INITIATION A STEP 7

INITIATION A STEP7

A) Les automates SIMATIC

Les automates **SIMATIC S7-300** sont modulaires et destinés à des applications d'entrée et de milieu de gamme. Ils disposent de nombreuses possibilités (alimentations, CPU, coupleurs de réseaux, cartes d'entrées et sorties, etc.) adaptées aux différentes tâches d'automatisation. Il existe une série spéciale appelée S7-300F pour les systèmes d'automatisation à haute sécurité.

Les automates S7-300 et S7-300F fonctionnent normalement entre 0 et 60°C, 95% d'humidité au maximum. Ils supportent des accélérations de 1g en cas de choc accidentel.

La série S7-300 OUTDOOR est prévue pour les environnements hostiles et les systèmes embarqués. Elle fonctionne entre –25 et +70°C, 95% d'humidité avec condensation possible et supporte des vibrations fortes et des accélérations de 15g au maximum, en cas de choc accidentel.

Architecture :

Dans la série S7-300, on a le choix entre une vingtaine de CPU standard, avec des spécificités différentes. Il existe aussi des CPU pour des usages particuliers. Certaines comprennent dans le même boîtier, des entrées, des sorties tout ou rien ou analogiques, des ports séries, etc..

L'ensemble des cartes additionnelles est très riche. En plus des cartes de communication, il existe des cartes d'entrées et de sorties tout ou rien, des cartes combinant entrées et sorties, des modules de comptage, des modules de positionnement, des modules permettant des calculs booléens programmables, très rapides, des modules de commande de moteur, de puissance, de régulation, etc..

Les automates **SIMATIC S7-400** sont destinés à des applications moyennes et supérieures.

Ils ont des possibilités beaucoup plus importantes (plusieurs CPU pour le même automate, dispositifs de redondance, de calculs rapides, de mémoire de masse, etc.).

L' Atelier logiciel utilisable pour la programmation des automates SIMATIC s'appelle STEP 7. Il est disponible à 3 niveaux : STEP 7 LITE (automates S7-300), STEP 7 (S7-300, S7-400) et STEP 7 Professional.

B) Principes de conception d'un programme

B-1) Généralités :

En fonctionnement, deux programmes différents s'exécutent dans une CPU, le système d'exploitation et le programme utilisateur.

Le système d'exploitation organise toutes les fonctions et procédures dans la CPU, qui ne sont pas liées à une tâche spécifique d'automatisation. Il assure la procédure de mise en fonctionnement, l'actualisation de la mémoire image des entrées et la création de la mémoire image des sorties, l'appel du programme utilisateur, les gestions des alarmes, des traitements d'erreurs, des zones mémoire à utiliser. Quelques paramètres du système d'exploitation sont modifiables par l'utilisateur.

Le programme utilisateur, chargé dans la CPU, contient tous les traitements nécessaires à la tâche d'automatisation. Il sera organisé en BLOCS.

- Blocs d'organisation (OB) qui déterminent la structure du programme utilisateur ;

- Blocs fonctionnels (FB) associés automatiquement à une « mémoire » (DB d'instance) ;
- Blocs fonctions (FC) contenant des sous programmes pour les fonctions fréquemment utilisées ;
- Blocs fonctionnels système (SFB) intégrés à la CPU, ils permettent de réaliser quelques fonctions système importantes ;
- Blocs fonctions système (SFC) intégrés à la CPU ;
- Blocs de données d'instance (DB d'instance) associés aux FB et aux SFB quand ces derniers sont appelés ;
- Blocs de données globaux (DB) contenant les données utilisateur communes à tous les blocs.

Le nombre de blocs autorisés par type de bloc, et la longueur possible de ces blocs, dépend de la CPU utilisée.

B-2) Traitement cyclique du programme :

L'exécution cyclique du programme utilisateur constitue le traitement normal pour les automates programmables. Le système d'exploitation appelle le bloc OB1 cycliquement. Ce dernier appellera d'autres blocs (voir le schéma plus loin), et déclenchera ainsi le traitement cyclique du programme utilisateur.

Déroulement :

-Le système d'exploitation démarre le dispositif de surveillance de la durée du cycle (« chien de garde » réglé à 150 ms pour la CPU utilisée en TP) ;
-La CPU écrit les valeurs de la mémoire image des sorties dans les modules de sortie ;
-La CPU lit l'état des entrées dans les modules d'entrées et met à jour la mémoire image des entrées ;
-La CPU traite le programme utilisateur (appel de OB1) ;
-A la fin d'un cycle, le système d'exploitation traite les tâches en attente (ex : communications) ;
-La CPU revient alors au début du cycle et relance la surveillance du temps de cycle.

Lorsque le programme utilisateur se sert d'entrées (Ei.j) et de sorties (Ak.m), la CPU n'accède pas directement aux modules d'entrées et de sorties mais utilise une zone de mémoire interne contenant une image de ces entrées et de ces sorties (MIE et MIS). Cette image sera cohérente pendant l'exécution d'un cycle.
Le temps maximal de cycle est variable selon les CPU. Il peut être modifié. En cas de dépassement accidentel de ce temps maximal, un programme d'erreur peut être exécuté (OB80).

Possibilité d'interruption :

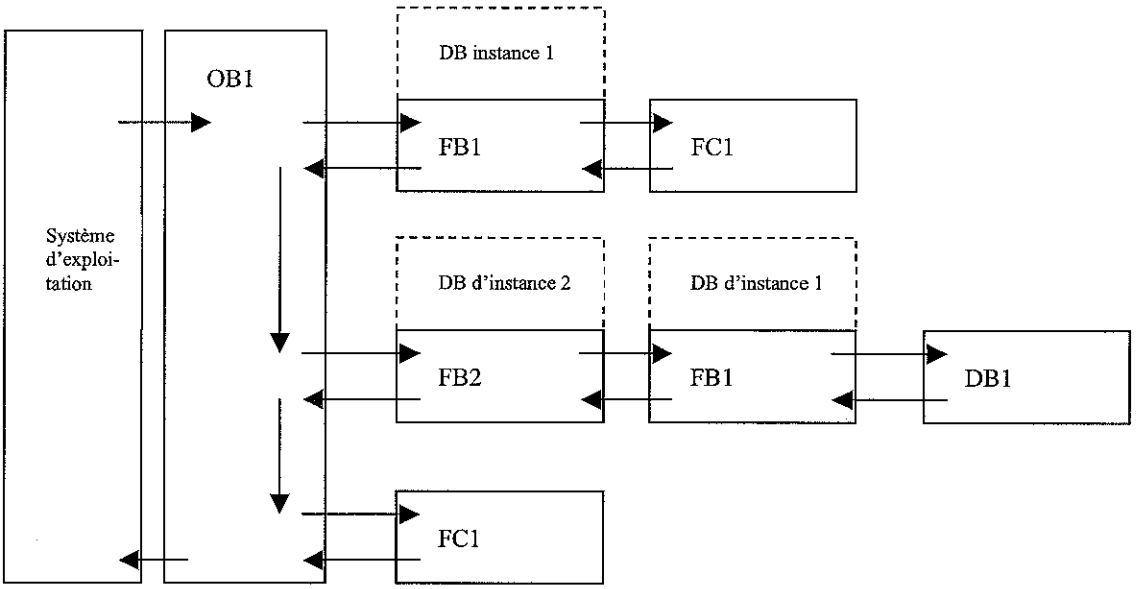
Des événements peuvent provoquer une interruption du traitement cyclique, et dérouter l'exécution du programme en cours vers un autre, plus prioritaire. Ces événements peuvent être horaires (OB10 à OB17), temporisés (OB20 à OB23), cycliques (OB30 à OB38), liés au processus (OB40 à OB47), liés aux erreurs possibles (OB80 à OB87), etc.. Il existe des blocs spéciaux qui s'exécutent lors du démarrage à chaud (OB100) et lors du démarrage à froid (OB102).

Tâche de fond :

Il est possible d'exécuter un programme utilisateur (de priorité faible) en « tâche de fond » (OB90). Tout ou partie de ce programme s'exécutera à la fin du cycle, s'il reste du temps entre la valeur maximale du temps de cycle possible défini, et le temps de cycle effectif. Il est alors nécessaire de fixer judicieusement le temps de cycle maximal possible.

De très nombreuses autres possibilités existent mais elle sortent du cadre de cette initiation à STEP 7.

B-3) Exemple de structure de programme :



OB1 est un « chef d'orchestre », il organise le travail des autres blocs.

FB1 et FB2 disposent de leurs données propres (contenues dans leur DB d'instance).

DB1 est un bloc de données global (les données contenues dans ce bloc sont éventuellement utilisables par tous les autres blocs) .

Pour la programmation, on procédera dans l'ordre inverse de cette structure. On programmera d'abord FC1, DB1, puis FB1 avec son bloc DB d'instance, puis FB2 avec son bloc DB d'instance, et enfin OB1.

La « profondeur d'imbrication » possible dépend de la CPU utilisée (8 pour la CPU314 utilisée en TP).

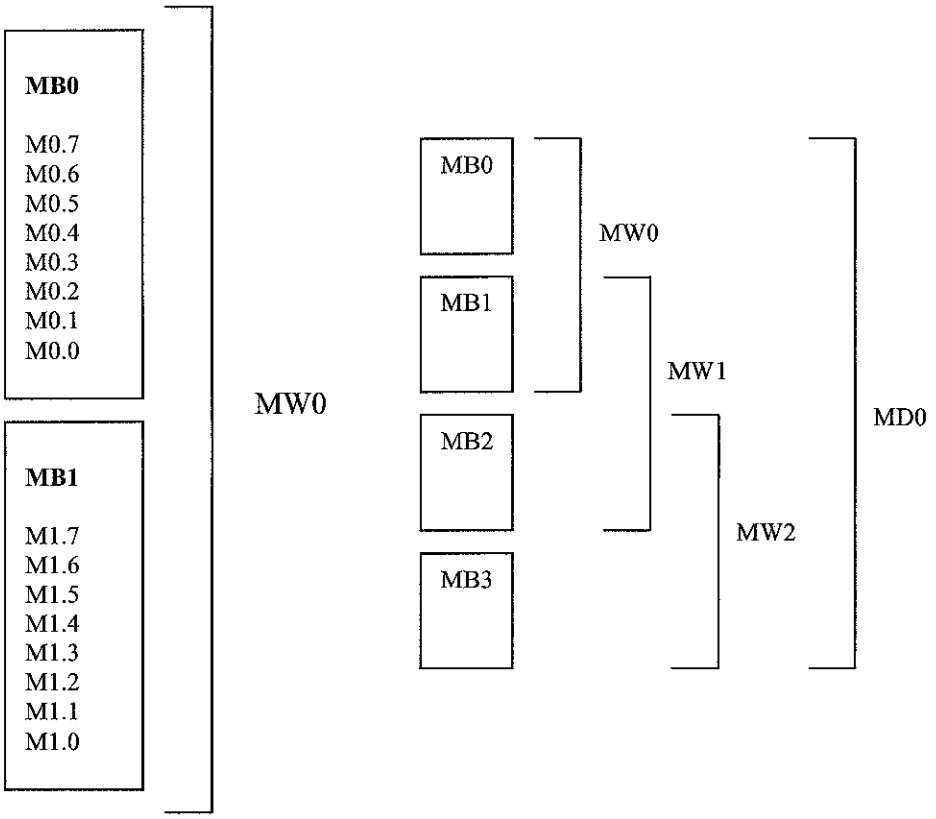
Remarque : Un programme utilisateur court pourrait être intégralement codé dans OB1 ...

Temps de cycle : Connaissant la structure du programme, on peut calculer le temps de cycle automate. Certaines CPU sont conçues pour être particulièrement rapides.

B-4) Utilisation de la mémoire.

STEP 7 utilise des bits ou « memento » (exemple : M0.0), des octets (exemple : MB3), des mots de 16 bits (exemple : MW1), des mots doubles (exemple : MD4). (les 16 premiers octets, de MB0 à MB15, sont rémanents dans notre CPU 314).

Exemple : le mot MW0 est composé de l'octet MB0 et de l'octet MB1. L'octet MB0 est composé des 8 bits de M0.0 à M0.7 et l'octet MB1 est composé des 8 bits de M1.0 à M1.7. (voir ci-dessous)



ATTENTION : Dans MW0, MB1 est l'octet de poids faible et MB0 est l'octet de poids fort.

Dans un octet, on peut écrire en hexadécimal (de B#16#0 à B#16#FF). Dans un mot de 16 bit on peut écrire des entiers de –32768 à +32767, en binaire (de 2#0 à 2#1111.1111.1111.1111), en hexadécimal (de W#16#0 à W#16#FFFF). Dans les mots doubles, on peut écrire des entiers sur 32 bits (de –2147483648 à +2147483647) et des nombres à virgule flottante, etc..

Il faudra veiller, lors de la programmation, à ne pas utiliser un bit appartenant non intentionnellement à un mot déjà utilisé par ailleurs, cela conduirait à une panne difficilement détectable.

Entrées et sorties

Les mots images des entrées s'appellent EW. Par exemple, le mot EW0 est composé des deux octets EB0 et EB1. EBO est composé des 8 entrées E0.0 à E0.7 et EB1 est composé des 8 entrées suivantes, de E1.0 à E1.7. (Ces mots sont déterminés au moment de la configuration matérielle de l'automate).

Les mots images des sorties s'appellent AW et se décomposent comme les mots d'entrées.

Le nom des entrées et des sorties est déterminé par la position des cartes d'entrées et de sorties sur le fond de panier de l'automate. Ici, chaque emplacement dispose de 4 octets. Si la première carte est à 32 entrées, celles ci s'appelleront :

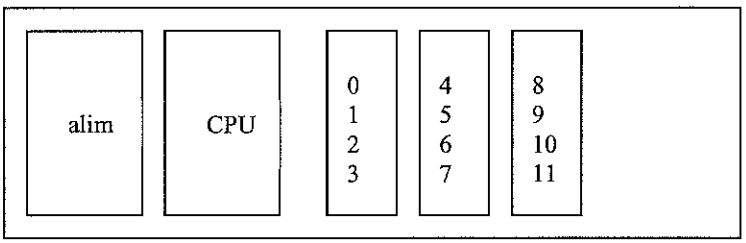
E0.0, E0.2, E0.3 ... E0.7 / E1.0, E1.2, ... E1.7 / E2.0, ..., E2.7 / E3.0, ... , E3.7
(4 octets d'entrée : EB0, EB1, EB2, EB3, mots EW0 et EW2)

Pour le second emplacement, si c'est une carte à 16 sorties :

A4.0, A4.1, ..., A4.7 / A5.0, ... , A5.7 (octets AB4 et AB5, mot AW4)

Pour le troisième emplacement, si c'est une carte à 16 sorties :

A8.0, A8.1, ..., A8.7 / A9.0, ..., A9.7 (octet AB8 et AB9, mot AW8)



C) Langages de programmation

Il existe trois langages de programmation appelés LIST, CONT et LOG. Les blocs peuvent être écrits dans ces trois langages, au choix, indépendamment les uns des autres.

CONT est le langage à contacts ou langage à relais ou « ladder », pour les utilisateurs habitués aux schémas électriques.
LOG utilise les boîtes logiques de l'algèbre de Boole.
LIST est un langage en liste d'instructions permettant une programmation la plus proche possible du langage du processeur. C'est le langage le plus efficace quant à l'utilisation de la mémoire de l'automate et la réduction de la durée du temps de cycle. Il permet d'utiliser toutes les ressources de STEP7 contrairement aux deux précédents. Néanmoins, il est souvent d'usage dans les entreprises, de programmer en CONT les blocs de code susceptibles de subir des maintenances de la part de programmeurs occasionnels.
Le langage LIST est très riche. Les éléments les plus utilisés seulement sont cités ci-dessous.

C-1) Eléments de programmation simple, en LIST

Fonction ET

```
U     E1.0      //si l'entrée E1.0 est à « 1 »
U     E1.1      //et si l'entrée E1.1 est à « 1 »
=     A4.0      //alors, la sortie A4.0 est mise à « 1 »
```

Le résultat logique de l'opération (RLG) entre les deux entrées, est affecté (signe =) à la sortie A4.0.

Autre exemple :

```
U     E1.0      //si l'entrée E1.0 est à « 1 »
U     E1.1      //et si l'entrée E1.1 est à « 1 »
U     M3.0      //et si le bit M3.0 est à « 1 »
U     M3.1      //et si le bit M3.1 est à « 1 »
=     M3.2      //alors le bit M3.2 est mis à « 1 »
=     M3.3      //et le bit M3.3 est mis à « 1 »
```

Le RLG de l'opération logique est affecté au bit M3.2 et au bit M3.3.

Négation de ET

```
U     E1.0      //si l'entrée E1.0 est à « 1 »
UN    E1.1      //et si l'entrée E1.1 est à « 0 »
=     A4.0      //alors, la sortie A4.0 est mise à « 1 »
```

Fonction OU

```
O     E1.0      //si l'entrée E1.0 est à « 1 »
O     E1.1      //ou si l'entrée E1.1 est à « 1 »
=     A4.0      //alors, la sortie A4.0 est mise à « 1 »
```

Négation de OU

```
O     E1.0      //si l'entrée E1.0 est à « 1 »
ON    E1.1      //ou si l'entrée E1.1 est à « 0 »
=     A4.0      //alors, la sortie A4.0 est mise à « 1 »
```

OU exclusif

```
X     E1.0      //si l'entrée E1.0 = « 1 » et E1.1= « 0 »
X     E1.1      //ou si l'entrée E1.0 = « 0 » et E1.1= « 1 »
=     A4.0      //alors A4.0 = « 1 »
```

Combinaisons de ET et de OU

```
U     E0.0
U     M10.0
O
U     E0.2
U     M0.3
O     M10.1
=     A4.0
```

en langage Booléen, on écrirait : (E0.0 . M10.0)+(E0.2 . M0.3)+M10.1 = A4.0

Dans le cas suivant, des parenthèses sont nécessaires :

```
U(
O     E0.0
O     M10.0
)
U(
O     E0.2
O     M10.3
)
U     M10.1
=     A4.0
```

en langage Booléen, on écrirait : (E0.0+M10.0).(E0.2+M10.3).M10.1 = A4.0

il est toujours possible d'écrire le problème ci-dessus avec des bits intermédiaires :

```
O     E0.0
O     M10.0
=     M0.0
O     E0.2
O     M10.3
```

```
=     M0.1
U     M0.0
U     M0.1
U     M10.1
=     A4.0
```

SET et RESET

```
U     E1.0      //si E1.0 = « 1 »
S     A4.0      //alors A4.0 = « 1 »
U     E1.1      //si E1.1 = « 1 »
R     A4.0      //alors A4.0 = « 0 »
```

Actions sur le RLG

NOT inverse la valeur du RLG

SET met à « 1 » la valeur du RLG

CLR met à « 0 » la valeur du RLG

Exemple :

```
...
SET
=     M10.0
```

M10.0 prendra la valeur « 1 » quel que soit le programme qui précède.

Front montant, front descendant (Rappel : l'utilisation de fronts montants ou descendants provenant de capteurs, apporte un risque et nécessite des précautions de câblage particulières des entrées).

```
U     E1.0      //si E1.0 = « 1 »
FP    M1.0      //et si M1.0 présente un front montant
=     A4.0      //A4.0 sera mis à « 1 » pendant un cycle de OB1

U     E1.0      //si E1.0 = « 1 »
FN    M1.0      //et si M1.0 présente un front descendant
=     A4.0      //A4.0 sera mis à « 1 » pendant un cycle de OB1
```

Saut vers une étiquette

Il existe un saut inconditionnel (SPA) et un saut conditionnel (SPB ou SPBN). Le saut peut s'exécuter vers l'avant ou vers l'arrière (risqué !) mais absolument à l'intérieur d'un même bloc. La destination du saut, dans ce bloc doit être unique. L'étiquette comporte 4 caractères au maximum.

```
...
...
SPA   ETIQ  //saut inconditionnel vers l'étiquette ETIQ
...
ETIQ: ...

U     M1.1
U     M1.2
SPB   ABCD  //le saut vers l'étiquette ABCD aura lieu si le RLG vaut « 1 »
...
ABCD: ...

U     M1.3
U     M1.4
```

SPBN EFGH //le saut vers l'étiquette EFGH aura lieu si le RLG vaut « 0 »
...
EFGH: ...

Il existe plusieurs autres façons d'effectuer des sauts. Elles sortent du cadre de cette initiation à STEP 7.

C-2) Opérations utilisant les accumulateurs

Selon la CPU utilisée, on peut disposer de 2 accumulateurs de 16 bits ou de 2 accumulateurs de 32 bits ou de 4 accumulateurs de 16 bits ou de 4 accumulateurs de 32 bits.
Les exemples suivants concerneront la CPU314 (T.P.) qui a 2 accumulateurs de 32 bits.

Les 2 accumulateurs s'appellent **ACCU1** et **ACCU2**.

On peut représenter les 4 octets de ACCU1 (ou de ACCU-2) comme ceci :

ACCU1 H-H	ACCU1 H-L	ACCU1 L-H	ACCU1 L-L

ATTENTION : Tous les ordres décrits ci-dessous s'exécutent <u>sans tenir compte</u> du résultat logique (RLG) c'est à dire, <u>à chaque cycle</u> du programme.

L **MB10** // On charge l'octet MB10 dans ACCU1. Il va venir occuper ACCU1 L-L

L **MW6** // On charge le mot MW6 dans ACCU1. Il occupera ACCU1 L-H et ACCU1 L-L

L **MD8** // On charge le mot double MD8 dans ACCU1 qu'il va occuper complètement.

L **EB0** // les 8 entrées (E0.0, ..., E0.7) passent dans ACCU1 L-L

L **50** // Le nombre entier 50 dans ACCU1 L-H et ACCU1 L-L (16 bits)

L **#1** // placer 1 dans ACCU1 sous forme d'une constante entière sur 32 bits

A chaque fois que l'on charge ACCU1, *son contenu préalable passe dans ACCU2* qui a la même structure. Si l'on charge une seconde fois ACCU1, le premier contenu est perdu.

T **MB0** // transfère le contenu de ACCU1 (ici ACCU1 L-L) dans l'octet MB0.

T **MW1** // transfère le contenu de ACCU1 (ACCU1 L-L et ACCU1 L-H) dans MW1

T **MD30** // tout ACCU1 passe dans MD30.

L **EW0** // les 16 entrées sont « recopiées dans ACCU1 puis transférées dans le mot
T **AW4** // de sortie AW4, ces sorties « recopient » alors les entrées.

TAK // permute les contenus de ACCU1 et ACCU2.

PUSH // passe ACCU1 dans ACCU2 sans modifier ACCU1.

POP // passe ACCU2 dans ACCU1 sans modifier ACCU2.

INC **n** // incrémenter ACCU1 L-L de la valeur entière n.

Exemple : incrémenter de 1, l'octet MB22

L **MB22**
INC **1**

T MB22

DEC n // soustraire la valeur entière n de ACCU1 L-L

UW fait un « ET » bit à bit, du contenu de ACCU1 L avec le contenu de ACCU2 L. Le résultat est placé dans ACCU1 L

Exemple :

```
L     MW20      // fait un « ET » bit à bit entre MW20 et MW40 et place le résultat
L     MW40      // dans le mot de sortie AW4
UW
T     AW4
```

OW fait un « OU » bit à bit, du contenu de ACCU1 L avec le contenu de ACCU2 L. Le résultat est placé dans ACCU1 L.

XOW fait un « OU exclusif » bit à bit, du contenu de ACCU1 L avec le contenu de ACCU2 L. Le résultat est placé dans ACCU1 L.

UD fait un « ET » bit à bit, du contenu de ACCU1 avec le contenu de ACCU2. Le résultat est placé dans ACCU1.

OD fait un « OU » bit à bit, du contenu de ACCU1 avec le contenu de ACCU2. Le résultat est placé dans ACCU1.

XOD fait un « OU exclusif » bit à bit, du contenu de ACCU1 avec le contenu de ACCU2. Le résultat est placé dans ACCU1.

+I additionne ACCU1 L à ACCU2 L et place le résultat dans ACCU1 L sans changer ACCU2 L (nombres entiers sur 16 bits).

-I soustrait ACCU1 L de ACCU2 L et place le résultat dans ACCU1 L sans changer ACCU2 L (nombres entiers sur 16 bits).

***I** multiplie ACCU1 L par ACCU2 L et place le résultat dans ACCU1 L sans changer ACCU2 L (nombres entiers sur 16 bits).

/I divise ACCU2 L par ACCU1 L et place le résultat dans ACCU1 L sans changer ACCU2 L (nombres entiers sur 16 bits).

+R additionne ACCU1 à ACCU2 et place le résultat dans ACCU1 sans changer ACCU2 (nombres à virgule flottante sur 32 bits).

+D même chose mais avec des entiers 32 bits.

-R soustrait ACCU1 de ACCU2 et place le résultat dans ACCU1 sans changer ACCU2 (nombres à virgule flottante sur 32 bits).

-D même chose mais avec des entiers 32 bits.

***R** multiplie ACCU1 et ACCU2 et place le résultat dans ACCU1 sans changer ACCU2 (nombres à virgule flottante sur 32 bits).

***D** même chose mais avec des entiers 32 bits.

/R divise ACCU2 par ACCU1 et place le résultat dans ACCU1 sans changer ACCU2 (nombres à virgule flottante sur 32 bits).

/D même chose mais avec des entiers 32 bits.

ABS donne la valeur absolue du nombre contenu dans ACCU1 et place le résultat dans ACCU1.

Il existe de nombreux autres ordres de ce type (racine carrée, exponentielle, fonctions trigonométriques, etc.).

+ a additionne la constante entière **a** (16 ou 32 bits) à ACCU1, et porte le résultat dans ACCU1.

Exemple :

```
L     MW10        // ajoute 5 à MW10
+     5
T     MW10
```

ATTENTION : Tous les ordres décrits ci-dessous s'exécutent sans tenir compte du résultat logique (RLG) c'est à dire, à chaque cycle du programme, mais ils influent, une fois exécutés, sur le résultat logique.

==I <>I >I <I >=I <=I comparent ACCU2 L à ACCU1 L (entiers 16 bits)

exemple :

```
L     MW10   // M0.0 sera mis à « 1 » si le contenu de MW10 est plus grand que celui de MW20.
L     MW20
>I           // fait RLG = « 1 » si MW10 > MW20
S     M0.0
```

==D <>D >D <D >=D <=D comparent ACCU2 à ACCU1 (entiers 32 bits).

==R <>R >R <R >=R <=R comparent ACCU2 à ACCU1 (réels 32 bits).

Exemple :

```
L     MD10
L     1.35 E+02
>R
=     M2.0        // égal « 1 » si MD10 > 1.35 E+02
```

Il existe de nombreux autres ordres pour convertir le contenu des accumulateurs en BCD, en GRAY, pour arrondir, tronquer, faire des décalages et des rotations, complémenter, inverser l'ordre de octets, etc..

C-3) Temporisations

On peut ouvrir plusieurs temporisations simultanément, leur nombre varie avec la CPU utilisée (en général 256 temporisations).

On dispose de 5 modes simples appelées **SI, SV, SE, SS, SA.**

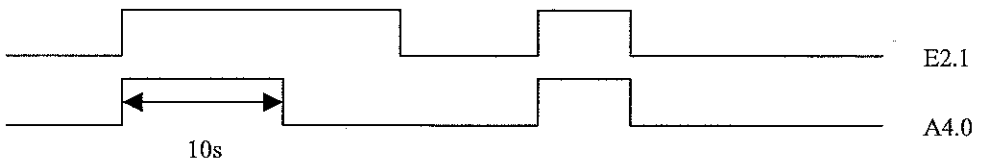
La durée de la temporisation doit être préalablement placée dans ACCU1. Il existe une façon évoluée de coder ce nombre, sous la forme : **S5T#*a*H_*b*M_*c*S_*d*MS**
H : heures, M : minutes, S : secondes et MS : millisecondes.
Temps maximal possible : 2H_46M_30S c'est à dire 9990 secondes.

Exemples : **S5T#10S** (10 secondes), **S5T#1M_35S_6MS** (1 minute et 35,006 secondes).

Temporisation SI :

```
U     E2.1        //si l'entrée E2.1 est à « 1 »
L     S5T#10S     //placer la durée prévue dans ACCU1 (ici : 10 secondes)
SI    T1          //lancer T1
```

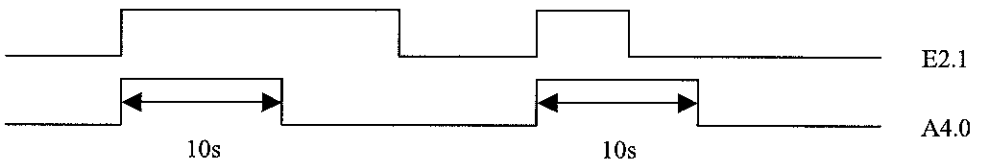
```
U     T1      //pendant que T1 est valide,
=     A4.0    //la sortie A4.0 doit être à « 1 »
```



Remarque : la chute de E2.1 entraîne l'arrêt de la temporisation T1.

Temporisation SV :

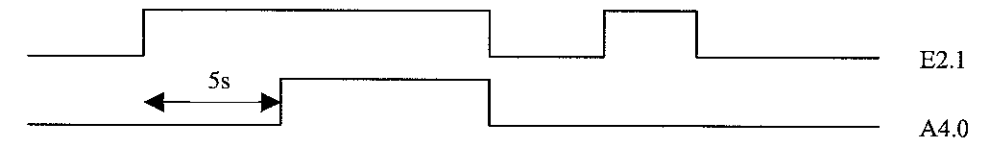
```
U     E2.1      //si l'entrée E2.1 est à « 1 »
L     S5T#10S   //placer la durée prévue dans ACCU1
SV    T2        //lancer T2
U     T2        //pendant que T2 est valide,
=     A4.0      //la sortie A4.0 doit être à « 1 »
```



Remarque : cette fois, la chute de E2.1 n'entraîne pas l'arrêt de la temporisation T2.

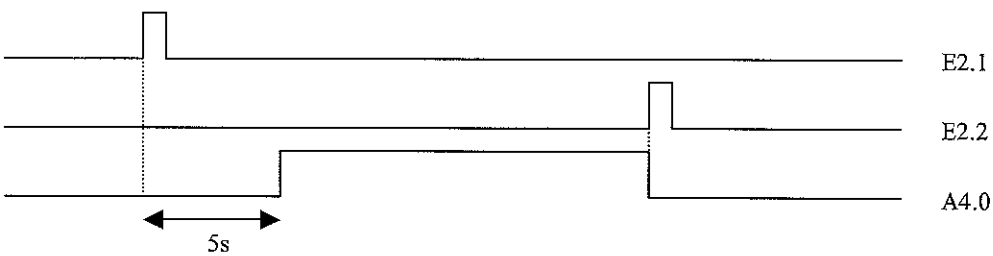
Temporisation SE :

```
U     E2.1      //si l'entrée E2.1 est à « 1 »
L     S5T#5S    //placer la durée prévue dans ACCU1
SE    T3        //lancer T3
U     T3        //pendant que T3 est valide,
=     A4.0      //la sortie A4.0 doit être à « 1 »
```



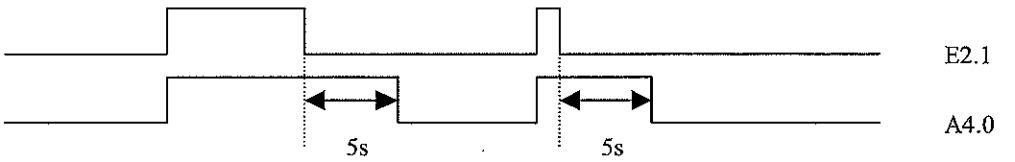
Temporisation SS :

```
U     E2.1      //si l'entrée E2.1 est à « 1 »
L     S5T#5S    //placer la durée prévue dans ACCU1
SS    T4        //lancer T4
U     E2.2      //si l'entrée E2.2 est à « 1 »
R     T4        //arrêter T4
U     T4        //pendant que T4 est valide
=     A4.0      //la sortie A4.0 doit être à « 1 »
```

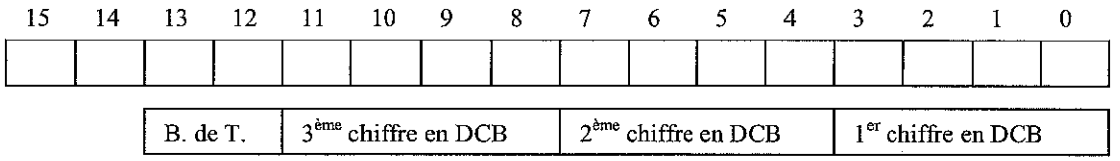


Temporisation SA :

```
U     E2.1      //si l'entrée E2.1 est à « 1 »
L     S5T#5S    //placer la durée prévue dans ACCU1
SA    T5        //lancer T5
U     T5        //pendant que T5 est valide,
=     A4.0      //la sortie A4.0 doit être à « 1 »
```



Complément d'information : Le nombre placé dans ACCU1-L, indiquant la durée d'une temporisation, est codé sur 12 bits, de la manière suivante :



Les bits 14 et 15 sont inutilisés.
Les bits 12 et 13 contiennent la base de temps. Il existe 4 bases de temps :

0	**0**	10 millisecondes	(0)
0	**1**	100 millisecondes	(1)
1	**0**	1 seconde	(2)
1	**1**	10 secondes	(3)

Les bits 0 à 11 contiennent 3 chiffres codés en DCB, constituant le nombre qui, multiplié par la base de temps choisie, donnera la durée de la temporisation.

Le codage ci-dessus est réalisé automatiquement par le format évolué cité plus haut, mais on peut aussi l'écrire directement dans ACCU1-L. Exemples :

L **W#16#2009** (9 secondes) ; **L** **W#16#0005** (50 millisecondes)

C-4) Compteurs

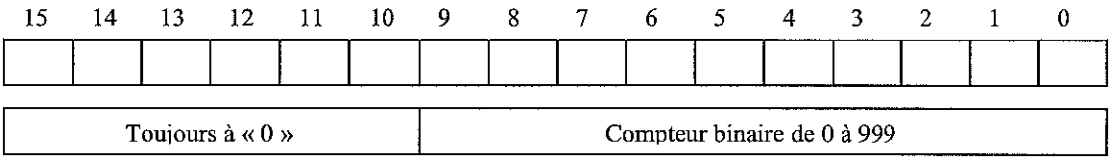
On peut ouvrir plusieurs compteurs simultanément, leur nombre varie avec la CPU utilisée (en général 256 compteurs). Ils comptent de 0 à 999.

Exemple :

```
U     E0.0    // si l'entrée E0.0 passe à "1"
L     C#3     // on place 3 dans ACCU1 (remarquer le format spécial)
S     Z1      // initialiser le compteur Z1 à 3
...
...
U     E0.1    // si E0.1 passe à "1"
```

```
ZV    Z1     // incrémenter Z1 (sur le front montant de E0.1).
...
...
U     E0.2   // si E0.2 passe à "1"
ZR    Z1     // décrémenter Z1 (sur le front montant de E0.2). ZR n'agit plus si Z1 est déjà à 0.
...
...
R     Z1     // remet Z1 à 0, comme si Z1 était un bit.
...
...
U     Z1     // met le RLG à « 1 » si Z1 différent de 0, comme si Z1 était un bit .
...
...
UN    Z1     // met le RLG à « 1 » si Z1 = 0, comme si Z1 était un bit.
...
...
L     Z1     // place le contenu de Z1 comme entier dans ACCU1- L.
...
...
```

Complément d'information : Les compteurs sont codés sur 10 bits de la manière suivante.



C-5) Ordres relatifs au déroulement d'un programme

Appel d'un bloc à partir d'un autre bloc : CALL

ATTENTION : CALL s'exécute toujours, à chaque cycle, sans tenir compte du RLG (comme L ou T ...).

Exemples :

```
CALL  FC2          // appel de FC2

CALL  FB1, DB3     // appel de FB1 avec son DB d'instance associé DB3.
```

On peut également appeler un bloc FC seulement, par :

```
UC    FC3          // appel de FC3 inconditionnel (équivalent à CALL  FC3)
```

Appel conditionnel d'un bloc FC :

```
U     E0.2         // si E0.2 = « 1 »
CC    FC6          // alors appeler FC6
```

Fin de bloc : BE

BE marque simplement la fin du bloc en cours et saute au bloc ayant appelé le bloc en cours.

Exemple : Si un bloc contient des sauts, il peut comporter plusieurs BE

```
        U       E0.0
        SPB     SUIV
        ...
        ...
        BE                      // fin provisoire
SUIV:   U       E0.1
        ...
        ...
        BE                      // fin définitive
```

BEB est une fin de bloc conditionnelle, exemple :

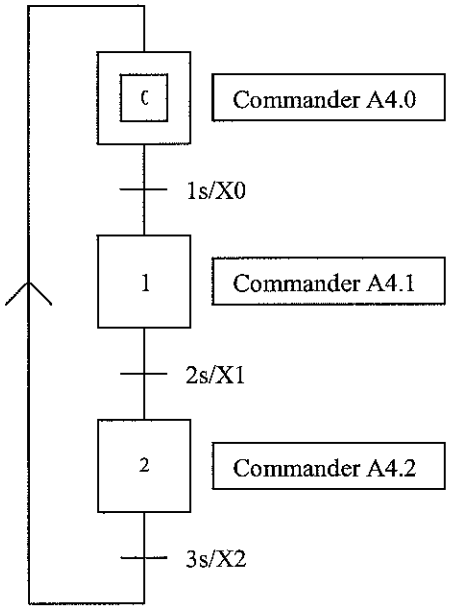
```
        U       E0.0
        BEB                     //  fin du bloc si E0.0 = « 1 »
        ...
        ...
```

BEA est une fin de bloc inconditionnelle (équivalente à BE), exemple :

```
        U       E0.2            // ce bloc s'arrêtera en BEA si E0.2 = « 0 »
        SPB     SUIV
        ...
        ...
        BEA
SUIV :  NOP0                    // NOP0 « non opération »
        ...
        ...
```

D) Quelques exemples simples

Exemple simple n°1 : (STEP7 ignore le grafcet, mais on peut toujours le « traduire »).



Une fois initialisé, ce grafcet va progresser seul, les 3 sorties A4.0, A4.1, A4.2, seront activées successivement (« chenillard »).

Pour commander l'initialisation de ce grafcet, on va utiliser une entrée auxiliaire E0.0.

On « matérialise » X0 par le bit M0.0, X1 par le bit M0.1 et X2 par le bit M0.2.

Premier essai de programme :

```
U     E0.0      //si l'entrée auxiliaire E0.0 est activée
S     M0.0      //alors, on initialise le grafcet.
R     M0.1
R     M0.2
NOP 0           //**************************
UN    E0.0      //si l'entrée auxiliaire E0.0 n'est pas activée
U     M0.0      //et si on est en X0
L     S5T#1S    //alors on lance une temporisation T1 de 1 seconde
SE    T1
U     T1        //au bout de 1 seconde
R     M0.0      //on passe de X0 à X1.
S     M0.1
NOP 0           //**************************
U     M0.1      //et si l'on est en X1
L     S5T#2S    //alors on lance une temporisation T2 de 2 seconde
SE    T2
U     T2        //au bout de 2 secondes
R     M0.1      //on passe de X1 à X2.
S     M0.2
NOP 0           //**************************
U     M0.2      //et si l'on est en X2
L     S5T#3S    //alors on lance une temporisation T3 de 3 secondes
SE    T3
U     T3        //au bout de 3 secondes
R     M0.2      //on passe de X2 à X0.
S     M0.0
NOP 0           //**************************
```

```
U     M0.0          //puis, on « recopie » M0.0 dans A4.0
=     A4.0
U     M0.1          // etc.
=     A4.1
U     M0.2
=     A4.2

BE                  //Fin du bloc
```

Ce bloc de code peut être écrit complètement dans OB1, le fonctionnement sera correct.

Il est toujours intéressant d'écrire une table d'assignation de mnémoniques (non obligatoire) :

Nom de la variable	**nouveau nom**
M0.0	X0
M0.1	X1
M0.2	X2
E0.0	entrée_auxi
A4.0	sortie0
A4.1	sortie1
A4.2	sortie2
T1	tempo1
T2	tempo2
T3	tempo3

Le bloc de code devient : (les noms donnés aux variables s'écrivent entre « »)

```
U     « entrée_auxi » //si l'entrée auxiliaire E0.0 est activé
S     « X0 »          //alors, on initialise le grafcet.
R     « X1 »
R     « X2 »
NOP 0                 //**************************
UN    « entrée_auxi » //si l'entrée auxiliaire E0.0 n'est pas activée
U     « X0 »          //et si on est en X0
L     S5T#1S          //alors on lance une temporisation T1 de 1 seconde
SE    « tempo1 »
U     « tempo1 »            //au bout de 1 seconde
R     « X0 »          //on passe de X0 à X1.
S     « X1 »
NOP 0                 //**************************
U     « X1 »          //et si on est en X1
L     S5T#2S          //alors on lance une temporisation T2 de 2 secondes
SE    « tempo2 »
U     « tempo2 »            //au bout de 2 secondes
R     « X1 »          //on passe de X1 à X2.
S     « X2 »
NOP 0                 //**************************
U     « X2 »          //et si on est en X2
L     S5T#3S          //alors on lance une temporisation T3 de 3 secondes
SE    « tempo3 »
U     « tempo3 »            //au bout de 3 secondes
R     « X2 »          //on passe de X2 à X0.
S     « X0 »
NOP 0                 //**************************
U     « X0 »          //commande des sorties
=     « sortie0 »
U     « X1 »
=     « sortie1 »
U     « X2 »
```

```
=       « sortie2 »

BE                      //Fin du bloc
```

On va modifier cette programmation en écrivant, en plus du bloc OB1, un bloc fonction FC1 qui va contenir l'initialisation du grafcet.

Bloc FC1 :

```
S       "X0"
R       "X1"
R       "X2"
BE
```

Cette fois, le bloc OB1 va s'écrire :

```
        UN      « entrée_auxi »  //si l'entrée auxiliaire E0.0 n'est pas activée
        SPB     AA               //alors sauter à l'étiquette AA
        CALL    FC1              //appel du bloc FC1 seulement quand E0.0 est activée

AA:     U       « X0 »           //si l'on est en X0
        L       S5T#1S           //alors on lance une temporisation T1 de 1 seconde
        .....
        ..... (suite identique)
```

On pourra également regrouper les 6 lignes de commandes des sorties, dans un bloc FC2 :

```
U       « X0 »
=       « sortie0 »
U       « X1 »
=       « sortie1 »
U       « X2 »
=       « sortie2 »
BE
```

On appellera ce bloc inconditionnellement, par : **CALL FC2**

On remarque que la structure :

```
U       « X0 »          //si l'on est en X0
L       S5T#1S          //alors on lance une temporisation T1 de 1 seconde
SE      « tempo1 »
U       « tempo1 »      //au bout de 1 seconde
R       « X0 »          //on passe de X0 à X1.
S       « X1 »
```

est répétée 3 fois dans le programme principal. On va regrouper ces lignes de code dans un bloc fonctionnel (FB), sorte de sous-programme, auquel on va passer des paramètres. Au moment de la création de ce FB1, le bloc DB1 d'instance associé va se créer automatiquement et on va y inscrire les paramètres selon leur catégorie :

Paramètres d'entrée : Le nom de la temporisation, qu'on appellera **tem** (type de paramètre : TIMER) et la durée que l'on appellera **duree** (type de paramètre : S5TIME).

Paramètres d'entrée et sortie : L'étape précédente et l'étape suivante qu'on appellera **etape_actuelle** et **etape_suivante** (type de paramètre : BOOL).

(les noms de variables s'écrivent sans accent)

Le bloc FB1 s'écrira alors :

```
U     #etape_actuelle
L     #duree
SE    #tem
U     #tem
R     #etape_actuelle
S     #etape_suivante
BE
```

(Le nom des variables internes au bloc FB1 s'écrit cette fois précédé de #)

Le programme principal, OB1, devient alors :

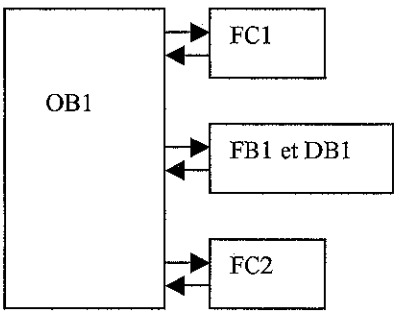
```
UN    « entrée_auxi »  //si l'entrée auxiliaire E0.0 n'est pas activée
SPB   AA               //alors sauter à l'étiquette AA
CALL  FC1              //appel du bloc FC1 seulement quand E0.0 est activée

AA :  CALL FB1, DB1
      tem              :=T1
      duree            :=S5T#1S
      etape_actuelle   := « X0 »
      etape_suivante   := « X1 »

      CALL FB1, DB1
      tem              :=T2
      duree            :=S5T#2S
      etape_actuelle   := « X1 »
      etape_suivante   := « X2 »

      CALL FB1, DB1
      tem              :=T3
      duree            :=S5T#3s
      etape_actuelle   := « X2 »
      etape_suivante   := « X0 »

      CALL FC2         // commande des sorties
      BE
```



Variante du programme avec l'utilisation de OB100 « démarrage à chaud » : Initialisation automatique du grafcet.

Le bloc OB100 s'exécute *automatiquement* (ce bloc ne peut pas être appelé), une seule fois, au moment du passage de STOP à RUN de la CPU. La durée d'exécution de ce bloc n'est pas surveillée par le « chien de garde » de la CPU. On peut l'utiliser ici pour initialiser le grafcet automatiquement :

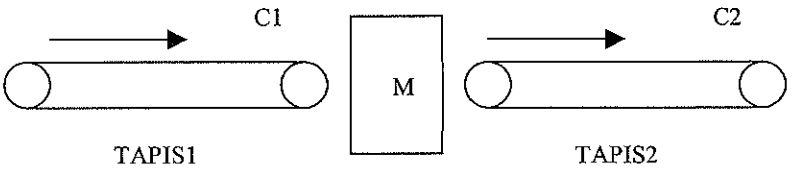
Programme à écrire dans OB100 :

```
SET          // pour mettre le RLG à « 1 » (sinon les ordres suivants ne s'exécutent pas)
S     "X0"
R     "X1"
R     "X2"
BE
```

Bien sûr, FC1 n'est plus utile, l'entrée auxiliaire non plus. Il faut modifier OB1 en conséquence. L'initialisation du grafcet se fera au « STOP-RUN » de l'automate.

Remarque : Le bloc OB100 est un bloc d'organisation. Il ne peut pas être appelé mais il peut appeler d'autres blocs FC... ou FB... lors de son exécution.

Exemple simple n°2 : Utilisation d'un compteur



Un tapis roulant TAPIS1 apporte des objets à placer dans le magasin M. Un autre tapis roulant TAPIS2 retire ces objets du magasin M et les emporte vers leur destination.

Le magasin M contient au maximum 50 objets. Un capteur C1 détecte les objets qui arrivent dans le magasin M et un autre capteur C2 détecte les objets qui quittent la machine.

Cette machine a deux modes de marche.

Marche de production normale : Le TAPIS2 fonctionne tant que le magasin M contient des pièces. Il s'arrête quand le magasin M est vide, dès que la dernière pièce a passé le capteur C2. Le TAPIS1 fonctionne en permanence apportant des pièces à un rythme imprévisible.

Un voyant lumineux s'allume quand le contenu du magasin M est supérieur ou égal à 40 pièces.

Le TAPIS1 est arrêté momentanément quand le magasin M contient 50 pièces.

Marche d'initialisation lancée à la suite *d'une pression* sur le bouton INIT : Le nombre de pièces en M est présumé nul mais en fait, inconnu (cas d'un arrêt anormal de la machine). Le TAPIS2 fonctionnera seul pendant 20 secondes, ce qui est suffisant pour faire sortir les pièces éventuellement restantes en M. Le compteur sera initialisé pendant cette manœuvre.

Affectations de mnémoniques :
E0.0 : initialisation ; **E0.1** : C1 ; **E0.2** : C2 ;
A4.0 : VOYANT 40 PIECES ; **A4.1** : TAPIS1 ; **A4.2** : TAPIS2
T1 : tempo (20 secondes).
Z1 : compteur.

```
U     « initialisation » // une pression sur le bouton d'initialisation
L     S5T#20S           // lance la tempo de 20 secondes.
SA    « tempo »
NOP 0
U     « initialisation » // une pression sur le bouton d'initialisation
L     C#0               // initialise le compteur à 0.
S     "compteur"
NOP 0
UN    « tempo »         // si l'on n'est pas en cours d'initialisation
U     C1                // C1 incrémente le compteur
ZV    « compteur »
UN    « tempo »         // si l'on n'est pas en cours d'initialisation
U     C2                // C2 « décrémente » le compteur
ZR    « compteur »
NOP 0
L     40
L     « compteur »      // 40 dans ACCU2, « compteur » dans ACCU1
<=I                     // si ACCU2 <= ACCU1, alors
=     « voyant »        // allumer le voyant
NOP 0
```

```
L     50
L     « compteur »     // 50 dans ACCU2, « compteur » dans ACCU1
>I                     // si ACCU2 > ACCU1,
UN    « tempo »        // et si l'on n'est pas en cours d'initialisation, alors
=     « tapis 1 »      // activer le tapis1
NOP 0
O     « compteur »     // si « compteur > 0
O     « tempo »        // ou si on est en cours d'initialisation, alors
=     « tapis2 »       // alors activer le tapis2
BE
```

On remarquera que les commandes sont **uniques** (tapis1, tapis2, voyant).

Electriquement, il faudra que les connexions à l'automate, des capteurs C1 et C2, soient correctement établies, pour éviter les décrémentations et incrémentations parasites du compteur (paires torsadées blindées).

Exemples simples n°3 : Manipulations de données, blocs de données globaux.

Un *bloc de données global* (qui peut être appelé par n'importe quel bloc de code du programme) est un tableau formé d'un certain nombre d' « étagères » (maximum 256) sur lesquelles on peut ranger des données sous forme de booléens, d'entiers simples ou doubles, de réels simples ou doubles, de caractères, etc.. On peut choisir la structure d'un bloc de données global. Il ne faut pas le confondre avec un *bloc de données d'instance* qui, lui, est affecté à un bloc FBi, et, est crée automatiquement au moment de la programmation de ce FBi.

Exemple de bloc de données :

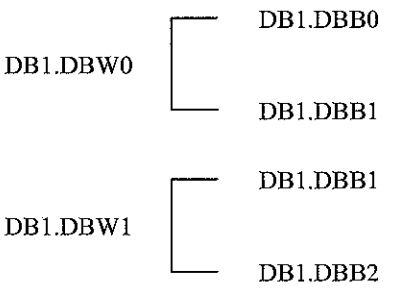
Adresse	**nom**	**type**	**valeur initiale**	**commentaire**
		STRUCT		
0.0	mot1	WORD	W#16#0	
2.0	poids	WORD	W#16#0	
4.0	numero1	INT	0	
6.0	zone	DWORD	DW#16#5	
10.0	passage	DINT	L#30	
14.0	mesure1	REAL	1.22e+01	
16.0	bit2	BOOL	FALSE	
18.0	byte1	BYTE	B#16#5	
19.0	durée	S5TIME	S5T#0H_0M_5S_12MS	
		END_STRUCT		

STEP7 gère automatiquement les adresses. Le nom est obligatoire, le commentaire est facultatif.

Dans le programme, on appelle un bloc de données DB1 par : **AUF DB1**

La première donnée de ce bloc s'appelle : **DB1.DBW0** si c'est un mot ou **DB1.DBB0** si c'est un byte.

ATTENTION :



DB1.DBW0 (16 bits) se subdivise en DB1.DBB0 (8 bits) et DB1.DBB1 (8 bits). *L'octet de poids faible* est DB1.DBB1 et *l'octet de poids fort* est DB1.DBB0.

Exemple :

```
AUF     DB1       // ouvrir DB1
L       W#16#20   // placer 20 en hexa dans ACCU1-L
T       DB1.DBW0  // transférer le contenu de ACCU1-L dans DB1.DBW0
```

Le résultat est que DB1.DBB1 contient 20 (ici, en hexa) et que DB1.DBB0 contient 0.

Exemple : On crée un bloc de données DB5 :

Adresse	nom	type	valeur initiale	commentaire
		STRUCT		
0.0	mot0	WORD	W#16#0012	
2.0	mot1	WORD	W#16#0034	
4.0	mot2	WORD	W#16#0056	
6.0	mot3	WORD	W#16#0078	
		END_STRUCT		

On écrit le transfert de valeurs de données vers des mots courants, en utilisant le bloc ci-dessus :
(il faut veiller à ne pas créer de « chevauchements »).

```
...
...
      AUF   DB5          // ouverture de DB5
      L     DB5.DBW0     // DB5.DBB0 contient 00     DB5.DBB1 contient 12
T     MW0                // MB0 contient 00          MB1 contient 12
      L     DB5.DBW2     // DB5.DBB0 contient 00     DB5.DBB1 contient 34
T     MW2                // MB2 contient 00          MB3 contient 34
      L     DB5.DBW4     // DB5.DBB0 contient 00     DB5.DBB1 contient 56
T     MW4                // MB4 contient 00          MB5 contient 56
      L     DB5.DBW6     // DB5.DBB0 contient 00     DB5.DBB1 contient 78
T     MW6                // MB6 contient 00          MB7 contient 78
...
...
```

Exemple : ajouter le premier nombre au deuxième, et multiplier le résultat obtenu par le troisième, mettre le résultat final dans le mot MW8 :

```
...
...
AUF   DB5
L     DB5.DBW0
T     MW0
L     DB5.DBW2
T     MW2
NOP   0
L     MW0          // MW0 dans ACCU1-L
L     MW2          // MW2 dans ACCU1-L ; MWO dans ACCU2-L
+I                 // somme des deux ACCU
T     MW6          // on passe le résultat dans MW6
NOP   0
L     DB5.DBW4
T     MW4
NOP   0
L     MW4          // MW4 dans ACCU1-L
L     MW6          // MW6 dans ACCU1-L ; MW4 dans ACCU2-L
*I                 // produit des deux ACCU
T     MW8          // on met le résultat final dans MW8
...
...
```

Exemple : Avec le mot de sortie AW4

Contenu du bloc de données : (DB5)

```
0.0   mot0   W#16#0001
2.0   mot1   W#16#0002
```

Programme : On utilise AW4 subdivisé en AB4 (poids fort) et AB5 (poids faible).

Expliquer la différence de résultat entre les trois petits programmes suivants :

```
AUF   DB5
L     DB5.DBW0      // DB5.DBB0 = 00  ;  DB5.DBB1 = 01
T     AW4           // A5.0 = VRAI
BE


AUF   DB5
L     DB5.DBW1      // DB5.DBB1 = 01  ;  DB5.DBB2 = 00
T     AW4           // A4.0 = VRAI
BE


AUF   DB5
L     DB5.DBW2      // DB5.DBB2 = 00  ;  DB5.DBB3 = 02
T     AW4           // A5.1 = VRAI
BE
```

Etc.

DB5.DBB3	DB5.DBB2	DB5.DBB1	DB5.DBB0
02	00	01	00

```
                                  ****DB5.DBW0*********
                    ****DB5.DBW1*********
****DB5.DBW2*********
```

Exemple simple n°4 : Interruptions.

On peut programmer avec STEP7, un ou plusieurs blocs qui seront exécutés à la suite d'une interruption. Cette possibilité dépend de la CPU utilisée.

Dans le cas de notre CPU (CPU314), si l'on écrit un programme dans le bloc OB35, celui-ci s'exécutera tous les dixièmes de secondes (cette périodicité est modifiable au moment de la configuration de la CPU). Des CPU plus importantes permettent de lancer jusqu'à 8 blocs (OB30 à OB38) grâce à 8 interruptions possibles de niveau de priorité variable et de périodicité réglable.

Exemple de programme utilisant OB35 :

On déclare le bloc de données global **DB5** (une seule « étagère ») :

Adresse	nom	type	valeur initiale	commentaire
		STRUCT		
0.0	mot	WORD	W#16#0	
		END_STRUCT		

Puis le bloc **OB100** (qui ne s'exécutera qu'une seule fois au moment du passage STOP à RUN) :

```
AUF   DB5          // ouverture de DB5
L     0            // 0 dans ACCU1
T     DB5.DBW0     // ACCU1 dans DB5.DBW0
T     AW4          // ACCU1 dans AW4 (mettre à 0 les sorties)
BE
```

Puis le bloc **OB35** qui s'exécutera tous les dixièmes de secondes :

```
AUF   DB5
L     DB5.DBW0     // DB5.DBW0 dans ACCU1
INC   1            // incrémentation de ACCU1
T     DB5.DBW0     // on replace dans DB5.DBW0
BE
```

DB5.DBW0 sera incrémenté tous les dixièmes de seconde.

Puis le programme principal complètement contenu dans **OB1** :

```
      AUF   DB5
      L     25           // 25 dans ACCU1
      L     DB5.DBW0     // DB5.DBW0 dans ACCU1 ; 25 dans ACCU2
      <I                 // si ACCU2 < ACCU1
      =     A4.0         // alors activer A4.0
      NOP 0
      L     50           // 50 dans ACCU1 ; DB5.DBW0 dans ACCU2
      <I                 // si ACCU2 < ACCU1
      SPB   ETI          // sauter à ETI
      L     0            // sinon, on met 0 dans ACCU1
      L     DB5.DBW0     // et on le place dans DB5.DBW0
ETI : BEA
      BE
```

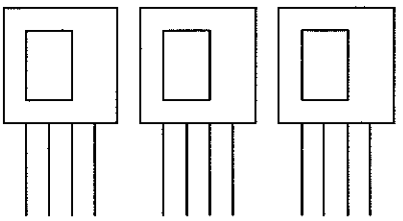
Pendant 2,5 secondes, A4.0 sera activé. Pendant les 2,5 secondes qui suivent, il sera désactivé (clignotement de A4.0).

Au bout de 5 secondes, DB5.DBW0 sera remis à 0, et le cycle recommence.

Exemple simple n°5 : Opérations combinatoires sur des mots.

En appuyant sur un bouton poussoir, on allume une lampe pendant un temps déterminé, réglé en secondes par trois roues codeuses :

lampe



E0.7	E0.6	E0.5	E0.4	E0.3	E0.2	E0.1	E0.0	E1.7	E1.6	E1.5	E1.4	E1.3	E1.2	E1.1	E1.0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Ci-dessus, le mot image de la carte d'entrée EW0 (subdivisé en EB0 – poids fort, et EB1 – poids faible).

Câblage des trois roues codeuses. Celle de droite (unités) est reliée à E1.3, E1.2, E1.1, E1.0, celle du milieu (dizaines) est reliée à E1.7 , E1.6, E1.5, E1.4, celle de gauche (centaines) est reliée à E0.3, E0.2, E0.1, E0.0.

Le bouton poussoir est relié à E0.7.

La lampe est commandée par A4.0.

```
U       T1          // si la temporisation s'exécute
=       A4.0        // allumer la lampe
BEB                 // si la temporisation s'exécute, arrêter le traitement ici. Ainsi, la
NOP 0               // temporisation T1 n'est pas redémarrée si le bouton poussoir est enfoncé.
L       EW0         // EW0 dans ACCU1-L
UW      W#16#0FFF   //  « masquer » les bits E0.7, E0.6, E0.5, E0.4 (ils seront mis à « 0 »).
NOP 0               // la valeur du temps est codée sur les 12 bits qui restent.
OW      W#16#2000   // mettre la bonne base de temps sur les bits 12 et 13 du mot de poids faible
NOP 0               // de ACCU1
NOP 0
U       E0.7        // poussoir
SV      T1          // lancer T1
...
...
```

Exemple simple n°6 : Utilisation de l'horodateur.

L'horodateur utilise un format spécial appelé « DATE_AND_TIME », formé de 8 octets codés en DCB.

Octet 0 : année (2 chiffres) ;
Octet 1 : mois ;
Octet 2 : jour ;
Octet 3 : heure ;
Octet 4 : minute ;
Octet 5 : seconde ;
Octet 6 : les deux chiffres de poids fort de millisecondes ;
Octet 8 : le chiffre de poids faible des millisecondes et le jour de la semaine (dimanche = 1, lundi = 2, etc.).

Il existe une façon évoluée de coder ce nombre, exemple : **DT#04-5-28-10 :12 :25:3567**

(Le mardi 28 mai 2004 à 12 heures, 25 minutes, 35 secondes, 567 millièmes de seconde).

Les CPU contiennent plusieurs dizaines de fonctions FC préprogrammées, appelées SFC. Les deux premières appelées SFC0 (« SET_CLK ») et SFC1 (« READ_CLK ») servent justement à utiliser l'horodateur. L'accès au programme source des SFCi est impossible.

Exemple d'utilisation :

On va écrire un premier bloc de données **DB1** : (mise à l'heure de l'horodateur).

Adresse	nom	type	valeur initiale	commentaire
0.0		STRUCT		
0.0	**depart**	**DATE_AND_TIME**	**DT#04-5-28-10:00:00:00**	
8.0		END_STRUCT		

(les parties à programmer sont en caractère gras)

Puis un second bloc de données **DB2** : (lecture de l'heure courante).

Adresse	nom	type	valeur initiale	commentaire
0.0		STRUCT		
0.0	**moment**	**DATE_AND_TIME**	**DT#90-1-1-0:0:0:**	
8.0		END_STRUCT		

Le bloc DB1 contient l'heure à laquelle on veut régler préalablement l'horodateur (ce dernier peut aussi se régler avec l'heure du P.C.), et le bloc DB2 contiendra l'heure courante lue effectivement (la valeur initiale indiquée, est ici sans intérêt) à chaque cycle automate.

Bloc OB1 (ou autre bloc appelant) :

```
      ...
      ...

      UN    E0.0                  // E0.0 : bouton de mise à l'heure
      SPB   ETIQ
      AUF   DB1
      CALL  « SET_CLK »           // ou SFC0
            PDT         :=  DB1.depart   // PDT valeur retournée par SET_CLK
            RET_VAL     :=  MW0          // indicateur d'exécution

ETIQ: AUF   DB2
      CALL «READ_CLK»               // ou SFC1
            CDT         :=  DB2.moment   // CDT valeur retournée par READ_CLK
            RET_VAL     :=  MW2          // indicateur d'exécution
```

```
L     DB2.DBB0
T     MB10        // MB10 contient l'année

L     DB2.DBB1
T     MB11        // MB11 le mois

L     DB2.DBB2
T     MB12        // MB12 le jour

L     DB2.DBB3
T     MB13        // MB13 l'heure

L     DB2.DBB4
T     MB14        // MB14 les minutes

L     DB2.DBB5
T     MB15        // MB15 les secondes

L     DB2.DBB6
T     MB16        // MB16 les 2 chiffres de poids fort des dixièmes de seconde

L     DB2.DBB7
T     MB17        // MB17 le chiffre de poids faible des dixièmes et jour de la semaine
...
...               // suite du programme
```

RET_VAL est une valeur entière, de contrôle, retournée par les blocs préprogrammés. Si cette valeur est positive ou nulle, l'exécution du bloc a été normale. Une valeur négative indique une erreur (les différentes valeurs négatives indiquent les différentes erreurs possibles – voir documentation relative à chaque bloc préprogrammé).

La lecture des valeurs contenues dans **MW0** et **MW2** permettra de diagnostiquer et d'identifier une éventuelle erreur, ou de lancer l'exécution d'un bloc d'erreur.