme **void** (vide).

Paramètres d'une fonction

La définition des paramètres (arguments) d'une fonction est placée entre parenthèses **( )** derrière le nom de la fonction. Si une fonction n'a pas besoin de paramètres, les parenthèses restent vides ou contiennent le mot **void**. La fonction minimale qui ne fait rien et qui ne fournit aucun résultat est alors:

```
void dummy() {}
```

Instructions

En C, *toute* instruction simple est terminée par un point-virgule **;** (même si elle se trouve en dernière position dans un bloc d'instructions). Par exemple:

```
printf("hello, world\n");
```

B.3.6. Discussion de l'exemple 'Hello_World'

- Exercice 2.3
- Exercice 2.5

Reprenons le programme 'Hello_World' et retrouvons les particularités d'un programme en C.

B.3.6.a. HELLO_WORLD en C

```
  #include <stdio.h>
main()
/* Notre premier programme en C */
{
   printf("hello, world\n");
   return 0;
}
```

B.3.6.b. Discussion

- La fonction **main** ne reçoit pas de données, donc la liste des paramètres est vide.

- La fonction **main** fournit un code d'erreur numérique à l'environnement, donc le type du résultat est **int** et n'a pas besoin d'être déclaré explicitement.

- Le programme ne contient pas de variables, donc le bloc de déclarations est vide.

- La fonction **main** contient deux instructions:

* l'appel de la fonction **printf** avec l'argument "hello, world\n";

Effet: Afficher la chaîne de caractères "hello world\n".

* la commande **return** avec l'argument 0;

Effet: Retourner la valeur 0 comme code d'erreur à l'environnement.

- L'argument de la fonction **printf** est une chaîne de caractères indiquée entre les guillemets. Une telle suite de caractères est appelée *chaîne de caractères constante (string constant).*

- La suite de symboles **'\n'** à la fin de la chaîne de caractères "hello, world\n" est la notation C pour '*passage à la ligne*' (angl: new line). En C, il existe plusieurs couples de symboles qui contrôlent l'affichage ou l'impression de texte. Ces *séquences d'échappement* sont toujours précédées par le caractère d'échappement **'\'**. (voir exercice 2.4).

printf et la bibliothèque <stdio>

La fonction **printf** fait partie de la bibliothèque de fonctions standard *<stdio>* qui gère les entrées et les sorties de données. La première ligne du programme:

```
#include <stdio.h>
```

instruit le compilateur d'inclure le fichier en-tête '`STDIO.H`' dans le texte du programme. Le fichier '`STDIO.H`' contient les informations nécessaires pour pouvoir utiliser les fonctions de la bibliothèque standard *<stdio>* .

B.3.6.c. Exercice 2.3

Modifiez le programme 'hello world' de façon à obtenir le même résultat sur l'écran en utilisant plusieurs fois la fonction **printf**.

B.3.6.d. Exercice 2.4

Expérimentez avec les séquences d'échappement que vous trouvez dans le tableau ci-dessous et complétez les colonnes vides.

séq. d'échapp.	*descr. anglaise*	*descr. française*
`\n`	`new line`	`passage à la ligne`
`\t`		
`\b`		
`\r`		
`\"`		
`\\`		
`\0`		
`\a`		

B.3.6.e. Exercice 2.5

Ci-dessous, vous trouvez un simple programme en C. Essayez de distinguer et de classifier autant que possible les éléments qui composent ce programme (commentaires, variables, déclarations, instructions, etc.)

```
#include <stdio.h>
/* Ce programme calcule la somme de 4 nombres entiers
   introduits au clavier.
*/
main()
{
    int NOMBRE, SOMME, COMPTEUR;

    /* Initialisation des variables */
    SOMME = 0;
    COMPTEUR = 0;
    /* Lecture des données */
    while (COMPTEUR < 4)
        {
         /* Lire la valeur du nombre suivant */
         printf("Entrez un nombre entier :");
         scanf("%i", &NOMBRE);
         /* Ajouter le nombre au résultat */
         SOMME += NOMBRE;
         /* Incrémenter le compteur */
         COMPTEUR++;
        }
    /* Impression du résultat */
    printf("La somme est: %i \n", SOMME);
    return 0;
}
```

B.3.7. Règles d'écriture des programmes C

Afin d'**écrire des programmes C lisibles**, il est important de respecter un certain nombre de règles de présentation :

- ne jamais placer plusieurs instructions sur une même ligne
- utiliser des identificateurs significatifs
- grâce à l'indentation des lignes, on fera ressortir la structure syntaxique du programme.
- Les valeurs de décalage les plus utilisées sont de 2, 4 ou 8 espaces.
- on laissera une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions.
- une accolade fermante est seule sur une ligne (à l'exception de l'accolade fermante du bloc de la structure do ... while) et fait référence, par sa position horizontale, au début du bloc qu'elle ferme.
- aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces.
  il est nécessaire de **commenter les listings**. Eviter les commentaires triviaux.

B.3.8. Exemple de programme

1ère ligne, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34ème ligne

```
/* **************************
   Programme de démonstration
   **************************/
#include <stdio.h>  /* directives au préprocesseur */
#define DEBUT -10
#define FIN 10
#define MSG "Programme de démonstration\n"
/***********************************/
int carre(int x);   /* déclaration des fonctions */
int cube(int x);
/***********************************/
main()             /* debut programme principal */
{                  /* début du bloc de la fonction main */
  int i;           /* définition des variables locales */
  printf(MSG);
  for ( i = DEBUT; i <= FIN ; i++ )
  {
     printf("%d   carré: %d   cube: %d\n", i
                                   , carre(i)
                                   , cube(i) );
  }                /* fin du bloc for  */
  return 0;
}                  /* fin du bloc de la fonction main */
int cube(int x) {   /* definition de la fonction cube */
   return x * carre(x);
}
int carre(int x) {  /* definition de la fonction carre */
   return x * x;
}
```

B.3.9. Identificateurs

Les identificateurs nomment les objets C (fonctions, variables ...)

- C'est une suite de lettres ou de chiffres.
- Le premier caractère est obligatoirement **une lettre**.
- Le caractère _ (souligné) est considéré comme une lettre.
- Le C distingue les minuscules des majuscules. Exemple : **carlu Carlu CarLu CARLU** sont des identificateurs valides et tous différents.
- La longueur de l'identificateur dépend de l'implémentation. La norme ANSI prévoit qu'au moins les 31 premiers caractères soient significatifs pour le compilateur.
- L'éditeur de liens peut limiter le nombre de caractères significatifs des identificateurs à un nombre plus petit.
- Un identificateur ne peut pas être un **mot réservé du langage** :
-

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Recommandations :

- Utiliser des identificateurs significatifs.
- Réserver l'usage des identificateurs en majuscules pour le préprocesseur.
  Réserver l'usage des identificateurs commençant par le caractère _ (souligné) à l'usage du compilateur.

B.3.10. Variables / identificateurs / adresse / pointeurs

On appelle variable une mémoire de l'ordinateur (ou plusieurs), à laquelle on a donné un nom, ainsi qu'un type (on a précisé ce qu'on mettra dans cette variable (entier, réel, caractère,...), pour que le compilateur puisse lui réserver la quantité de mémoire nécessaire. Dans cette variable, on pourra y stocker une valeur, et la modifier au cours du programme.

Exemple : int a; a est une variable entière, le compilateur va lui réserver en mémoire la place nécessaire à un entier (2 octets en Turbo C). Le nom de cette variable est choisi par le programmeur. On préfère utiliser le terme identificateur plutôt que le nom, car il permet d'identifier tout objet que l'on voudra utiliser (pas seulement les variables).

Les identificateurs doivent suivre quelques règles de base : il peut être formé de lettres (A à Z), de chiffres et du caractère _ (souligné). Le premier caractère doit être une lettre (ou _ mais il vaut mieux le réserver au compilateur). Par exemple valeur1 ou prem_valeur sont possibles, mais pas 1ere_valeur. En C, les minuscules sont différentes des majuscules (**SURFace et surFACE désignent deux objets différents**). ***Le blanc est donc interdit dans un identificateur*** (utilisez _) .

Les lettres **accentuées sont également interdites**. La plupart des compilateurs acceptent n'importe quelle longueur d'identificateurs (tout en restant sur la même ligne) mais seuls les **32 premiers caractères sont significatifs.**

On considère comme blanc : soit un blanc (espace), soit un retour à la ligne, soit une tabulation, soit un commentaire, soit plusieurs de ceux-ci. Les commentaires sont une portion de texte commençant par /* et finissant par le premier */ rencontré. les commentaires ne peuvent donc pas être imbriqués. mais un commentaire peut comporter n'importe quel autre texte, y compris sur plusieurs lignes.

Un identificateur se termine soit par un blanc, soit par un signe non autorisé dans les identificateurs (parenthèse, opérateur, ; ...). Le blanc est alors autorisé mais non obligatoire.

L'endroit où le compilateur a choisi de mettre la variable est appelé **adresse de la variable** (c'est en général un nombre, chaque mémoire d'un ordinateur étant numérotée de 0 à ?). Cette adresse ne nous

intéresse que rarement de manière explicite, mais souvent de manière indirecte. Par exemple, dans un tableau, composé d'éléments consécutifs en mémoire, en connaissant son adresse (son début), on retrouve facilement l'adresse des différentes composantes par une simple addition.

On appelle **pointeur** une variable dans laquelle on place (mémorise) une adresse de variable (où elle est) plutôt qu'une valeur (ce qu'elle vaut).

Les types de variables scalaires simples que l'on utilise le plus couramment sont le char (un caractère), l'int (entier) et le float (réel). Le char est en fait un cas particulier des int, chaque caractère étant représenté par son numéro de code ASCII.

B.3.11. Expressions / opérateurs

Une expression est un calcul qui donne une valeur résultat (exemple : 8+5). Une expression comporte des variables, des appels de fonction et des constantes combinés entre eux par des opérateurs (ex : **MaVariable*sin(VarAngle*PI/180)**).

- **Une expression** de base peut donc être un appel à une fonction (exemple **sin(3.1416)**).
- **Une fonction** est un bout de programme (que vous avez écrit ou faisant partie d'une bibliothèque) auquel on "donne" des valeurs (arguments), entre parenthèses et séparés par des virgules. La fonction fait un calcul sur ces arguments pour "retourner" un résultat. Ce résultat pourra servir, si nécessaire, dans une autre expression, voire comme argument d'une fonction exemple **atan(tan(x)**).
- **Les arguments** donnés à l'appel de la fonction (dits paramètres réels ou effectifs) sont recopiés dans le même ordre dans des copies (paramètres formels), qui elles ne pourront que modifier les copies (et pas les paramètres réels).
  Dans le cas de fonctions devant modifier une variable, il faut fournir en argument l'adresse (par l'opérateur &, voir plus bas), comme par exemple pour scanf.

Pour former une expression, les opérateurs possibles sont assez nombreux, nous allons les détailler suivant les types de variables qu'ils gèrent.

Séquences d'échappement: voir les deux bibliothèques : CONIO.H et STDIO.H.

Comme nous l'avons vu au chapitre 2, l'impression et l'affichage de texte peut être contrôlé à l'aide de ***séquences d'échappement***. Une séquence d'échappement est un couple de symboles dont le premier est le *signe d'échappement* '\'. Au moment de la compilation, chaque séquence d'échappement est traduite en un caractère de contrôle dans le code de la machine. Comme les séquences d'échappement sont identiques sur toutes les machines, elles nous permettent d'écrire des programmes portables, c.-à-d.: des programmes qui ont le même effet sur toutes les machines, indépendamment du code de caractères utilisé.

Séquences d'échappement

1) caractères de commande précédés de " \" :

Notation en C	Code ASCII (hexa)	Abréviat° usuelle	Signification
'\a'	07	BEL	Bip sonore (Bell)
'\b'	08	BS	Retour arrière (Back space)
'\f'	0C	FF	Saut de page (Form feed)
'\n'	0A	LF	Saut de ligne (Line feed)
'\r'	0D	CR	Retour chariot (Carriage return)
'\t'	09	HT	Tabulation horizontale (Horizontal tab)
'\v'	0B	VT	Tabulation verticale (Vertical tab)
'\0'	00	NULL	Caractère nul
'\''	2C	'	Apostrophe
'\\'	5C	\	Trait oblique
'\"'	22	"	Guillemet
'\?'	3F	?	Point d'interrogation
\xHH			valeur HH en hexadécimal (2 chiffres)

Si '\' est suivi par un caractère autre que ceux ci-dessus, '\' est ignoré.

Exemple printf("n = %d soit en hexa %x et en binaire %b",x,x,x);

Le caractère NUL

La constante **'\0'** qui a la valeur numérique zéro (ne pas à confondre avec le caractère '0' !!) a une signification spéciale dans le traitement et la mémorisation des chaînes de caractères: En C le caractère '\0' définit **la fin d'une chaîne** de caractères.

2) caractères précédés de " % " :

. %b : affiche en binaire
. %c : affiche un caractère
. %d : affiche en décimal
. %o : affiche en octal
. %s : affiche une chaîne
. %u : affiche en décimal non signé
. %x : affiche en hexadécimal

B.3.12. Les VARIABLES

B.3.12.a. Les types simples

- ***Ensembles de nombres et leur représentation***

En mathématiques, nous distinguons divers ensembles de nombres:

* l'ensemble des entiers naturels **IN**,
* l'ensemble des entiers relatifs **ZZ**,
* l'ensemble des rationnels **Q**,
* l'ensemble des réels **IR**.

En mathématiques l'ordre de grandeur des nombres est illimité et les rationnels peuvent être exprimés sans perte de précision.

Un ordinateur ne peut traiter aisément que des nombres entiers d'une taille limitée. Il utilise le système binaire pour calculer et sauvegarder ces nombres. Ce n'est que par des astuces de calcul et de représentation que l'ordinateur obtient des valeurs correctement approchées des entiers très grands, des réels ou des rationnels à partie décimale infinie.

- ***Les charges du programmeur***

Même un programmeur utilisant C ne doit pas connaître tous les détails des méthodes de codage et de calcul, il doit quand même être capable de:

- *choisir un type numérique approprié à un problème donné;*
c.-à-d.: trouver un optimum de précision, de temps de calcul et d'espace à réserver en mémoire

- *choisir un type approprié pour la représentation sur l'écran*

- *prévoir le type résultant d'une opération entre différents types numériques;*
c.-à-d.: connaître les transformations automatiques de type que C accomplit lors des calculs

- *prévoir et optimiser la précision des résultats intermédiaires au cours d'un calcul complexe;*
c.-à-d.: changer si nécessaire l'ordre des opérations ou forcer l'ordinateur à utiliser un type numérique mieux adapté

Exemple :Supposons que la mantisse du type choisi ne comprenne que 6 positions décimales (ce qui est très réaliste pour le type **float**):

$$\underbrace{(1.00001*10^8 + 850)} - 1*10^8$$
$$= 1.00001*10^8 - 1*10^8 = 1000$$

$$\underbrace{(1.00001*10^8 - 1*10^8)} + 850$$
$$= 1000 + 850 = 1850$$

B.3.12.b. Les variables simples :

Elles sont définies par leur nom, leur portée et leur type.
Il faut les déclarer avant toute utilisation.

- ***Portée d'une variable***

Portée	Déclaration	Accessibilité	Existence
Convention : en lettres minuscules.	A l'intérieur d'une fonction	Uniquement à l'intérieur de la fonction où elle a été déclarée.	Temporaire Pendant l'exécution de la fonction où elle a été déclarée
Convention : initiale en majuscule0	En dehors de toute fonction. Avant le main.	Par toute fonction du programme.	Permanente.

- ***Type d'une variable***

Type	Signification	Représentation	
		Taille (bits)	Limites
.	Entier	16	-32768 à +32767
.	Entier	16	-32768 à +32767
.	entier double précision	32	-2.10^{9} à $+2.10^{9}$
.	caractère	8	-128 à +127
.	Réel	32	$\pm10^{-37}$ à $\pm10^{38}$
.	réel double précision	64	$\pm10^{-307}$ à $\pm10^{308}$

• Remarque : il existe le modificateur de type : **unsigned** qui rend transforme un type signé en type non signé.

Type	Signification	Représentation	
		Taille (bits)	Limites
.	entier non-signé	16	0 à 65535
.	entier double précision non signé	32	0 à 4 294 967 295
.	Caractère non signé	8	0 à 255

• ***Attention : ces définitions changeront avec le compilateur utilisé pour le 68HC11.***

- ***Notations particulières***

Les entiers

Un entier peut être représenté selon 3 bases numériques :

Décimal	Octal	Hexadécimal
117	.	.

Si l'entier dépasse 16 bits, il est transformé en entier long sur 32 bits.
On peut forcer cette conversion en faisant suivre l'entier par l ou L.
Exemple : 14L sera codé sur 32 bits.

Les caractères

Ils sont codés sur 8 bits et peuvent prendre des valeurs entre –128 et +127.

Entre 0 et +127, ils peuvent représenter le code ASCII :'0'=48=code **ASCII** de 0.

Table des caractères imprimables (32 à 127) — ou table ASCII standard

ASCII	Caractère.	ASCII	Caractère	ASCII	Caractère
32	SP (*space*, espace)	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	DEL (*delete*, sup.)

Tableau 2 : codage ASCII.

Les réels

Les réels sont codés en simple ou double précision selon la technique de la virgule flottante : ±MM...MMEE...EE où ± : bit de signe, MM...MM : mantisse, EE...EE : exposant.

Précision	Mantisse (bits)	Exposant (bits)
.	8	23
.	11	52

Les réels peuvent s'écrire en :

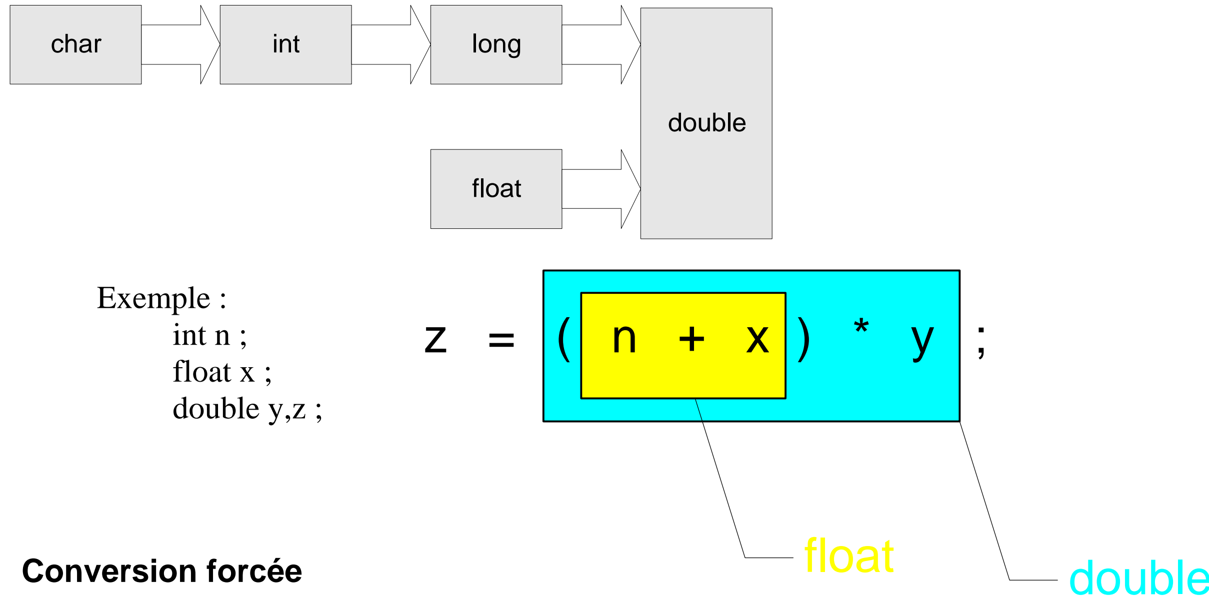
En français	En langage C
-1225,69	Notation décimale : -1225.69
-12,2569.10^{2}	Notation scientifique : -12.2569e2
25	Notation décimale : 25.

- ***Conversion de type***

Le langage C permet de manipuler des objets de type différent grâce au mécanisme de la conversion.

La conversion implicite

Un type est absorbé par un type occupant une plus grande place en mémoire.



Exemple :
int n ;
float x ;
double y,z ;

Conversion forcée

Elle apparaît lors des opérations d'affectation.
La règle est de transformer le résultat d'une affectation ou d'un calcul dans le type de l'objet qui reçoit l'affectation ou le résultat du calcul.
Ces conversions peuvent se faire dans n'importe quel sens : **elles ne provoquent aucune erreur de compilation alors que certaines d'entre elles produisent des résultats aberrants.**

Exemples :

Conversion	Commentaires
double vers float	Si la valeur est représentable par un float, on perd de la précision. Dans le cas contraire, message d'erreur « over ou underflow ».
float vers int	Si la valeur est représentable par un int, on obtient la partie entière. Sinon, on risque d'obtenir une valeur aberrante.
int vers char	L'octet de poids faible est placé dans le char.

B.3.12.c. Les tableaux

- ***Définition :***

Le langage C manipule des tableaux ou tables qui sont une liste d'éléments du même type, portant le même nom.
Chaque élément du tableau est repéré par un numéro unique : l'indice.

Attention : il commence toujours à 0.

Exemple :

```
void main()
{
int i,s,nb;
int tab[10]; /* Déclaration du tableau tab contenant 10 variables de type int */
float moy;

/* Saisie des notes dans le tableau et de la somme*/
for (i=0,s=0; i<10; i++)
	{
	printf("Donner la note n°%d: ",i+1);
	scanf("%d",&tab[i]); /* Saisie de l'élément n°i du tableau tab */
	.			/* calcul de la somme */
	}
/* Calcul de la moyenne */
.
printf("\n\nLa moyenne de la classe est %d /n toto  %d",moy, 2*moy+3);

/* Calcul du nombre d'élèves au dessus de la moyenne */
for (i=0,nb=0;i<10;i++)
	.
printf("%d élèves ont plus de la moyenne \"); .
}
```

- ***Généralités sur les tableaux :***

• t[5] repère l'élément n°5 du tableau t.

• L'élément d'un tableau s'utilise comme une variable simple : t[2]=6; t[1]++, --t[5], x=2*t[8]+t[10],...

• L'indice est de type entier (int) ou caractère (char) sous forme d'une constante ou d'une expression : t[4], t[i++], t[2*y-4],...
Un indice démarre toujours à partir de 0.

• Le langage C ne met en place aucun contrôle du débordement d'indice.

- ***Tableaux à plusieurs dimensions :***

Les tableaux à plusieurs dimensions sont autorisés.
Ils nécessitent plusieurs indices.
La déclaration "double tab[10][5]; " réserve un tableau de 10x5 éléments de type double.

Les règles des tableaux à une dimension s'appliquent pour les tableaux à plusieurs dimensions.

t[4][2] repère un élément de la 4ème ligne et de la 2ème colonne du tableau t.
Quelques opérations : t[i][j],t[i+5][8*j+i+5], y=t[i][j++],...

Progression des indices :
int t[3][4]; /* Déclaration d'un tableau de 3×4 éléments de type int */
Les éléments du tableau sont ordonnés comme suit :
t[0][0],t[0][1],t[0][2],t[1][0],t[1][1],t[1][2],t[2][0],t[2][1],t[2][2],t[3][0],t[3][1],t[3][2]

- ***Déclaration et initialisation de tableaux :***

• Avant toute utilisation un tableau doit être déclaré.

Exemples :

int t[5]; : Cette déclaration revient à réserver dans la mémoire la place pour 5 int successifs.
int t[5][6] : Idem pour un tableau 5 lignes par 6 colonnes c-à-d pour 30 int successifs.

• Un tableau peut être initialisé.

Exemples :

int t[5]={1,2,3,4,5}; : Déclaration et initialisation complète du tableau t.

int t[5]={1,,3,,5}; : Déclaration et initialisation partielle du tableau t.

int t[]={1,2,3,4,5}; : Déclaration et initialisation complète du tableau t dont on ne précise pas la dimension. Le compilateur la retrouve à l'aide de l'initialisation.

int t[3][4]={1,2,3,4,5,6,7,8,9,10,11,12}; ou
int t[3][4]={{1,2,3},{4,5,6},{7,8,9},{10,11,12}}; : Déclaration et initialisation complète du tableau t.

int t[3][4]={1,,,4,5,6,7,,,,11,12}; ou int t[3][4]={{1,,},{4,5,6},{7,,},{,11,12}}; : Déclaration et initialisation partielle du tableau t.

- ***Les chaînes de caractères***

Les chaînes de caractères sont des tableaux de caractères dont la dernières case contient le **caractère nul (valeur 0 : '\0')** appelé caractère de fin de chaîne.

• **Attention : ne pas confondre ce caractère avec le caractère '0'=48=0x30** qui représente le code ASCII de 0.

• L'intérêt de ces chaînes de caractères réside dans le fait qu'il existe un grand nombre de fonctions standard fournies avec le compilateur qui permettent de réaliser des opérations de base : copie d'une chaîne vers une autre, concaténation de 2 chaînes, transformation en format numérique, transformation d'un nombre en chaîne, calcul de la longueur d'une chaîne, …
Ces fonctions sont à utiliser avec le fichier d'entête string.h.(Cf. paragraphe traitant des fonctions).

- ***Exercice 3.1***

Quel(s) type(s) numérique(s) pouvez-vous utiliser pour les groupes de nombres suivants? Dressez un tableau et marquez le choix le plus économique:

```
(1)    1     12       4        0    -125
(2)    1     12      -4        0     250
(3)    1     12       4        0     250
(4)    1     12      -4        0.5   125
(5)    -220                    32000        0
(6)    -3000005     .000000001
(7)    410    50000      2
(8)    410    50000     -2
(9)    3.14159265              1015
(10)    2*10^7               10000001
(11)    2*10^-7             10000001
(12)    -1.05*1050             0.0001
(13)    305.122212             0     -12
```

B.3.13. LES OPERATEURS :

B.3.13.a. Les opérateurs arithmétiques :

On retrouve les opérateurs traditionnels :

+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Reste de la division entière ou modulo

REMARQUE : % ne s'applique pas aux floats et aux doubles

B.3.13.b. Les opérateurs relationnels et logiques :

Ils portent sur des propositions et des expressions.
On trouve dans l'ordre croissant :

>, >=, <, <=	Test de supériorité et infériorité
== , !=	Test d'égalité et de différence
&& , ||	"Et" logique, "ou" logique
!	Complément logique (Non)

Exemple :

```
void main()
{
int i,j;
printf("i et j ? ");
scanf("%d %d",&i,&j);
if .        printf("Egalité");
if (.          ) printf("Positifs");
if (.          ) printf("i non nul");
if (.      ) printf("Condtion vraie si j≠0");
if .        printf("j n'est pas strictement négatif");
}
```

C considère que la valeur 0 correspond à FAUX et que toute valeur non nulle est VRAIE.

B.3.13.c. Opérations de traitement de bits :

Ce sont les opérations qui permettent de mettre à 0 ou à 1 des zones de un ou plusieurs bits d'une donnée.

&	Et
|	Ou
^	Ou exclusif
<<	Décalage à gauche (suivi du nombre de décalages)
>>	Décalage à droite (suivi du nombre de décalages)
~	Complément à 1

Exemples :

```
int i,j,k;
i=2;
j=7;

k=i & j; ⇒ k=.
k=i | j; ⇒ k=.
k=i ^ j; ⇒ k=.
```

* Pour les nombres signés, les décalages préservent le bit de signe : décalage arithmétique

```
i= - 3;  /* i=1111 1111 1111 1101 */

k=i<<2; ⇒ k=.
```

: on fait entrer des '0' à droite et on garde le MSB (bit de signe).

```
i=-32768; /* i=1000 0000 0000 0000 */
k=i>>2; ⇒ i .
```

: Propagation du bit de signe

* Pour les nombres non signés, les décalages ne font jouer aucun rôle particulier au MSB : décalage logique.

```
unsigned i,k;
i=32768; /* i=.                    */
k=i>>2; ⇒ k = .
```

B.3.13.d. Opérations d'incrémentation et de décrémentation :

Dans le tableau qui suit, on suppose initialement que i=3 et j=15.

Opération	Exemple	Equivalent	Résultat	Commentaire
Pré incrémentation	i=++j;	.	.	L'incrémentation précède l'affectation
Post incrémentation	i=j++;	.	.	L'incrémentation suit l'affectation
Pré décrémentation	i=--j;	.	.	La décrémentation précède l'affectation
Post décrémentation	i=j--;	.	.	La décrémentation suit l'affectation

B.3.14. LES STRUCTURES DE CONTROLE :

B.3.14.a. Symbolique des algorigrammes

Quelques symboles utilisés dans la construction d'un algorigramme :

SYMBOLE	DESIGNATION	SYMBOLE	DESIGNATION
Symboles de traitement		Symboles auxiliaires	
	Symbole général Opération ou groupe d'opérations sur des données, instructions, pour laquelle il n'existe aucun symbole normalisé.		**Renvoi** Symbole utilisé deux fois pour assurer la continuité lorsqu'une partie de ligne de liaison n'est pas représentée.
	Sous-programme Portion de programme considérée comme une simple opération.		**Début, fin , interruption** Début, fin ou interruption d'un algorigramme.
	Entrée-Sortie Mise à disposition d'une information à traiter ou enregistrement d'une information traitée.		**Commentaire** Symbole utilisé pour donner des indications sur les opérations effectuées.
Symbole de test		Les différents symboles sont reliés entre eux par des lignes de liaisons.	
	Branchement Exploitation de conditions variables impliquant un choix parmi plusieurs.		
Sens conventionnel des liaisons			
Le sens général des lignes de liaison doit être : • De haut en bas • De gauche à droite Lorsque le sens général ne peut pas être respecté, des pointes de flèche à cheval sur la ligne indiquent le sens utilisé.			

B.3.14.b. Structures conditionnelles ou alternatives :

On utilise soit le terme d'alternative ou le terme de condition.

Alternative réduite :	**si (condition vraie)**	**if (condition vraie)**
Lorsqu'il n'y a qu'une seule instruction dans le bloc qui suit la condition, les parenthèses ne sont pas obligatoires (Cf exemple)	**{** **action1 ;** **action2 ;** **…** }	**{** **action1;** **action2;** **...** }

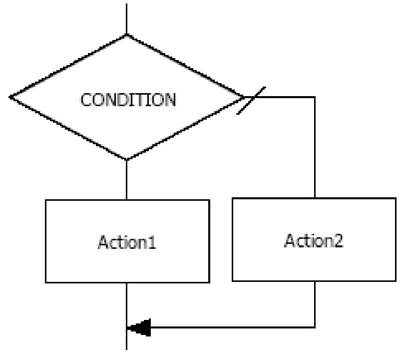
Alternative complète	**Si (condition vraie)** **{** **action1** **action2** **...** **}** **Sinon** **{** **action3** **action4** **...** **}**	**if (condition vraie)** **{** **action1;** **action2;** **...** **}** **else** **{** **action3;** **action4;** **...** **}**

Exemple :

```
void main()
{
if (i<=j) printf("i est plus petit que j");  /*Parenthèses pas obligatoires */
else
  {
  if (i<j) printf("i est plus grand que j");
else printf("i et j sont égaux");
  }
}
```

Remarquer les styles de présentation !

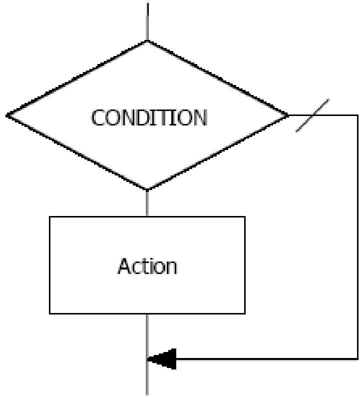
- si la condition est vérifiée seul le premier traitement est exécuté ;
- si la condition n'est pas vérifiée seul est effectué le second traitement.



Notation :	si condition alors action1 ; sinon action2 ; fsi ;

alternative réduite

La structure alternative réduite se distingue de la précédente par le fait que seule la situation correspondant à la validation de la condition entraîne l'exécution du traitement, l'autre situation conduisant systématiquement à la sortie de la structure.

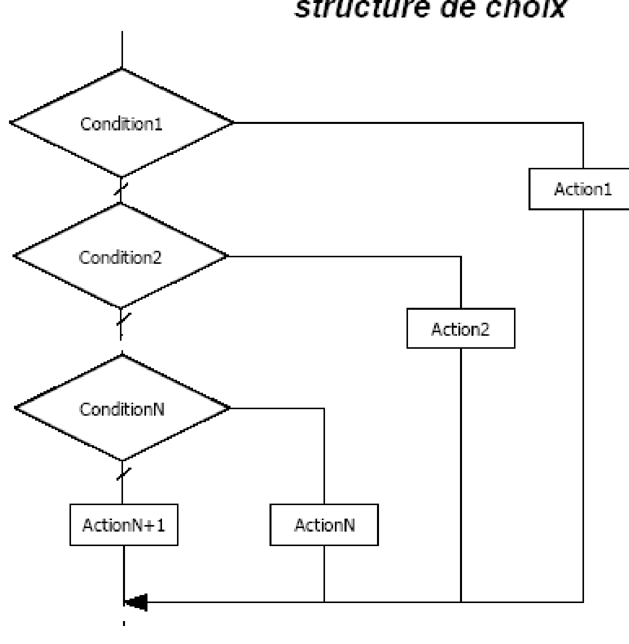


B.3.14.c. Sélection :

La structure de sélection permet d'aiguiller vers différentes instructions en fonction de la valeur d'une variable.

Sélection Si la variable prend la valeur 1, on exécute l'action 11 puis on sort. Si la variable prend la valeur 2, on exécute l'action 21 puis on sort. Si la variable ne prend aucune valeur de la liste, on exécute l'action 3 par défaut.	**selon que (variable)** **{** **valeur1 :** **action11;** **…** **sortir;** **valeur2 :** **action21;** **…** **sortir;** **…** **...Par défaut : action3;** **}**	**switch(variable)** **{** **case valeur1 :** **action11;** **...** **break;** **case valeur2 :** **action21;** **...** **break;** **...** **default : action3;** **}**

L'omission de "break" provoque l'exécution de l'option suivante.
Le sélecteur ne peut être que de type entier ou caractère.



Exemple :

```
void main()
{
int k;
scanf("%d ",&k);
switch(k)
  {
   case 0 : printf("Zéro");
            printf("k est nul");
            break;
    case 1: printf ("Un");
            break;
   default : printf("Autres valeurs");
  }
}
```

B.3.14.d. Structures répétitives :

Tant que Tant que la condition est vraie, on exécute en boucle les actions 1, 2, … Dés que la condition devient fausse, on sort de la boucle.	**tant que (condition vraie)** **{** **action1;** **action2;** **...** **}**	**while (condition vraie)** **{** **action1;** **action2;** **...** **}**

<table>
<tr><td>Répéter
On exécute en boucle les actions 1, 2, …, tant que la condition est vraie.
Dés que la condition devient fausse, on sort de la boucle.</td><td>Répéter
{
action1
action2
…
} tant que (condition vraie);</td><td>do
{
action1;
action2;
...
} while (condition vraie);</td></tr>
<tr><td>Pour
On réalise les initialisations prévues.
Tant que la condition est vraie, on réalise les actions du corps de la boucle. On réalise alors les actions de fin de boucle.</td><td>Pour (initialisation(s) ; condition vraie; action(s) de fin de boucle)
{
action1;
action2;
…
}</td><td>for (initialisation(s) ; condition vraie ; action(s) de fin de boucle)
{
action1;
action2;
...
}</td></tr>
</table>

- **Différence entre while (tant que) et do … while (répéter) :**

  - **while** (tant que) : la condition de boucle est testée avant d'exécuter le corps de la boucle.
    Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.
  - **do … while** (répéter … tant que) : le corps de boucle est exécuté. Puis la condition de boucle est testée. On exécute de nouveau le corps de boucle si la condition est vraie.
    Si la condition est fausse au départ, le corps de la boucle est exécuté une fois.

- Traduction d'une structure pour : Exemple :

<table>
<tr><td>

```
void main()
{
int i,s;

.
  {
  s=s+i;
  printf("i=%d  s=%d\n",i,s);
  }
}
```

</td><td>

```
void main()
{
int i,s;

.


{
  s=s+i;
  printf("i=%d  s=%d\n",i,s);
  .
  }
}
```

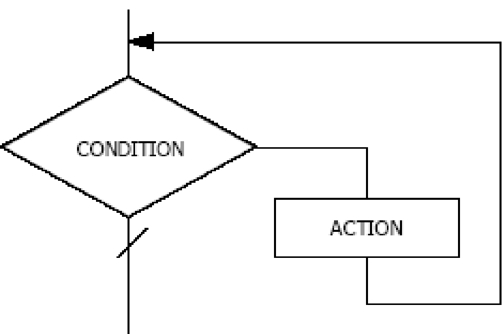
</td></tr>
</table>

structure TANT QUE ... FAIRE...

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

Par traitement on entend :

- soit une structure isolée,
- soit une succession d'instructions.

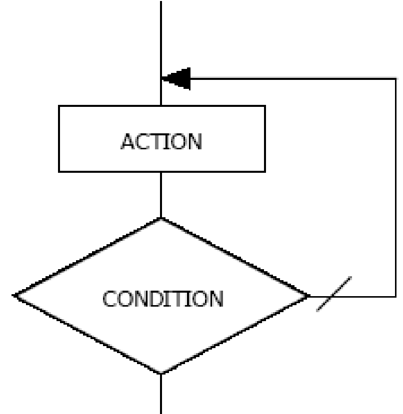


structure RÉPÉTER JUSQU'À

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

Par traitement on entend :

- soit une structure isolée,
- soit une succession d'instructions.



L'action est toujours exécutée au moins une fois.

B.3.14.e. Sortie de boucle :

Il est parfois nécessaire de sortir d'une boucle en cours d'exécution sans attendre que la condition de fin de boucle soit atteinte.
Cette structure est utilisée dans les structures Tant que, Répéter, Sélection.

Sortie de boucle	Sortir	break;

Exemple : Calculer le produit des entiers entre 1 et 10 tant que ce produit est inférieur à 1000, quel que soit le nombre d'entiers. Si l'entier est en dehors de la plage [0, 10], on arrête.

```
void main()
{
int i,p;

for (i=1, p=1; i<=10; i++)
    {
    p=p*i;
    printf("i=%d  p=%d\n",i,p);
.
            {
            printf("Limite atteinte : stop\n";
            .
            }
    }
```

B.3.15. POINTEUR

B.3.15.a. Définition

- Un pointeur contient **l'adresse** d'un autre *objet* manipulé par le programme. Cet *objet* peut être une variable, une fonction, …
- Un pointeur est déclaré par l'expression suivante : <type> *<nom du pointeur>;

***Exemple :* int *ptr;**

Cette expression déclare un pointeur appelé ptr qui pointe (ou contient l'adresse) d'un int.

- Un pointeur est non initialisé lors de sa déclaration : il pointe sur du *vide*.

Avant de s'en servir, il faut l'initialiser pour qu'il pointe sur quelque chose.

Exemple :

```
char c;
char *ptr;      /* Déclaration du pointeur sur un char (1 octet) */
ptr=&c;         /* Opérateur & donne l'adresse de la variable. Ne pas confondre avec le ET
                logique */
*ptr=0x04;      /* On écrit la valeur 4 en hexa dans la variable dont l'adresse est contenue dans
                ptr */
```

Remarque : Pour la programmation du 68HC11 en langage C, on travaille avec ses registres.
Il faut donc utiliser leurs adresses placées dans des pointeurs.
On procède la manière suivante :

```
unsigned char *porta;   /*Déclaration d'un pointeur sur un unigned char si on travaille en non
                        signé ou char *porta si on travaille en signé*/

porta = (unsigned char)0x1000;
```

On affecte au pointeur porta la valeur **numérique** de l'adresse au moyen d'une opération de *cast* qui, dans ce cas, force la transformation d'une simple valeur numérique entière en un pointeur sur un unsigned char.

Rassurez-vous toutes ces opérations barbares sont déjà réalisées dans un fichier d'entête ***hc11.h*** *que l'on inclut au début de nos programmes*

B.3.15.b. Utilisation

Attention : l'erreur fréquente avec les pointeurs est de confondre le pointeur avec la variable pointée. Un pointeur ne réserve aucune place pour la variable pointée.

Exemples :

```
char k,n;
char *cptr;
cptr=&n         /* Opérateur & donne l'adresse de la variable. Ne pas confondre avec le ET
                logique */
*cptr=2;        /* Affectation */
k=*cptr+1;      /* Opération algébrique sur la variable pointée : k=3 */
cptr=cptr+1;    /* Opération algébrique sur le pointeur lui-même : cptr =0x1008 car cptr pointe
                sur un char (1 octet) */

long *lptr;
lptr=0x1000;
lptr =lptr+2;   /* lptr = 0x1000 + 2*4 = 0x1008 car un long occupe 4 octets en mémoire */
```

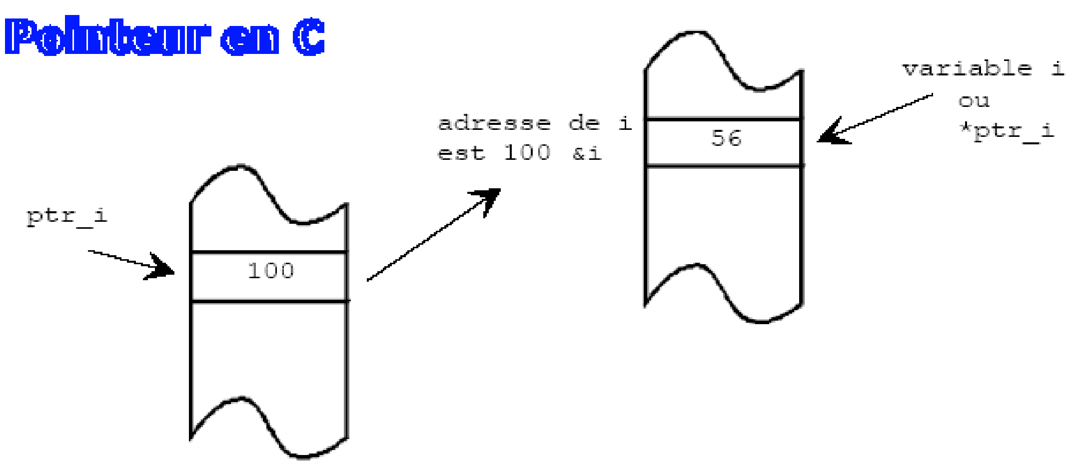
123456				00abcdef				0011223344				…			
1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	100a	100b	100c	…		

Prochaine   Prochaine

La ligne grise représente les adresses. Les valeurs sont hexadécimal.

B.3.15.c. Exemples

Représentation mémoire.



Ici `ptr_mes` sera augmenté de 5 octets car le type `char` réserve un octet.

Attention: `*(ptr_mes+5)` **est complétement différent de** `*ptr_mes+5`.
`*(ptr_mes+5)` **c'est la valeur de l'objet pointé par** `ptr_mes+5`.
`*ptr_mes+5` **c'est l'addition de la valeur de l'objet pointé par** `ptr_mes` **et de 5.**

Exemple:

```
#include <stdio.h>

/* Déclaration et initialisation d'une chaîne de caractères */
char mes[10]="Morgan";

/* Déclaration d'un pointeur de caractère */
char *ptr_mes;

main()
{
     /* Initialisation de ptr_mes sur l'adresse de mes */
     ptr_mes=mes; /* Equivalent à ptr_mes=&mes[0] */

     /* Affichage du dernier caractère de mes */
     printf("Le dernier caractère de mes est %c\n",*(ptr_mes+5));

     /* Affichage du caractère ayant pour code ASCII(M+5) -> R */
     printf("Le caractère de code ASCII(M+5) %c",*ptr_mes+5);
}
```

- **Additionner ou soustraire une constante `n`, qui permet au pointeur de se déplacer de `n` éléments dans un tableau. Le pointeur sera augmenté ou diminué de `n` fois le nombre d'octet(s) du type de variable.**

Exemple:

```
#include <stdio.h>
/* Déclaration et initialisation d'un tableau d'entiers */
char tab[4]={4,3,2,-5};
/* Déclaration d'un pointeur d'entier */
char *ptr_tab;

main()
{
    /* Initialisation de ptr_tab sur l'adresse de tab */
    ptr_tab=tab; /* Equivalent à ptr_tab=&tab[0] */

    /* Déplacement de 3 éléments de ptr_tab */
    ptr_tab=ptr_tab+3;

    /* Affichage du quatrième élément de tab */
    printf("tab[3]= %d\n",*ptr_tab);

    /* Déplacement de 2 éléments de ptr_tab */
    ptr_tab-=2; /* Equivalent à ptr_tab=ptr_tab-2; */

    /* Affichage du deuxième élément de tab */
    printf("tab[1]= %d\n",*ptr_tab);
}
```

Attention: Le changement de valeur du pointeur doit-être contrôlé. Si le pointeur pointe sur une zone mémoire non définie, la machine peut se planter à tout moment. Pour l'exemple ci-dessus le pointeur `ptr_tab` doit être compris entre `&tab[0]` et `&tab[3]`.

B.3.15.d. Pointeur et tableau

Les notions de tableau et de pointeurs sont liés.
Un tableau est géré comme un pointeur constant que l'on ne peut pas modifier et qui contient l'adresse de la 1ère case du tableau.

Exemple :

```
int i;
int t1[10];   /* déclaration d'un tableau de 10 int. t1 représente l'adresse de la 1ère case du tableau */
int t2;       /* déclaration d'un pointeur sur un int */

t2=t1;        /* t2 est initialisé avec l'adresse de la 1ère case du tableau t1 */

for (i=0; i<10; i++,t2++)
   {
   *t2=5*i;        /* On écrit dans chaque case du tableau une valeur à l'aide du pointeur t2 */
   printf("%d ";t1[i]);  /* On affiche chaque valeur à l'aide de la variable indicée t1 */
   }
```

C. Généralités : les microprocesseurs

C.1. Le microprocesseur : c'est quoi ?

Le **processeur** (**CPU**, pour *Central Processing Unit*, soit *Unité Centrale de Traitement*) est le cerveau de l'ordinateur. Il permet de manipuler des informations numériques, c'est-à-dire des informations codées sous forme binaire, et d'exécuter les instructions stockées en mémoire.

Le premier **microprocesseur** (Intel 4004) a été **inventé en 1971**. Il s'agissait d'une unité de calcul de 4 bits, cadencé à 108 kHz. Depuis, la puissance des microprocesseurs augmente exponentiellement.

Comment peut-il exécuter une grande variété de fonctions ?

parce qu'il est programmable. Il exécute une suite d'instructions qui peut être modifiée à souhait.

Pourquoi son domaine d'application est-il si étendu ?

parce qu'on peut le coupler, via des interfaces d'entrée et de sortie, à une grande variété d'organes extérieurs (fig. 1).

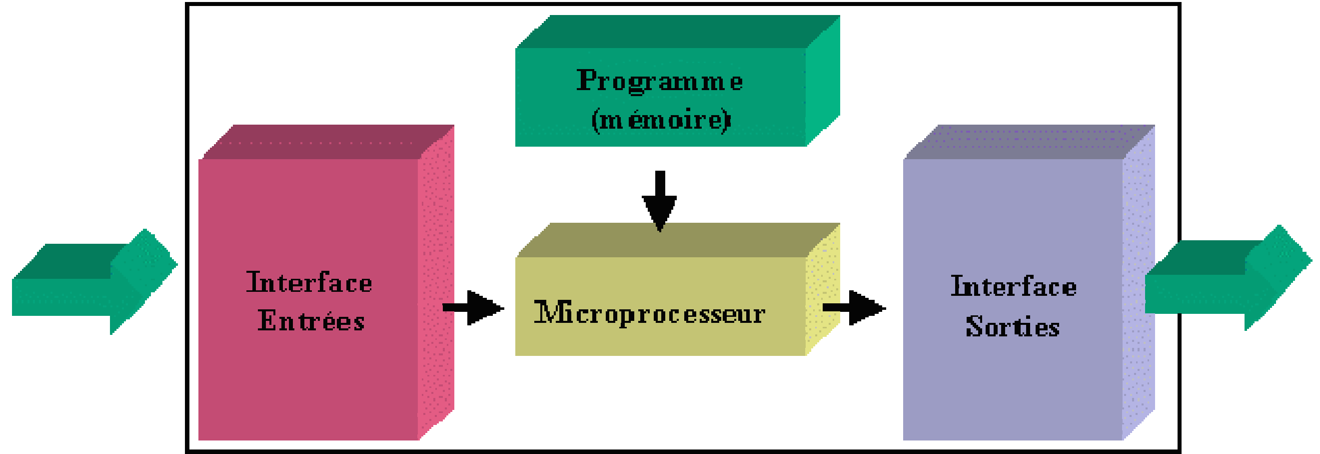


fig. 1 Utilisation d'un microprocesseur

L'activité du microprocesseur est alors de répondre aux entrées pour produire des sorties, d'une façon déterminée par une . (le programme) qui est stockée dans une .

Conclusion

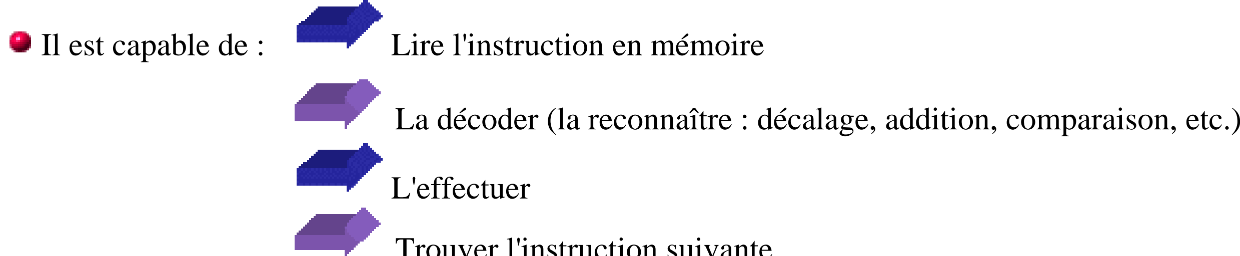
Un microprocesseur est constitué d'un ou plusieurs circuits LSI ou VLSI qui réalisent des fonctions de traitement.

On l'appelle également unité centrale de traitement (CPU : central processing unit).

Un micro-ordinateur est un ordinateur construit autour d'un ou plusieurs microprocesseurs (fig. 1).

C.2. Qu'est-ce qu'une Unité de Traitement ?

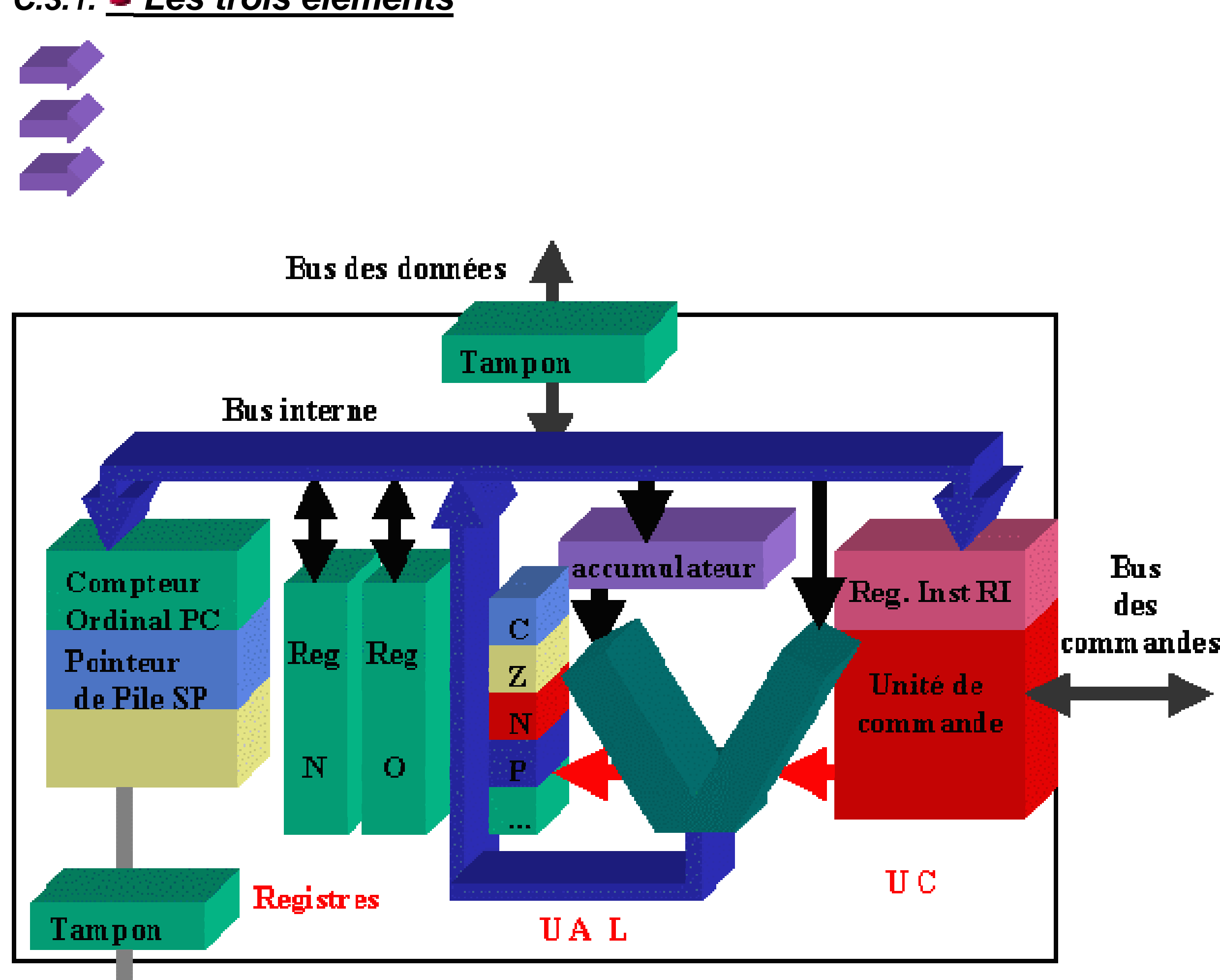
C'est l'organe qui effectue la séquence d'instructions stockée en mémoire.



- Il est capable de :
  - Lire l'instruction en mémoire
  - La décoder (la reconnaître : décalage, addition, comparaison, etc.)
  - L'effectuer
  - Trouver l'instruction suivante

C.3. Composants d'un microprocesseur

C.3.1. Les trois éléments



Exemple d'architecture d'un microprocesseur

Ces trois éléments sont reliés entre eux par un bus interne, celui-ci permettant les échanges de données entre les différentes parties du.

C.3.2. *L'unité de commande*

Elle permet de "séquencer" le déroulement des instructions. Elle effectue la recherche en mémoire de l'instruction, le décodage, l'exécution et la préparation de l'instruction suivante. L'unité de commande élabore tous les signaux de synchronisation internes ou externes (bus des commandes) au microprocesseur

C.3.3. *L'unité arithmétique et logique (UAL)*

C'est l'organe qui effectue les :
arithmétiques : addition, soustraction, multiplication, ...
logiques : et, ou, non, décalage, rotation, ...

Deux registres sont associés à l' : l'accumulateur et le registre d'état.

L'accumulateur (nommé : A)

C'est une des deux entrées de l'UAL. Il est impliqué dans presque toutes les opérations réalisées par l'UAL. Certains constructeurs ont des microprocesseurs à deux accumulateurs (Motorola : 6800)

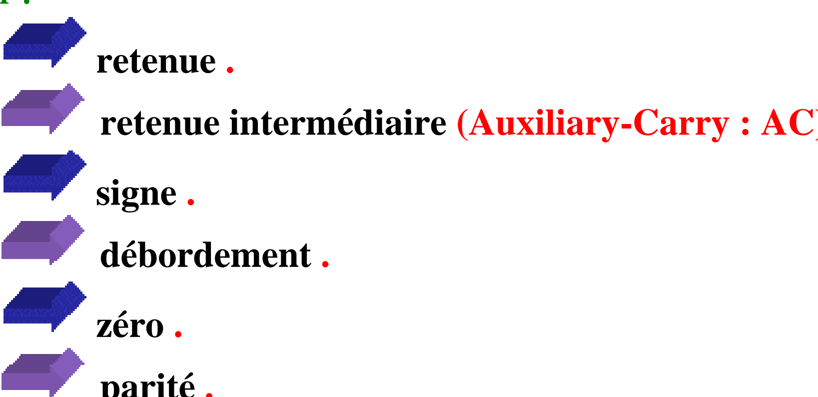
Exemple : A étant l'accumulateur et B un registre, on peut avoir : A+BÄ A (ADD A,B : addition du contenu du registre A avec celui du registre B, le résultat étant mis dans A)

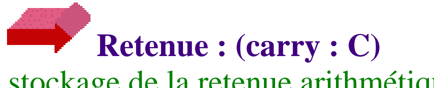
Le registre d'état (Flags : F)

A chaque opération, le microprocesseur positionne un certain nombre de bascules d'état. Ces bascules sont appelées aussi indicateurs d'état ou drapeaux (status, flags).

Par exemple, si une soustraction donne un résultat nul , l'indicateur de zéro (Z) sera mis à 1. Ces bascules sont regroupées dans le registre d'état

On peut citer comme indicateur :





Retenue : (carry : C)

stockage de la retenue arithmétique

Exemple: addition de nombres binaire sur 8 bits

	11111100		
+	**10000010**	**FCH** + **82H**	= **17EH**
carry : **1** =	**01111110**	(252)10 + **(130)10**	= **(382)10**

La bascule C (.) sert aussi à capter le bit expulsé lors d'une opération de décalage ou de rotation

exemple :

décalage à gauche d'un bit de cet octet : 10010110
la « carry » recueille le 1 du bit de poids fort carry : 1 00101100

Retenue intermédiaire : (Auxiliary Carry : AC)

Sur les opérations arithmétiques, ce drapeau signale une retenue entre groupes de 4 bits (Half-byte : demi-octet) d'une quantité de 8 bits.

Signe: (S)

Cette bascule est mise à 1 lorsque le résultat de l'opération est négatif (bit de plus fort poids du résultat à 1)

Débordement : (overflow : O)

Cet indicateur est mis à 1, lorsqu'il y a un dépassement de capacité pour les opérations arithmétiques en complément à 2. Sur 8 bits, on peut coder de -128 (1000 0000) à +127 (0111 1111).

104	**0110 1000**	**- 18**	**1110 1110**
+ **26**	+ **0001 1010**	**- 118**	**1000 1010**
=130	= **1000 0010 (-126)**	**-136**	**0111 1000 (120) avec C=1**

L' . est une fonction logique (OU exclusif) de la retenue (C) et du signe (S).

Zéro : (Z)

Cette bascule est mise à 1 lorsque le résultat de l'opération est nul.

Parité : (P)

Cette bascule est mise à 1 lorsque le nombre de 1 de l'accumulateur est pair.

Remarque

La plupart des instructions modifient le registre d'état

exemple :

ADD A, B positionne les drapeaux :
OR B, C (B ou C -> B) positionne S, Z, P tandis que
MOV A, B (Move, Transférer le contenu de B dans A) n'en positionne aucun

C.3.4. *Les registres*

Deux type de registres : les registres d'usage général, et les registres d'adresses (pointeurs)

Les registres d'usage général

Ce sont des mémoires rapides, à l'intérieur du microprocesseur, qui permettent à l'UAL de manipuler des données à vitesse élevée. Ils sont connectés au bus données interne au microprocesseur.

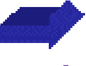
L'adresse d'un registre est associée à son nom (on donne généralement comme nom une lettre) A,B,C...

Exemple : MOV C,B : transfert du contenu du registre "d'adresse" B dans le registre "d'adresse" C

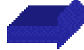
Les registres d'adresses (pointeurs)

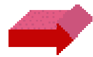
Ce sont des registres connectés sur le bus adresses.

On peut citer comme registre:

 Le compteur ordinal (pointeur de programme PC)

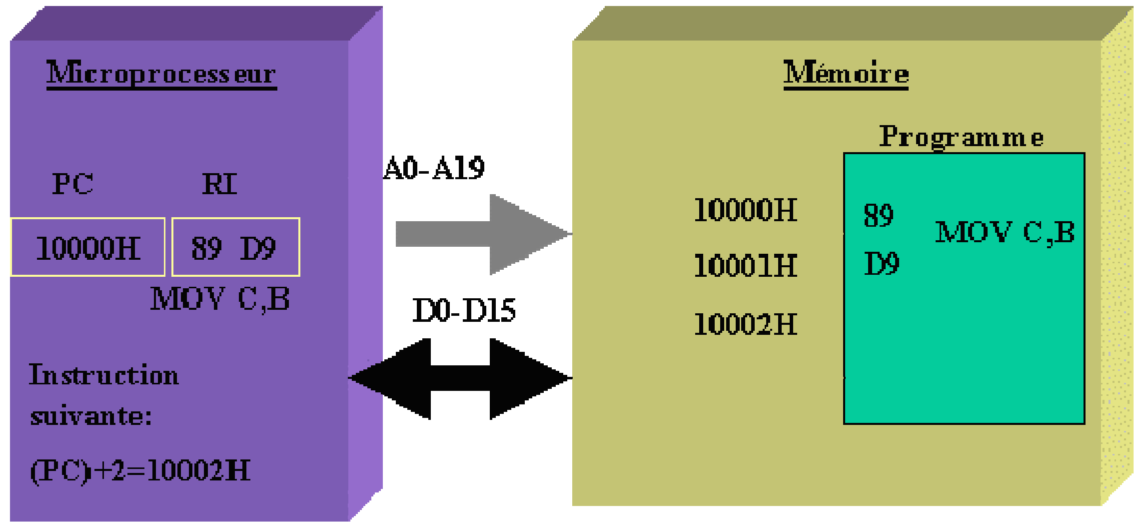
Le pointeur de pile (stack pointer SP)

 Les registres d'index (index source SI et index destination DI)

 **Le compteur ordinal (pointeur de programme PC)**

Il contient l'adresse de l'instruction à rechercher en mémoire. L'unité de commande incrémente le compteur ordinal (PC) du nombre d'octets sur lequel l'instruction, en cours d'exécution, est codée. Le compteur ordinal contiendra alors l'adresse de l'instruction suivante.

exemple : (PC)=10000H ; il pointe la mémoire qui contient l'instruction MOV C,B qui est codée sur deux octets (89 D9H) ; l'unité de commande incrémentera de deux le contenu du PC : (PC) = 10002H (la mémoire sera supposée être organisée en octets).



Compteur de Programme (PC)

Les registres d'adresses (pointeurs)

Le pointeur de pile (stack pointer SP)

➡Il contient l'adresse de la pile. Celle-ci est une partie de la mémoire, elle permet de stocker des informations (le contenu des registres) relatives au traitement des interruptions et des sous-programmes.

➡La pile est gérée en **LIFO** : (Last IN First Out) dernier entré premier sorti. Le fonctionnement est identique à une pile d'assiette

➡Le pointeur de pile SP pointe le haut de la pile (31000H fig. 1), il est décrémenté avant chaque empilement, et incrémenté après chaque dépilement.

➡Il existe deux instructions pour empiler et dépiler : PUSH et POP.

➡exemple: PUSH A empilera le registre A et POP A le dépilera.

Sur la figure 1, on va montrer le fonctionnement de la pile lors d'instructions comme PUSH et POP.

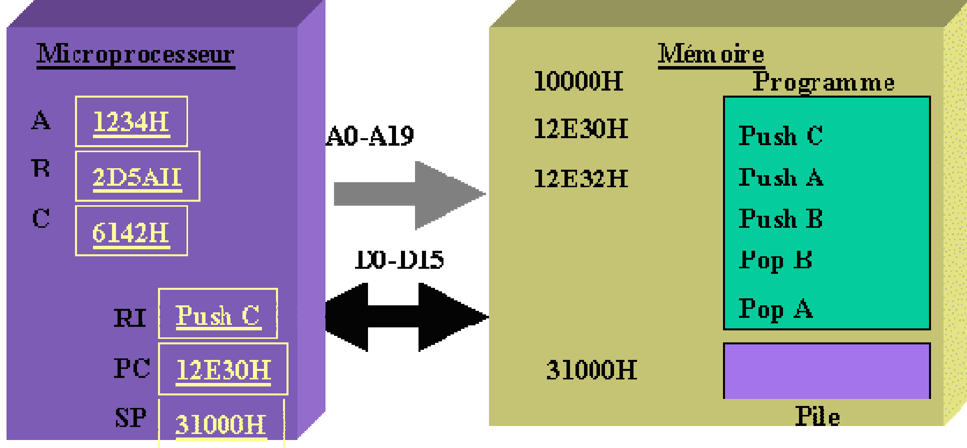


Fig. 1 Sauvegarde sur la pile

Question?

Que se passera-t-il durant l'exécution du programme commençant en 12E30H? Que vaudra SP et que contiendra la pile à cette adresse, à la fin du programme?

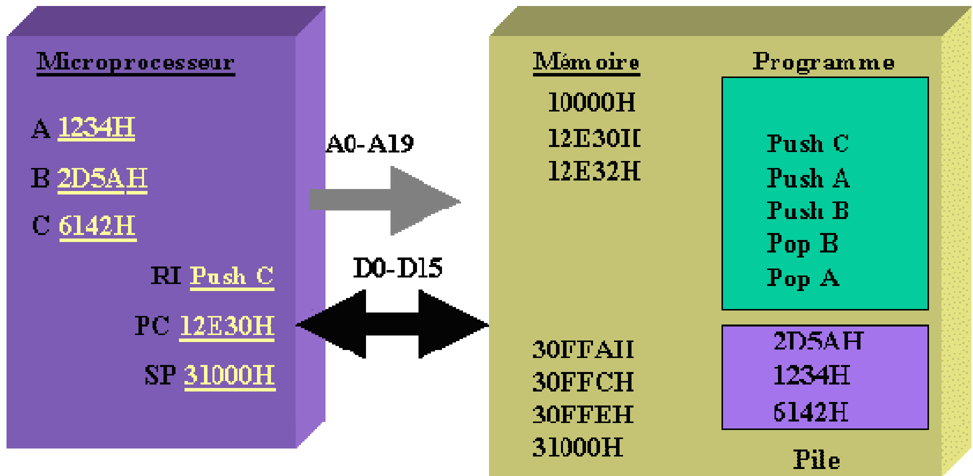
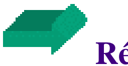


Fig. 2 exercice sur la pile

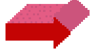


Réponse.

Le programme commence par sauvegarder le contenu de C dans la pile (PUSH C). Pour cela (SP) est décrémenté de deux((SP)=31000H-2=30FFEH), puis on effectue l'écriture de (C) dans la mémoire à l'adresse (SP) : (30FFEH) = 6142H.

Pour PUSH A on obtient : (30FFCH)=1234H, et pour PUSH B : (30FFAH)=2D5AH.

Pour l'instruction POP B, ((SP)) est chargé dans le registre B ((SP)=30FFAH ; (B)=2D5AH) puis (SP) est incrémenté de deux ((SP)= 30FFAH+2=30FFCH). Enfin, pour POP A on obtient : (A)=1234H et (SP)=30FFCH + 2 = 30FFEH

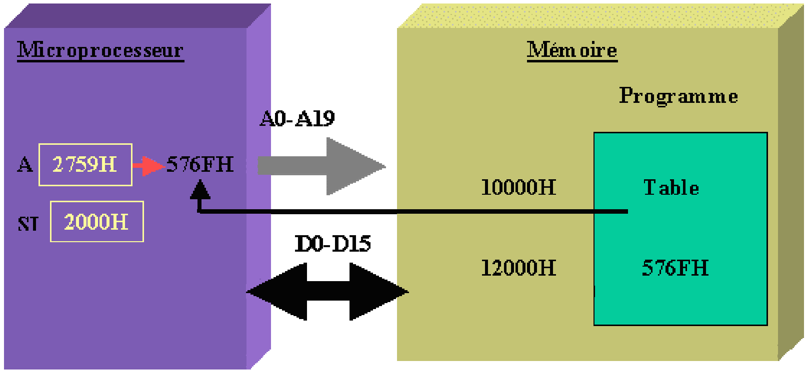


Les registres d'index (index source SI et index destination DI)

Les registres d'index permettent de mémoriser une adresse particulière (par exemple : début d'un tableau).

Ces registres sont aussi utilisés pour adresser la mémoire de manière différente. C'est le mode d'adressage indexé.

exemple :
MOV A,[SI+10000H] place le contenu de la mémoire d'adresse 10000H+le contenu de SI, dans le registre A.



exemple sur le registre d'index

C.4. Les BUS

C.4.1. Définition

Le Bus est un ensemble de fils électriques (cuivre) où apparaît une information binaire (0 ou 1) c'est à dire (0V ou 5V) sur chaque fil.

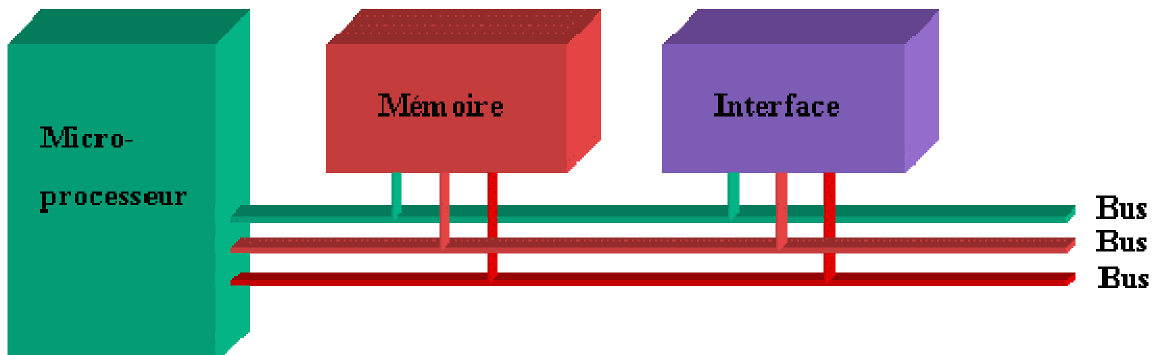


Fig. 1 les bus

Chaque **bus** a une fonction particulière

C.4.2. Les trois Bus

Le bus de données

Le bus d'adresses

Le bus de commandes

C.4.2.a. Bus des données

Il permet de véhiculer des données du microprocesseur vers un composant ou d'un composant vers le microprocesseur. Il est donc bidirectionnel. Le nombre de fils de ce bus varie suivant les microprocesseurs (8 / 16 / 32 / 64 bits). Dans la littérature, les différents fils de ce bus sont appelés D0, D1, ..., Dp-1, si le bus a "p" fils.

C.4.2.b. Bus des adresses

La mémoire est composée de nombreuses cases mémoires. Chaque case est repérée par une adresse. Lorsque le microprocesseur veut, par exemple, lire une case, il doit indiquer à quelle adresse elle se trouve. Il met cette adresse sur le bus des adresses. La case mémoire reconnaît alors son adresse et met sur le bus données son contenu.

exemple : Bus adresses 16 bits - données sur 8 bits

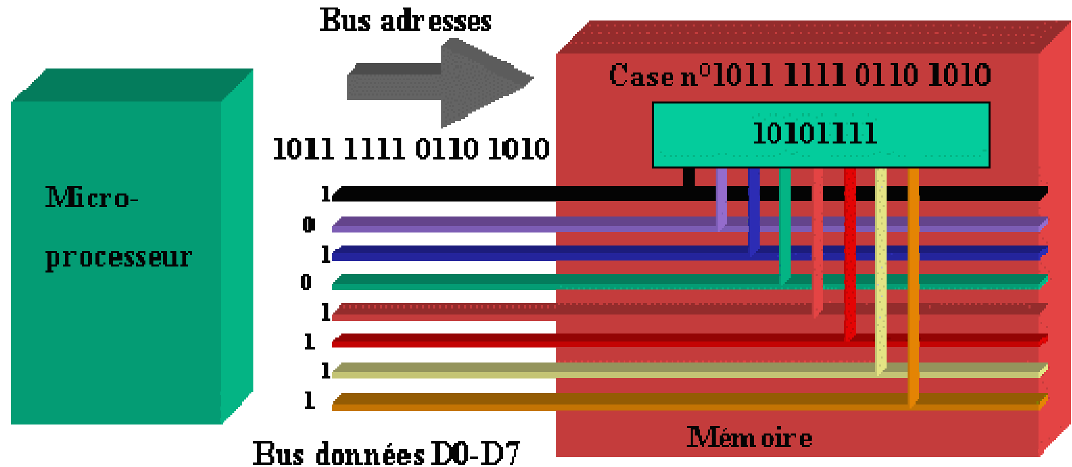


Fig. 2 bus adresses

Dans l'exemple précédent, le microprocesseur écrit la donnée 10101111 dans la case mémoire d'adresse 1011 1111 0110 1010.

Le bus adresses est unidirectionnel : du microprocesseur vers les autres composants.
Il se compose de 16 à 32 fils suivant les microprocesseurs que l'on nomme A0, A1, ..., An-1.

16 bits		adressage de 2^{16}		64x1024 mots = 64 Kmots
20 bits		adressage de 2^{20}		1024x1024 mots = 1Mmots
32 bits		adressage de 2^{32}		4096x1024 x1024 mots = 4 Gmots

C.4.2.c. Bus des commandes

Le bus des commandes est constitué d'un ensemble de fils de "commandes", permettant la synchronisation et biensûr la commande des boîters mémoires et entrées/sorties par le microprocesseur.

Dans le cas précédent, la cellule mémoire doit savoir à quel instant elle doit mettre son contenu sur le bus données. Pour cela, le microprocesseur possède une broche appelée Read ($\overline{RD}$) qu'il met à 0 (0v) lorsque la cellule doit agir. De même, lors d'une écriture du microprocesseur vers la cellule, il met sa broche Write ($\overline{WR}$) à 0 (0V). Les signaux RD et WR sont des signaux de synchronisation, de contrôle, de commande. Ils sont reliés aux autres composants par un bus : le bus des commandes. Celui-ci comporte d'autres signaux de commandes.

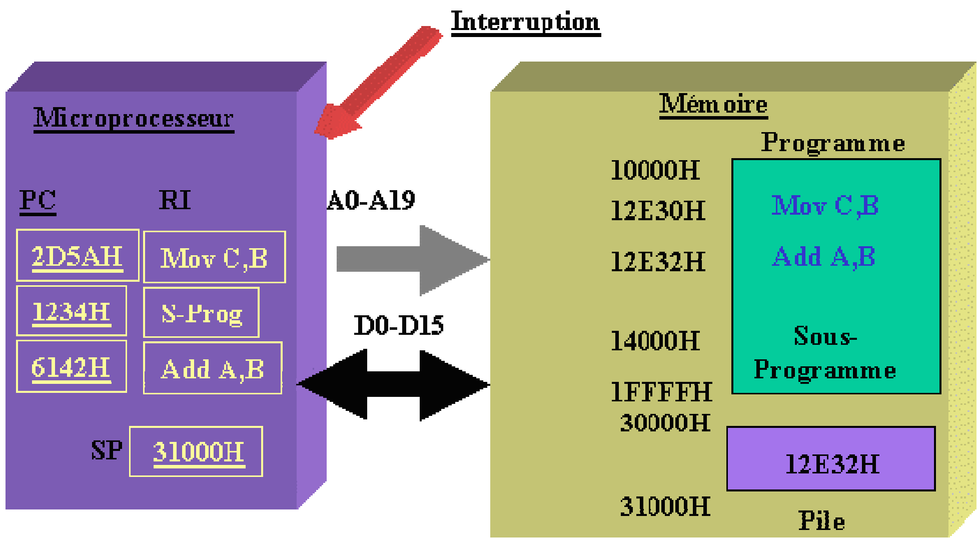
C.5. L'interruption

C.5.1. Niveau Hard

Le microprocesseur possède des broches spécialisées pour les interruptions. Si un composant extérieur au microprocesseur veut l'interrompre, il lui suffit de changer la valeur d'une de ces broches (0 ➡ 1 ou 1 ➡ 0 suivant les broches). Le programme qui traite l'interruption (appelé sous-programme d'interruption) a été placé en mémoire par le concepteur à une adresse connue du microprocesseur.

C.5.2. Niveau Soft

Lors d'une interruption, le microprocesseur exécute l'instruction en cours, puis charge le (PC) (pointeur de programme) à l'adresse de la première instruction du sous-programme d'interruption. De manière à pouvoir terminer le programme en cours et revenir à PC +m (adresse de l'instruction suivante), cette adresse est mémorisée dans la pile.



Sauvegarde sur la pile

Dans notre exemple, l'adresse de l'instruction en cours est 12E30H. Le microprocesseur exécute l'instruction MOV C,B. Il s'aperçoit de la demande d'interruption.

Avant de la traiter, il termine l'instruction, sauvegarde l'adresse de retour dans la **pile**, c'est à dire l'adresse de l'instruction suivante (dans notre exemple 12E32H). Et enfin il se branche à l'adresse du sous-programme d'interruption.

A la fin de celui-ci, nous pourrons revenir au programme principal en reprenant l'adresse sauvegardée dans la **pile**, grâce à une instruction de retour (interrupt return IRET).

Le sous-programme d'interruption peut utiliser les registres et donc les modifier. Pour éviter tous problèmes, on sauvegarde sur la **pile** (empiler: PUSH) tous les registres qui contiennent une donnée.

Cet empilement s'appelle : sauvegarde du "contexte". Le microprocesseur se charge de sauvegarder le (PC) et aussi le registre d'état F. Les autres registres doivent être gérés par le programmeur du micro-ordinateur.

C.6. Familles de processeurs

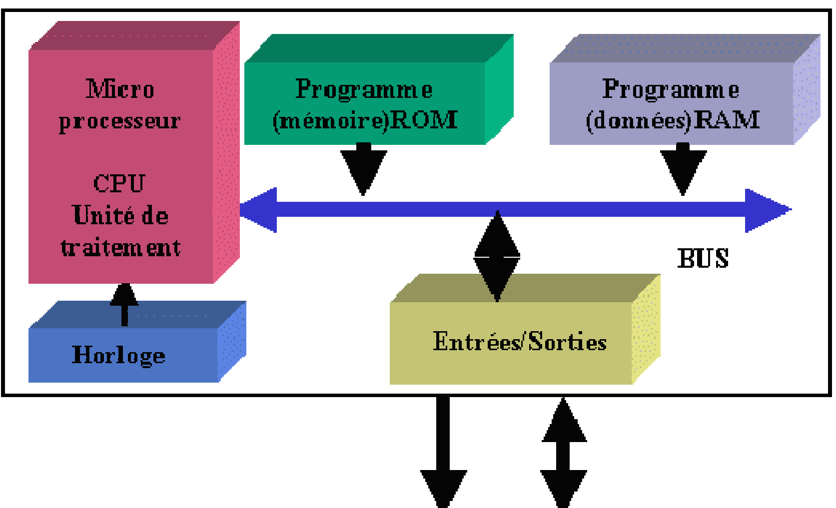
Chaque type de processeur possède son propre jeu d'instruction. On distingue ainsi les familles de processeurs suivants, possédant chacun un jeu d'instruction qui leur est propre :

- Intel 80x86 : le « x » représente la famille. On parle ainsi de 386, 486, 586, 686, etc.
- ARM
- IA-64
- MIPS
- **Motorola** 6800
- PowerPC
- SPARC
- Zilog
- ...

Cela explique qu'un programme réalisé pour un type de processeur ne puisse fonctionner directement sur un système possédant un autre type de processeur, à moins d'une traduction des instructions, appelée **émulation**. Le terme « **émulateur** » est utilisé pour désigner le programme réalisant cette traduction.

C.7. Architecture d'un micro-contrôleur

L'architecture d'un micro-contrôleur est la suivante :



Constituants

Un BUS : ensemble de fils sur lesquels circulent toutes les informations échangées entre les circuits constituant le micro-ordinateur.

Le Microprocesseur.

Deux types de **Mémoires**

- mémoire **ROM** (Read Only Memory) ou mémoire morte, accessible uniquement en lecture, dont le contenu n'est pas perdu en cas de coupure d'alimentation (mémoire non volatile). Ce type de mémoire est utilisé pour stocker des informations qui ne seront pas modifiées (par exemple le programme).
- mémoire **RAM** (Random Acces Memory) ou mémoires vives, accessibles en lecture et en écriture dont le contenu est perdu en cas de coupure de courant (mémoire volatiles). Ce type de mémoire est utilisé pour stocker des informations temporaires.

une Horloge ("cadencement" et synchronisation de l'ensemble)

D. Le HC12 de Motorola

D.1. Avantages / inconvénients : Les 68HC12 et 68HC912

D.1.1. Donnons tout d'abord les principales caractéristiques de la famille 68HC12 :

- Les références du 68HC12 sont assez compliquées. En fait, le terme 68HC12 (ou M68HC12 ou HC12) désigne une famille. Dans la réalité, tous les 68HC12 ont une **FLASH intégrée** et la famille devient 68HC912. MOTOROLA a, depuis peu, supprimé le "68" et la référence devient alors HC912. Il existe les déclinaisons suivantes de cette famille : HC912B32, HC912BC32, HC912D60A, HC912DG128A. Dans la suite de ce document, nous continuerons à utiliser l'ancienne référence 68HC12 ou HC12.

- La principale différence entre le 68HC11 et le 68HC12 est la présence d'une mémoire FLASH dans ce dernier microcontrôleur. Cette mémoire est importante et permet de programmer directement en langage C ou C++.

- Le microcontrôleur 68HC12 est en quelque sorte un super 6809. Son jeu d'instructions et ses modes d'adressages ressemblent à s'y méprendre à ceux du 6809. En prime, il a emprunté quelques instructions au 68000 : MOVW, MOVB.... Ceux connaissant le 6809 ne seront donc pas dépaysés par le 68HC12 et auront même le plaisir de retrouver les LEAX qui leur étaient si utiles.

- Le 68HC12 accepte les syntaxes du 68HC11. Le passage du HC11 au HC12 s'effectuera généralement sans grandes difficultés car il suffit de ré-assembler les anciens programmes 68HC11. Seuls, certains modes d'adressage tel que le 10,X+ ou certaines instructions comme la MOVW nécessiteront une petite étude pour ceux qui ne viennent pas du 6809 ou du 68000.

- Le 68HC12 dispose d'un jeu d'instructions mathématiques (16x16, 32/16, EMACS, FDIV...) qui simplifie les calculs.

- Le 68HC12 dispose également de nouvelles instructions sur la logique floue (fuzzy logic). Il s'agit d'une nouvelle approche dont le principe est expliqué brièvement ci-après. Parfois, les variables ne sont pas 0 ou 1 mais sont comprises entre 0 et 1. Une variable peut devenir de moins en moins vraie tandis qu'une autre adjacente de plus en plus vraie. Ces variables suivent certaines règles définies par le programmeur. Cette méthode est empruntée aux systèmes experts sans pour autant en garder la complexité. Par exemple, dans un sèche-linge contrôlé par un microcontrôleur, le linge n'est jamais complètement humide (niveau 0) ou sec à 100% (niveau 1) mais plus ou moins sec. Le problème devient complexe lorsque plusieurs règles analogiques interviennent, comme par exemple la température du linge. Dans des cas comme celui-ci, la machine s'arrête à de l'intersection de deux droites (intersection de l'hypoténuse de deux triangles rectangles mis en opposition, l'un pour l'humidité, l'autre pour la température). Le 68HC12 permet de programmer ces conditions sous la forme de "règles d'inférence" qui suivent les algorithmes sur la logique floue.

- L'adressage est, comme tous les autres microcontrôleurs de la gamme, sur 64 Ko. . Cependant, il est prévu une pagination qui permet de dépasser cette barrière de 64 Ko.. A noter également que l'ASIC complémentaire au 68HC12 et utilisé sur les cartes DiaVerre permet aussi de faire de la pagination.

- Le processeur interne du 68HC12 est en 16 bits réels et les octets pairs/impairs des mémoires sont alignés automatiquement. Au niveau extérieur, le 68HC12 peut se mettre en **monochip, en étendu 8 bits ou en étendu 16 bits.**

- Le 68HC12 offre quatre canaux **PWM** (pulse width modulator) d'une grande utilité pour la commande de moteurs. Ces sorties, associées à un CI du genre L293 ou L297, permettent de réaliser rapidement des variateurs de vitesse et asservissements divers. Le 68HC12 réalise automatique toute cette gestion de rapport cyclique qui, avec les microcontrôleurs du genre 68HC11, s'effectuait par timer sous interruptions de programmes.

- La grande nouveauté du 68HC12 est incontestablement le **bus CAN V2.0** qui, en quelques années, est devenu le standard du réseau industriel. Ce bus, mis au point initialement par BOSCH pour les automobiles, s'est très vite généralisé, notamment en raison de sa fiabilité. Il en résulte que le bus CAN se trouve aujourd'hui dans de nombreux domaines autres que celui de l'automobile. Ce bus bifilaire, de conception moderne, extrêmement fiable et dont la vitesse est de 1 Mbps, remplace les autres bus bifilaires lorsqu'une mise en réseau à la fois performante et fiable est requise. Aujourd'hui, il existe une telle demande au niveau de l'industrie pour le bus CAN que cette technique est devenue incontournable pour les formations BAC+.

- Tous les microcontrôleurs se disent supporter les langages de haut niveau (HLL) mais il s'agit souvent plus d'un argument commercial qu'une réalité. Le support d'HLL nécessite la réentrance et une **gestion correcte de la pile**. Le 68HC12, tout comme le 6809, a été conçu dans cette optique, ce qui n'est pas le cas des autres microcontrôleurs bas de gamme comme le 68HC11. En effet, lors de l'appel d'une fonction HLL, les paramètres sont stockés dans la pile S. La fonction appelée doit pouvoir les récupérer facilement. Pour cela, il lui faut des modes d'adressage adéquats, et en particulier l'indexé par S avec offset, avec pré/post incrémentation ou décrémentation (ex.: nn,S++). De même, la création d'espaces dynamiques pour les variables nécessite la fameuse LEAS xx,S. Même problème pour la réentrance. Avec les microcontrôleurs non adaptés aux HLL, le compilateur "se débrouille" mais au prix d'un "source-assembleur" extrêmement lourd puisqu'il faut 5 à 10 instructions là où le 68HC12, 6809 ou 68000 n'en nécessite qu'une seule.

- Le port **BDM** du 68HC12 permet de faire du débogage en temps réel grâce à une sonde très faible coût. Avec ce système il n'est pas nécessaire de disposer d'un émulateur onéreux. Le débogage peut également se faire avec **les langages de haut niveau tel que le C**.

PK-HCS12C32
STARTER KIT pour Motorola MC9S12C32 (USB).

PK-HCS12E128
STARTER KIT pour Motorola MC9S12E128 (USB).

Outil temps réel. Debogueur «In-Circuit" (ICD) interface BDM*.
Environnement Metrowerks CodeWarrior IDE.
* *Background Debug Module*

- Le 68HC12 est particulièrement bien conçu pour la CEM. En milieu scolaire ce point n'est pas important mais dans l'industrie, il est tout à fait capital. En effet, une société ne peux exporter du matériel s'il n'est pas conforme aux normes CE dont la CEM. En milieu industriel, il est donc capital de porter une attention particulière à la CEM, d'autant qu'une carte conçue dans cette optique est beaucoup plus fiable qu'une carte traditionnelle car les signaux sont plus propres.

- Enfin, le 68HC12 emprunte au 68HC11 de nombreux modules (multiples timers, CAN...) avec des performances supérieures. Par exemple, les convertisseurs A/D sont sur 10 bits au lieu de 8, et les **TIMERS** nettement plus sophistiqués.

D.1.2. Inconvénients du 68HC12 :

- La vitesse d'horloge est élevée (...25 MHz). Les fronts sont donc très raides et produisent des harmoniques de rang élevé qui entrent en résonance avec les pistes ou fils si le 68HC12 est utilisé en mode étendu (en mode monochip, il existe une possibilité de lisser les fronts en vue d'une meilleure immunité CEM). Si le 68HC12 est utilisé en mode étendu, les cartes doivent être en multicouches, ce qui augmente considérablement le prix.

- Les 68HC12 ne sont vendus actuellement qu'en plateaux de 50 pièces CMS.

- En 68HC11 il existait un petit débogueur gratuit Debug11. En 68HC12, il existe bien Debug12 mais ce logiciel gratuit doit être programmé dans la FLASH du 68HC912 qui doit alors fonctionner en mode monochip. Le problème est qu'en mode monochip, la RAM ne fait que 2 Ko. ce qui est suffisant pour l'assembleur mais nettement insuffisant pour y mettre le moindre programme sérieux en langage C. Il est donc impossible d'utiliser conjointement Debug12 et une application écrite en langage C, à moins d'utiliser un émulateur qui remplace la FLASH par de la RAM ; dans ce cas, on peut mettre dans cette RAM de substitution à la fois DEBUG12 et une application écrite en C. Ce problème est d'ailleurs général à tous les microcontrôleurs : si l'on veut utiliser le langage C en pédagogie, il faut un minimum de RAM, ce qui **exclut le mode monochip**.

- Enfin, l'initialisation du microcontrôleur est particulièrement complexe.

Il en résulte que la réalisation d'une petite maquette 68HC12 demande des moyens importants en coût et en temps, surtout si la maquette fonctionne en mode étendu. De plus, le temps passé pour la mise au point est également important du fait de la complexité du 68HC12. Aussi, les enseignants souhaitant réaliser à peu de frais de petites maquettes autonomes ont intérêt à garder le 68HC11, le 68HC08, les 8051 ou encore les PIC.

Le 68HC12 se présente donc comme une évolution de 6809 et 68HC11, avec quelques traces de 68000. Au niveau logiciel, on pourrait penser que c'est le 6809 qui revient au devant de la scène. Quant à l'aspect matériel, c'est probablement le 68HC11 qui lui a servi de modèle. Le 68HC12 est donc un microcontrôleur assez compliqué mais très intéressant.

En résumé, on s'aperçoit que MOTOROLA a pris ce qu'il y avait de meilleur dans ses différents produits antérieurs : les instructions et modes d'adressage du 6809 qui étaient remarquablement bien pensés, les registres du 68HC11, la dualité 8/16 bits du 68008, quelques instructions du 68000... En y ajoutant toutes les tendances actuelles (PWM, logique floue, CAN ...), MOTOROLA en a fait un produit extrêmement intéressant.

D.2. La famille HC12

Microcontrollers > 16-Bit (HCS12, M68HC12, M68HC16, 56800/E) > HCS12 Family

Product	Internal RAM	EEPROM	Flash	Serial Interface	Timers	Bus Frequency	Supply Voltage	A/D Converter	A/D Converter	Pulse Width Modulator	Pulse Width Modulator	I/O Pins	Other Peripherals
				Type	Channels			Channels	Bits	Channels	Bits		
	(Byte)	(Byte)	(Byte)			(MHz)	(V)		(bit)		(bit)		
MC9S12A32	4096	1024	32000	**CAN 2.0 A/B, IIC, SCI, SPI**	8	25	5	4, 8	10, 16	4, 8	8, 16	91	
MC9S12A64	4096	1024	65536	**IIC, SCI, SPI**	8	25	5	8	10	4, 7, 8	8, 16	59, 91	
MC9S12A128	8192	2048	131072	**IIC, SCI, SPI**	8	25	5	8	10	4, 8	8, 16	59	Addressable External Memory
MC9S12A512	4096	1024	512000	**IIC, SCI, SPI**	8	25	5	8	10	7	8	59	
MC9S12C32	2000000		32000	**CAN, SCI, SPI**	8	16, 25	3.3, 5.5	8	10	6	8, 10	60	Low Voltage Inhibit
MC9S12C64	4000		64000	**CAN, SCI, SPI**	8	25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12C96	4000		96000	**CAN, SCI, SPI**	8	25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12C128	4000		128000	**CAN, SCI, SPI**	8	25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12D32	4096	1024	32000	**CAN 2.0 A/B, IIC, SCI, SPI**	8	25	5	8	10	4, 8	8, 16	91	
MC9S12D64	4096	1024	65536	**CAN 2.0 A/B, IIC, SCI, SPI**	8	25	5	8	10	4, 7, 8	8, 16	59, 91	
MC9S12DJ64	4096	1024	65536	**CAN 2.0 A/B, IIC, J1850, SCI, SPI**	8	25	5	8	10	4, 7, 8	8, 16	59, 91	
MC9S12DB128	8192	2048	131072	**BYTEFLIGHT, CAN 2.0 A/B, SCI, SPI**	8	25	5	8	10	8	8	91	Addressable External Memory
MC9S12DG128	8192	2048	131072	**CAN 2.0 A/B, IIC, SCI, SCP, SPI**	7, 8	25	5	8	10	8	8	59, 91	Addressable External Memory
MC9S12DJ128	8192	2048	131072	**CAN 2.0 A/B, IIC, J1850, SCI, SPI**	7, 8	25	5	8	10	4, 8	8, 16	59, 91	Addressable External Memory
MC9S12DP512	12288	4096	512000	**CAN 2.0 A/B, IIC, J1850, SCI, SPI**	8	25	5	8, 10	8, 10	8	8	91	Addressable External Memory
MC9S12DT128	8192	2048	131072	**CAN 2.0 A/B, IIC, SCI, SPI**	8	25	5	8	10	4, 8	8, 16	91	Addressable External Memory
MC9S12E32	2000	1024	32000	**IIC, SCI, SPI**	8	25	5	8	10	7	8	60	
MC9S12E64	4096	1024	65536	**IIC, SCI, SPI**	8	25	5	8	10	7	8	59, 91	
MC9S12E128													
MC9S12GC16	2000		32000	**SCI, SPI**	8	16, 25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12GC32	2000		32000	**SCI, SPI**	8	16, 25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12GC64	4000		64000	**SCI, SPI**	8	25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12GC96	4000		96000	**SCI, SPI**	8	25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12GC128	4000		128000	**SCI, SPI**	8	25	3.3, 5.5	8	10	6	8	60	Low Voltage Inhibit
MC9S12H128	6000	4096	131072	**CAN 2.0 A/B, IIC, SCI, SPI**	8	16	5	8	10	3, 6	8, 16	99	Addressable External Memory, Low Voltage Inhibit
MC9S12H256	12288	4096	262144	**CAN 2.0 A/B, IIC, SCI, SPI**	8	16	5	8, 16	10	3, 6	8, 16	99	Addressable External Memory, Low Voltage Inhibit
MC9S12NE64	8000		64000	**Ethernet, IIC, SCI, SPI**		25	3.3, 5	8	10				Real-Time Interrupt, Watchdog Timer
MC9S12T64	2048	2048	65536	**SCI, SPI**	8			8	10	4, 8	8, 16		
MC9S12UF32	3584		32768	**SCI, USB 2.0**	8	30	5					75	Watchdog Timer
MC9S12A128B	8192	2048	131072	**IIC, SCI, SPI**	8	25	5	8	10	4, 8	8, 16	59, 91	Addressable External Memory

MC9S12A256B	12288	4096	262144	**IIC, SCI, SPI**	8	25	5	2, 3, 8	10	4, 8	8, 16	59, 91	Addressable External Memory
MC9S12DB128B	8192	2048	131072	**BYTEFLIGHT, CAN, CAN 2.0 A/B, SCI, SPI**	8	25	5	8	10	8	8	91	Addressable External Memory
MC9S12DG128B	8192	2048	131072	**CAN 2.0 A/B, IIC, SCI, SPI**	7, 8	25	5	8	10	8	8	59, 91	Addressable External Memory
MC9S12DG256B	12288	4096	262144	**CAN 2.0 A/B, IIC, SCI, SPI**	8	25	5	8	10	4, 8	8, 16	91	Addressable External Memory, Low Voltage Inhibit
MC9S12DJ128B	8192	2048	131072	**CAN 2.0 A/B, IIC, J1850, MSCAN12, SCI, SPI**	7, 8	25	5	8	10	4, 8	8, 16	59, 91	Addressable External Memory
MC9S12DJ256B	12288	4096	262144	**CAN 2.0 A/B, IIC, J1850, SCI, SPI**	8	25	5	8	10	4, 8	8, 16	59, 91	Addressable External Memory
MC9S12DP256B	12288	4096	262144	**CAN 2.0 A/B, IIC, J1850, SCI, SPI**	8	25	5	8	10	8	8	91	Addressable External Memory
MC9S12DT128B	8192	2048	131072	**CAN 2.0 A/B, IIC, SCI, SPI**	8	25	5	8	10	4, 8	8, 16	91	Addressable External Memory
MC9S12DT256B	12288	4096	262144	**CAN 2.0 A/B, IIC, SCI, SPI**	8	25	5	8	10	8	8	91	Addressable External Memory, Low Voltage Inhibit

D.3. Package HC12C12

PACKAGE OPTIONS

PART NUMBER	PACKAGE	TEMPERATURE RANGE
MC9S12C32CFA	48 LQFP	-40°C to +85°C
MC9S12C32VFA	48 LQFP	-40°C to +105°C
MC9S12C32MFA	48 LQFP	-40°C to +125°C
MC9S12C32CPB	52 LQFP	-40°C to +85°C
MC9S12C32VPB	52 LQFP	-40°C to +105°C
MC9S12C32MPB	52 LQFP	-40°C to +125°C
MC9S12C32CFU	80 QFP	-40°C to +85°C
MC9S12C32VFU	80 QFP	-40°C to +105°C
MC9S12C32MFU	80 QFP	-40°C to +125°C

FA
48-Pin LQFP
.5 mm Pitch
7 mm x 7 mm Body

PB
52-Pin LQFP
.65 mm Pitch
10 mm x 10 mm Body

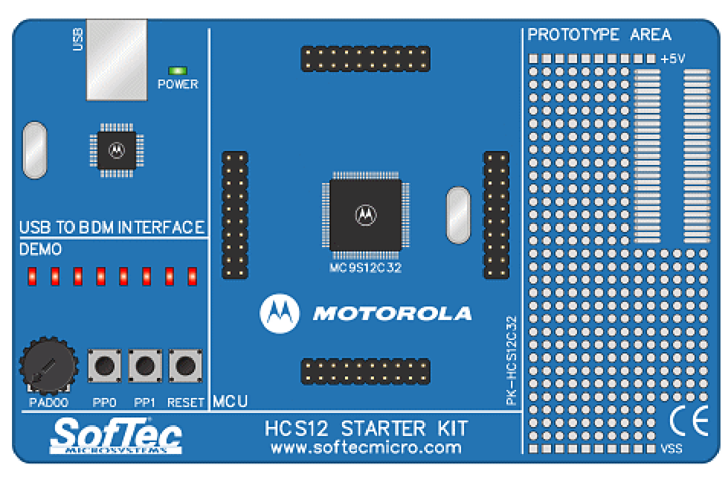
FU/PU
80-Lead QFP/LQFP
.65 mm Pitch
14 mm x 14 mm Body

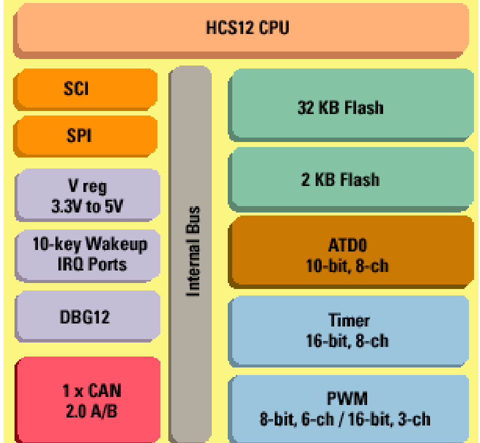
MOTOROLA

D.4. Kit PK-HCS12xxx

Remarques: Ce microcontrôleur possède les avantages du 6809, 68HC11 et 68000.

D.4.1. PK-HCS12C32 Starter Kit pour Motorola MC9S12C32





True BDM Debugging through USB Interface

Metrowerks CodeWarrior (Special Edition) Included, with Editor, Assembler, C Compiler and Debugger

Working Experiments in minutes (C, Asm, Processor Expert)

Overview

The PK-HCS12C32 Starter Kit has been designed for the evaluation of the MC9S12C32 microcontroller and the debugging of small user applications. The Starter Kit takes advantage of the Metrowerks CodeWarrior Integrated Development Environment (which groups an Editor, Assembler, C Compiler and Debugger) and the Motorola BDM interface, which allows the download and debug of the user application into the microcontroller FLASH memory. Together with CodeWarrior, the Starter Kit provides you with everything you need to write, compile, download, in-circuit emulate and debug user code. Full-speed program execution allows you to perform hardware and software testing in real time. The Starter Kit is connected to the host PC through a USB port. A prototyping area allows you to wire your own small application.

The Starter Kit Box

Board Features

The PK-HCS12C32 board has the following hardware features:

A MCU section. It contains a soldered, 80-pin MC9S12C32 device (in QFP package) with connectors to access the I/O pins of the microcontroller for expansion prototyping. A 16 MHz crystal oscillator (in Colpitts configuration) is

Supported Devices

MC9S12C32

At a Glance

CodeWarrior IDE

Editor; Assembler; C Compiler (12-KB Limited); Linker; Source Level and Symbolic Debugger; Windows 98/2000/XP Compatible

Debugging Capabilities

Reset, Start, Stop, Single Step, Step Over, Step Out; Hardware Breakpoints; Trace; Full Handling of Target' s On-Chip Debug Peripheral; Real-Time Refresh of RAM, Variables and Peripherals

provided, and the microcontroller is configured to work in single-chip mode (MODA = MODB = 0).

A Demo section. It features a RESET push-button, two user push-buttons, a potentiometer and eight user LEDs.

A USB to BDM Interface section. It contains the circuitry needed to electrically and logically translate BDM-like commands sent by the host PC through the USB cable to the BDM interface of the microcontroller. The PK-HCS12C32 board is powered (at 5 V) by the USB bus.

A Prototype section. You can wire your own circuit here. The prototype section features both a standard, thru-hole area (for mounting traditional components) and a SMD area (for soldering SMD components in SOIC package).

Communication

USB Connection to the Host PC, BDM Connection to the Built-In Target Microcontroller

Power Supply

Provided by the USB Bus, No External Power Supply Needed

Package Contents

Starter Kit Board, USB Cable, Metrowerls CodeWarrior CD, SofTec Microsystems CD w/ Microcontroller Datasheets…

D.4.2. HC9S12E128 : Structure interne

128K Flash EEPROM
8K RAM
Voltage Regulator
HCS12 CPU
Periodic Interrupt
COP Watchdog
Clock Monitor
Debugger (DBG12)
Breakpoints
Single-Wire Background Debug Module
Clock and Reset Generation Module (CRG)
System Integration Module (SIM)
Multiplexed Address/Data Bus
PMF
SCI0
SCI1
SPI
TIM0
TIM1
PWM
MUX
TIM2
ADC
DAC0
DAC1
SCI2
IIC

D.4.3. Pile et registre: memory map:

Table 7-2. Stacking Order on Entry to Interrupts

Memory Location	CPU Registers
SP + 7	$RTN_H : RTN_L$
SP + 5	$Y_H : Y_L$
SP + 3	$X_H : X_L$
SP + 1	B : A
SP	CCR

7 A 0	7 B 0	8-BIT ACCUMULATORS A AND B
15 D 0		16-BIT DOUBLE ACCUMULATOR D (A : B)
15 X 0		INDEX REGISTER X
15 Y 0		INDEX REGISTER Y
15 SP 0		STACK POINTER
15 PC 0		PROGRAM COUNTER
S X H I N Z V C		CONDITION CODE REGISTER

- C — CARRY
- V — OVERFLOW
- Z — ZERO
- N — NEGATIVE
- I — INTERRUPT MASK (DISABLE)
- H — HALF-CARRY FOR BCD ARITHMETIC
- X — XIRQ INTERRUPT MASK (DISABLE)
- S — STOP DISABLE (IGNORE STOP INST.)

Memory Map

$0000
$0400
$2000
$4000
$8000
$C000
$FF00
$FFFF

EXTERNAL
VECTORS
VECTORS
VECTORS

EXPANDED MODE
NORMAL SINGLE CHIP MODE
SPECIAL SINGLE CHIP MODE

$0000 1K Bytes Register Space (mappable to any 2K boundary) $03FF
$2000 8K Bytes RAM (mappable to any 8K boundary) $3FFF
$4000 0.5K, 1K, 2K or 4K Bytes Protected Sector
16K Bytes Fixed Flash EEPROM $7FFF
$8000 16K Bytes Page Window Eight 16K Bytes Flash EEPROM Pages $BFFF
$C000 16K Fixed Flash EEPROM
2K, 4K, 8K or 16K Bytes Protected Boot Sector $FFFF
$FF00 BDM (if active) $FFFF

Note: this is a useful map, which is not the map out of reset. After reset the map is:
$0000 ~ $03FF: Register Space
$0000 ~ $3FFF: 16K RAM (only 15K RAM visible $0400 ~ $3FFF)

Table 7-1. CPU12 Exception Vector Map(1)

Vector Address	Source
$FFFE–$FFFF	System reset
$FFFC–$FFFD	Clock monitor reset
$FFFA–$FFFB	COP reset
$FFF8–$FFF9	Unimplemented opcode trap
$FFF6–$FFF7	Software interrupt instruction (SWI)
$FFF4–$FFF5	$\overline{\text{XIRQ}}$ signal
$FFF2–$FFF3	$\overline{\text{IRQ}}$ signal
$FF00–$FFF1	Device-specific interrupt sources (HCS12)
$FFC0–$FFF1	Device-specific interrupt sources (M68HC12)

1. See Device User Guide and Interrupt Block Guide for further details

D.4.4. Mécanisme d'interruption

START

OPCODE TRAP? — YES → T.1 - I INTERNAL CALCULATIONS

NO

1.0 - V FETCH VECTOR

RESET? — YES

NO

INTERRUPT? — NO

YES

2.0 - I NO BUS ACCESS
SET S, X, AND I

3.0 - P FETCH PROGRAM WORD
START TO FILL INSTRUCTION QUEUE

4.0 - P FETCH PROGRAM WORD
CONTINUE TO FILL INSTRUCTION QUEUE

5.0 - P FETCH PROGRAM WORD
FINISH FILLING INSTRUCTION QUEUE

END

2.1 - S PUSH RETURN ADDRESS
ADDRESS OF INST THAT WOULD HAVE EXECUTED IF NO INTERRUPT

3.1 - P FETCH PROGRAM WORD
START TO FILL INSTRUCTION QUEUE

4.1 - S PUSH Y

5.1 - S PUSH X

6.1 - P FETCH PROGRAM WORD
CONTINUE TO FILL INST. QUEUE
TRANSFER B:A TO 16-BIT TEMP REG

7.1 - S PUSH B:A

8.1 - s PUSH CCR (BYTE)
SET I BIT
IF XIRQ, SET X BIT

9.1 - P FETCH PROGRAM WORD
FINISH FILLING INSTRUCTION QUEUE

END

2.2 - S PUSH RETURN ADDRESS
ADDRESS OF INST AFTER SWI OR UNIMPLEMENTED OPCODE

3.2 - P FETCH PROGRAM WORD
START TO FILL INSTRUCTION QUEUE

4.2 - S PUSH Y

5.2 - S PUSH X

6.2 - P FETCH PROGRAM WORD
CONTINUE TO FILL INST. QUEUE
TRANSFER B:A TO 16-BIT TEMP REG

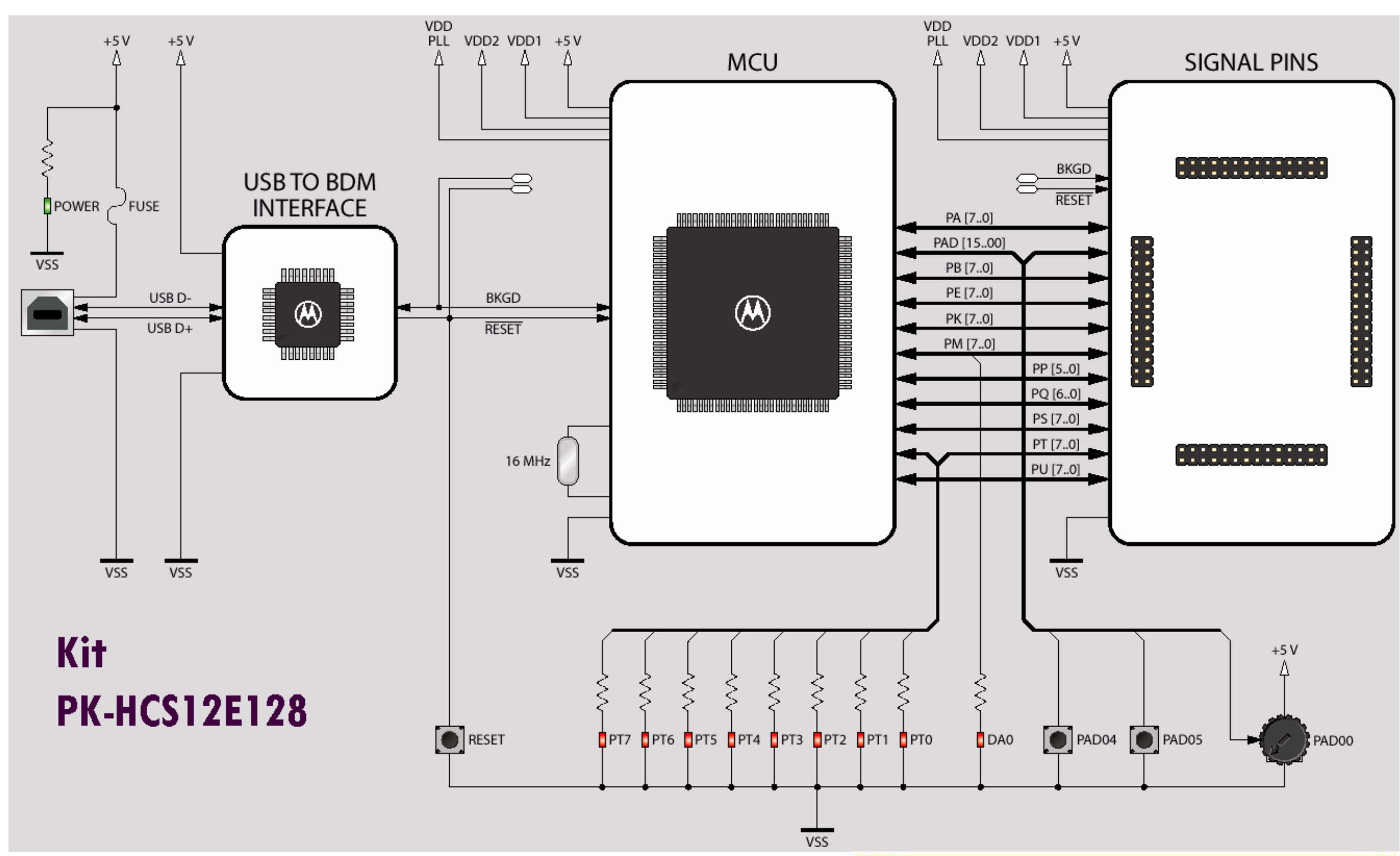
7.2 - S PUSH B:A

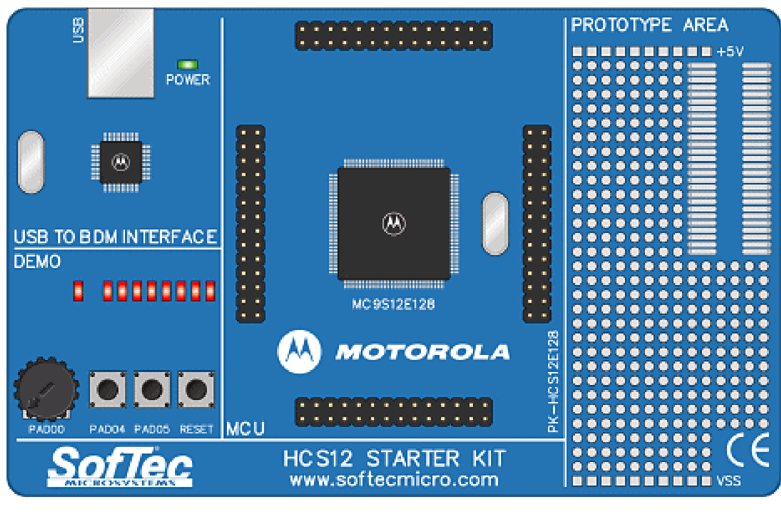
8.2 - s PUSH CCR (BYTE)
SET I BIT

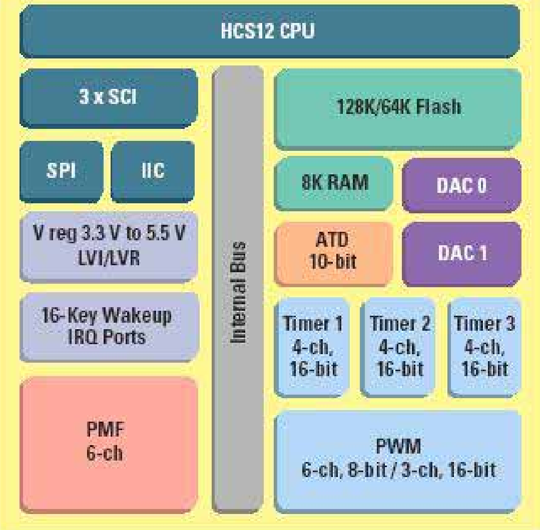
9.2 - P FETCH PROGRAM WORD
FINISH FILLING INSTRUCTION QUEUE

END

Figure 7-1. Exception Processing Flow Diagram


MCU
SIGNAL PINS
USB TO BDM
INTERFACE
POWER
FUSE
VSS
USB D-
USB D+
BKGD
RESET
16 MHz
PA [7..0]
PAD [15..00]
PB [7..0]
PE [7..0]
PK [7..0]
PM [7..0]
PP [5..0]
PQ [6..0]
PS [7..0]
PT [7..0]
PU [7..0]
Kit
PK-HCS12E128
PT7
PT6
PT5
PT4
PT3
PT2
PT1
PT0
DA0
PAD04
PAD05
PAD00


USB
POWER
PROTOTYPE AREA
USB TO BDM INTERFACE
DEMO
MC9S12E128
MOTOROLA
PK-HCS12E128
PAD00 PAD04 PAD05 RESET
MCU
SofTec
HCS12 STARTER KIT
www.softecmicro.com


HCS12 CPU
3 x SCI
128K/64K Flash
SPI
IIC
8K RAM
DAC 0
V reg 3.3 V to 5.5 V
LVI/LVR
ATD
10-bit
DAC 1
Internal Bus
16-Key Wakeup
IRQ Ports
Timer 1
4-ch,
16-bit
Timer 2
4-ch,
16-bit
Timer 3
4-ch,
16-bit
PMF
6-ch
PWM
6-ch, 8-bit / 3-ch, 16-bit

CodeWarrior Integrated Development Environment

PK-HCS12C32 comes with (and are seamlessly integrated into) a free version of CodeWarrior Development Studio for HC(S)12 Microcontrollers, Special Edition. The CodeWarrior Development Studio for Motorola HC(S)12 Microcontrollers enables you to build and deploy HCS12 systems quickly and easily. CodeWarrior Development Studio for HC(S)12 Microcontrollers, Special Edition, includes the CodeWarrior integrated development environment (IDE); 12 KB code-size limited C compiler and C source-level debugger; macro assembler and Assembly-level debugger. The Special Edition allows you to evaluate CodeWarrior Development Studio for HC(S)12 Microcontrollers at no cost.

Metrowerks CodeWarrior - [main.c]

File Edit View Search Project Debug Processor Expert Window Help

Path: D:\Prog_FileD\Programmation\Metrowerks\CW12V31\(CodeWarrior_Examples)\MC9S12\SofTec

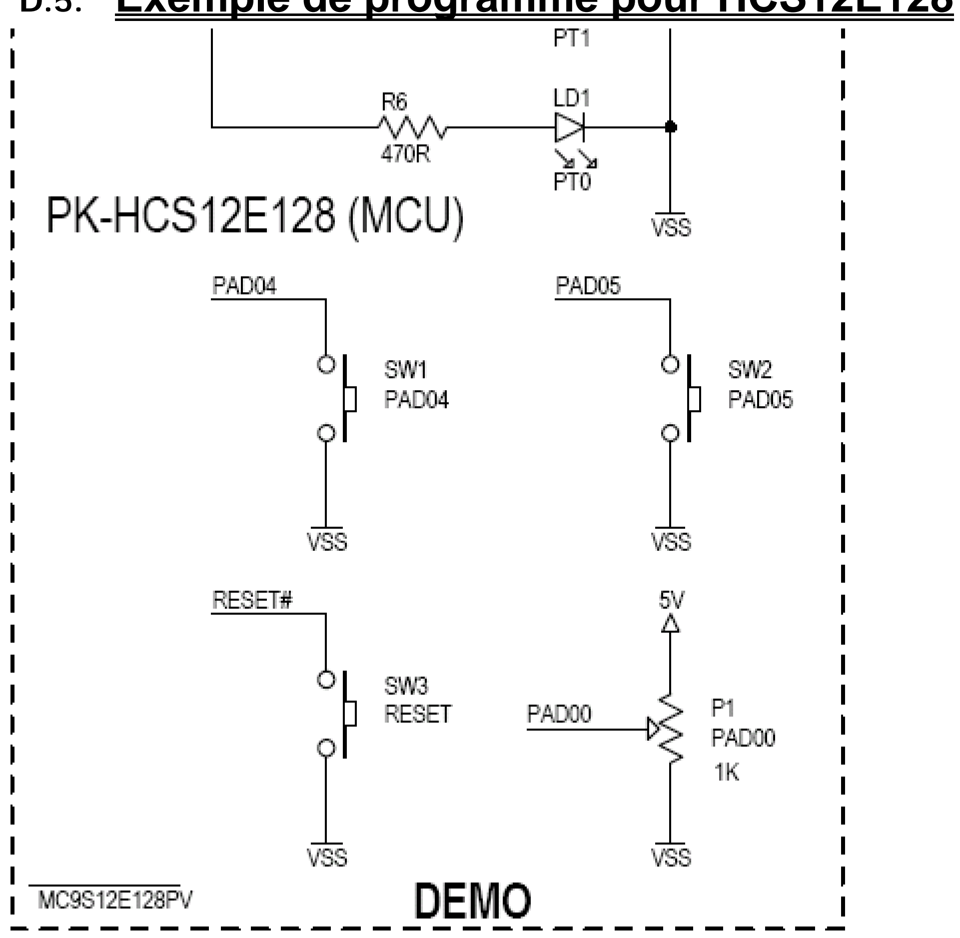
```
//////////////////////////////////////////////////////////////////////////////
// ADC Sample for SofTec Microsystems PK-HCS12E128 Starter Kit
//
// By rotating the potentiometer (PAD00 channel 0), you affect the results
// of the A/D conversion, and the value of each conversion is displayed on the
// LEDs on PT[7..0].
// --------------------------------------------------------------------------
// Copyright (c) 2003 SofTec Microsystems
// http://www.softecmicro.com/
//////////////////////////////////////////////////////////////////////////////
#include <hidef.h>                  // Common defines and macros
#include <mc9s12e128.h>             // Derivative information
#pragma LINK_INFO DERIVATIVE "SampleS12"
// --------------------------------------------------------------------------
// Puts the following code into NOT BANKED FLASH MEMORY
// --------------------------------------------------------------------------
#pragma CODE_SEG __NEAR_SEG NON_BANKED
// --------------------------------------------------------------------------
// Peripheral Initialization Routine
// --------------------------------------------------------------------------
void PeriphInit(void)
{
    PTT = 0x00;                     // Clears port T[7..0]
    DDRT = 0xFF;                    // Configures port T[7..0] as output

    ATDCTL3 = 0x08;                 // One conversion per sequence
    ATDCTL4_SRES8 = 1;              // Sets A/D conversion to 8 bit resolution
    ATDCTL5_SCAN = 1;               // Continuous conversion (scan mode)
    ATDCTL2 = 0x80;                 // Normal ATD functionality
}
// --------------------------------------------------------------------------
// Main
// --------------------------------------------------------------------------
void main(void)
{

    PeriphInit();                   // Microcontroller initialization
    EnableInterrupts;               // Enables interrupts

    for(;;)                         // Forever
    {
        if(ATDSTAT1_CCF0)           // If ATD conversion complete flag 0
            PTT = ATDDR0H;          // Writes the ATD value on Port T
    }
}
```

D.5. Exemple de programme pour HCS12E128




```
////////////////////////////////////////////////////////////////////////////////
// @ Denis MICHAUD le 07 02 06          version 7
// FILE : dm7_KB.c : gestion touche et affichage
// ADC Sample for SofTec Microsystems PK-HCS12E128 Starter Kit
//     PAD05 et PAD04 connectés à 2 Bouton Poussoir: 0 actif
//
// By rotating the potentiometer (PAD00 channel 0), you affect the results
// of the A/D conversion, and the value of each conversion is displayed on the
// LEDs on PT[7..0].
// This example varies the luminosity of the DAC0 LED by generating a triangle
// wave through the DAC output on channel 0.
// IMA : issue de ADC.mcp
////////////////////////////////////////////////////////////////////////////////

#include <hidef.h>              // Common defines and macros
#include <mc9s12e128.h>         // Derivative information
 // ---------------------------------------------------------------------------
  char ATEMP;

#pragma LINK_INFO DERIVATIVE "SampleS12"

// ---------------------------------------------------------------------------
// Puts the following code into NOT BANKED FLASH MEMORY
// ---------------------------------------------------------------------------
#pragma CODE_SEG __NEAR_SEG NON_BANKED

// ---------------------------------------------------------------------------
#define BP__ 3  // Aucune touche  APPUYEES
#define BP_4 2  // touche PAD04  APPUYEE
#define BP5_ 1  // touche PAD05  APPUYEE
#define BP54 0   // 2 touches PAD04 et PAD05  APPUYEES
#define mask54 0x30  // masque deux touches
```

```
// ---------------------------------------------------------------------------
void delay_(int duree_ms)              // Temporisation
  {
    unsigned long i,j;
    for(j = 0; j < duree_ms; j++)
      {
        for(i = 0; i < 200; i++)    // Software delay
        ;     // Ne rien faire
      }
  }
// ---------------------------------------------------------------------------
// Peripheral Initialization Routine
// ---------------------------------------------------------------------------
void PeriphInit(void)
{
// Congig port T 8 LEDs
    PTT = 0x55;                     // port T[7..0]         1 sur 2 allumés
    DDRT = 0xFF;                    // Configures port T[7..0] as output
             delay_(1000);
             PTT  = 0xAA;                                                      // port T[7..0]
             1 sur 2 allumés, décalage
// Config du ATD
   ATDCTL3 = 0x08;                 // One conversion per sequence
   ATDCTL4_SRES8 = 1;              // Sets A/D conversion to 8 bit resolution
   ATDCTL5_SCAN = 1;               // Continuous conversion (scan mode)
   ATDCTL2 = 0x80;                 // Normal ATD functionality
    PERAD |= 0x30;               // Enables pull-ups on Port PAD[05..04]
//    PIEAD |= 0x30;               // Enables key interrupts on Port PAD[05..04]
    ATDDIEN1 |= 0x30;            // Enables digital input buffer on Port PAD[05..04]
// Congig du DAC
    DAC0_DACDRight = 0x00;           // Sets DAC data to 0
    DAC0_DACC0 |= 0x89;              // Enables DAC: DACE = 1, DSGN = 1, DACOE = 1
    DAC0_DACDRight = 0xFF;     // Allumer LED maxi
             delay_(5000);
             DAC0_DACDRight = 0x00;     // Allumer LED mini
             PTT  = 0x33;              // port T[7..0]           0011 0011
}
// ---------------------------------------------------------------------------
// InvertV                 : Bascule, INVERSION d'un charactere
// ------------------------------------------------------------------------- */
unsigned char InvertV_(unsigned char mon_seuil,unsigned char old_dir)
{
              unsigned char ma_dir;
    {
        if(old_dir)
        {
                DAC0_DACDRight = mon_seuil;
                ma_dir = 0;
        }
        else
        {
                DAC0_DACDRight = ~mon_seuil;  // inversion logique de la variable
                ma_dir = 1;
        }
        return ma_dir;
    }
}
// ---------------------------------------------------------------------------
// PWMmain                 : VOBULATION, dent de scie de la sortie DAC
// ---------------------------------------------------------------------------
void PWMmain(char vitesse)
{
    unsigned char up_dir = 1;
    {
        delay_(vitesse);
        if(up_dir)
        {
            DAC0_DACDRight++;       // Increases the DAC output
             if(DAC0_DACDRight == 0xFF)
                up_dir = 0;
        }
```

```
        else
        {
            DAC0_DACDRight--;       // Decreases the DAC output
            if(DAC0_DACDRight == 0x60)
                up_dir = 1;
        }
    }
}
//***************************************************************************
char inCLAV ;
unsigned char etatVAR,temp_;
// --------------------------------------------------------------------------
// Main : Gestion clavier et ATD  et DAC  ********************************
void main(void)
{
    PeriphInit();                // Microcontroller initialization
    etatVAR = 0;
    do
      {
        PWMmain(4);                            // vobulation DAC
        inCLAV = PORTAD1&mask54;              // variable clavier PAD04 et PAD05      }
    while (inCLAV==mask54);  // REPETER jusqu'à appuye 1 touche
    delay_(100);
    for(;;)                      // Forever
    {
       inCLAV = (PORTAD1&mask54)>>4;    // variable clavier PAD04 et PAD05
       switch(inCLAV)                   // choix action 8+1 LED en fonction 2 BP et pot
        {
          case BP_4:               // PA4=0  ON appuye  SEUL, ( PA5 OFF =1)
                                                PTT=0x0F;
                             DAC0_DACDRight = 0xD0;        // luminosite 3 quart
                                                break;
          case BP5_:                                         // PA5 et PA4 appuyes
                   do {
                    if(ATDSTAT1_CCF0)    // If ATD conversion complete flag 0
                        PTT = ~ATDDR0H; // Writes the ATD Complemented on Port T
                        temp_ =  InvertV_(ATDDR0H,etatVAR);
                        etatVAR =temp_;
                        delay_(5000);
                        inCLAV = (PORTAD1&mask54)>>4;      // variable clavier PAD04 et PAD05
                        } while (inCLAV == BP5_);
                    // recopie et clignotement du pot sur dac
                    break;
          case BP54:
                    PTT=0xF0;                    // PA4=1 NON appuye et PA5=0 appuye
                    DAC0_DACDRight = 0x90;            // luminosite moitie
                                        break;
          default:                        // case BP__; :OFF, AUCUN Bp appuyes
                    if(ATDSTAT1_CCF0) // If ATD conversion complete flag 0
                      {
                            PTT = ATDDR0H;  // Writes the ATD value on Port T
                            DAC0_DACDRight = ATDDR0H; // recopie pot sur DAC
                      }
        }
    }
} // FIN main ********************************** @dm 2006 ***********
```

E. Définitions :

Les préfixes pour des multiples binaires (Mo, Ko, Octets…)

Le débit :

En décembre 1998 la Commission Électrotechnique Internationale (International Electrotechnical Commission = IEC), l'organisation internationale principale pour la standardisation mondiale dans electrotechnology, approuvé comme une Norme Internationale IEC nomment et des symboles pour des préfixes pour des multiples binaires pour l'utilisation dans les champs (domaines) de traitement de données et la transmission de données. Les préfixes sont comme suit :

Les préfixes pour des multiples binaires

Facteur	Nom	Symbole	Origine	Derivation
2^{10}	kibi	Ki	kilobinary: $(2^{10})^1$	kilo: $(10^3)^1$
2^{20}	mebi	Mi	megabinary: $(2^{10})^2$	mega: $(10^3)^2$
2^{30}	gibi	Gi	gigabinary: $(2^{10})^3$	giga: $(10^3)^3$
2^{40}	tebi	Ti	terabinary: $(2^{10})^4$	tera: $(10^3)^4$
2^{50}	pebi	Pi	petabinary: $(2^{10})^5$	peta: $(10^3)^5$
2^{60}	exbi	Ei	exabinary: $(2^{10})^6$	exa: $(10^3)^6$

Exemples et comparaisons avec les prefixes

one **kibibit**	1 Kibit = 2^{10} bit = **1024 bit**
one **kilobit**	1 kbit = 10^3 bit = **1000 bit**
one **mebibyte**	1 MiB = 2^{20} B = **1 048 576 B**
one **megabyte**	1 MB = 10^6 B = **1 000 000 B**
one **gibibyte**	1 GiB = 2^{30} B = **1 073 741 824 B**
one **gigabyte**	1 GB = 10^9 B = **1 000 000 000 B**

Il est suggéré qu'en anglais, la première syllabe du nom du préfixe binaire-multiple devrait être prononcée de la même manière comme la première syllabe du nom du préfixe de SI correspondant et que la deuxième syllabe devrait être prononcée comme "bee" (en anglais).

Il est important de reconnaître que les nouveaux préfixes pour des multiples binaires ne font pas partie du Système International d'Unités (SI), le système métrique moderne. Cependant, pour le bien-être de compréhension et le rappel, ils ont été tirés des préfixes de SI pour les pouvoirs positifs de dix. Comme peut voir de la susdite table, le nom de chaque nouveau préfixe est tiré du nom du préfixe de SI correspondant en conservant les deux premières lettres du nom du préfixe de SI et ajoutant les lettres "bi", qui se rappelle le mot "le fichier binaire". De même le symbole de chaque nouveau préfixe est tiré du symbole du préfixe

de SI correspondant en ajoutant la lettre "i", qui se rappelle de nouveau le mot "le fichier binaire". (Pour la cohérence avec les autres préfixes pour des multiples binaires, le symbole Ki est utilisé pour 2^{10} plutôt que ki.)

La publication officielle :

Ces préfixes pour des multiples binaires, qui ont été développés par le Comité Technique IEC (TC) 25, des Quantités et des unités et leurs symboles de lettre, avec l'appui fort du Comité International pour des Poids et des Mesures (CIPM) et l'Institut d'Électriques et des Électroniciens (IEEE), a été d'abord adoptée par l'IEC comme l'Amendement 2 à la Norme Internationale IEC IEC 60027-2 : des symboles de lettre à être utilisés dans la technologie électrique - la Partie 2 : Télécommunications et électronique.

Le plein contenu de l'Amendement 2, qui a une date de publication de 1999-01, est reflété dans les tables ci-dessus et la suggestion quant à la prononciation. Par la suite le contenu de cet Amendement était incorportated dans la deuxième édition d'IEC 60027-2, qui a une date de publication de 2000-11 (la première édition a été publiée en 1972). La citation complète pour cette norme révisée est IEC 60027-2, la Deuxième édition, 2000-11, des symboles de Lettre à être utilisé dans la technologie électrique - la Partie 2 : Télécommunication et électronique

Historique :

Une fois, des professionnels informatiques ont remarqué que 2^{10} était très presque égal à 1000 et a commencé à utiliser SI préfixent "le kilo" pour signifier 1024. Cela a travaillé assez bien pendant une décennie ou deux parce que chacun qui a parlé des kilo-octets savait que le terme a impliqué 1024 octets. Mais, les nombreux acheteurs d'ordinateurs et les professionnels informatiques commerciaux ont eu besoin de parler aux physiciens et des ingénieurs et même aux gens ordinaires, la plupart de ce qui savent qu'un kilomètre est 1000 mètres et un kilogramme est 1000 grammes.

Alors le stockage de données pour des gigaoctets et même terabytes, est devenu pratique et les dispositifs de stockage n'ont pas été construits sur des arbres binaires, qui ont signifié que, pour beaucoup de buts pratiques, l'arithmétique binaire était moins commode que l'arithmétique décimale. Le résultat est qu'aujourd'hui personne ne sait pas quel est la valeur d'un méga-octet. Dans l'histoire de l'informatique, le méga-octet signifié 2^{20} = 1048576 octets, mais les fabricants de dispositifs de stockage informatiques utilisent d'habitude le terme pour signifier 1000000 d'octets. Quelques designers de réseaux locaux ont utilisé le mégabit par seconde pour signifier 1048576 bit/s, mais tous les ingénieurs de télécommunications l'utilisent pour signifier 10^6 bit/s. Et si deux définitions du méga-octet ne sont pas assez, un troisième méga-octet de 1024000 octets est le méga-octet a eu l'habitude de formater 90 millimètres familiers (3 pouce 1/2), la disquette "de 1.44 MO". La confusion est réelle, comme est le potentiel pour l'incompatibilité dans des normes et dans des systèmes mis en oeuvre.

Face à cette réalité, le Conseil de Normes IEEE a décidé que des normes IEEE utiliseront le conventionnel, internationalement adopté, les définitions des préfixes de SI. Mega signifiera 1000000, sauf que la base deux définition peut être utilisée (si une telle utilisation est explicitement désignée au cas par cas) jusqu'à un tel temps que des préfixes pour des multiples binaires est adoptée par un corps de normes(standards) approprié.

Issue de http://www.corsaire.org/consulting/multiplesbinaires.html

F. Bibliographie :

Claude Delannoy, Le Livre du C, Premier langage, Eyrolles

Un grand classique. Claude Delannoy est un pédagogue reconnu. Il présente les concepts du langage C progressivement. Un livre à conseiller aux débutants, qui n'ont aucune idée de ce qu'est la programmation.

Claude Delannoy, Le Livre du C, Premier langage, Eyrolles

Un autre grand classique. Claude Delannoy est un pédagogue reconnu. Il présente les concepts du langage C progressivement. Un livre à conseiller aux débutants.

Alan FEUER, langage C, problèmes et exercices, Masson

Mon livre d'exercices préféré. Des exercices simples jusqu'aux plus tordus permettent de comprendre le langage C en profondeur. La lecture des explications associées aux corrections est particulièrement enrichissante et comment résister aux croustillants

`++x || ++y && ++z;`

et autres

`printf("%s ", *--*++cpp+3 );`

B. Kernighan & D. Ritchie, Le langage C norme ANSI, DUNOD

La Bible originelle du langage C. Le langage expliqué par ses inventeurs. Certains l'adorent, moi, ce n'est pas mon préféré. Reste que c'est l'ouvrage de référence par excellence. Il ne fera pas honte à votre bibliothèque. Une autre Bible du langage C, conseillée par Sébastien Moutault.

Claude Delannoy, Le langage C, Eyrolles

Une autre référence du langage C, vivement conseillée par Sébastien Moutault. Tout sur le langage C. Un livre de référence à conserver toute sa vie.

J.-P. Braquelaire, Méthodologie de la programmation en C, DUNOD.

*Encore un référence à conserver. **Mon livre de cours préféré**. Il n'explique pas comment fonctionne le langage C mais comment bien s'en servir. C'est aussi une introduction à une fome de programmation orientée objet en C. Ce n'est pas un livre pour débuter, mais on ne s'en sépare plus. On y trouve aussi l'API POSIX.*

Brian W. Kernighan, Dennis M. Ritchie
Programmieren in C, 2. Ausgabe, ANSI-C
Carl Hanser Verlag, 1990

Borland C++ Version 3.1, Programmer's Guide, 1992
Borland C++ Version 3.1, User's Guide, 1992
Borland C++ Version 3.1, Library Reference, 1992

Claude Delannoy
Exercices en Langage C
Edition EYROLLES, 1992

C auf der Überholspur, Kurs: C-Programmierung
DOS International (Sept.1992 - Apr.1993)

Jean-Michel Gaudin
Infoguide Turbo - C
Editions P.S.i, 1988

Niklaus Wirth
Algorithmen und Datenstrukturen
B.G.Teubner, 1983

Jürgen Franz, Christoph Mattheis
Objektorientierte Programmierung mit C++
Franzis Verlag, 1992

EST - Langage algorithmique - Manuel de référence
Ministère de l'Education Nationale, 1991

EST - Informatique, Initiation à l'algorithmique - Classe de 12e
EST - Informatique, Initiation à l'algorithmique - Classe de 13e
Ministère de l'Education Nationale, 1992

- ***HC12***

Hard: SofTec MICROSYSTEMS http://www.softecmicro.com
Soft: CodeWarrior http://www.MetroWerks.com

FOURNISSEUR: **PROgrammation** 92 av du Général de Gaulle, 92 250 La Garenne Colombes

Web: http://www.programmation.fr

- Sites sur le C:

ftp://ftp.imag.fr/pub/labo-CLIPS/commun/C/Introduction_ANSI_C.pdf

http://www-ipst.u-strasbg.fr/pat/program/tpc.htm
http://www.ltam.lu/Tutoriel_Ansi_C/home.htm
http://www.multimania.com/dancel/c/c.html
http://www.loria.fr/~mermet/CoursC/coursC.html
http://www-inf.enst.fr/~charon/CFacile/default.htm

- Sites µC:

http://www.polytech-lille.fr/~rlitwak/Cours_MuP/
http://www.ief.u-psud.fr/~jok/iut/doc.html

G. Glossaire : sigles & acronymes.