

INTRODUCTION PROGRESSIVE AU C

À UTILISER CONJOINTEMENT AVEC L'ABRÉGÉ DU LANGAGE C ET LES TPs

Cette introduction permet de présenter progressivement les grandes notions les plus simples du langage C.

Ce document ne doit être exploité qu'après avoir pris connaissance des notions présentées pages 1 à 5 de l'abrégé du langage C.

1) STRUCTURE MINIMALE D'UN PROGRAMME POUR SYSTÈME EMBARQUÉ	2
2) OPÉRATIONS SIMPLES SUR LES ENTRÉES / SORTIES	2
Opérations de sortie.....	2
Opérations d'entrées.....	3
Le programme pédagogique le plus simple avec une opération d'entrée et une opération de sortie.....	3
3) DÉFINITIONS SIMPLES DE DONNÉES.....	4
4) UTILISATION DE STRUCTURES SIMPLES DE CONTRÔLE DE FLUX.....	5
Lecture d'un bit d'entrée avec une structure if	5
Attente du changement d'état d'une entrée avec une boucle while.....	5
Attente de changement d'état d'une entrée et d'une fin de temporisation	5
Transfert de 12 octets vers un affichage multiplexé à LEDs.....	5
Réalisation d'une boucle sans fin	6
5) UTILISATION DE FONCTIONS	6
5.1) utilisation de fonctions pour la diminution de la taille du code produit.....	6
Fonction sans paramètre	7
Fonction avec paramètre / Passage de paramètre.....	7
Fonctions avec plusieurs paramètres.....	8
5.2) Utilisation de fonctions pour la lisibilité d'un programme	9
5.3) Gestionnaire d'interruption (interrupt handler)	10
Gestionnaire d'interruption avec le PIC et le compilateur HiTech.....	10
Exemple simple.....	10
6) DÉCLARATIONS DE DONNÉES. VARIABLES LOCALES ET GLOBALES	11
7) PASSAGES D'INFORMATIONS VERS UNE FONCTION	11
8) TABLEAUX, POINTEURS	12
8.1) Exemple d'utilisation d'un tableau pour générer des signaux	13
8.2) Exemple d'utilisation d'un tableau pour des motifs à afficher.....	14
8.3 Utilisation d'un pointeur pour accéder aux éléments d'un tableau	14

1) STRUCTURE MINIMALE D'UN PROGRAMME POUR SYSTÈME EMBARQUÉ

<pre>#include <pic.h></pre>	permet d'inclure (lors de la phase pré processeur) le fichier texte qui contient la déclaration des registres à fonctions spéciales du μ C (port d'E/S, registres de contrôle, etc.) ¹
<pre>void main(void) { /* instructions pour l'initialisation */ ... DebBclSansFin : /* instructions de la boucle sans fin */ goto DebBclSansFin ; }</pre>	programme principal devant obligatoirement s'appeler main Étiquette de début de la boucle sans fin Saut à l'étiquette de début

1 : chaque compilateur est livré avec un ou plusieurs fichiers contenant chacun les déclarations des registres à fonction spéciale. Ces fichiers sont dits fichiers « en-tête » (header). Voir abrégé § *Structure simplifiée d'un programme source / Inclusion de fichier*

Les déclarations des registres mentionnent les adresses. Ceci est présenté plus loin dans ce document.

Remarque :

Les programmeurs n'utilisent jamais une étiquette + un « goto » pour réaliser une boucle sans fin. La forme employée nécessite de connaître les structures de contrôle de flux. Voir plus loin.

2) OPÉRATIONS SIMPLES SUR LES ENTRÉES / SORTIES

Les opérations simples d'entrées / sorties font intervenir les affectations avec quelques opérateurs élémentaires.

OPÉRATIONS DE SORTIE

Modification des 8 bits d'un port	Mise à 0 de certains bits d'un port	Mise à 1 d certains bits d'un port
<code>PORTx = valeur ;</code>	<code>PORTx = PORTx & MASQUE ;</code>	<code>PORTx = PORTx MASQUE ;</code>

PORTx doit avoir été au préalable déclaré (par exemple dans un fichier en-tête inclus avec `#include`).

& et **|** sont des opérateurs logiques du langage C. Voir abrégé § *Opérateurs*

MASQUE peut être une valeur numérique ou un symbole équivalent à une valeur numérique défini avec la directive `#define`. On écrit en général les constantes avec des majuscules, pour les

distinguer des variables, écrites en minuscules (voir abrégé § Données : variables & constantes / Constantes / Constante symbolique).

Exemple d'extrait de programme :

<pre>#include <pic.h> ... #define MASQUE 0x25 ... void main(void) { ... TRISD=0xFF ; DebBclSansFin : ... PORTD=PORTD & MASQUE ; ... goto DebBclSansFin ... }</pre>	<p>définition du symbole MASQUE. En début de compilation, chaque occurrence de MASQUE sera remplacée par la valeur 0x25¹</p> <p>Configuration du port D du µC en sortie. TRISD est déclaré dans pic.h. TRISD est un registre, donc une variable. Il est écrit en majuscule, car c'est ainsi qu'il est déclaré</p> <p>Instruction décrite plus haut. Chaque bit à 0 de MASQUE → bit correspondant du port D à 0. Les autres bits sont inchangés.</p>
--	--

1 : 0x25 est une constante écrite en hexadécimal. Voir abrégé § Données : variables & constantes / Constantes / Règles d'écriture des constantes.

Remarques :

Certains compilateurs ont une extension du C ANSI qui permet de manipuler un seul bit sur une variable (port de sortie, mot en mémoire).

Une déclaration spécifique permet de déclarer un bit. Une fois cette déclaration effectuée, on peut utiliser une instruction du type :

```
RD1 = 0 ; /* mise à 0 du bit 1 du port D */
```

Les compilateurs HiTech et CC5X admettent cette syntaxe.

De plus le compilateur CC5X admet d'accéder à un bit d'une variable (port ou case mémoire quelconque) avec la syntaxe suivante PORTD.0 pour le bit 0 du port D.

OPÉRATIONS D'ENTRÉES

VarE est une variable préalablement définie destinée à contenir la valeur d'un port d'entrée.

Lecture des 8 bits d'un port	Lecture avec masquage	
VarE = PORTx ;	VarE = PORTx & MASQUE	

LE PROGRAMME PÉDAGOGIQUE LE PLUS SIMPLE AVEC UNE OPÉRATION D'ENTRÉE ET UNE OPÉRATION DE SORTIE

Ce programme n'a qu'une valeur pédagogique. C'est souvent le 1^{er} programme écrit, compilé, etc. avec un système embarqué.

On prend comme exemple une maquette pédagogique avec un μC PIC dont le port D est connecté à des LEDs et le port C est connecté à des micro interrupteurs

<pre>#include <pic.h> void main(void) { TRISD=0 ; TRISC=0xFF ; DebBclSansFin : PORTD= PORTC ; goto DebBclSansFin ; }</pre>	<p>programmation du port D en sortie programmation du port C en entrée</p> <p>le port C est d'abord lu, puis la valeur est envoyée sur le portD. Si une variable intermédiaire est nécessaire, elle est créée lors de la compilation¹.</p>
---	---

1 : avec une instruction de ce type, une variable intermédiaire n'est pas nécessaire : la valeur sur le port D est d'abord recopiée dans l'accumulateur (ou registre de travail avec le PIC) puis le contenu de l'accumulateur est transféré sur le port C.

3) DÉFINITIONS SIMPLES DE DONNÉES

A chaque fois qu'une donnée est utilisée, il faut auparavant indiquer au compilateur son type pour que celui-ci réserve la mémoire nécessaire (entre autres). C'est la définition (aussi appelée déclaration) d'une donnée qui permet de réserver de la mémoire.

L'emplacement de la définition d'une donnée est important. On se contente ici d'un seul emplacement : en début de fichier, avant le programme main.

Les données les plus utilisées sont les entiers sur 8 bits et 16 bits. Les codes peuvent correspondre à des entiers non signés (code binaire « naturel ») ou signés (code complément à 2).

Une définition s'effectue en écrivant le type suivi du nom de la donnée.

exemples :

<code>unsigned¹ char i ;</code>	définition d'un entier non signé sur 8 bits
<code>unsigned¹ char i, j, k ;</code>	définition de 3 entiers non signés sur 8 bits
<code>unsigned¹ char i, j, k ;</code>	même chose que ci-dessus, mais avec ce style d'écriture, il est possible de placer un commentaire en face de chaque variable.
<code>signed¹ char u ;</code>	définition d'un entier signé sur 8 bits. Il est possible d'utiliser u dans <code>if(u<0) {...}</code> .
<code>int m ;</code>	définition d'un entier signé sur 16 bits

1 : si seulement char est mentionné, le compilateur utilise un type signé ou non signé par défaut, selon les options de compilation. Voir la documentation du compilateur utilisé.

4) UTILISATION DE STRUCTURES SIMPLES DE CONTRÔLE DE FLUX

Les expressions utilisées dans les structures de contrôles de flux sont ici simples. *Pour le détail, voir l'abrégé sur le langage C.*

LECTURE D'UN BIT D'ENTRÉE AVEC UNE STRUCTURE IF

1ère possibilité	2 ^{ème} possibilité à préférer
<pre>VarTemp = PORTx & MASQUE ; /* VarTemp doit avoir au préalable été déclaré */ if (VarTemp ==1) { ...} else { ...}</pre>	<pre>if ((PORTx & MASQUE) == 1) { ...} else { ...}</pre>

VarTemp : pour Variable Temporaire. Cette variable doit au préalable avoir été déclarée, comme variable locale de préférence. *Voir § 6.*

== est un opérateur relationnel à ne pas confondre avec l'opérateur d'affectation. L'expression qui contient cet opérateur vaut 1 si l'expression est vraie, 0 si elle est fausse.

ATTENTE DU CHANGEMENT D'ÉTAT D'UNE ENTRÉE AVEC UNE BOUCLE WHILE

Exemple donné à titre pédagogique. Souvent à éviter (explication donnée plus tard).

<pre>while ((PORTx & MASQUE) != 1) ;</pre>	Tant que la condition n'est pas vraie, ne rien faire (; = instruction vide)
--	--

ATTENTE DE CHANGEMENT D'ÉTAT D'UNE ENTRÉE ET D'UNE FIN DE TEMPORISATION

```
while ( (FinTempo == FAUX) && ((PORTx & MASQUE) != 1))
;
```

L'expression qui correspond à la condition entre () comprend 2 expressions à valeur booléenne connectées par un ET logique. La deuxième de ces expressions ((PORTx & MASQUE) != 1) réalise un masquage avec un ET bit à bit qui n'est pas de même nature que le ET logique.

TRANSFERT DE 12 OCTETS VERS UN AFFICHAGE MULTIPLEXÉ À LEDS

```
for(NumAff = 0 ; NumAff <12 ; NumAff ++)
```

```

/* instruction pour extraire un octet d'un tableau et l'envoyer vers l'affichage. La position
de l'octet à extraire dépend de NumAff */
/* diverses autres instructions */
}

```

La suite d'instructions ci-dessous réalise la même chose que ci-dessus :

```

NumAff =0 ;
while (NumAff <12)
{...
NumAff ++ ;
...}

```

Une variante utilisant la possibilité de post-incrémentation

```

NumAff=0 ;
while (NumAff++<12) /* NumAff est incrémenté
après évaluation de l'expression */
{...
...}

```

RÉALISATION D'UNE BOUCLE SANS FIN

Un programme pour système embarqué contient une boucle sans fin. Celle-ci peut être réalisée de différentes façons.

<pre> for (; ;) {...} /* corps de la boucle sans fin */ </pre>	<pre> while (1) /* ou n'importe quelle valeur sauf 0*/ {...} /* corps de la boucle sans fin */ </pre>
---	---

Pour les explications, voir l'abrégé sur le langage C.

5) UTILISATION DE FONCTIONS

Une fonction correspond à un sous programme¹.

Voir le § sur les fonctions dans l'abrégé sur le C (définition, utilisation, valeur retournée, etc.)

Les fonctions sont utilisées pour diminuer la taille du code produit et/ou pour améliorer la lisibilité d'un programme

5.1) UTILISATION DE FONCTIONS POUR LA DIMINUTION DE LA TAILLE DU CODE PRODUIT

Lorsqu'une portion de programme est utilisée plusieurs fois, l'emploi d'une fonction permet de réduire la taille du code produit.

Le code pour la définition de la fonction existe une seule fois. Chaque appel de la fonction correspond à un appel de sous programme.

¹ Dans quelques cas exceptionnels, une fonction ne correspond pas à un sous programme. C'est le cas avec certains compilateurs pour des µCs avec très faibles ressources.

FONCTION SANS PARAMÈTRE

Ex :

```
void main(void) {
...
for ( ;;) {
...
    Tempo1() ; // appel de la fonction
...
    Tempo1() ;
...
}
}
```

Avant l'appel de la fonction, il faut absolument placer soit la définition de la fonction, soit une déclaration si la définition est placée après ou sur un autre fichier.

La définition décrit ce que fait la fonction et donne des informations au compilateur sur la valeur de retour et sur le ou les paramètres.

La déclaration donne au compilateur des informations sur la valeur de retour et le ou les paramètres pour que le compilateur prévoie les emplacements mémoire nécessaires.

Structure du programme avec définition avant l'appel

```
...
void Tempo1 (void) /* en-tête de la fonction*/
{...} /* corps de la fonction qui décrit ce que
réalise la fonction */
...
void main(void) {
...
for ( ;;) {
...
    Tempo1() ; // appel de la fonction
...
    Tempo1() ;
...
}
}
```

Structure du programme avec définition après l'appel

```
...
void Tempo1 (void) ; /* déclaration fonct.*/
...
void main(void) {
...
for ( ;;) {
...
    Tempo1() ; // appel de la fonction
...
    Tempo1() ;
...
}
}
...
void Tempo1 (void) /* en-tête de la fonction*/
{...} /* corps de la fonction qui décrit ce que
réalise la fonction */
```

FONCTION AVEC PARAMÈTRE / PASSAGE DE PARAMÈTRE

La fonction de l'exemple précédent n'utilise aucun paramètre et ne renvoie aucune valeur. Il est possible d'utiliser la même fonction pour différentes temporisations. Lors de l'appel, on utilise un paramètre qui correspond ici à la durée de la temporisation.

Ex :

```
void main(void) {
```

```

...
for ( ;;) {
    ...
    Tempo(10) ; // appel de la fonction avec le paramètre 10
    ...
    Tempo1(100) ; // appel de la fonction avec le paramètre 100
    ...
}

```

La définition de la fonction est la suivante :

```

void Tempo1(unsigned char Nb_ms) {
    unsigned char i; /* variable locale → voir plus loin §6 et abrégé C, § Constitution d'un
programme / Donnée / Portée */
    for (i=0 ; i<Nb_ms ;i++) {
        .... // instructions pour tempo élémentaire 1 ms
    }
}

```

Le paramètre passé lors de l'appel de la fonction s'appelle le paramètre effectif ou l'argument. Le paramètre utilisé lors de la définition est dit paramètre muet. Son nom n'est utilisé que dans la définition.

Lors de l'exécution du programme, avant l'appel du sous programme (qui correspond à la fonction), le paramètre est placé dans un ou plusieurs registres ou cases mémoire. Si dans le programme source le paramètre est une variable, une **copie** de cette variable est faite. Le sous programme lit le ou les registres ou cases mémoire qui contiennent la copie de la variable.

Ce type de passage de paramètre est dit par copie de valeur. C'est le plus fréquent dans le cas de programmes simples.

Si on raisonne uniquement au niveau du programme source, on peut dire que lors de l'appel de la fonction, le code de la fonction est exécuté avec le paramètre formel qui reçoit une copie du paramètre effectif ou argument.

FONCTIONS AVEC PLUSIEURS PARAMÈTRES

Une fonction peut avoir plusieurs paramètres. Pour la transmission des paramètres, seul importe l'ordre des paramètres.

Ex :

```
AfficheUneLigneLCD(« Bonjour », LIGNE_1) ; // appel de la fonction avec 2 paramètres
```

L'en-tête de la définition de la fonction est :

```
void AfficheUneLigneLCD(unsigned char* PtMsg, unsigned char NumLigne)
```

Lors de l'exécution de la fonction, si on raisonne uniquement au niveau du programme source, PtMsg reçoit une copie du 1^{er} paramètre (*type non détaillé ici*) et NumLigne reçoit la valeur LIGNE_1 (constante définie auparavant. Voir plus loin pour la définition)

Pour se souvenir du mécanisme de passage, on peut utiliser le schéma ci-dessous :

Appel	Définition
<pre>AfficheUneLigneLCD(« Bonjour » LIGNE_1)</pre>	<pre>void AfficheUneLigneLCD(unsigned char* PtMsg, unsigned char NumLigne)</pre>

L'appel et l'en-tête de la définition sont placés sur plusieurs lignes pour une meilleure lisibilité.

5.2) UTILISATION DE FONCTIONS POUR LA LISIBILITÉ D'UN PROGRAMME

Les fonctions peuvent être utilisées pour organiser de façon simple et lisible un programme en C.

Ex : thermomètre à affichage alphanumérique

```
void main(void) {
  Init();
  AffichageMsgAccueil();
  for (;;) {
    AcquisitionAN1(); // AN1 est l'entrée analogique image de la température
    Calcul();
    Affichage();
    Tempo();
  }
}
```

Dans l'exemple ci-dessus, les fonctions utilisées n'ont pas de valeurs de retour et pas de paramètres.

Les passages de données entre fonctions s'effectuent par variables globales. Voir plus loin le § 6 et l'abrégi du langage C, § Constitution d'un programme/Donnée/Portée.

Par exemple la fonction AcquisitionAN1 range dans une variable globale le résultat de la conversion analogique / numérique. Cette variable globale est utilisée par la fonction Calcul(). Cette dernière fonction place le résultat du calcul dans une autre variable globale, etc. Les variables globales sont de classe de mémorisation statique → un emplacement mémoire leur est réservé durant toute l'exécution du programme.

Au lieu d'utiliser les variables globales pour le passage de paramètres, on peut utiliser la copie de valeur, comme présenté en 5.2)

Dans l'exemple précédent, à la place de :

```
AcquisitionAN1();
Calcul();
Affichage();
```

On peut utiliser :

```
Affichage(Calcul(AcquisitionAN1()));
```

Les définitions des fonctions doivent bien entendu être différentes dans les 2 cas.

Dans le 2^{ème} cas, la fonction AcquisitionAN1() est d'abord appelée. Elle renvoie un résultat qui est utilisé en paramètre de la fonction Calcul() qui est ensuite appelée. Cette fonction Calcul renvoie elle même une valeur qui est le paramètre de la fonction Affichage() appelée en dernier.

5.3) GESTIONNAIRE D'INTERRUPTION (INTERRUPT HANDLER)

Un gestionnaire d'interruption (Interrupt Handler) est une fonction qui correspond à un programme de traitement d'interruption (ISR = Interrupt Service Routine)
Voir document d'introduction aux µP et µC, § 9.3) Interruption.

Voir l'abrégé sur le C, § Particularité pour les systèmes embarqués / Gestionnaire d'interruption.

Rappel : pour que les interruptions puissent être prises en compte, il faut les autoriser en positionnant correctement certains bits de registres de contrôle.
Voir pour le PIC les bits GIE, PIE pour les autorisations globales et les différents bits pour les interfaces et les périphériques intégrés.

GESTIONNAIRE D'INTERRUPTION AVEC LE PIC ET LE COMPILATEUR HITECH

Comme le PIC n'a qu'un seul programme de traitement d'interruption, il faut absolument commencer par rechercher la source de l'interruption, lorsque plusieurs sources sont possibles. Ceci s'effectue en testant les drapeaux (bits d'état). Une interruption provoque une mise à 1 du drapeau correspondant.

Pour ne pas retourner dans le programme de traitement d'interruption, il faut remettre le drapeau à 0 avant de revenir au programme principal.

<code>void interrupt NomQuelconque(void) {</code>	L'identificateur de la fonction n'est utilisé que dans la définition Son non importe peu
<code>....</code>	Déclarations éventuelles des variables locales
<code>if (drapeau1==1) {.... drapeau1=0 ;} if (drapeau2==1) {... drapeau2=0;}</code>	recherche de la source d'interruption si plusieurs sources possibles acquiescement indispensable
<code>}</code>	

EXEMPLE SIMPLE

Avec un PIC, une interruption fait clignoter 4 fois un voyant situé sur le PORTD.
Le programme source du gestionnaire d'interruption est le suivant :

```
interrupt void TraitInter(void)
{
  unsigned int i ; // variable locale
  unsigned char NbClignot ; // variable locale
  if (INTF ==1) {
    for(NbClignot=4;NbClignot!=0;NbClignot--)
    {
      RD0=1; // RD0 de type bit (extension C ANSI) défini auparavant
```

```

        for(i=0xFFFF;i!=0;i-) // temporisation à usage pédagogique
            ;
        RDO=0;
        for(i=0xFFFF;i!=0;i-)
            ;
    }
    INTF=0;
}
... // recherche pour les autres sources d'interruption
}

```

6) DÉCLARATIONS DE DONNÉES. VARIABLES LOCALES ET GLOBALES

Le compilateur peut allouer un emplacement pour une variable, lors de l'exécution du programme :

- pour toute la durée de l'exécution
- ou seulement durant l'exécution d'une partie du programme.

La partie du fichier du programme source où cette variable peut être utilisée est respectivement :

- tout le fichier source
- une fonction

Le premier type est une **variable globale** ; le second type est une **variable locale**.

C'est l'emplacement qui permet de déterminer la catégorie de variable.

Une variable globale est définie en dehors de toute fonction.

Une variable locale est définie au début d'une fonction, avant les instructions proprement dites.

Une variable globale peut être utilisée dans plusieurs fichiers source. La réservation de mémoire ne doit s'effectuer que dans un des fichiers source.

Ce fichier source contient la définition de la variable globale. Exemple `char VarGlob ;`

Le ou les autres fichiers source contiennent la déclaration. Exemple `extern char VarGlob.`

Les variables globales peuvent être utilisées pour le passage d'informations entre fonctions. Elles sont indispensables pour le passage d'informations au gestionnaire de traitement d'interruption.

Pour plus de détail, voir l'abrégé sur le C.

7) PASSAGES D'INFORMATIONS VERS UNE FONCTION

Le passage d'informations vers une fonction s'effectue par

- variables globales
- paramètres

Il existe 2 types de passages de paramètres :

- par copie de valeur (déjà présenté au § 5.1)
- par adresse

Le passage de paramètres par copie de valeur est le plus utilisé pour les cas simples. Avec un passage de paramètres de ce type, la fonction ne peut modifier la valeur de la variable dont la copie est utilisée. En effet l'emplacement de la variable n'est pas connu de la fonction, seul l'emplacement de la copie est connu.

Avec le passage de paramètre par adresse, c'est l'adresse d'une donnée qui est passée à la fonction. Ce type de passage de paramètre est utilisé pour passer un tableau ou une chaîne de caractère en paramètre. C'est l'adresse de ce tableau ou de la chaîne de caractère qui est réellement passée.

Pour plus de détail, voir Abrégé du langage C / Fonctions / Transmission et type de paramètres. Voir aussi Types de données complexes / Tableaux en paramètres d'une fonctions.

8) TABLEAUX, POINTEURS

Dans les systèmes embarqués, les tableaux sont principalement utilisés pour mémoriser une succession de :

- signaux à délivrer
- informations à afficher (codes des caractères à afficher, etc.)
- résultats de mesures

Les tableaux peuvent être manipulés :

- directement avec leurs identificateurs et des indices
- indirectement par l'adresse de l'emplacement mémoire utilisé

Dans ce dernier cas, on utilise un pointeur. Un pointeur est obligatoire pour le passage de tableau en paramètre d'une fonction.

Dans certains cas, l'utilisation d'un pointeur à la place de l'identificateur et d'indice(s) permet de diminuer la taille du code produit.

8.1) EXEMPLE D'UTILISATION D'UN TABLEAU POUR GÉNÉRER DES SIGNAUX

On désire délivrer les signaux ci-contre :

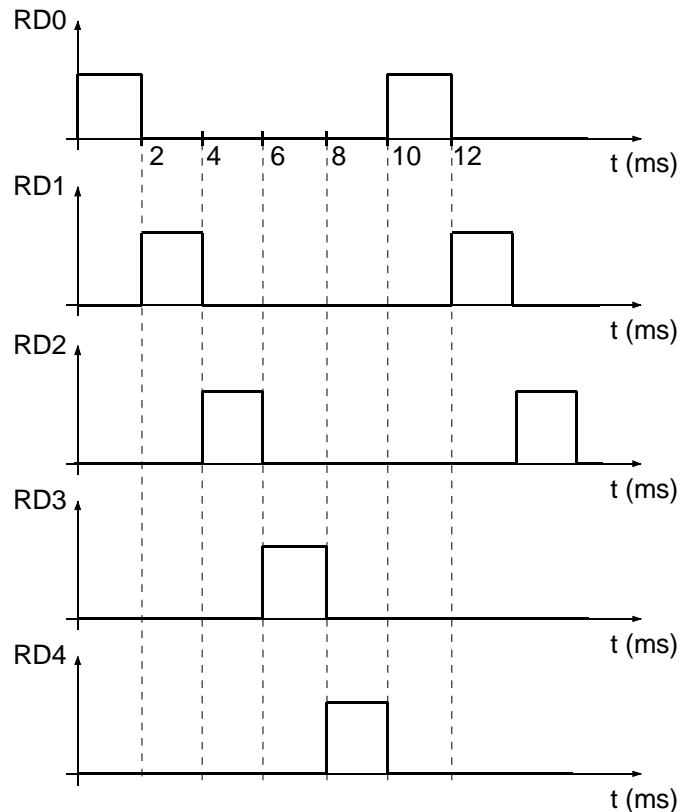
Toutes les sorties utilisées sont sur le même port : le port D.

Les différents états des sorties peuvent être mémorisés dans le tableau suivant :

```
CdesSorties[5]={  
0b00000001,  
0b00000010,  
0b00000100,  
0b00001000,  
0b00010000 }
```

A chaque fois que la sortie doit être modifiée, il faut utiliser le tableau.

Une variable, dont la valeur est conservée entre 2 appels est utilisée. Cette variable vaut de 0 à 4 ; elle est incrémentée avant ou après l'utilisation du tableau. Elle est réinitialisée à 0 lorsque tous les éléments du tableau ont été utilisés.



Ex de code

```
PORTD= CdesSorties[EtatSorties] ;  
EtatSorties += 1 ; // ou EtatSorties ++  
if (EtatSorties == 5)  
    EtatSorties = 0 ; // réinitialisation
```

```
variante  
PORTD= CdesSorties[EtatSorties++] ;
```

Ce code remplace de façon très compacte les lignes suivantes :

```
switch (EtatSorties) {  
case 0 : PORTD = 0b00000001 ; break ;  
case 1 : PORTD = 0b00000010 ; break ;  
case 2 : PORTD = ....  
....
```

Remarque : le compilateur ne produit pas nécessairement un code compact à partir d'une partie de programme source compacte.

8.2) EXEMPLE D'UTILISATION D'UN TABLEAU POUR DES MOTIFS À AFFICHER

Un module d'affichage est constitué de 12 afficheurs à LEDs de 5*8 LEDs. L'affichage est multiplexé groupe de colonnes par groupe de colonnes.

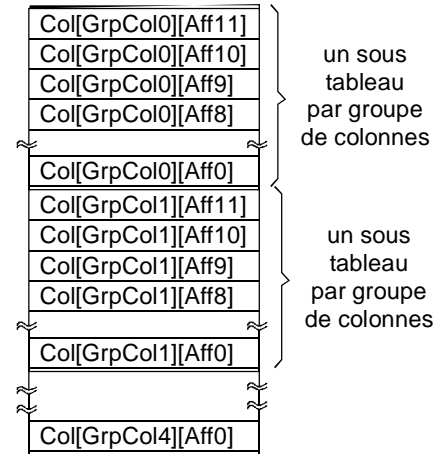
Le tableau qui contient les motifs à afficher est organisé de façon à faciliter l'écriture de la portion de programme de gestion du multiplexage.

L'organisation du tableau est la suivante.

Pour un accès avec les indices du tableau, on utilise 2 variables GrpCol et Aff qui valent respectivement de 0 à 4 et de 0 à 11.

Chaque accès en lecture s'effectue par :

... = Col[GrpCol][Aff] ;



8.3 UTILISATION D'UN POINTEUR POUR ACCÉDER AUX ÉLÉMENTS D'UN TABLEAU

Voir l'abrégé sur le C.

L'utilisation décrite ici n'est pas l'emploi le plus fréquent d'un pointeur. Elle permet cependant de diminuer sensiblement la taille du code produit dans certains cas (µC PIC et compilateur Hitech par exemple).

On reprend l'exemple du 8.1)

Il peut être traduit par :

<pre>unsigned char* PtCdesSorties ; PtCdesSorties = CdesSorties ; PORTD=*PtCdesSorties ; PtCdesSorties +=1 ;</pre>	<pre> déclaration d'un pointeur sur un caractère CdesSorties est l'adresse de début du tableau. PtCdesSorties est initialisé à cette adresse. PORTD = le contenu de l'adresse donnée par PtCdesSorties PtCdesSorties est incrémenté de la taille en octet de la donnée pointée (ici un octet).</pre>
--	---

Il faut ensuite réinitialiser le pointeur avec l'adresse de début du tableau lorsque tous les éléments ont été utilisés

```
if (PtCdesSorties == PtCdesSorties + 5) // on utilise ici l'arithmétique des pointeurs.  
    PtCdesSorties = CdesSorties ;
```