E I P E700

MANUEL UNIPROG

22.05.2012 – Français

E.I.P. S.A. **Rte de l'Intyamon 23** **CH–1667 ENNEY**	**Tél : 026 / 921 80 40** **Fax : 026 / 921 80 49**	*www.eipsa.ch* *info@eipsa.ch*

Table des matières

1 Présentation du manuel Uniprog

Ce manuel a pour but d'aider dans la mise en œuvre du E700 d'un point de vue de la programmation en langage Uniprog, pour permettre de développer et de gérer les différentes tâches à réaliser sur le E700 et les équipements qui lui sont associés.

Il décrit chacune des fonctions ou instructions Uniprog. Des exemples concrets sont associés à chacune des instructions Uniprog, en général directement applicables sur le E700.

Nous sommes disponibles pour toute précision ou suggestion par e-mail à l'adresse ***info@eipsa.ch***.

2 Introduction sur la syntaxe UNIPROG E700

- En UNIPROG E700, les caractères spéciaux (é, è, à etc, ainsi que **TAB** le caractère de tabulation) ne sont pas reconnus par l'éditeur. Il faut donc éviter de les utiliser. Ces caractères spéciaux sont, à part le TAB, tous ceux qui ont un code ASCII supérieur à 127 (7Fh). Consulter une table ASCII pour plus de détails.

- Il y a une distinction entre les majuscules et les minuscules. Il faut éviter d'utiliser des minuscules pour les noms de variables car on ne pourra pas examiner leur contenu en temps réel (touche TRACE puis WATCH (F1)). En effet, l'éditeur du E700 n'est pas prévu pour entrer des minuscules.

- Les noms de variables, constantes ou labels contiennent au maximum 10 caractères (lettres, chiffres ou caractère souligné (_). Ils doivent impérativement commencer par une lettre de A à Z.

- Les commentaires sont précédés d'un point-virgule (;). Donc tout ce qui suit un point-virgule est ignoré par le compilateur. Et ce, jusqu'au bout de la ligne courante.

- En UNIPROG E700, il n'y a pas de type de données. On peut donc mélanger des réels, des entiers et des booléens. Le E700 travaille tout en réels. Il n'y a donc pas besoin de d'ajouter le point décimal. Par exemple, on peut écrire simplement 4 à la place de 4.0.

- L'UNIPROG E700 permet d'exécuter jusqu'à 10 tâches simultanées. Ces tâches sont numérotées de 0 à 9. La tâche 9 est appelée tâche AUTOMATIQUE ou AUTOMAT. C'est une boucle qui est exécutée en permanence, indépendante du bouton START. (Le bouton START permet d'exécuter le cycle écrit en ISO ou en UNIPROG.)
  Lorsque le bouton START est pressé, l'exécution commence automatiquement dans la tâche 0. La tâche 0 peut ensuite activer d'autres tâches (de 1 à 8). Lorsqu'une tâche se termine, elle redevient libre et peut être affectée à d'autres travaux. Lorsqu'une tâche est une boucle infinie, l'instruction KSIM (Kill SIMultaneous task) permet de «tuer» une tâche. Une tâche peut se tuer elle-même.
  Le programme (cycle) se termine lorsque toutes les tâches (de 0 à 8) sont terminées. La tâche 9 (AUTOMAT) continue, bien évidemment, à s'exécuter.

- Il existe 8 registres R0, R1, R2, …, R7. Les registres sont en quelque sorte des variables pré-déclarées disponibles pour le programmeur. Il existe un jeu de registres par tâche. Donc la tâche n ne peut pas modifier le contenu des registres de la tâche m.
  Lorsque des registres sont utilisés et modifiés dans une procédure (routine, sous-programme), il ne faut pas oublier de les sauvegarder au début de la procédure et de les restaurer en fin de procédure. Une pile est à disposition pour ces sauvegardes temporaires. Il existe un 9^{ième} registre R8 en lecture seulement et utilisable en index direct [R8] qui vaut PNB, le numéro de la tâche simultanée courante.

- Chaque tâche possède sa propre pile.

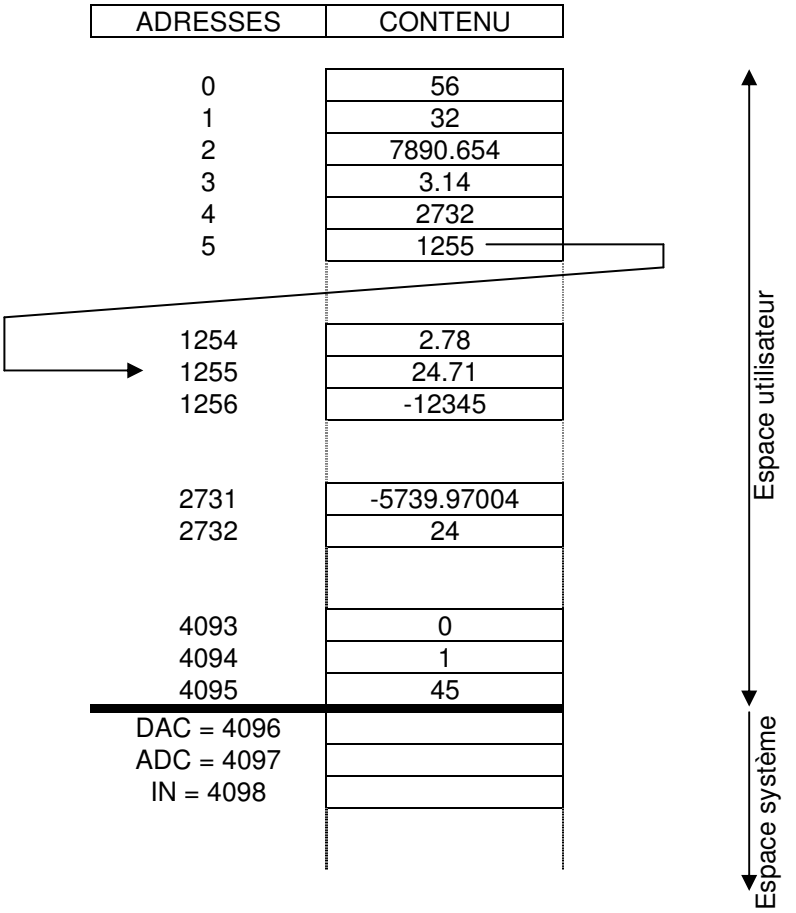
- Il existe trois modes d'adressage en UNIPROG : l'adressage immédiat, l'adressage direct et l'adressage indirect.

```
Adressage immédiat :   MOV   R0 5    ; On met la valeur (immédiate) 5 dans R0. Donc R0
                                     ; vaut maintenant 5

Adressage direct :     MOV   R0 #5   ; On met le contenu de l'adresse 5 dans R0. Donc R0
                                     ; vaut maintenant 1255 (voir exemple ci-dessous)

Adressage indirect :   MOV   R0 @5   ; On met le contenu du contenu de l'adresse 5 dans
                                     ; R0. Donc R0 vaut maintenant 24.71
```

  En effet, dans l'exemple ci-dessous, on a vu que le contenu de 5 valait 1255. Le contenu du contenu, c'est en fait le contenu de 1255. Et le contenu de 1255 vaut bien 24.71 !

  L'adressage indirect est très utile lorsqu'un programmeur désire travailler avec la notion de pointeurs.

Note : Pour les registres (R0 à R7), il n'existe que les adressages directs (Ri) et indirects (@Ri).



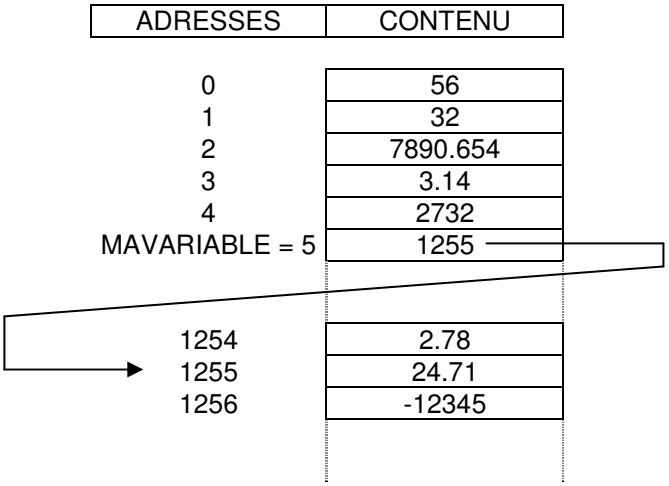
La mémoire disponible pour l'utilisateur a une taille de 4096. Les adresses vont donc de 0 à 4095. Le contenu à chaque adresse est un nombre codé sur 4 octets. On rappelle ici qu'il n'y a pas de type de données en UNIPROG; tout est en réel.

Au delà de l'adresse 4095, c'est un espace réservé qui commence : la mémoire système. C'est une portion de mémoire qui est partagée entre l'utilisateur et le système d'exploitation. Chacune des adresses au-delà de 4095 porte un nom. Ce nom peut être retrouvé dans le fichier appelé SYSTEM.E7M.

L'utilisateur peut donner un nom de son choix à chacune des adresses comprises entre 0 et 4095. C'est ce qu'on appelle une déclaration de variable :

Par exemple : **`MAVARIABLE = 5`**

En reprenant l'exemple précédent on aura :



Déclarer une variable, c'est donner un nom symbolique à une adresse. Dans cet exemple, on a donné MAVARIABLE comme nom symbolique à l'adresse 5. On pourra désormais invariablement travailler avec 5 ou avec MAVARIABLE. C'est la même chose !

On pourra désormais écrire :

```
MOV     R0 MAVARIABLE     ; R0 vaut maintenant 5

MOV     R0 #MAVARIABLE    ; R0 vaut maintenant 1255

MOV     R0 @MAVARIABLE    ; R0 vaut maintenant 24.71
```

Afin d'éviter de donner plusieurs noms symboliques à une même adresse, on peut laisser au compilateur le soin de choisir lui-même une adresse pour une variable. Pour cela, il suffit de déclarer :

MAVARIABLE =

Si on ne donne pas de valeur à droite du signe égal (=), le compilateur recherchera lui-même le premier emplacement de libre et il lui affectera cette valeur. Lorsqu'on utilise plusieurs variables dans un programme, une bonne habitude à prendre est donc de laisser au compilateur le soin de déterminer lui-même l'adressed'une variable en ne décrivant rien à droite du signe égal (=).

Maintenant qu'on sait déclarer une variable, profitons-en pour introduire la notion de vecteur ou de tableau.

Un tableau se déclare de la manière suivante :

```
MONTABLEAU[6] =
```

On vient de déclarer un tableau de 6 éléments. Ces éléments sont numérotés de 0 à 5.

ADRESSES	CONTENU
MONTABLEAU[0]	568.24
MONTABLEAU[1]	-359
MONTABLEAU[2]	0
MONTABLEAU[3]	0
MONTABLEAU[4]	23
MONTABLEAU[5]	-321.009

```
MOV     R0 #MONTABLEAU[4]          ; Maintenant, R0 vaut 23
```

Les indices d'un tableau sont soit fixes (ce sont alors des constantes) ou soit variables (ce ne peut être que des registres).

Déclarons une constante :

```
TABINDEX = 1
```

puis exécutons :

```
MOV     R0 #MONTABLEAU[TABINDEX]    ; Maintenant, R0 vaut –359
```

Parcours (lecture) d'un tableau (à l'envers dans l'exemple suivant : de 5 à 0) :

```
        MOV     R0 6

BOUCLE: DEC     R0
        MOV     R1 #MONTABLEAU[R0]
        BRIN1   R0 BOUCLE
        . . .
```

Quelques explications sur ce premier petit bout de programme :

R0 est l'indice variant entre 5 et 0. Il est initialisé à 6, mais comme il est décrémenté directement après, il ne vaut plus que 5 (DEC R0 décrémente R0. Donc R0 ← R0 – 1).

Ensuite, on va mettre dans R1 le contenu de MONTABLEAU[R0] = MONTABLEAU[5] = -321.009.

La dernière instruction signifie «Branche-toi à BOUCLE si R0 n'est pas nul», ce qui sous-entend «continue si R0 est nul».

BOUCLE est ce qu'on appelle un **label**. Plutôt que de numéroter les lignes et de lui dire de faire un saut (ou branchement) à la ligne numéro n, on préfère ne «numéroter» que les lignes qui doivent l'être. Et plutôt que de donner des numéros, on donne un nom symbolique que l'on appelle **label**.

Pour en revenir à notre programme, on l'a vu, R0 vaut 5. Il n'est donc pas nul et va donc faire un saut (branchement) au label BOUCLE.

Nouvelle décrémentation de R0. Donc R0 vaut maintenant 5 – 1 = 4.
Ensuite, on va mettre dans R1 le contenu de MONTABLEAU[R0] = MONTABLEAU[4] = 23.
R0 vaut 4. Il n'est donc pas nul. Le programme saute donc au label BOUCLE.

Nouvelle décrémentation de R0. Donc R0 vaut maintenant 4 – 1 = 3.
Ensuite, on va mettre dans R1 le contenu de MONTABLEAU[R0] = MONTABLEAU[3] = 0.
R0 vaut 3. Il n'est donc pas nul. Le programme saute donc au label BOUCLE.

Nouvelle décrémentation de R0. Donc R0 vaut maintenant 3 – 1 = 2.
Ensuite, on va mettre dans R1 le contenu de MONTABLEAU[R0] = MONTABLEAU[2] = 0.
R0 vaut 2. Il n'est donc pas nul. Le programme saute donc au label BOUCLE.

Nouvelle décrémentation de R0. Donc R0 vaut maintenant 2 – 1 = 1.
Ensuite, on va mettre dans R1 le contenu de MONTABLEAU[R0] = MONTABLEAU[1] = -359.
R0 vaut 1. Il n'est donc pas nul. Le programme saute donc au label BOUCLE.

Nouvelle décrémentation de R0. Donc R0 vaut maintenant 1 – 1 = 0.
Ensuite, on va mettre dans R1 le contenu de MONTABLEAU[R0] = MONTABLEAU[0] = 568.24.
R0 vaut 0. Il est donc nul. Le programme ne saute pas au label BOUCLE, mais il va continuer et exécuter la suite symbolisée par les 3 points de suspension.

3 Fonctionnement formel de l'exécuteur UNIPROG

L'exécuteur UNIPROG effectue l'instruction UNIPROG pointée par SIMPTR. SIMPTR est un marqueur qui pointe sur l'instruction à exécuter à cet instant. Quand l'instruction est complètement terminée, l'exécuteur UNIPROG incrémente la valeur de SIMPTR pour que celui-ci pointe sur la prochaine instruction à effectuer.

Dans le jargon des initiés, SIMPTR est le PC (Program Counter).

Le nombre maximum de tâches simultanées est défini par une constante : *SimNb*. Cette constante est interne et donc inaccessible au programmeur. Elle est fixée à 10. Le tâches sont donc numérotées de 0 à *SimNb* – 1.

La tâche numéro *SimNb* – 1 est la tâche de fond; la tâche AUTOMAT.
La tâche 0 est la tâche qui sera exécutée lors d'une pression sur le bouton START.

Comme il peut y avoir jusqu'à *SimNb* tâches simultanées, il faut avoir autant de SIMPTR. Il y en a un par tâche simultanée. Un simultané est libre, ou inactif si son SIMPTR vaut 0.

Les SIMPTR sont des variables SYTEM, accessibles au programmeur en lecture seulement. Elles doivent être indexées par le numéro du simultané correspondant.

Exemple :
```
        MOV     R0 0
        MOV     R1 0

LOOP:   BRIN0   #SIMPTR[R0] LOOP1
        INC     R1

LOOP1:  INC     R0
        CMP     R0 10
        JL      LOOP
```

Le petit exemple de la page précédente calcule le nombre de tâches simultanées en cours d'exécution à l'instant ou le code est exécuté.

Le résultat est mémorisé dans le registre R1.

Le registre R0 est utilisé comme indice de boucle et vaudra 0, puis 1, puis 2 et ainsi de suite, jusqu'à *SimNb*.

L'instruction BRIN0 teste si SIMPTR[R0] vaut 0 ou non. Si SIMPTR[R0] vaut 0, alors on saute au label LOOP1. Sinon, si SIMPTR[R0] ne vaut pas 0, alors, on incrémente R1. Donc R1 ne sera incrémenté que si SIMPTR[R0] est non-nul, ce qui signifie que la tâche simultanée en question est en cours d'exécution.

L'instruction CMP compare R0 avec 10 (*SimNb*).

L'instruction JL effectue un saut au label LOOP si le résultat de la comparaison précédente est du type «inférieur à» (Less en anglais). Donc on ne saute au label LOOP que si R0 est plus petit que 10 (*SimNb*). Sinon, l'exemple se termine.

L'instruction INC incrémente R0 ou R1. Incrémenter signifie «ajouter 1».

Très schématiquement, voici comment fonctionne l'exécuteur UNIPROG :

Initialisations (tous les SIMPTR sont mis à 0)

SIMPTR [*SimNb* - 1] ← Première instruction de la tâche de fond (tâche AUTOMAT);

```
Boucler indéfiniment
{
        Mise à jour des variables de l'écran;
        s ← 0;
        Maintenir la LED AUTO allumée;

        Si une tâche ISO est en exécution, alors compiler la prochaine ligne ISO;

        Tant que s < SimNb
        {
                Mise à jour de l'écran TRACE;

                Si SIMPTR [s] est différent de 0, alors
                {
                        Exécuter l'instruction pointée par SIMPTR [s];
                }

                Incrémenter s;
        }
```

4 Les instructions Uniprog du E700

Détail de chacune des instructions Uniprog, en commençant par la présentation d'une description type :

EXEMPLE par1 [par2] (*Présentation syntaxique)*

« Définition et description de l'instruction **EXEMPLE** »

- par1 « Description du paramètre par1 de l'instruction **EXEMPLE**. »
- [par2] « Description du paramètre optionnel par2 de l'instruction **EXEMPLE**, qui peut être présent zéro ou une fois »
- {par2} « Description du paramètre optionnel par2 de l'instruction **EXEMPLE**, qui peut être présent zéro, une ou plusieurs fois »
- Lorsque le paramètre est intitulé **dst**, cela signifie que l'instruction va modifier sa valeur.

Notes :

- « Notes associées à l'instruction **EXEMPLE** »
- IMPORTANT, « * » et « ** » devant le mnémonique de l'instruction :

  EXEMPLE : en l'absence du signe * devant le mnémonique de l'instruction, l'instruction est dite « atomique », ce qui signifie que son exécution est exclusive, et que les instructions s'exécutant en parallèle dans les autres simultanés ne sont pas exécutées en même temps (Exemple: un ***MOV*** #VARIABLE 10 est exclusif, et on est sûr qu'aucune instruction dans les tâches en parallèle va interférer jusqu'à la fin de l'affectation de VARIABLE avec la valeur 10.

  EXEMPLE** : en revanche, lorsque le signe * apparaît devant le mnémonique de l'instruction, elle est dite « non atomique », ce qui signifie que son exécution n'est pas exclusive, et que d'autres instructions peuvent éventuellement être exécutées en même temps dans d'autres simultanés. Des précautions devront donc être prises pour s'assurer du bon fonctionnement global (Exemple: positionnement avec ***POSA, les axes sont sollicités jusqu'à la réalisation du positionnement, alors que les instructions dans d'autres simultanés seraient exécutées en même temps et pendant le positionnement).

  ****EXEMPLE** : de même, lorsque le signe ** apparaît devant le mnémonique de l'instruction, elle est aussi « non atomique », et utilise de plus le timer TMR[R8] associé au simultané. Le programme utilisant cette instruction doit tenir compte du fait que TMR[R8] est modifié par cette instruction (Exemple: l'instruction ***WAIT*** utilise le timer du simultané, on peut utiliser TMR[R8] avant et/ou après ***WAIT***, mais le ***WAIT*** réinitialise TMR[R8]).

Voir aussi « Groupe d'instructions auquel appartient l'instruction **EXEMPLE**, Variables systèmes, Instructions, ou autres points pouvant être intéressants dans le cadre de l'instruction **EXEMPLE** »

Exemple :

« Exemple Uniprog utilisable sur le E700 pour l'instruction EXEMPLE, exemple que l'on peut retrouver dans le dossier des Exemples Uniprog / Instruction EXEMPLE / Exemple fournis sur le CD de la Documentation et des Outils livrés avec le E700 : »

```
LABEL:      « INSTRUCTIONS »              ; « Commentaires »
            .
            .
            .
            EXEMPLE par1 par2             ; "Instruction EXEMPLE"
            .
            .
            .
            END                           ; Fin programme
```

« Explications de l'exemple en Uniprog ci-dessus »

L'instruction étudiée apparaît en évidence dans l'exemple Uniprog, pour permettre de visualiser rapidement la mise en œuvre de l'instruction.

Note, en ce qui concerne les fichiers exemples fournis sous forme de fichiers à charger sur le E700 :

Un répertoire commun « Fichiers E700 communs aux exemples » regroupe les fichiers à charger premièrement sur le E700.

Pour chacune des instructions Uniprog, un répertoire du même nom que le mnémonique de l'instruction, contient les fichiers spécifiques de l'exemple pour l'instruction étudiée : ces fichiers sont à charger ensuite sur le E700.

Pour d'autres exemples, ne pas oublier de recharger les fichiers communs avant de charger ceux de l'exemple (cas de l'AUTOMAT.E7M qui est spécifique pour certaines instructions, et qu'il est nécessaire de recharger avant la mise en œuvre de l'exemple).

ABS dst

L'instruction **ABS** retourne la valeur absolue de son argument *dst*.

- *dst* : valeur modifiée, **dst ← ABS (dst)**

Exemple :

```
;              DECLARATION CONSTANTES ET VARIABLES
               VAL =                          ; Variable VAL
;
;                           INITIALISATIONS
               MOV    R0   8                  ; Init registre R0 a 8
               MOV    #VAL -0.7               ; Init VAL a -0.7
;
;                          CORPS DU PROGRAMME
               ABS    R0                      ; R0 vaut 8
               ABS    #VAL                    ; VAL vaut 0.7
;
               END                            ; Fin du programme
```

ADD dst src

L'instruction **ADD** effectue l'addition de ses deux arguments et retourne le résultat dans le premier argument.

- *dst* : valeur modifiée, **dst ← dst + src**
- *src* : valeur à additionner avec *dst*.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
                PI  =  3.141593                 ; Constante Pi
                VAL =                           ; Variable VAL
;
;                         INITIALISATIONS
                MOV     R0   8                  ; Init registre R0 a 8
                MOV     #VAL -0.7               ; Init VAL a -0.7
;
;                       CORPS DU PROGRAMME
;
                ADD     R0   PI                 ; R0 vaut 8 + Pi = 11.141593
                ADD     #VAL R0                 ; VAL vaut -0.7 + 8 + Pi
                                                ;          = 10.441593
;
                END                             ; Fin du programme
```

Il est important de noter que la constante PI n'est pas précédée du signe # puisque PI n'est pas une variable, mais une constante. PI, en tant que constante n'a pas de contenu. C'est une valeur immédiate. C'est une substitution : on a substitué 3.141593 par le symbole PI.

La commande suivante provoquerait une erreur :

```
                ADD     PI #VAR
```

Car on ne peut pas aller écrire dans PI, puisque PI est une constante et une constante n'a pas de contenu !

ADDD src

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **ADDD** effectue l'addition de la valeur de *src* dans l'accumulateur (**ADD**ition **D**irecte).

- *src* : valeur à ajouter à l'accumulateur, **accum[PNB] ← accum[PNB] + *src*.**
- *accum[PNB]* : représente la mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, non utilisable directement en Uniprog).

  Est modifié par l'instruction pour contenir le résultat.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
                PI  =  3.141593                 ; Constante Pi
;
;                             INITIALISATIONS
                MOV     R0   53                 ; Init registre R0 a 53
;
;                            CORPS DU PROGRAMME
                LOAD    R0                      ; Charge le contenu de R0
                                                ; dans l'accumulateur
                                                ; Accumulateur vaut ici 53
                ADDD    PI                      ; accum vaut accum + Pi
                                                ; = 53 + 3.141593 = 56.141593
                STORE   R0                      ; Stocke l'accumulateur
                                                ; dans R0, R0 vaut 56.141593
;
                END                             ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Initialiser le registre R0 avec 53.
- Effectuer les opérations R0 <- (R0 + Pi) en utilisant l'accumulateur.

AND dst src

L'instruction **AND** exécute un ET logique bit-à-bit de ses deux arguments (AND, noté aussi « & »), et retourne le résultat dans le premier de ces arguments.

- *src* : une des deux valeurs pour le **AND**.
- *dst* : l'autre des deux valeurs pour le **AND**, contient le résultat : **dst ← dst AND src**

Notes :

dst et *src* sont des nombres réels dont les valeurs sont d'abord copiées en interne du système, puis arrondies vers l'entier le plus proche et codées en entier relatif sur 4 octets avant d'être utilisées dans l'exécution du **AND**.

Exemple en logique :

```
4.58 & 15   = 5 & 15

            = 0000 0000  0000 0000  0000 0000  0000 0101
            & 0000 0000  0000 0000  0000 0000  0000 1111

            = 0000 0000  0000 0000  0000 0000  0000 0101

            = 15
```

Donc 4.58 & 15 = 15

Exemple en Uniprog :

Voici une procédure qui comporte deux paramètres : R0 et R1. R0 est un paramètre “valeur” (c’est-à-dire que son contenu ne va pas varier; il sera donc restauré avec sa valeur initiale d’avant l’appel en fin de procédure) R0 est un nombre entier positif quelconque. La procédure proposée ci-dessous retourne dans R1 la réponse à la question R0 est-il impair ? R1 est donc un paramètre “variable” puisque son contenu sera modifié par l’exécution de la procédure.

Appel de notre procédure :

```
            MOV     R0 35                   ; Ici R0 vaut 35
                                            ; (R0 est donc impair)
            CALL    EVEN                    ; Appel et execution
                                            ; de la proc. EVEN
           . . .                            ; Ici R1 vaut 1
                                            ; (car R0=35 est impair)

            MOV     R0 34                   ; Ici R0 vaut 34
                                            ; (il est donc pair)
            CALL    EVEN                    ; Appel et execution
                                            ; de la proc. EVEN
           . . .                            ; Ici R1 vaut 0
                                            ; (car R0=34 est pair)
```

Code de la procédure EVEN (R1 vaudra 1 si R0 est impair et R1 vaudra 0 si R0 est pair) :

```
EVEN:       MOV     R1 R0           ; Ici R1 vaut R0
            AND     R1 1            ; Masque=1 pour ne garder que bit 0
            END                     ; Fin de la procedure et retour au
                                    ; programme principal.
```

Donc EVEN est un symbole inventé par le programmeur. C’est un label. L’instruction ***CALL*** est un appel de procédure, procédure qui débute au label passé comme argument du ***CALL***. Une procédure, appelée aussi routine ou sous-programme se termine par une instruction ***END***. Lorsque le ***END*** est exécuté, le programme retourne au point d’où il a été appelé. Ce point est symbolisé dans notre exemple par «. . .».

Pour que ce retour puisse effectivement avoir lieu, ce point (appelé *adresse de retour*) est stocké sur la pile. Le ***END***, lui, va dépiler l’adresse de retour et ainsi permettre un retour au bon endroit. On pourrait donc remplacer une instruction ***CALL*** par le code suivant :

```
            MOV     R0 34                   ; Ici R0 vaut 34
                                            ; (il est donc pair)
            PUSH    RETOUR                  ; Empile adresse label RETOUR:
            JMP     EVEN                    ; effectue EVEN puis
                                            ; effectue RETOUR apres EVEN
                                            ; (equivalent    CALL EVEN
                                            ;  puis          JMP RETOUR)
;
RETOUR:     . . .                           ; A la fin de EVEN, R1 vaut 0
                                            ; (car R0=34 est pair)
```

Nous verrons plus loin la signification exacte de ***PUSH*** et ***JMP***. Pour l’instant, il suffit de savoir que ***PUSH*** dépose une valeur sur la pile (ici l’adresse correspondant au symbole RETOUR). ***JMP*** (**J**u**MP**) exécute un saut inconditionnel à l’adresse passée en argument (EVEN dans notre exemple).

ASIM [num] label

L'instruction **ASIM** crée une nouvelle tâche simultanée en exécution (**A**ctivate **SIM**ultaneous task).

- *label* : *label* est l'étiquette de la première instruction à exécuter dans la nouvelle tâche.
- *[num]* : numéro de la nouvelle tâche, *num* est optionnel.

  Si *num* est omis, c'est l'exécuteur du E700 qui choisira lui-même un numéro libre pour y exécuter la tâche demandée.
- *ASIMUSER* Variable système, permet de choisir lorsque *num* est omis, qui gère le cas où il n'y a plus de place libre comme numéro de nouvelle tâche :

  0 : gestion par le système d'exploitation (Runtime error 25 si erreur).

  1 : gestion en Uniprog avec la variable système SIMACT qui vaut -1 en cas d'erreur.

Notes :

La tâche courante qui exécute l'instruction **ASIM** continue normalement à s'exécuter à l'instruction suivant le **ASIM** : il y a donc exécution en parallèle des deux tâches, celle contenant le **ASIM**, et celle démarrée par le **ASIM**, qui ont bien sûr deux numéros différents.

Il y a possibilité de faire exécuter jusqu'à 10 tâches Uniprog simultanément :

- Ces tâches sont numérotées de 0 à 9. La tâche 0 est la tâche par défaut pour l'exécution du programme CYCLE. C'est aussi dans la tâche 0 que s'exécute par défaut un programme écrit en ISO, ainsi que le programme de démarrage (Power ON program).
- En appuyant sur le bouton START, c'est la tâche 0 qui démarre.
- La tâche 9 est la tâche dite «AUTOMATE». Elle s'exécute automatiquement à l'enclenchement, sans être influencée par START ou STOP. C'est le bouton AUTO qui gère l'exécution de la tâche AUTOMATE.
- ASIM peut être combinée avec BRIN1 dans certains cas particuliers. Voir l'extension de syntaxe de BRIN1 pour de plus amples renseignements.

Voir aussi instructions ***KSIM***, ***PSIM***, ***RSIM*** et ***BRIN1*.**

Exemple :

Imaginons que pendant l'exécution du cycle, sur une machine d'usinage, une pédale permette d'ouvrir les portes de protection. On va donc construire une tâche simultanée qui, en parallèle avec l'exécution du cycle, gèrera la pédale et les portes.

On suppose que la pédale est branchée sur l'entrée numéro 0 et que les portes sont commandées par la sortie 1.

En deuxième partie de ce manuel, nous analyserons toutes les variables système disponibles. De petits exemples seront donnés pour illustrer l'utilisation de ces variables. La liste de ces variables se trouve dans le fichier SYSTEM.E7M.

En attendant, pour notre exemple, sachons simplement que la variable qui désigne une entrée s'appelle **IN** et la variable qui désigne une sortie s'appelle **OUT**.

Comme il y a 8 entrées et 8 sorties internes sur le E700, ces variables sont indexables par un index de 0 à 7.

Note : CIN / COUT pour les entrées sorties sur **c**arte I/O.

MIN / MOUT pour les entrées sorties sur **m**odule I/O.

RIN / ROUT pour les entrées sorties sur **r**emote (panneau clavier/écran du E700).

En Uniprog :

```
            PORTES = 1                      ; Sortie 1 : portes
            PEDALE = 0                      ; Entree 0: pedale
                                            ; Programme CYCLE (tache 0)
          . . .                             ; demarre avec bouton START
            ASIM   1 DRIVEDOOR              ; Activation tache sim. 1
          . . .                             ; Execution CYCLE (tache 0)
            KSIM    1                       ; Arret tache 1
            END                             ; Fin programme CYCLE
                                            ; (tache 0)

DRIVEDOOR:  MOV    #OUT[PORTES] #IN[PEDALE]
            BRIN1  #OUT[PORTES] DRIVE1
DRIVE0:     WAIT0  #IN[PEDALE]
            ON     #OUT[PORTES]

DRIVE1:     WAIT1  #IN[PEDALE]
            OFF    #OUT[PORTES]
            JMP    DRIVE0
```

Quelques explications s'imposent :

L'idée est de copier l'état de l'entrée PEDALE dans la sortie PORTES. On aurait pu faire très simple avec le code suivant :

```
DRIVEDOOR:  MOV    #OUT[PORTES] #IN[PEDALE]
            JMP    DRIVEDOOR
```

L'avantage du premier code proposé est qu'il n'exécute l'affectation que lorsque l'état de l'entrée PEDALE change (passe de 0 à 1 ou de 1 à 0). Cela évite d'aller écrire des milliers de fois la même valeur sur la sortie PORTES; ces écritures prenant un certain temps.

BRIN1 signifie «Branche-toi au label DRIVE1 si la sortie PORTES n'est pas nulle», ce qui sous-entend «continue si la sortie PORTES est nulle».

WAIT0 signifie «Attend tant que l'entrée PEDALE est à 0. Continue dès qu'elle devient non-nulle».

ON signifie «Mettre à 1». On aurait pu remplacer

```
            ON     #OUT[PORTES]
```

Par

```
            MOV    #OUT[PORTES] 1
```

WAIT1 signifie «Attend tant que l'entrée PEDALE est non-nulle. Continue dès qu'elle devient nulle».

OFF signifie «Mettre à 0». On aurait pu remplacer

```
            OFF    #OUT[PORTES]
```

Par

```
            MOV    #OUT[PORTES] 0
```

JMP DRIVE0 signifie «Saute au label DRIVE0». On dit que le **J**u**MP** est le saut ou branchement inconditionnel car le saut a toujours lieu, quelles que soient les conditions. On verra plus loin dans ce manuel qu'il existe des sauts ou branchement conditionnels, à savoir que le saut n'a lieu que si certaines conditions sont remplies. On a déjà vu un branchement conditionnel : ***BRIN1***. Dans l'instruction ***BRIN1*** *src1 src2*, le branchement à *src2* n'a lieu que si *src1* est non-nul. Ici, la condition à remplir est «*src1* non-nul».

On se convainc facilement que la tâche simultanée 1 (DRIVEDOOR) ne se termine jamais. Elle est ce qu'on appelle une boucle infinie puisque aucune condition ne permet de lui dire de terminer sa boucle.

Donc, juste avant que le cycle (tâche 0) s'arrête, cette tâche 0 devra «tuer» les éventuelles tâches activées. En l'occurrence, elle devra tuer la tâche 1 puisqu'elle (la tâche 1) ne peut pas s'arrêter toute seule ! On se rappelle qu'un programme de cycle ne se termine que quand toutes les tâches (sauf la tâche AUTOMAT numéro 9) sont terminées. L'instruction qui permet de tuer une tâche est ***KSIM*** *src1* ou *src1* est le numéro de la tâche.

Dans la plupart des cas, les diverses tâches devant être exécutées en simultané ne durent que quelques instants. C'est pourquoi un simultané pourra probablement exécuter plusieurs tâches au cours du temps. Mais il est parfois difficile de savoir si à tel ou tel instant un simultané est libre ou occupé. Certes tout est disponible en Uniprog pour construire son propre dispatcher, mais ***ASIM*** offre cette possibilité très facilement. Il suffit d'activer la tâche sans lui donner de numéro de simultané. C'est l'exécuteur qui cherchera un simultané de libre à cet instant pour y exécuter la tâche.

```
            ASIM    DRIVEDOOR                ; Activation tache sim.
```

Il existe une variable SYSTEM qui permet de savoir quel numéro de simultané a été choisi par l'exécuteur E700. Cette variable s'appelle SIMACT.

Lorsque ***ASIM*** a été exécuté, SIMACT contiendra le numéro (de 0 à 8) choisi par l'exécuteur. Si tous les simultanés sont occupés, il n'y aura pas d'activation et SIAMCT contiendra –1.

Bien évidemment, il y a une variable SIMACT par simultané. Il faut donc l'indexer par ce numéro de simultané. Il existe une autre variable qui s'appelle PNB (Program NumBer) qui retourne le numéro du simultané.

```
            MOV     R0 #PNB
LOOP:       ASIM    DRIVEDOOR                ; Activation tache sim.
            CMP     #SIMACT[R0] -1
            JE      LOOP                    ; Recommencer tant que
                                             ; l'activation echoue
```

Dans cet exemple, on active la tâche DRIVEDOOR dans un simultané libre quelconque. Si tous les simultanés sont occupés au moment de l'essai d'activation, on recommence indéfiniment jusqu'à que l'un d'entre eux se libère pour y exécuter DRIVEDOOR.

La variable système ASIMUSER permet de laisser le système d'exploitation générer lui-même une erreur en cas d'échec de ASIM.

ATN dst

L'instruction **ATN** calcule l'arc tangente de son argument *dst*.

- *dst* : valeur modifiée, **dst ← TAN-1 (dst)**

  « **ATN** *dst* » est définie pour tout *dst* réel.

Voir aussi Variable système DEG, Instructions : ***COS***, ***SIN***, ***TAN***.

Notes :

C'est la seule fonction trigonométrique inverse car elle permet d'en déduire les autres !

Les unités de mesure dans lesquelles travaillent les fonctions trigonométriques sont définies dans la variable système # DEG :

Avant l'instruction **ATN**, on affectera donc la variable DEG avec l'une des instructions ***ON***, ***OFF***, ***MOV*** de la valeur :

- ON pour travailler en degrés (valeur 1, par défaut au démarrage du E700).
- OFF pour travailler en radians (valeur 0).

Si on met DEG à 0, alors l'argument dst doit contenir une valeur d'angle en radians.

Exemple :

```
ON      #DEG            ; Travail en degres
MOV     R0 1            ; R0 vaut 1
ATN     R0              ; R0 vaut 45 (en degres)

OFF     #DEG            ; Travail en radians
MOV     R0 1            ; R0 vaut 1
ATN     R0              ; R0 vaut 0.7854 = PI/4
                        ; (en radians)

END                     ; Fin du programme
```

BRIN0 in label

L'instruction **BRIN0** effectue un saut conditionnel à l'instruction indexée par *label* si la condition *in* vaut 0 (**BR**anch if **IN** is **0**).

- *in* : *in* est la condition pour effectuer le saut, **saut si *in* = 0**
- *label* : *label* est l'étiquette de l'instruction suivante à exécuter si la condition est remplie.

Notes :

Dans l'ancien Uniprog, le premier argument (la condition) ne pouvait être qu'une entrée (IN). C'est pourquoi cette instruction s'appelle **BRIN0**. Dans le nouvel Uniprog E700, le premier argument (la condition) peut être n'importe quelle sorte de paramètre, valeur, variable, tableau, registre.

Voir aussi Instructions ***CALIN0***, ***CALIN1***, ***BRIN1*****.**

Exemple :

Principe de l'exemple, attente avec timeout :

- L'idée est que l'on doive attendre qu'une entrée, disons l'entrée 3, passe à 0.
- Si au bout de 5 secondes, elle n'est toujours pas passée à 0, on exécute un code d'erreur.
- Pour cela, on va charger le **TiMeR** TMR avec 5000 millisecondes (il y a un timer TMR par tâche simultanée) et attendre que ce temps soit écoulé.
- Pour tester si le timer est actif ou non, on a la variable TAF (**T**imer **A**ctive **F**lag) qui change d'état quand le timer s'arrête : chaque tâche simultanée possède sa propre variable TAF.

Une autre idée est que cette procédure soit accessible depuis n'importe quelle tâche simultanée :

- La procédure doit donc savoir dans quelle tâche elle est en train de s'exécuter.
- Pour le savoir, il y a la variable PNB (**P**rogram **N**um**B**er). PNB = 0 dans la tâche 0, PNB = 1 dans la tâche 1, etc.

Le fait qu'une procédure puisse être exécutée depuis plusieurs tâches simultanées s'appelle la réentrance.

En Uniprog :

```
; ------------------------------------------------------------------
;  PROCEDURE WAIT1TOUT (Wait with timeout if #IN[3])
;  Timeout de 5 secondes
; ------------------------------------------------------------------

WAIT1TOUT:  PUSH    R0                      ; Sauvegarde R0 pour
                                            ; appeler la procedure
                                            ; depuis n'importe ou.
            MOV     R0 #PNB                 ; R0 indice de tableau.
                                            ; Rappel: les indices ne
                                            ; peuvent etre que des
                                            ; constantes ou registres:
                                            ; Or PNB est une variable!
            MOV     #TMR[R0] 5000           ; Init timer avec 5000 ms
                                            ; Automatiquement: TAF = 1
LOOP:       BRIN0   #IN[3] ENDWAIT          ; IN[3] passe a 0:
                                            ; ok, termine!
            BRIN1   #TAF[R0] LOOP           ; Timeout pas encore
                                            ; ecoule: on attend

            .                               ; Si on arrive ici, c'est
                                            ; que IN[3] n'a pas ete nulle
            .                               ; durant les 5 secondes
                                            ; C'est donc ici que devra
            .                               ; etre ecrit le code
                                            ;  d'erreur a executer.

ENDWAIT:    POP     R0                      ; Restauration registre R0

            END                             ; Fin de la procedure
```

Remarque : le registre R8 contient la valeur de PNB, et R8 peut être utilisé comme indice de tableau, ce qui signifie que l'exemple donné ci-avant s'écrit aussi :

```
WAIT1TOUT:  MOV     #TMR[R8] 5000           ; Init timer avec 5000 ms
                                            ; Automatiquement: TAF = 1
LOOP:       BRIN0   #IN[3] ENDWAIT          ; IN[3]=0? oui: ok,termine!

            BRIN1   #TAF[R8] LOOP           ; Timeout? non: on attend

            . . .                           ; code fonction erreur ici

ENDWAIT:    END                             ; Fin de la procedure
```

BRIN1 in label [num label$_1$]

L'instruction **BRIN1** effectue un saut conditionnel à l'instruction indexée par *label* si la condition *in* ne vaut pas 0 (**BR**anch if **IN** is **1**).

- *in* : *in* est la condition pour effectuer le saut, **saut si *in* ≠ 0**
- *label* : *label* est l'étiquette de l'instruction suivante à exécuter si la condition est remplie.

Extension de syntaxe : Il est possible de condenser BRIN1 et ASIM en une seule instruction, ceci pour garantir qu'entre le test (BRIN1) et l'activation (ASIM), aucune tâche ne pourra s'exécuter.

- *num* : numéro d'une nouvelle tâche à activer.
- *label$_1$* : *label$_1$* est l'étiquette de la première instruction à exécuter dans la nouvelle tâche.

Notes :

Voir ***BRIN0*** ci-dessus qui est similaire à **BRIN1** pour les détails de fonctionnement et pour l'exemple. Dans le cas du **BRIN1**, la condition est remplie si le premier argument est non-nul.

Voir aussi Instructions ***CALIN0***, ***CALIN1***, ***BRIN0*.**

Concernant l'extension de syntaxe (BRIN1 avec 4 arguments) :

Les deux arguments num et label$_1$ sont optionnels.

Avant d'activer une nouvelle tâche simultanée, il peut être utile de tout d'abord vérifier que celle-ci soit libre. Si le numéro choisi est libre, alors on peut l'activer. Sinon, on saute à *label*. Le problème est que test et activation se font en deux temps. Or, entre ces deux temps, il se pourrait qu'une autre tâche décide d'occuper le numéro choisi. Il est donc impératif d'atomiser ces deux instructions dans ce cas précis. C'est ce que propose le BRIN1 avec 4 arguments :

```
BRIN1  #SIMPTR[num] label    ; Teste si la tâche numéro num est libre.
ASIM   num label1            ; Si c'est le cas, alors activer la tâche numéro num au label label1.
```

Malheureusement, si une autre tâche active, elle aussi, la tâche numéro num à un autre label que label$_1$, le programme ne s'exécutera pas correctement. Pour éviter ce cas particulier, pour que personne ne puisse agir entre le test et l'activation, on réalise ces deux instructions en une seule fois :

```
BRIN1  #SIMPTR[num] label num label1
```

Notes :

- BRIN1 admet deux ou quatre arguments. La syntaxe à trois arguments n'est pas valide. Cela signifie qu'on ne peut pas condenser BRIN1 #SIMPTR[num] label et ASIM label$_1$.
- Cette extension de syntaxe à quatre arguments n'est pas autorisée pour BRIN0. Elle n'est valable que pour BRIN1.
- Voir l'instruction ASIM pour plus de détails concernant l'activation de tâches simultanées.

(BRKPT)

À ne pas utiliser !

BRM label

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **BRM** effectue un saut conditionnel à l'instruction indexée par *label* si l'accumulateur est strictement plus petit que 0 (**BR**anch if accumulator is **M**inus **0**).

- *label* : *label* est l'étiquette de l'instruction suivante à exécuter si la condition est remplie.
- *accum[PNB]* : mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, en lecture et écriture).

  Est lu par cette instruction pour effectuer le test.

Exemple :

Principe de l'exemple :

- On écrit une procédure qui enclenche la sortie 0 pendant 1 seconde si le paramètre R0 est inférieur à 0.
- On enclenche la sortie 1 pendant 1 seconde si le paramètre R0 est plus grand ou égal à 0.

En Uniprog :

```
MINUS  = 0
PLUS   = 1
TESTPM:     LOAD   R0                    ; Accumulateur vaut R0
            BRM    TESTM                 ; Saut a TESTM si accum < 0

            ON     #OUT[PLUS]            ; Enclencher sortie PLUS
            WAIT   1                     ; Attente d'une seconde
            OFF    #OUT[PLUS]            ; Eteindre sortie PLUS
            JMP    TESTPM1               ; Termine. Saut a la fin

TESTM:      ON     #OUT[MINUS]           ; Enclencher sortie MINUS
            WAIT   1                     ; Attente d'une seconde
            OFF    #OUT[MINUS]           ; Eteindre sortie MINUS

TESTPM1:    END                          ; Fin de la procedure
```

BRNZ label

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **BRNZ** effectue un saut conditionnel à l'instruction indexée par *label* si l'accumulateur est différent de 0 (**BR**anch if accumulator is **N**on-**Z**ero).

- *label* : *label* est l'étiquette de l'instruction suivante à exécuter si la condition est remplie.
- *accum[PNB]* : mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, en lecture et écriture).

  Est lu par cette instruction pour effectuer le test.

Exemple :

Principe de l'exemple :

- On écrit une procédure qui enclenche la sortie 0 pendant 1 seconde si le paramètre R0 est différent de 0.
- On enclenche la sortie 1 pendant 1 seconde si le paramètre R0 est égal à 0.

En Uniprog :

```
MINUS  = 0
PLUS   = 1
TESTPM:     LOAD   R0                    ; Accumulateur vaut R0
            BRNZ   TESTM                 ; Saut a TESTM si accum<>0

            ON     #OUT[PLUS]            ; Enclencher sortie PLUS
            WAIT   1                     ; Attente d'une seconde
            OFF    #OUT[PLUS]            ; Eteindre sortie PLUS
            JMP    TESTPM1               ; Termine. Saut a la fin

TESTM:      ON     #OUT[MINUS]           ; Enclencher sortie MINUS
            WAIT   1                     ; Attente d'une seconde
            OFF    #OUT[MINUS]           ; Eteindre sortie MINUS

TESTPM1:    END                          ; Fin de la procedure
```

BRP label

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **BRP** effectue un saut conditionnel à l'instruction indexée par *label* si l'accumulateur est strictement plus grand que 0 (**BR**anch if accumulator is **P**ositive).

- *label* : *label* est l'étiquette de l'instruction suivante à exécuter si la condition est remplie.
- *accum[PNB]* : mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, en lecture et écriture).

  Est lu par cette instruction pour effectuer le test.

Exemple :

Principe de l'exemple :

- On écrit une procédure qui enclenche la sortie 0 pendant 1 seconde si le paramètre R0 est supérieur à 0.
- On enclenche la sortie 1 pendant 1 seconde si le paramètre R0 est inférieur ou égal à 0.

En Uniprog :

```
MINUS  = 0
PLUS   = 1
TESTPM:     LOAD   R0                    ; Accumulateur vaut R0
            BRP    TESTM                 ; Saut a TESTM si accum > 0

            ON     #OUT[PLUS]            ; Enclencher sortie PLUS
            WAIT   1                     ; Attente d'une seconde
            OFF    #OUT[PLUS]            ; Eteindre sortie PLUS
            JMP    TESTPM1               ; Termine. Saut a la fin

TESTM:      ON     #OUT[MINUS]           ; Enclencher sortie MINUS
            WAIT   1                     ; Attente d'une seconde
            OFF    #OUT[MINUS]           ; Eteindre sortie MINUS

TESTPM1:    END                          ; Fin de la procedure
```

BRZ label

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **BRZ** effectue un saut conditionnel à l'instruction indexée par *label* si l'accumulateur est égal à 0 (**BR**anch if accumulator is **Z**ero).

- *label* : *label* est l'étiquette de l'instruction suivante à exécuter si la condition est remplie.
- *accum[PNB]* : mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, en lecture et écriture).

  Est lu par cette instruction pour effectuer le test.

Exemple :

Principe de l'exemple :

- On écrit une procédure qui enclenche la sortie 0 pendant 1 seconde si le paramètre R0 est égal à 0.
- On enclenche la sortie 1 pendant 1 seconde si le paramètre R0 est différent de 0.

En Uniprog :

```
MINUS  = 0
PLUS   = 1
TESTPM:     LOAD    R0                  ; Accumulateur vaut R0
            BRZ     TESTM               ; Saut a TESTM si accum=0

            ON      #OUT[PLUS]          ; Enclencher sortie PLUS
            WAIT    1                   ; Attente d'une seconde
            OFF     #OUT[PLUS]          ; Eteindre sortie PLUS
            JMP     TESTPM1             ; Termine. Saut a la fin

TESTM:      ON      #OUT[MINUS]         ; Enclencher sortie MINUS
            WAIT    1                   ; Attente d'une seconde
            OFF     #OUT[MINUS]         ; Eteindre sortie MINUS

TESTPM1:    END                         ; Fin de la procedure
```

CALIN0 in label { param$_i$ }

L'instruction **CALIN0** effectue un appel conditionnel à la procédure désignée par *label* en effectuant les instructions à partir de *label* si la condition *in* vaut 0 (**CAL**L if **IN** is **0**).

- *in* : *in* est la condition pour effectuer le saut, **saut si *in* = 0**
- *label* : *label* est l'étiquette de la première instruction de la procédure à exécuter si la condition est remplie.
- *param$_i$:* Passage d'un nombre illimité de paramètres. Voir CALL pour plus de détails.

Notes :

L'argument condition *in* peut être une valeur, une variable, un tableau, un registre.

CALIN0 complète l'instruction de saut ***BRIN0*** pour simplifier le code dans certaines situations.

CALIN0 peut générer une erreur 6 en cas de débordement de pile (Stack Overflow).

Voir aussi Instructions ***CALIN1***, ***BRIN0***, ***BRIN1***.

Exemple :

Principe de l'exemple :

- Si la LED STOP est éteinte :
  - Une pédale est connectée sur l'entrée 1 (IN[1]).
  - Si la pédale est enfoncée, elle enclenche pendant 2 secondes une soufflette connectée sur la sortie 0 (OUT[0]).

En Uniprog :

```
            PEDALE = 1                      ; Constante PEDALE
            SOUFFL = 0                      ; Constante SOUFFL
            ...                             ; Bloc instructions ici
            CALIN0 #LED[LSTART] PEDAL       ; Appeler PEDAL
                                            ; si led START eteinte.
            END                             ; Fin du programme
; Procedure PEDAL:
PEDAL:      BRIN0  #IN[PEDALE] PEDALEND     ; Si IN[1]=0, fin!
            ON     #OUT[SOUFFL]             ; OUT[0] a 1
            WAIT    2                       ; Attendre 2 secondes
            OFF    #OUT[SOUFFL]             ; OUT[0] a 0
PEDALEND:   END                             ; Fin de la procedure
```

Explications :

- LSTOP est l'index de la LED verte du bouton STOP (la variable LED retourne l'état d'une led du panneau E700, la liste des index possibles pour LED se trouve dans le fichier E700KEY.E7M)
- On n'exécute donc la procédure PEDAL que si la LED du bouton STOP est éteinte.

CALIN1 in label { param$_i$ }

L'instruction **CALIN1** effectue un appel conditionnel à la procédure désignée par *label* en effectuant les instructions à partir de *label* si la condition *in* ne vaut pas 0 (**CAL**L if **IN** is **1**).

- *in* : *in* est la condition pour effectuer le saut, **saut si *in* ≠ 0**
- *label* : *label* est l'étiquette de la première instruction de la procédure à exécuter si la condition est remplie.
- *param$_i$:* Passage d'un nombre illimité de paramètres. Voir CALL pour plus de détails.

Notes :

Voir ***CALIN0*** ci-dessus qui est similaire à **CALIN1** pour les détails de fonctionnement et pour l'exemple. Dans le cas du **CALIN1**, la condition est remplie si le premier argument est non-nul.

Voir aussi Instructions ***CALIN0***, ***BRIN0***, ***BRIN1***.

CALL label { param$_i$ }

L'instruction **CALL** effectue un appel de sous-programme.

- *label* : *label* est l'étiquette de la première instruction de la procédure à exécuter.
- *param$_i$:* Passage d'un nombre illimité de paramètres.

Notes :

Sauvegarder les registres utilisés dans la prcédure avec les instructions ***PUSH*** et ***POP***.

Un sous-programme (procédure ou fonction) se termine par une instruction ***END*** : en fin de sous-programme, l'exécution se poursuit avec l'instruction qui suit le **CALL**.

CALL peut retourner une erreur 6 en cas de débordement de pile (Stack Overflow).

Voir aussi Instructions ***END***.

Exemple :

Procédure Uniprog d'arrêt de l'Uniprog en cas d'alarme, dans la tâche AUTOMAT (simultané 9) :

```
            . . .                           ; Instructions avant le CALL
            CALL    STOPUNI
            . . .                           ; Instructions apres le CALL
            END;                           ; Fin du programme

; Procedure STOPUNI:
STOPUNI:    PUSH    R0                      ; Sauvegarde de R0 sur pile

            MOV     R0 0                    ; Initialisation R0 a 0

; Arret des simultanes 0 a 8:
STPUNI1:    CMP      #PNB R0                ; R0 = tache courante ?
            JE       STPUNI2                ; Oui: ne pas arreter la tache
            RSIM     R0                     ; Reveil eventuel simultane R0
            KSIM     R0                     ; Arret du simultane R0
STPUNI2:    INC      R0                     ; Increment numero simultane
            CMP      R0 8                   ; R0 = 8 ?
            JLE      STPUNI1                ; R0 different de 8, on boucle

            STOPM   -1 0                    ; Arret des mouvements

            POP     R0                      ; Restitution R0 depuis pile
            END                             ; Fin de la procedure
```

Explications :

- La procédure STOPUNI est appelée depuis le programme principal.
- Elle utilise le registre R0 dont elle sauvegarde la valeur au début, puis la restitue à la fin avec les instructions ***PUSH*** et ***POP***.
- Dans une boucle où R0 va de 0 à 8, si R0 ne correspond pas à la tâche courante, elle termine la tâche R0 avec les instructions ***RSIM*** puis ***KSIM***.
- Elle termine enfin les mouvements avec l'instruction ***STOPM***.

Passage de paramètres

Exemple : Appel d'une fonction d'addition (ADDIT). On désire additionner deux nombres N1 et 5 et mettre le résultat dans une variable RES.

```
RES             =
N1              =
AUX             =

        ...
        CALL    ADDIT #RES #N1 5
        ...
        END
                      OUT   IN    IN
ADDIT:  MOV     #PARAM[0] #PARAM[1]
        ADD     #PARAM[0] #PARAM[2]
        END
```

> Note
> Tous les paramètres doivent être sur la même ligne

Les paramètres se trouvent dans les variables système #PARAM[i]. Le premier paramètre porte le numéro 0. On peut passer un nombre indéfini de paramètres. Les paramètres peuvent être variables (#RES) ou valeur (#N1 et 5). Le sens des flèches dans l'exemple ci-dessus illustre ces deux possibilités. Le paramètre #RES sera modifié après exécution de la procédure, alors que les deux autres paramètres restent constants (#N1 et 5).

Les paramètres utilisent une pile interne au système d'exploitation. Ils ne sont donc pas situés dans la variable système STACK.

Attention, il n'est pas possible d'utiliser récursivement les variables système PARAM[i]. Cela signifie que l'on ne peut pas faire un CALL proc #PARAM[i]. Il n'est pas interdit d'utiliser les registres (R0..R7) comme paramètres, mais cela est fortement déconseillé. En effet, dans l'exemple ci-dessous, cela fonctionnera si la procédure SUB2 n'utilise pas R0. Par contre, si elle l'utilise, cela ne fonctionnera plus du tout.

Par exemple : On désire soustraire 2 à notre précédente opération (SUB2).

Interdit

```
ADDIT:  MOV     #PARAM[0] #PARAM[1]
        ADD     #PARAM[0] #PARAM[2]
        CALL    SUB2 #PARAM[0]
        END

SUB2:   SUB     #PARAM[0] 2
        END
```

Déconseillé

```
ADDIT:  PUSH    R0
        MOV     R0 #PARAM[1]
        ADD     R0 #PARAM[2]
        CALL    SUB2 R0
        MOV     #PARAM[0] R0
        POP     R0
        END

SUB2:   SUB     #PARAM[0] 2
        END
```

Interdit

```
ADDIT:  PUSH    R0
        MOV     R0 #PARAM[1]
        ADD     R0 #PARAM[2]
        CALL    SUB2 R0
        MOV     #PARAM[0] R0
        POP     R0
        END

SUB2:   PUSH    R0
        SUB     #PARAM[0] 2
        POP     R0
        END
```

Utilisation correcte

```
ADDIT:  MOV     #AUX #PARAM[1]
        ADD     #AUX #PARAM[2]
        CALL    SUB2 #AUX
        MOV     #PARAM[0] #AUX
        END

SUB2:   (PUSH   R0)
        SUB     #PARAM[0] 2
        (POP    R0)
        END
```

Note :

Cette nouvelle fonctionnalité de passage de paramètre (version 3.06 et suivantes) n'entraîne pas de problème de compatibilité avec les programmes Uniprog écrits avec les anciennes versions (->3.05), à une exception près :

Pour les programmeurs qui ont remplacé des END par des POP & JMP, il y aura un problème car, comme cité précédemment, la pile des paramètres est indépendante et interne au système d'exploitation. Cette pile interne est alimentée par l'instruction CALL (CALIN0/CALIN1). C'est l'empilement des paramètres. Elle est vidée par l'instruction END (dépilement des paramètres).

Donc, si dans un programme, un END a été remplacé par les instructions POP et JMP, le dépilement des paramètres ne se fera pas. A la longue, il y a un risque de débordement de pile qui conduira à un blocage du système. A noter que même si aucun paramètre n'est passé, il y a quand même un empilement pour respecter les niveaux de profondeur d'appel.

Le remplacement d'un END par les instructions POP et JMP doit donc être modifié comme suit :

Utilisation normale	Remplacement du END par POP/JMP. Valable jusqu'à la version 3.05. Invalide dès la version 3.06	Remplacement du END par POP/PUSH/JMP. Valable dans toutes les versions
PROC: . . . **END**	**PROC:** . . . **POP** **#VAR** **JMP** **label**	**PROC:** . . . **POP** **#AUX** **PUSH** **label** **END**

CASE val label retour

L'instruction **CASE** effectue un test de cas dans un contexte ***SWITCH*** - ***ENDS*** (voir ces instructions). Dans le cas où la variable définie dans le ***SWITCH*** vaut *val*, effectue la procédure *label*, puis à la fin de la procédure *label*, continue à l'adresse *retour*.

- *val* : *val* est la valeur à comparer au contenu de la variable désignée dans le ***SWITCH***.
- *label* : *label* est l'étiquette de la première instruction de la procédure à exécuter si *val* est égale à la valeur de la variable désignée dans le ***SWITCH***.
- *retour* : à la fin de la procédure *label*, *retour* est l'étiquette de la première instruction à exécuter pour la suite des opérations.

Notes :

Se rapporter à l'instruction ***SWITCH*** en page ***167*** pour le détail sur cette instruction.

Voir aussi Instructions ***SWITCH***, ***ENDS***.

Exemple :

Implémentation de touches de fonctions dans l'écran utilisateur.

On implémente une fonction M3 au-dessus de la touche F3 et une fonction M5 au-dessus de la touche F5. Rappel : M3 enclenche la broche et M5 arrête la broche. La broche est sur la sortie 0.

Tout d'abord, il faut éditer le fichier DISPLAY.INI et y définir l'écran 0. Voici un exemple :

Dans DISPLAY.INI :

```
[Display0]
f3 = "M3"
f5 = "M5"
f6 = "FIN"
```

Ainsi, l'écran principal affichera le texte ci-dessus. Noter qu'après avoir modifié DISPLAY.INI, il faut éteindre le E700, puis le rallumer pour que les changements soient pris en compte.

Il faut maintenant implémenter le code qui sera appelé lorsqu'on pressera sur les touches F3, F5, ou F6 !

En Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
            BROCHE = 0                      ; Constante BROCHE=0
;
;                         INITIALISATIONS
            CALL   MACROM5                  ; Arret broche
;
;                       CORPS DU PROGRAMME
;
; Boucler sur touches F3, F5 et F6:
MENU0:      SWITCH #CURKEY                  ; Tester touche pressee
             CASE  KF3 MACROM3 MENUEND      ; Cas touche=F3 :
                                            ;   faire procedure MACROM3
                                            ;   puis aller a MENUEND
             CASE  KF5 MACROM5 MENUEND      ; Cas touche=F5 :
                                            ;   faire procedure MACROM5
                                            ;   puis aller a MENUEND
                                            ; Autre cas: touche=F6 :
                                            ;   continuer
MENUEND:    ENDS                            ; Fin de test
;
            CMP     #CURKEY KF6             ; Est-ce F6 ?
            JNE     MENU0                   ; non: reprendre MENU0
;
FIN:        CALL   MACROM5                  ; Arret broche
            END                             ; Fin du programme
;
;                             PROCEDURES
;
MACROM3:    ON     #OUT[BROCHE]             ; Activer broche avec OUT[0]=1
            WAIT1  #KEY[KF3]                ; Attendre tant que F3 pressee
            END                             ; Fin procedure MACROM3

MACROM5:    OFF    #OUT[BROCHE]             ; Arret broche avec OUT[0]=0
            WAIT1  #KEY[KF5]                ; Attendre tant que F5 pressee
            END                             ; Fin procedure MACROM5
```

Explications :

Dans la boucle MENU0 - MENUEND :

- La variable CURKEY contient le code de la touche actuellement pressée. Si aucune touche n'est pressée à cet instant, alors CURKEY vaut 255.
- Les constantes KF3, KF5 et KF6 proviennent du fichier E700KEY.E7M. Ce sont les codes des touches F3, F5 et F6.

Donc si la touche F3 est enfoncée, le programme va appeler MACROM3 qui enclenchera la broche, puis qui attendra que la touche F3 soit relâchée.

Si la touche F5 est enfoncée, le programme va appeler MACROM5 qui arrêtera la broche, puis qui attendra que la touche F5 soit relâchée.

Si la touche F6 est enfoncée, le programme ne boucle plus et se termine.

On sous-entend ici que la broche n'est pas pilotée par un convertisseur de fréquence 0-10V, mais par une simple sortie, ce qui ne permet pas de varier la vitesse de la broche dans notre exemple.

CINR dst

L'instruction **CINR** arrondit le contenu de *dst* à l'entier le plus proche (**C**onversion to **IN**teger **R**ound).

- *dst* : valeur modifiée, **dst ← Round (dst)**

Notes :

Différences entre **CINR** et ***CINT*** :

Dst	1.55	1.5	1.45	-1.45	-1.5	-1.55
CINT dst	1	1	1	-1	-1	-1
CINR dst	2	2	1	-1	-2	-2

Voir aussi Instruction ***CINT***.

Exemple :

```
MOV    R0 1.55              ; Ici R0 vaut 1.55
CINR   R0                   ; Ici R0 vaut 2
MOV    R1 -1.55             ; Ici R1 vaut -1.55
CINR   R1                   ; Ici R1 vaut -2

CINR   3.14                 ; Provoque une erreur en
                            ; execution car la destination
                            ; du resultat est illegale
```

CINT dst

L'instruction **CINT** tronque le contenu de *dst* en supprimant la partie fractionnaire du nombre (**C**onversion to **IN**teger **T**runc).

- *dst* : valeur modifiée, **dst ← Round (dst)**

Notes :

Différences entre ***CINR*** et **CINT** :

Dst	1.55	1.5	1.45	-1.45	-1.5	-1.55
CINT *dst*	1	1	1	-1	-1	-1
CINR *dst*	2	2	1	-1	-2	-2

Voir aussi Instruction ***CINR***.

Exemple :

```
MOV     R0 1.55                 ; Ici R0 vaut 1.55
CINT    R0                      ; Ici R0 vaut 1
MOV     R0 -1.55                ; Ici R0 vaut -1.55
CINT    R0                      ; Ici R0 vaut -1

CINT    3.14                    ; Provoque une erreur en
                                ; execution car la destination
                                ; du resultat est illegale
```

*CIRA ax1 val1 ax2 val2 [cx cy mode]

L'instruction **CIRA** effectue une interpolation circulaire en mode absolu (en ISO : G90, G02, ou G03) et génère un arc de cercle entre les axes *ax1* et *ax2*, dont le point de départ est la position courante, et le point d'arrivée a pour coordonnées absolues (*val1*; *val2*).

Deux syntaxes possibles :

- *cx*, *cy* et *mode* sont omis : définir préalablement un rayon avec l'instruction «***RAD*** *rayon mode*».
- *cx*, *cy* et *mode* sont précisés : on définit un centre (*cx*; *cy*) en coordonnées relatives à la position actuelle et un sens de rotation avec *mode*. Ce sont I et J en ISO.

- *ax1* : Numéro ou nom de l'axe désignant le premier axe du plan contenant le cercle.

  NB : pour simplifier les explications dans la suite, nous appellerons cet axe X.
- *val1* : Coordonnée absolue du point d'arrivée en X.
- *ax2* : Numéro ou nom de l'axe désignant le second axe du plan contenant le cercle.

  NB : pour simplifier les explications dans la suite, nous appellerons cet axe Y.
- *val2* : Coordonnée absolue du point d'arrivée en Y.

Les arguments suivants sont optionnels. Mais si on les utilise, ils doivent y figurer tous les trois !

- *cx :* Coordonnée en X du centre du cercle par lequel passe l'arc. Cette coordonnée est relative au point de départ de l'arc. C'est la coordonnée I de l'ISO.
- *cy :* Coordonnée en Y du centre du cercle par lequel passe l'arc. Cette coordonnée est relative au point de départ de l'arc. C'est la coordonnée J de l'ISO.
- *mode* contient le sens de rotation :

  0 : pour le sens anti-horaire.

  1 : pour le sens horaire.

  Par analogie, *mode* = 1 pour un G02 en ISO, et *mode* = 0 pour un G03 en ISO.

Notes :

C'est toujours le chemin le plus court qui est choisi pour exécuter l'arc.

Pour générer un arc de 360^{o} (cercle complet), il suffit d'utiliser la deuxième syntaxe (avec *cx*, *cy* et *mode*) et de faire coïncider exactement (*val1*; *val2*) avec le point de départ.

Explications :

Un arc de cercle est défini par son point de départ, son point d'arrivée, son sens et :

- par son centre
- ou par son rayon.

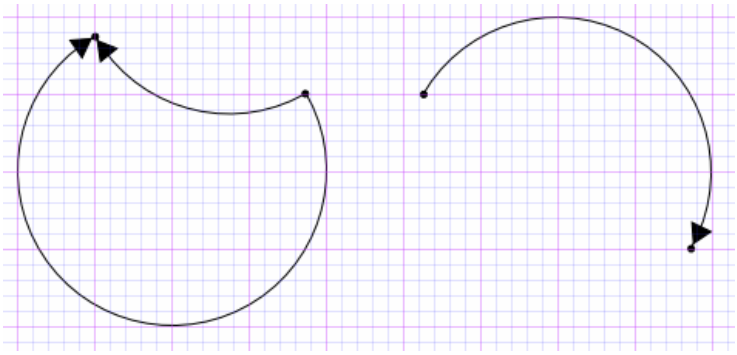
Le point de départ est la position au moment ou **CIRA** va être exécuté (rien à préciser).

Le point d'arrivée est donné par *val1* et *val2*.

Centre ou rayon :

- Si on donne son centre plutôt que son rayon, alors le centre est donné par *cx* et *cy* (comme I et J en ISO). Le sens est donné par *mode*.
- Si on donne son rayon plutôt que son centre, alors définir le rayon et le sens avec ***RAD***.

Dans l'option «rayon plutôt que centre», il y a une indétermination. En effet, Un point de départ, un point d'arrivée, un rayon et un sens, ne définissent pas qu'un seul arc, mais deux ! **CIRA** choisira dans cette situation l'arc le plus court.



Dans la figure ci-dessus, on voit bien qu'il existe toujours deux arcs (à gauche), sauf dans le cas ou le rayon est exactement la moitié de la distance entre les deux points (à droite). Uniprog choisit toujours l'arc le plus petit (le plus court).

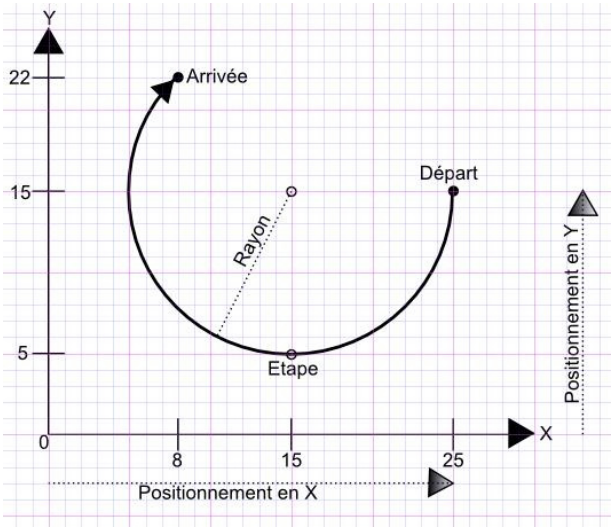
NB : afin de minimiser les erreurs de calcul, il est préférable de ne jamais programmer un arc de 180 degrés. Il vaut mieux programmer un tel arc en deux fois 90 degrés. Comme tous les cas ne sont pas défavorables, la possibilité de programmer un arc de 180 degrés n'est pas interdite.

Voir aussi Instructions ***DPATH***, ***ENDP***, ***RAD***, ***CIRR***.

Exemple :

Nous utiliserons les instructions ***DPATH***, ***ENDP***, ***POSA***, ***LINA2***, ***RAD*** (s'y rapporter si nécessaire).

Nous désirons réaliser le chemin ci-dessous, la position actuelle de l'outil étant l'origine (0 , 0).



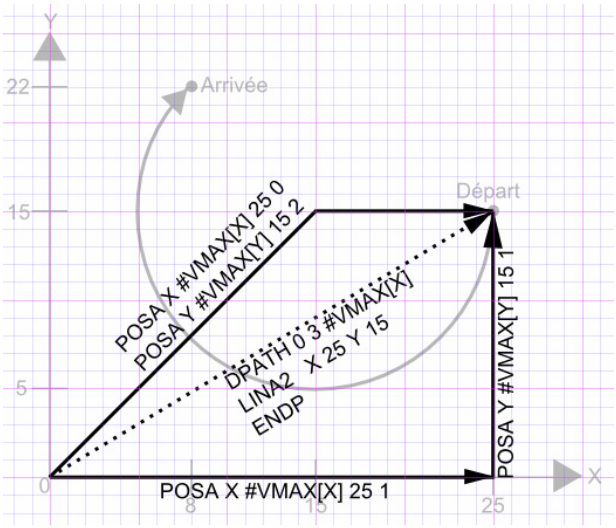
La première étape est de faire un positionnement au point de départ du chemin en (25 , 15) :

Positionnement depuis l'origine au point de départ de l'axe X, puis de l'axe Y :

```
POSA   X #VMAX[X] 25 1  ; Position de X a vitesse rapide
POSA   Y #VMAX[Y] 15 1  ; Position de Y a vitesse rapide
```

Ou bien pour mouvoir X et Y simultanément, il faut écrire :

```
POSA   X #VMAX[X] 25 0  ; Positionnement de X ...
POSA   Y #VMAX[Y] 15 2 ; ... et Y a vitesse rapide
```



La figure ci-dessus représente les trois chemins différents possibles pour arriver au point (25 , 15) :

En bas à droite, positionnement indépendant de X puis Y.

En haut à gauche, positionnement simultané de X et Y sans interpolation des deux axes. On suppose que les vitesses maximum en X et en Y sont identiques, c'est pourquoi la première phase du mouvement est dessinée exactement à 45 degrés.

Au milieu, positionnement par interpolation de X et Y et en utilisant l'instruction ***LINA2***.

NB : les positionnements simultanés ne sont pas des interpolations. C'est-à-dire que dans un positionnement simultané, il n'y a aucun asservissement de vitesse. Chacun des axes va à sa propre vitesse sans s'occuper des autres. Au contraire, en interpolation, les vitesses interdépendent entre chaque axe pour qu'ils arrivent en même temps au but, la vitesse du chemin est constante.

La deuxième étape est de décrire l'arc de cercle et aller au point d'arrivée (8 , 22). Il y a donc les deux possibilités, en définissant soit le centre, soit le rayon.

Par la méthode du rayon, il faudra passer par un point intermédiaire, donc en deux étapes, la première allant par exemple jusqu'au point (15 , 5), afin que le E700 choisisse le bon arc de cercle à chacune des étapes.

```
DPATH   0 3 #VMAX[X]      ; Def. de l'espace d'interpolation
 RAD    10 1              ; Rayon 10 et sens horaire
 CIRA   X 15 Y  5         ; Etape 1
 CIRA   X  8 Y 22         ; Etape 2
ENDP                      ; Fin du contour -> execution
```

Interpolation circulaire par la méthode du centre définit relativement au point de départ, avec le sens horaire.

Par rapport au point de départ, le centre est situé à (15 – 25 , 15 – 15) = (-10 , 0) :

```
DPATH   0 3 #VMAX[X]      ; Def. espace interpolation X et Y
CIRA   X 8 Y 22 -10 0 1   ; Arc par le centre
ENDP                      ; Fin du contour -> execution
```

Important :

Il est utile de savoir que l'instruction **CIRA** calcule les vitesses à appliquer à chaque instant pour que le chemin correct soit effectué. Un arc de cercle sera découpé en petits segments de droites. Ces segments seront stockés dans un tampon. Dès que le tampon sera plein, les mouvements

commenceront. Chaque petit mouvement libère de la place dans le tampon, ainsi, d'autres segments pourront y être stockés.

La raison pour laquelle il y a cette phase de stockage dans un tampon est d'éviter que les mouvements ne rattrapent le calcul. En effet, si le tampon est vide, il n'y a plus de mouvement à exécuter. Les moteurs s'arrêtent !

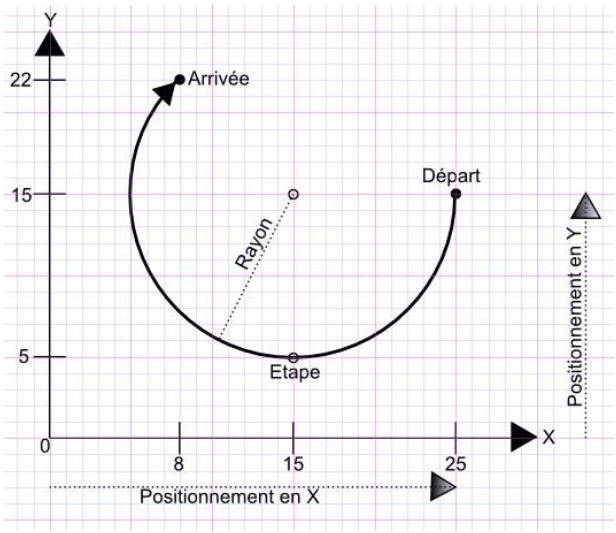
L'inconvénient de cette méthode est qu'il y a un décalage entre la ligne de code qui s'exécute (**CIRA** par exemple) et les mouvements physiquement accomplis.

Que se passe-t-il si le nombre de segments est si faible que le tampon ne sera jamais complètement rempli ?

Il y a deux conditions qui sont susceptibles de démarrer les mouvements : la première, nous l'avons déjà vue, est que le tampon soit rempli. La seconde est l'exécution de l'instruction ***ENDP***.

C'est ce qui explique l'obligation de terminer un contour par ***ENDP***. D'une part, on assure que le mouvement s'exécutera, et d'autre part, ***ENDP*** attend la fin du contour. C'est la fin du décalage abordé ci-dessus.

Pour illustrer ceci, reprenons l'exemple précédent :



Imaginons que l'on fasse ce mouvement en deux étapes. Entre l'étape 1 et l'étape 2, on doit activer une sortie, la sortie 0. Le code suivant ne fonctionnera probablement pas de manière satisfaisante :

```
DPATH   0 3 #VMAX[X]      ; Def. de l'espace d'interpolation
 RAD    10 1              ; Rayon et sens horaire
 CIRA   X 15 Y  5         ; Etape 1
 ON     #OUT[0]
 CIRA   X  8 Y 22         ; Etape 2
ENDP                      ; Fin du contour -> execution
```

Le problème est que lorsqu'il aura terminé de calculer le premier arc de cercle (du point *Départ* au point *Étape*), le tampon ne sera probablement pas encore rempli. Il va donc activer la sortie 0 alors que les axes seront encore physiquement au point de départ !

Pour que cela fonctionne, il y a deux possibilités : la première est de s'arrêter au point *Étape*, d'activer la sortie 0 puis de redémarrer pour terminer au point *Arrivée*.

```
            DPATH   0 3 #VMAX[X]      ; Def. de l'espace d'interpolation
             RAD    10 1              ; Rayon et sens horaire
            CIRA    X 15 Y  5         ; Etape 1
            ENDP                      ; Arret du mouvement a 15; 5

            ON      #OUT[0]           ; Active la sortie 0

            DPATH   0 3 #VMAX[X]      ; Def. de l'espace d'interpolation
            CIRA    X  8 Y 22         ; Etape 2
            ENDP                      ; Fin du contour -> execution jusqu'a
                                      ; l'arrivee.
```

L'inconvénient de cette méthode est qu'il y a un arrêt. Selon la nature de la matière usinée, cela peut se traduire par une marque indésirable au point d'arrêt (*Étape*).

L'autre méthode est d'activer une tâche simultanée qui surveille la position de l'axe X. Dès que sa position devient inférieure ou égale à 15, on enclenche la sortie 0.

```
            ASIM    ONOUT             ; Activation de la surveillance

            DPATH   0 3 #VMAX[X]      ; Def. de l'espace d'interpolation
             RAD    10 1              ; Rayon et sens horaire
            CIRA    X 15 Y  5         ; Etape 1
            CIRA    X  8 Y 22         ; Etape 2
            ENDP                      ; Fin du contour -> execution

            ...                       ; Bloc d'instructions eventuel ici

            END                       ; Fin du programme

; Procedure ONOUT de surveillance et d'activation OUT[0]:
ONOUT:      CMP     #FPABS[X] 15      ; Position X > 15?
            JG      ONOUT             ; Oui, alors attendre
            ON      #OUT[0]           ; Non, alors activer la sortie 0
            KSIM    #PNB              ; Arreter la tache de surveillance
```

Explications :

- Dans le programme principal, l'instruction ***ASIM*** crée une autre tâche simultanée dont l'exécution débute au label ONOUT, avant d'effectuer la suite des instructions du programme principal (ici le ***DPATH***) : Nous avons alors deux tâches qui s'exécutent en parallèle sur le E700, la principale qui va réaliser du contour, et l'autre qui effectue les instructions de surveillance à partir de ONOUT.
- Au niveau de la procédure ONOUT exécutée en parallèle du programme principal :
- La variable FPABS indexée par un numéro d'axe contient la position instantanée de l'axe, en unités choisies dans la configuration des axes.
- ***CMP*** compare FPABS[X] avec 15, et ***JG*** exécute un saut au label ONOUT si X > 15.
- Sinon, on a donc X ≥ 15, on positionne la sortie OUT[0] à 1.
- Puis l'instruction ***KSIM*** (Kill SIMultaneous task) arrête ou tue la tâche PNB (numéro de la tâche courante), ce qui signifie que la tâche s'arrête ou se tue elle-même, le programme principal continue son exécution normalement.

*CIRR ax1 val1 ax2 val2 [cx cy mode]

L'instruction **CIRR** effectue une interpolation circulaire en mode relatif (référentiels origines G91, G02, ou G03) et génère un arc de cercle entre les axes *ax1* et *ax2*, dont le point de départ est la position courante, et le point d'arrivée a pour coordonnées relatives (*val1*; *val2*).

Le fonctionnement de **CIRR** étant similaire à celui de ***CIRA***, le lecteur est renvoyé à l'instruction ***CIRA*** page ***45*** pour les explications et précautions d'utilisation de cette instruction.

Les différences avec CIRA sont :

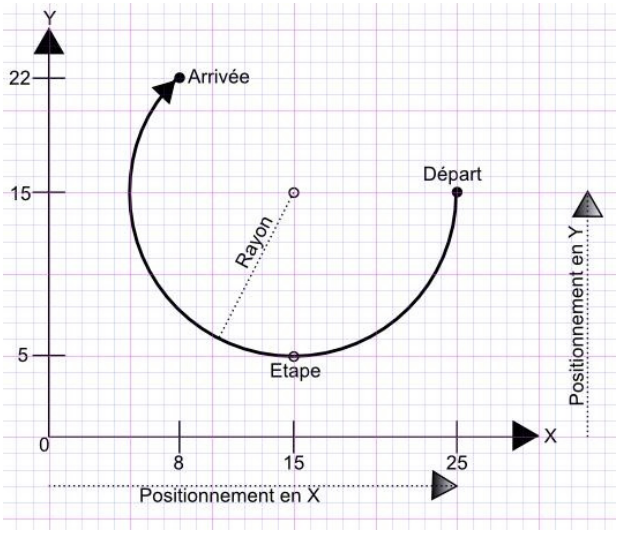
- *val1* : Coordonnée relative à la position actuelle du point d'arrivée en X.
- *val2* : Coordonnée relative à la position actuelle du point d'arrivée en Y.

Notes :

Voir aussi Instructions ***DPATH***, ***ENDP***, ***RAD***, ***CIRA***.

Exemple :

Nous reprenons simplement l'exemple choisi avec ***CIRA*** et nous l'écrivons avec **CIRR** pour montrer les différences.



Comme pour ***CIRA***, nous supposons que l'outil est initialement en (0 , 0).

Utilisation de **CIRR** en définissant le centre :

Point d'arrivée en X : 8 – 25 = -17

Point d'arrivée en Y : 22 – 15 = 7

```
POSA   X #VMAX[X] 25 0     ; Positionnement de X ...
POSA   Y #VMAX[Y] 15 2     ; ... et Y a vitesse rapide

DPATH  0 3 #VMAX[X]        ; Def. espace interpolation X et Y
 CIRR  X -17 Y 7 -10 0 1   ; Arc par le centre
ENDP                       ; Fin du contour -> execution
```

Utilisation de **CIRR** en définissant le rayon :

1ère étape en X : 15 – 25 = -10 2ième étape en X : 8 – 15 = -7

1ère étape en Y ; 5 – 15 = -10 2ième étape en Y : 22 – 5 = 17

```
POSA   X #VMAX[X] 25 0     ; Positionnement de X ...
POSA   Y #VMAX[Y] 15 2     ; ... et Y a vitesse rapide

DPATH  0 3 #VMAX[X]        ; Def. de l'espace d'interpolation
 RAD   10 1                ; Rayon et sens horaire
 CIRR  X -10 Y -10         ; Etape 1
 CIRR  X  -7 Y  17         ; Etape 2
ENDP                       ; Fin du contour -> execution
```

CLR232

L'instruction **CLR232** réinitialise la mémoire de réception des données via la liaison série RS232, ce qui correspond à l'état « aucune données reçues ».

Notes :

Cette instruction s'applique dans le cas où le DIP switch numéro 5 de la carte CPU du E700 est sur ON, cas où la liaison série RS232 est dédiée aux programmes Uniprog (dans ce mode, pas d'utilisation possible de la liaison série entre un ordinateur et le E700 pour les applications EmbosView, E700 File, et Flashage avec FDT).

Voir aussi Variables systèmes RXOPT, TXOPT, Instructions ***RX232***, ***TX232***.

Exemple :

```
;              DECLARATION CONSTANTES ET VARIABLES
             MESSAGE[256] =                  ; Tableau elements RS232
;
;                           INITIALISATIONS
             MOV    #XTOUT 15                ; Timeout RS232 15 secondes
             CLR232                          ; Init reception RS232
;
;                          CORPS DU PROGRAMME
; Recevoir 5 caracteres depuis liaison serie RS232 et les afficher:
             ON     #RXOPT                   ; Mode RS232 longueur message
             RX232  5 MESSAGE                ; Recevoir 5 elements RS232
             DISPN  #XERR 2 0                ; Affiche statut XERR
             CMP    #XERR 2                  ; Reception ok ?
             JNE    SUITE3                   ; Non: ne pas afficher message
SUITE2:      WAIT   0.3                      ; Pause de 0.3 seconde
             ENDRP                           ; Fin de repetition
;
; Recevoir depuis liaison serie RS232 jusqu'au RETURN:
SUITE3:      OFF    #RXOPT                   ; Mode RS232 car. terminal
             RX232  13 MESSAGE               ; Recevoir RS232 jusque RETURN
             DISPN  #XERR 2 0                ; Affiche statut XERR
             CMP    #XERR 2                  ; Reception ok ?
             JNE    SUITE7                   ; Non: ne pas afficher message
;
SUITE7:      WAIT   1                        ; Pause de 1 seconde
             END                             ; Fin du programme
```

Ce programme en Uniprog effectue les opérations suivantes:

- Positionner le timeout pour l'attente des échanges via RS232 à 15 secondes.
- Recevoir 5 caractères depuis la liaison série RS232, puis afficher le satut XERR.
- Recevoir des caracteres depuis la liaison série RS232 jusqu'au caractère 13=RETURN, puis afficher le satut XERR.
- Faire une pause de 1 seconde et terminer le programme.

Le fichier exemple fournit « Exemples Uniprog\RX232\RX232.E7U », permet de visualiser les éléments reçus. Utiliser par exemple l'application « Hyper Terminal » (sous Microsoft Windows, dans Tous Les Programmes / Accessoires / Communications) pour échanger les éléments avec le E700 via la liaison série RS232.

CLRORG sim [espace]

L'instruction **CLRORG** annule l'utilisation de référentiels d'origines pour la tâche simultanée *sim*, et pour l'espace *espace* (ou espace ISODEF par défaut si le paramètre *espace* est omis).

CLRORG est équivalent à la mise en œuvre successive de l'ensemble des commandes ISO : G53, et T-1, et G60<D-1>, et G59.

- *sim* : numéro de la tâche simultanée concernée pour l'annulation de l'ensemble des référentiels d'origines
- [*espace*] : espace d'interpolation concerné pour l'annulation de l'ensemble des référentiels d'origines (valeur de 0 à 4 pour l'un des 5 espaces d'interpolation possible).

  Si espace omis, c'est l'espace ISODEF qui est concerné.

Notes :

En principe, il ne faudrait pas affecter les origines de cette manière. Cette instruction dangereuse ne devrait être utilisée qu'en très bonnes connaissances de cause.

CMP src1 src2

L'instruction **CMP** compare ses deux arguments src1 et src2, en vue d'une instruction de test et saut conditionnel qui suivra **CMP**.

- *src1* : premier nombre à comparer
- src2 : deuxième nombre à comparer

Notes :

Pour avoir un sens, **CMP** doit précéder une instruction de saut conditionnel. Les instructions utilisant le résultat de la comparaison sont :

- ***JE*** *label* : Saut à l'étiquette *label* si **src1 = src2** (Jump if Equal)
- ***JG*** *label* : Saut à l'étiquette *label* si **src1 > src2** (Jump if Greater)
- ***JGE*** label : Saut à l'étiquette label si **src1 ≥ src2** (Jump if Greater or Equal)
- ***JL*** label : Saut à l'étiquette label si **src1 < src2** (Jump if Less)
- ***JLE*** label : Saut à l'étiquette label si **src1 ≤ src2** (Jump if Less or Equal)
- ***JNE*** label : Saut à l'étiquette label si **src1 ≠ src2** (Jump if Not Equal)

CMP est modale, à savoir que tant qu'une autre instruction **CMP** n'est pas exécutée, la valeur du test est conservée dans la tâche simultanée courante, et les instructions de sauts conditionnels qui seraient exécutées par la suite en tiendraient compte.

Voir aussi Instructions ***JE***, ***JG***, ***JGE***, ***JL***, ***JLE***, ***JNE***.

Exemple :

```
BOUCLE:     CMP     #FPABS[X] 15     ; Est-ce que la position de l'axe X
                                     ; est superieure ou egale a 15 ?
            JGE     LABEL            ; Si oui, sauter a LABEL
            JMP     BOUCLE           ; Sinon, attendre

LABEL :     ...                      ; Suite des operations
```

Ce petit exemple implémente une boucle d'attente. On attend dans la boucle tant que la position en X n'a pas atteint la position de 15 mm.

COS dst

L'instruction **COS** retourne le cosinus de son argument *dst*.

- *dst* : valeur modifiée, **dst ← COS (dst)**

  Après l'exécution de **COS**, l'argument *dst* contient une valeur comprise entre –1.0 et 1.0.

Notes :

Les unités de mesure dans lesquelles travaillent les fonctions trigonométriques sont définies dans la variable système # DEG :

Avant l'instruction **COS**, on affectera donc la variable DEG avec l'une des instructions ***ON***, ***OFF*** de la valeur :

- ON pour travailler en degrés (valeur 1, par défaut au démarrage du E700).
- OFF pour travailler en radians (valeur 0).

Si on met DEG à 0, alors l'argument dst doit contenir une valeur d'angle en radians.

Voir aussi Variables systèmes DEG, variables groupe fonctions « Calculs, Cinématique, Interpolation », Instructions ***ATN***, ***SIN***, ***TAN***.

Exemple :

```
PI                  = 3.14159        ; Constante Pi

                                     ; Par defaut, on travaille en degres
                MOV     R0 60        ; Ici R0 vaut 60 degres
                COS     R0           ; Ici R0 vaut Cos(60) = 0.5

                OFF     #DEG         ; On travaille maintenant en radians
                MOV     R0 PI        ; Ici, R0 vaut Pi
                DIV     R0 3         ; Ici, R0 vaut Pi / 3 radians
                COS     R0           ; Ici R0 vaut Cos(Pi/3) = 0.5

                END                  ; Fin du programme
```

CPL dst

L'instruction **CPL** retourne le complément à 1 de son argument *dst*, 1 si *dst* vaut 0, 0 si *dst* est non nul.

- *dst* : valeur modifiée, **dst ← Non (dst)**

  Si *dst* = 0, alors **CPL** retourne la valeur *dst* = 1

  Si *dst* ≠ 0, alors **CPL** retourne la valeur *dst* = 0

Exemple :

Dans notre exemple, on fait clignoter la sortie 0 du E700 avec une fréquence de 1Hz (une demi-seconde entre chaque changement d'état, soit une période d'une seconde) :

```
BOUCLE:     CPL     #OUT[0]          ; Complementer sortie 0
            WAIT    PERIODE          ; Attendre une demi seconde
            JMP     BOUCLE           ; Boucler continuellement
```

CYCLN sim msg

L'instruction **CYCLN** permet de modifier le programme P.ON actif (programme de prise de référence) ou le programme CYCLE actif (programme lancé par action sur le bouton START) pour le simultané *sim*.

L'instruction **CYCLN** permet également de démarrer un programme ISO alors qu'un autre est déjà en exécution. Jusqu'à 5 programmes ISO peuvent ainsi être exécutés simultanément.

- *sim :* sélection du type du programme et du simultané concernés (argument en lecture seule):

  -1 : affecter le programme P.ON

  0 : affecter le programme CYCLE pour la tâche principale (simultané 0)

  n : affecter le programme CYCLE pour la tâche *n*, *n* valant 1, 2, 3 ou 4 (est utilisé dans le cas où plusieurs programmes ISO sont exécutés en parallèle)

- *msg :* numéro de la chaîne de caractères contenant le nom du programme Uniprog choisi.

  Exemple: *msg* = 21, avec m21 = « PIECE12.E7U » déclaré dans le fichier « MSG.INI », désigne le programme Uniprog 'PIECE12.E7U'.

Notes :

CYCLN correspond à la fonction du panneau de commande du E700: touche MEM du menu PAGE : sélection du programme avec les flèches, puis touches F1 pour P.ON ou F2 pour CYCLE.

CYCLN ne permet pas d'adresser les simultanés de 5 à 9.

Si *sim* =0 ou *sim* = -1, alors les actions de l'utilisateur sur les boutons START et STOP déclenchent les programmes P.ON et CYCLE sélectionnés avec l'instruction **CYCLN**.

Pour lancer le programme désiré par programmation:

- programme CYCLE Uniprog ou ISO en simultané 0: **CYCLN** avec sim = 0, puis démarrage avec la variable système #STARTFLG à 1
- programme ISO en simultané 1 à 4: instruction **CYCLN** puis instruction ***ISORUN***
- programme Uniprog en simultané 1 à 4: utiliser directement l'instruction ***ASIM***

Exemple :

```
[Msg]
m24 = "PART14.E7I"
m25 = "PART15.E7I"
```

En Uniprog :

```
; Selection CYCLE suivant valeur de l'entree IN[3]:
            BRIN1  #IN[3] SELECT5          ; IN[3]=1: aller a SELECT5
;
            CYCLN  0 24                    ; CYCLE=m24 en simultane 0
            END                            ; Fin du programme
;
SELECT5:    CYCLN  0 25                    ; CYCLE=m25 en simultane 0
            END                            ; Fin du programme
```

Cet exemple en Uniprog teste l'entrée numérique 3:

- Si IN[3] = 0, alors on continue et on affecte à CYCLE le programme m24 = PART14.E7I (programme ISO), pour le simultané 0
- Si IN[3] = 1, alors on va à l'étiquette SELECT5 et on affecte à CYCLE le programme m25 = PART15.E7I (autre programme ISO), pour le simultané 0
- En appuyant ensuite le bouton START, le programme sélectionné avec **CYCLN** est démarré.

DEC dst

L'instruction **DEC** retranche 1 à la variable passée en argument *dst* (**DEC**rémentation).

- *dst* : valeur modifiée, **dst ← dst - 1**.

Voir aussi Instruction ***INC***.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
              VAL =                           ; Variable VAL
;
;                           INITIALISATIONS
              MOV     R0    8                 ; Init registre R0 a 8
              MOV     #VAL 0.7                ; Init VAL a 0.7
;
;                          CORPS DU PROGRAMME
; Decrementer R0 et VAL:
              DEC     R0                      ; R0 vaut 7
              DEC     #VAL                    ; VAL vaut -0.3
;
              END                             ; Fin du programme
```

Cet exemple en Uniprog utilise le registre R0 et une variable VAL :

- On affecte 8 à R0 et 0.7 à VAL.
- On décrémente R0 qui contient alors 7.
- On décrémente VAL qui contient alors -0.3

DISPC src

L'instruction **DISPC** affiche un caractère à partir de son code ASCII sur l'écran du E700 à la place actuelle du curseur (**DISP**lay **C**haracter). Pour certains codes ASCII de contrôle, **DISPC** effectue des opérations spécifiques.

- src : valeur numérique du code ASCII du caractère à afficher.

  32 à 126 : affiche le caractère ASCII correspondant à la valeur numérique contenue dans *src*. Le curseur est déplacé d'un caractère vers la droite (suit le caractère inséré), sauf dans le cas où la colonne est égale à 41 avant **DISPC**, auquel cas le curseur reste en butée à droite de l'écran (conserve la même position)

  0 : Effacer l'écran courant. Le curseur se trouve ensuite en haut à gauche de l'écran (ligne = 0 et colonne = 0)

  1 : Effacer toute la ligne où est situé le curseur. Le curseur conserve la même position

  2 : Effacer le début de la ligne jusqu'à la gauche du curseur (caractère au niveau du curseur non compris). Le curseur conserve la même position

  3 : Effacer la fin de la ligne depuis le curseur (caractère au niveau du curseur compris). Le curseur conserve la même position

  7 : Émettre un son (Bell). Le curseur conserve la même position

  8 : Effacer le caractère à gauche du curseur (Backspace) et déplacer le texte qui suit le caractère supprimé d'un cran vers la gauche (jusqu'à la fin de la ligne). Après **DISPC** 8, le curseur conserve la même position. Si le curseur est en colonne 0, **DISPC** 8 ne fait aucune opération

  127 : Effacer le caractère au niveau du curseur (Delete) et déplacer le texte qui suit le caractère supprimé d'un cran vers la gauche (jusqu'à la fin de la ligne). Après **DISPC** 127, le curseur conserve la même position

Voir aussi ***DISPS***, ***DISPC***, ***DISPN***, ***DISPST***.

Exemple :

```
            GTXY   18 3                  ; Curseur ligne3 colonne18
            DISPC  79                    ; Afficher O=79 ASCII
            DISPC  75                    ; Afficher K=75 ASCII
            DISPC  7                     ; Emettre un son
;
            END                          ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Positionner le curseur au milieu de l'écran
- Afficher « OK » lettre par lettre à partir du code ASCII de chacune des lettres
- Émettre un son et terminer le programme.

DISPN src [digits] précision

L'instruction **DISPN** affiche un nombre *src* sur l'écran du E700 à la place actuelle du curseur. *précision* définit le nombre de décimales à afficher (**DISP**lay **N**umber). *digits* est optionnel et définit le nombre de caractères désirés pour la partie entière.

- *src* : valeur à afficher.
- *[digits]* : optionnel, donne le nombre de caractères voulus pour la partie entière :

  Omis : correspond à une valeur de 0

  0 : alignement à gauche, affiche autant de chiffres que nécessaire

  1 à 9 : complète le nombre de chiffres de la partie entière par des espaces pour afficher *digits* caractères, et commence par afficher le signe à l'aide d'un espace (si *src* ≥ 0) ou un signe moins (si *src* < 0).
  La taille totale de la partie entière est donc dans ce cas de *digits* + 1 caractères en tenant compte du caractère signe.
  Exemple: *digits* = 3 et *src* =-1, **DISPN** affichera les caractères « ..-1 » (deux espaces suivis de « -1 »)

- *précision* : donne le nombre de caractères désirés pour la partie décimale :

  0 : affiche *src* sous forme d'entier.
  Exemple: avec *src* = 3.1415927 et *précision* = 0, **DISPN** affiche « 3 »
  Autre exemple: avec *src* = 2.7182818 et *précision* = 0, **DISPN** affiche « 3 »

  1 à 6 : affiche *précision* caractères après le point de la partie décimale.
  Exemple: avec *src* = -3.1415927 et *précision* = 3, **DISPN** affiche « -3.142 »

Notes :

Les paramètres *digits* et *précision* doivent être tels que ***digit + précision* ≤ 9**. Dans le cas contraires, des étoiles sont affichées.

Dans le cas de dépassement des limites d'affichage du nombre *src*, ou impossibilité d'afficher *src* dans le format désiré, **DISPN** affiche des étoiles à la place des chiffres.

Voir aussi Instructions ***DISPS***, ***DISPC***, ***DISPST***.

Exemple :

Soient dans le fichier « DISPLAY.INI », les lignes :

```
[Display0]
l0 = "EIP SA - Exemple instruction DISPN:      "
l1 = "                                         "
l2 = "Nombre PI            = #x.xxxxxx#        "
l3 = "PI avec 3 decimales  = #xxxx.xxx#        "
l4 = "Partie entiere de PI = #xxxxxxxx#        "
l5 = "                                         "
```

En Uniprog :

```
;              DECLARATION CONSTANTES ET VARIABLES
             PI =   3.141593                ; Nombre PI
             VAL =                          ; Variable VAL
;
;                          CORPS DU PROGRAMME
             GTXY   24 2                    ; Curseur ligne2 colonne24
             DISPN  PI 6                    ; Afficher PI:
                                            ; format -XXX.XXXXXX
             GTXY   24 3                    ; Curseur ligne3 colonne24
             DISPN  3.141593 3 3            ; Afficher valeur PI:
                                            ; format -XXX.XXX
             GTXY   24 4                    ; Curseur ligne4 colonne24
             MOV    #VAL PI                 ; Definir VAL avec PI
             DISPN  #VAL 7 0                ; Afficher PI:
                                            ; format -XXXXXXX
;
             END                            ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Utiliser une constante PI valant 3.141593 et une variable VAL.
- Afficher PI au format plein avec 6 décimales : format « -XXX.XXXXXX » (alignement à gauche), à partir de la constante.
- Afficher PI avec 3 décimales et 3 chiffres pour la partie entière : format « -XXX.XXX » (alignement à droite), à partir d'une valeur immédiate.
- Afficher la partie entière de PI avec 7 chiffres : format « -XXXXXXX » (alignement à droite), à partir de la variable VAL.

DISPS src

L'instruction **DISPS** affiche l'écran utilisateur *src* sur l'écran du E700 (**DISP**lay **S**creen). L'écran utilisateur est défini auparavant dans le fichier « DISPLAY.INI ».

- *src* : numéro de l'écran à afficher entre 0 et **dpl** – 1 = 9 par défaut (10 écrans utilisateurs, **dpl** est défini dans le fichier « E700.INI »)

Notes :

Après **DISPS**, le curseur est positionné en haut à gauche de l'écran (ligne = 0, colonne = 0)

L'appel à un écran non défini dans « DISPLAY.INI » provoque l'affichage d'une erreur et l'arrêt de l'exécution des programmes en cours sur le E700.

Voir Instructions ***DISPC***, ***DISPN***, ***DISPST***

Exemple :

```
 [Display1]
l0 = "EIP SA – Exemple instruction DISPS:      "
l3 = "             E C R A N   1               "
[Display2]
l0 = "EIP SA – Exemple instruction DISPS:      "
l3 = "             E C R A N   2               "
f3  = "FIN"
```

En Uniprog :

```
TOUCHE =                                ; Variable touche pressee
;
BOUCLE:     DISPS  1                    ; Affichage ecran 1
            WAIT   2                    ; Attendre 2 secondes
            DISPS  2                    ; Affichage ecran 2
            WAIT   2                    ; Attendre 2 secondes
;
            CMP    #CURKEY KF3          ; Touche F3 pressee?
            JNE    BOUCLE               ; Boucler si non F3
;
            END                         ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Déclaration variable contenant le code de la touche pressée.
- Effectuer une boucle tant que la touche F3 = FIN n'est pas pressée:
  - Afficher l'écran utilisateur 1 pendant 2 secondes.
  - Afficher l'écran utilisateur 2 pendant 2 secondes.

DISPST msg

L'instruction **DISPST** affiche la chaîne de caractères numéro *msg* sur l'écran du E700 à la place actuelle du curseur (**DISP**lay **ST**ring).

- *msg* : numéro du message utilisateur à afficher (entre 0 et 119, 120 messages possibles par défaut).

  La chaîne de caractères « m<*msg*> » aura été définie dans le fichier « MSG.INI » (exemple: *msg* = 10, définition « m10 = "Texte exemple" » dans « MSG.INI »).

Notes :

Après l'instruction **DISPST**, le curseur est positionné au caractère suivant le dernier caractère écrit.

L'affichage d'un caractère ou d'un espace efface le caractère précédemment affiché à la place courante du curseur.

Si la définition d'une variable utilisateur est omise et que le m<*msg*> n'existe pas, il est affiché la chaîne de caractères « Illegal msg » (exemple: si *msg* = 3 et m3 non défini dans « MSG.INI », alors affichage des caractères « Illegal msg » à la position actuelle du curseur)

Voir aussi Instructions ***DISPS***, ***DISPC***, ***DISPN***

Exemple :

```
[Msg]
m10 = "L'entree numerique IN[3] vaut 0"
m11 = "L'entree numerique IN[3] vaut 1"
```

En Uniprog :

```
            GTXY   18 3                    ; Curseur ligne3 colonne10
;
            BRIN1  #IN[3] VAL1             ; Lire entree IN[3]
VAL0:       DISPST 10                      ; IN[3]=0: Affichage msg10
            END                            ; Fin du programme
VAL1:       DISPST 11                      ; IN[3]=1: Affichage msg11
            END                            ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Positionner le curseur au milieu de l'écran.
- Lire l'entée numérique IN[3] et en fonction de sa valeur :
- Si IN[3] = 0, afficher le message m10 avant de terminer le programme.
- Si IN[3] = 1, afficher le message m11 avant de terminer le programme.

DIV dst src

L'instruction **DIV** divise *dst* par *src* et met le résultat dans *dst* (**DIV**ision).

- *dst* : dividende, valeur modifiée, **dst** ← **dst** / **src**.
- *src* : diviseur.

  Si *src* est nul (*src* = 0), **DIV** produit l'erreur 19 dans le cas d'une division par zéro, avec arrêt des programmes du E700, et sur appui de la touche EXIT comme confirmation de la lecture du message d'erreur par l'utilisateur, relance de la tâche AUTOMAT.

Voir aussi Instructions ***MUL***, ***INV***.

Exemple :

Soient dans le fichier « MSG.INI », les lignes :

```
[Msg]
m11 = "Le cercle correspondant\na pour rayon #VAL"
```

En Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
             VAL =                          ; Perimetre, puis Rayon
;
;                          CORPS DU PROGRAMME
             INP1    #VAL 22 4 5 6 0 6000000 0; Saisie nombre
             DIV     #VAL 2                 ; VAL <- VAL / 2
             DIV     #VAL 3.1415927         ; VAL <- VAL / Pi
RESULT:      MSG     11                     ; Afficher resultat
             END                            ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Saisir un nombre à partir du clavier dans la variable VAL.
- Divise ensuite VAL par 2 puis par le nombre PI (RayonCercle = PérimètreCercle ÷ 2 ÷ Pi).
- Affiche le résultat avec le message m11

DIVD src

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **DIVD** divise l'accumulateur de la tâche courant*e accum[PNB]* par *src* et met le résultat dans l'accumulateur de la tâche courante (**DIV**ision **D**irecte).

- *src* : diviseur, **accum[PNB] ← accum[PNB] ÷ src**.

  Si *src* est nul, **DIVD** produit l'erreur d'exécution 19 (runtime error) dans le cas d'une division par zéro, avec arrêt des programmes du E700, et sur appui de la touche EXIT, comme confirmation de la lecture du message d'erreur par l'utilisateur, relance de la tâche AUTOMAT.

- *accum[PNB]* : représente la mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, non utilisable directement en Uniprog).

  Est modifié par l'instruction pour contenir le résultat.

Exemple:

```
;               DECLARATION CONSTANTES ET VARIABLES
              VAL =                          ; Variable
;
;                          CORPS DU PROGRAMME
              MOV    #VAL 30                 ; VAL <- 30
              LOAD   #VAL                    ; Ici accum[PNB] = 30
              DIVD   2                       ; Ici accum[PNB] = 30/2 = 15
              MOV    #VAL 5                  ; VAL <- 5
              DIVD   #VAL                    ; Ici accum[PNB] = 15/5 = 3
              STORE  #VAL                    ; VAL<-accum[PNB] (ici VAL=3)
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Initialiser la variable VAL avec 30
- Effectuer les opérations VAL <- ((VAL / 2) / 5) en utilisant l'accumulateur.

DPATH groupe espace vitesse

L'instruction **DPATH** définit un groupe d'interpolation entre plusieurs axes physiques, en vue de la réalisation d'un contour à une vitesse de consigne donnée, le contour étant un chemin dans l'hyperplan constitué des axes du groupe (**D**efine **PATH**).

- *groupe* : numéro du groupe choisi pour l'interpolation (de 0 à 4, soit 5 interpolations simultanées possibles). Dans le cas général qui est celui d'une seule interpolation à la fois, mettre *groupe* à 0.
- *espace :* définit les axes pour ce groupe d'interpolation.

  espace est constitué d'un code binaire sur 16 bits où le $i^{ème}$ bit « b_i » (i de 0 à 15, 16 axes physiques) est tel que :

  « b_i » vaut 1 si l'axe physique « i » est concerné,

  « b_i » vaut 0 si l'axe « i » n'est pas concerné :

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1

  Exemples :

  Si les axes concernés sont les axes 0, 1, 2 et 3, *espace* vaudra

  0000 0000 0000 1111 = 15

  Si les axes concernés sont les axes 0, 1, 2, 3, 7, 10 et 12, *espace* vaudra

  0001 0100 1000 1111 = 5263

  Si les axes concernés sont les axes 0 et 1, *espace* vaudra

  0000 0000 0000 0011 = 3

- *vitesse* : consigne de vitesse le long du contour en unités de vitesse (unité des axes de *espace*).

  Si les axes du groupe d'interpolation n'ont pas les mêmes unités de vitesse, l'unité de cet argument n'est pas significative.

Notes :

Les instructions **DPATH** et ***ENDP*** doivent être utilisées conjointement.

Définition d'un contour :

- commencer par **DPATH**
- puis des instructions de positionnement et de mouvement définissent le profil du contour (instructions ***LINA***, ***LINA2***, ***LINR***, ***LINR2***, ***RAD***, ***CIRA***, ***CIRR***, …)
- l'instruction ***ENDP*** finalise la définition du contour.

La réalisation du contour se termine à l'issue de l'instruction ***ENDP***.

Spécifier le groupe est nécessaire car il est possible d'exécuter jusqu'à cinq interpolations simultanément. Dans la plupart des cas, une seule interpolation est utilisée, le groupe 0. S'il y a plusieurs interpolations simultanées, on peut alors les mettre dans différents groupes de 0 à 4.

Voici une petite procédure qui calcule le paramètre *groupe* en fonction des axes utilisés :

```
SPACE   =                          ; Variable qui contiendra l'espace
;
CALCSP:     PUSH    R0             ; Debut de procedure CALCSP

            MOV     #SPACE 1       ; Ici SPACE = 1
            SHL     #SPACE X       ; Donner ici le 1er axe (ici X=0)

            MOV     R0 1
            SHL     R0 Y           ; Donner ici le 2eme axe (ici Y=1)
            OR      #SPACE R0      ; Ici SPACE = 1 OR 2 = 3

            MOV     R0 1
            SHL     R0 A           ; Donner ici le 3eme axe (ici A=3)
            OR      #SPACE R0      ; Ici SPACE = 3 OR 8 = 11

            MOV     R0 1
            SHL     R0 C           ; Donner ici le 4eme axe (ici C=5)
            OR      #SPACE R0      ; Ici SPACE = 11 OR 32 = 43
                                   ; avec bits (0, 1, 3, 5) a 1

; etc pour chaque axe ...
            POP     R0
            END                    ; Fin de procedure CALCSP
```

Voir aussi **Point « Important »** après l'exemple de l'instruction ***CIRA*** page ***45***.

Instructions ***ENDP***, ***LINA***, ***LINA2***, ***LINR***, ***LINR2***, ***RAD***, ***CIRA***, ***CIRR***.

Exemple :

```
; Decrire une helice en relatif a la position actuelle:
            DPATH  0 7 1                ; Interpolation espace 0
                                        ; axes 0=X 1=Y 2=Z a 1 m/min
              LINR   Z -10 0            ; Droite axe Z vers Z-10
              CIRR   X 10  Y 10  10 0 0 ; Cercle vers X+10 Y+10
                                        ; centre X+10 Y+0
                                        ; rotation anti-horaire
            ENDP                        ; Terminer le contour
;
            END                         ; Fin du programme
```

Cet exemple en Uniprog effectue une hélice :

- Définit une interpolation entre les axes X, Y et Z:
  - Définit une droite en Z vers les coordonnées Z-10
  - Définit un cercle en X,Y en relatif vers les coordonnées (X+10,Y+10)
- Finaliser le contour.

END

L'instruction **END** termine un programme ou un sous-programme (un sous-programme est aussi appelé fonction ou procédure) :

Dans le cas d'un programme (tâche simultanée de 0 à 9), le **END** termine le programme et le E700 arrête l'activité du programme.

dans le cas d'un sous-programme, la fonction ou procédure qui contient le **END** a été appelée par une instruction d'appel à une fonction ou procédure identifiée par son « Label » (instructions ***CALL***, ***CALIN0***, ***CALIN1***) : l'instruction **END** alors rencontrée dans le sous-programme provoque le retour à l'instruction suivant le ***CALL***.

Voir aussi instructions ***CALL***, ***CALIN0***, ***CALIN1***.

Exemple :

Soient dans le fichier « MSG.INI », les lignes :

```
[Msg]
m11 = "Avant appel fonction"
m12 = "Execution fonction"
m13 = "Apres appel fonction"
```

En Uniprog :

```
;                          CORPS DU PROGRAMME
             MSG    11                        ; Afficher message msg11
             CALL   FONCTION                  ; Appeler FONCTION
             MSG    13                        ; Afficher message msg13
;
             END                              ; Fin du programme
;
;                               PROCEDURES
FONCTION:    MSG    12                        ; Afficher msg12
             END                              ; Fin procedure
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Afficher msg11 = « Avant appel fonction »
- Appeler le sous-programme identifié par le label « FONCTION: » :
  - Afficher msg12 = « Execution fonction »
  - L'instruction **END** termine FONCTION et permet de revenir après le « ***CALL*** FONCTION »
- Afficher msg13 = « Apres appel fonction »
- L'instruction **END** termine le programme.

ENDP

L'instruction **ENDP** termine une interpolation débutée par l'instruction ***DPATH***, et finalise le contour défini par les instructions de mouvements situées entre les instructions ***DPATH*** et **ENDP** (**END P**ath).

Notes :

En fonction des consignes de contour demandées entre les instructions ***DPATH*** et **ENDP**, et en particulier si le contour est de petite longueur, le contour peut ne démarrer qu'à l'exécution de l'instruction **ENDP** : mais dans tous les cas, le contour est terminé à l'issue de l'instruction **ENDP**.

Voir Instructions ***DPATH***, ***LINA***, ***LINA2***, ***LINR***, ***LINR2***, ***RAD***, ***CIRA***, ***CIRR***.

Exemple :

```
            POSA   X #VMAX[X]  0 0          ; Aller en X=+00 attente
            POSA   Y #VMAX[Y]  0 2          ; Aller en Y=+00 go
;
            CALL   RECORDON                 ; Memoriser chemin contour
            DPATH  0 3 1                    ; Interpolation espace 0
                                            ; axes 0=X 1=Y a 1 m/min
             LINA2 X 20 Y 0                 ; Droite vers X=+20 Y=+0
             LINA2 X 50.500 Y 8.458         ; Droite vers X=50.500 Y=8.458
             RAD   13 0                     ; Rayon 13 sens anti-horaire
             CIRA  X 60 Y 21                ; Cercle vers X=+60 Y=+21
             LINA2 X 60 Y 25                ; Droite vers X=+60 Y=+25
             CIRA  X 39.900 Y 35.903        ; Cercle vers X=39.9 Y=35.903
             LINA2 X 0  Y 10                ; Droite vers X=+00 Y=+10
             LINA2 X 0  Y 0                 ; Droite vers X=+00 Y=+00
            ENDP                            ; Terminer le contour
;
            END                             ; Fin du programme
```

Cet exemple en Uniprog effectue un contour particulier (identique à l'exemple de l'instruction ***LINA***) :

- Définit une interpolation entre les axes X et Y:
  - Effectue un contour avec droites et arcs de cercles
- Finaliser le contour.

L'exemple « ENDP.E7U » livré avec la documentation (dossiers « Exemples Uniprog/Fichiers E700 communs aux exemples » et « Exemples Uniprog\ENDP »), permet de réaliser l'exemple sur un E700 et de visualiser le résultat du contour sur l'écran du E700.

ENDRP

L'instruction **ENDRP** termine une boucle de répétition définie par l'instruction ***REP*** (**END R**e**P**eat).

Notes :

Se rapporter aux explications données pour l'instruction ***REP*** page ***141***.

Voir aussi Instruction ***REP***.

Exemple :

Remettre à 0 les sorties 0 à 7 :

```
REP    8                    ; Boucle 8 fois, i=R9 de 0 a 7
 OFF   #OUT[R9]             ; Sortie
ENDRP                       ; Fin boucle REP
```

ENDS

L'instruction **ENDS** termine un bloc d'instructions initié par l'instruction ***SWITCH*** (choix suivant la valeur) et contenant une ou plusieurs instructions ***CASE*** (**END S**witch).

Notes :

Se rapporter à l'instruction ***SWITCH*** page ***167*** pour le détail sur cette instruction.

Voir aussi Instructions ***SWITCH***, ***CASE***.

Exemple :

Se rapporter à l'exemple de l'instruction ***SWITCH*** page ***167***.

ERROR msg mode

L'instruction **ERROR** affiche à l'écran du E700 la chaîne de caractères numéro *msg*, avec demande de confirmation de la part de l'utilisateur.

- *msg* : De 0à 119, numéro du message utilisateur à afficher.

  La chaîne de caractères « m<*msg*> » aura été définie dans le fichier « MSG.INI » (exemple: *msg* = 10, définition « m10 = "Texte exemple" » dans « MSG.INI »).

- *mode* : définit le mode arrêt définitif des programmes ou possibilité de continuer l'exécution des programmes.

  0 : alors tous les programmes Uniprog et ISO s'arrêtent ainsi que les mouvements (avec décélération). Le message d'erreur s'affiche et l'utilisateur doit le quittancer avec une seule possibilité :

  Presser la touche F1 = EXIT. Il n'y a pas la possibilité de continuer le programme après avoir quittancé, l'utilisateur est obligé de recommencer le programme au début.

  1 : alors tous les programmes Uniprog et ISO s'arrêtent ainsi que les mouvements (avec décélération). Le message d'erreur s'affiche et l'utilisateur doit le quittancer avec deux possibilités :

  Presser la touche F1 = CONT pour continuer l'exécution là où elle s'était arrêtée.

  Presser la touche F2 = STOP pour arrêter l'exécution. Il n'y a pas la possibilité de continuer le programme après avoir quittancé, l'utilisateur est obligé de recommencer le programme au début.

Notes :

Cette instruction est utilisée en cas de problème, comme un outil cassé par exemple.

Avec *mode* = 0, la tâche AUTOMAT est arrêtée lors de l'instruction **ERROR**, puis relancée depuis le début après l'instruction **ERROR**.

Si la définition d'une variable utilisateur est omise et que le m<*msg*> n'existe pas, il est affiché le message d'erreur R17 « Msg inexistant » (exemple: si *msg* = 3 et m3 non défini dans « MSG.INI », alors affichage du message d'erreur R17 « Msg inexistant »).

Voir aussi Instruction ***MSG***.

Exemple :

Soient dans le fichier « MSG.INI », les lignes :

```
[Msg]
m10 = "L'entree numerique  IN[3] vaut 0     "
m11 = "Attention: l'entree IN[3] vaut #IN[3]"
m12 = "ARRET:     l'entree IN[3] vaut #IN[3]"
```

En Uniprog :

```
            BRIN1  #IN[3] VAL1          ; Lire entrée IN[3]
VAL0:       GTXY   0 3                  ; Curseur ligne3 colonne0
            DISPST 10                   ; IN[3]=0: Affichage msg10
            WAIT   3                    ; Attendre 2 secondes
            END                         ; Fin du programme
VAL1:       ERROR  11 1                 ; IN[3]=1: Erreur msg11
                                        ; avec arret programmes,
                                        ; ce qui suit est execute
            DISPC  7                    ; Emettre un beep
            ERROR  12 0                 ; IN[3]=1: Erreur msg12
                                        ; avec arret programmes,
                                        ; le beep n'est pas fait:
            DISPC  7                    ; Emettre un beep
            END                         ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Test de l'entrée numérique IN[3] :
  - Si IN[3]=0:
    - afficher « IN[3]=0 » pendant 3 secondes
    - terminer le programme
  - Si IN[3]=1:
    - afficher une erreur avec possibilité de continuer
      - Si l'utilisateur choisit F2=STOP:
        - arrêt du programme, ce qui suit n'est pas effectué
      - Si l'utilisateur choisit F1=CONT:
        - émettre un beep
        - afficher message d'erreur avec quittance F1=EXIT, arrêt des programmes.
        - le beep qui suit n'est pas exécuté.

G5358 src [sim]

L'instruction **G5358** met en œuvre le système de coordonnées de pièces G53 à G58 indiqué dans *src* sur l'espace d'interpolation *sim* (workpiece coordinate system). Si *sim* est omis, mise en œuvre sur l'espace d'interpolation 0.

- *src* : numéro du référentiel à utiliser :

  53 : annule l'utilisation du précédent correcteur G54 à G58

  54 à 58 : annule l'utilisation du précédent correcteur G54 à G58 et applique le correcteur G*src* (G54, G55, G56, G57, ou G58 suivant *src*).

- *[sim]* : numéro de l'espace désiré (paramètre optionnel).

  0 : valeur par défaut si paramètre *sim* omis, utilisation de l'espace d'interpolation 0 (cf. instructions ***ISODEF***, ***DPATH***).

  1 à 4 : utilisation de l'espace d'interpolation *sim* (cf. instructions ***ISODEF***, ***DPATH***).

Voir aussi Variable système LASTG54

Exemple :

```
              G5358  54                    ; Appliquer G54
              POSA   X #VMAX[X] 0 0        ; Aller en X=0, preparer
              POSA   Y #VMAX[Y] 0 2        ; Aller en Y=0, executer
              DPATH  0 7 1                 ; Interpolation espace 0
                                           ; axes 0=X 1=Y 2=Z a 1 m/min
                LINR   Z -30 0             ; Droite axe Z vers Z-30
                CIRA   X 0  Y 0  10 0 1    ; Cercle vers X=0 Y=0
                                           ; centre X=10 Y=0
                                           ; rotation horaire
              ENDP                         ; Terminer le contour
              G5358  53                    ; Annuler G54 a G58
;
              END                          ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Utiliser le correcteur G54.
- Aller en (X,Y) = (0,0) à vitesse maximum.
- Définit une interpolation en X, Y et Z:
  - Définit une droite en Z vers Z-10.
  - Définit un cercle en X,Y vers (X+10,Y+10).
- Finalise le contour.
- Annule l'utilisation du référentiel G54.

Dans les dossiers « Exemples Uniprog/Fichiers E700 communs aux exemples » et « Exemples Uniprog/G5358 », l'exemple permet de visualiser les déplacements puis le contour sur le E700. Pour NB : définir les coordonnées du référentiel G54 avant de lancer le programme « G5358.E7U » (menu **TRACE** / **MDI** / G54 **EXEC**, puis menu **TOOLPOS** fonction **TEACH**, …).

*GETKF dst

L'instruction **GETKF** attend que l'utilisateur ait pressé une des touches actives du menu utilisateur F1 à F6, et renvoie le numéro de la touche pressée dans *dst* (**GET K**ey **F***i*).

- *dst* : Chiffre de 1 à 6, à l'issue de l'instruction **GETKF**, *dst* contient le numéro de la touche de fonction active pressée F<*dst*>, de *dst* = 1 pour F1 à *dst* = 6 pour F6.

  Les touches de fonctions sont actives si elles ont un libellé associé :

  - Le libellé a été défini dans le Display actif sous la forme (fichier « DISPLAY.INI », *i* de 1 à 6, *nom* jusqu'à 6 caractères):

```
f<i> = "nom"
```

  - Et si la touche de fonction définie dans le Display actif n'a pas été désactivée par l'instruction ***MENU***.
  - Ou si la touche de fonction a été activée avec l'instruction ***MENU***.

  L'action sur une touche de fonction non active est sans effet.

Notes :

Pendant l'attente de la touche de fonction, le voyant AUTO reste allumé, les autres tâches simultanées continuent à tourner.

Lorsque l'utilisateur maintient la pression sur la touche F1 à F6 au-delà d'une seconde environ, les programmes s'interrompent, le voyant AUTO s'éteint, jusqu'à ce que l'utilisateur ait libéré cette touche de fonction.

Exemple :

```
 [Display1]
l0 = "EIP SA - Exemple instruction GETKF:      "
l3 = "            E C R A N   1                "
f1 = "ECRAN1"
f2 = "ECRAN2"
f6 = "FIN"
[Display2]
l0 = "EIP SA - Exemple instruction GETKF:      "
l3 = "            E C R A N   2                "
f1 = "ECRAN1"
f3 = "ECRAN3"
f6 = "FIN"
[Display3]
l0 = "EIP SA - Exemple instruction GETKF:      "
l3 = "            E C R A N   3                "
f1 = "ECRAN1"
f4 = "BEEP"
f6 = "FIN"
```

En Uniprog :

```
;              DECLARATION CONSTANTES ET VARIABLES
             TOUCHE =                        ; Code touche clavier E700
;
;                          INITIALISATIONS
             CALL   ECRAN1                   ; Afficher ecran 1
;
;                         CORPS DU PROGRAMME
; Attendre appui touches F1 a F6:
ATTENTEFI:  GETKF  #TOUCHE                   ; Attendre touche F1 a F6
             SWITCH #TOUCHE                  ; Test la touche pressee
               CASE   1  ECRAN1 FINSWI       ; F1: Afficher ecran 1
               CASE   2  ECRAN2 FINSWI       ; F2: Afficher ecran 2
               CASE   3  ECRAN3 FINSWI       ; F3: Afficher ecran 3
               CASE   6  RIEN   FINSWI       ; F6: Terminer programme
               DISPC  7                      ; F4 ou F5: Emettre un son
FINSWI:      ENDS
;
             CMP    #TOUCHE 6                ; F6 pressee ?
             JNE    ATTENTEFI                ; Non: Boucle attendre touche
                                             ; Oui: Terminer programme
;
FINFI:       END                             ; Fin du programme
;
;                              PROCEDURES
;
ECRAN1:      DISPS  1                        ; Affichage ecran 1
             END                             ; Fin procedure
;
ECRAN2:      DISPS  2                        ; Affichage ecran 2
             END                             ; Fin procedure
;
ECRAN3:      DISPS  3                        ; Affichage ecran 3
             END                             ; Fin procedure
;
RIEN:        END                             ; Fin procedure
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Afficher l'écran 1 avec menus F1 = ECRAN1, F2 = ECRAN2, F6 = FIN.
- Attendre une touche de fonction F1 à F6 (seules celles définies dans le menu utilisateur sont visibles et actives):
  - Dans le cas où F1 est pressée, afficher l'écran utilisateur 1.
  - Dans le cas où F2 est pressée et que F2 est définie dans le menu utilisateur actuel, afficher l'écran utilisateur 2.
  - Dans le cas où F3 est pressée et que F3 est définie dans le menu utilisateur actuel, afficher l'écran utilisateur 3.
  - Dans le cas où F6 = FIN est pressée, terminer le programme.
  - Dans les autres cas (F4 ou F5), on émet un son : ce cas n'est actif que pour le menu 3 qui contient un F4 = "BEEP".
- On reprend la boucle d'attente d'une touche de fonction avec l'instruction **GETKF**.

GTXY colonne ligne

L'instruction **GTXY** positionne le curseur de l'écran du E700 à la ligne *ligne* et à la colonne *colonne* (**G**o**T**o coordonnées (**X**,**Y**)).

- *colonne* De 0 à 41, ordonnée de la position désirée du curseur (numéro de colonne).
- *ligne* De 0 à 7, abscisse de la position désirée du curseur (numéro de ligne).

Voir aussi Instructions ***DISPS***, ***DISPC***, ***DISPN***, ***DISPST***, ***MSG***, ***ERROR***.

Exemple :

```
[Msg]
m10 = "Etat IN[3] = xx (0=ouvert, 1=ferme)"
```

En Uniprog :

```
            GTXY   3  2                  ; Curseur ligne2 colonne3
            DISPST 10                    ; Afficher m10
            GTXY   16 2                  ; Curseur ligne2 colonne18
            DISPN  #IN[3] 1 0            ; Afficher IN[3]
                                         ; 1 entier 0 decimales
                                         ; affiche 2 caracteres avec
                                         ; le caractere de signe
            WAIT   5                     ; Attendre 5 secondes
;
            END                          ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Afficher l'écran 1
- Afficher la chaine m10
- Écrire l'état de l'entrée IN[3] à la bonne place dans m10, c'est-à-dire au niveau des deux caractères « xx » après le signe « = ». Le ***DISPN*** écrit deux caractères, le signe (écrit un espace si le nombre à afficher est positif), puis un caractère pour la valeur.

ICNT

Cette instruction n'est pas encore implémentée

L'instruction **ICNT** est prévue pour effectuer l'incrément du compteur général.

INC dst

L'instruction **INC** ajoute 1 à la variable passée en argument *dst.*

- *dst* : valeur modifiée, **dst ← dst + 1**.

Voir aussi Instruction ***DEC***.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
              VAL =                          ; Variable VAL
;
;                           INITIALISATIONS
              MOV    R0   8                  ; Init registre R0 a 8
              MOV    #VAL -0.7               ; Init VAL a -0.7
;
;                          CORPS DU PROGRAMME
; Incrementer R0 et VAL:
              INC    R0                      ; R0 vaut 9
              INC    #VAL                    ; VAL vaut 0.3
;
              END                            ; Fin du programme
```

Cet exemple en Uniprog utilise le registre R0 et une variable VAL :

- On affecte 8 à R0 et -0.7 à VAL.
- On incrémente R0 qui contient alors 9.
- On incrémente VAL qui contient alors 0.3

INP val inf sup précision

L'instruction **INP** permet à l'utilisateur de saisir un nombre à partir du clavier du E700, la valeur est retournée dans *val*.

Important : l'instruction **INP** interrompt la tâche AUTOMAT, ce qui peut s'avérer très dangereux pour la surveillance et la gestion de l'équipement du E700.
Il est donc vivement conseillé de ne pas employer **INP**, mais plutôt ***INP1*** qui au contraire permet de saisir une valeur sans interrompre ni la tâche AUTOMAT ni les tâches en cours.
Cette instruction n'existe que par soucis de compatibilité avec les premières versions de E700

- *val* contient la valeur saisie par l'utilisateur.
- *inf* valeur minimale que l'utilisateur peut entrer.
- *sup* valeur maximale que l'utilisateur peut entrer.
- *précision* 0 si *val* doit être un entier

  1 à 6 pour spécifier le nombre de chiffres à droite du point décimal.

Notes :

Pendant la saisie avec l'instruction **INP**, la tâche AUTOMAT est interrompue.

Voir aussi Instruction ***INP1***.

*INP1 [dst posX posY fl fr min max mode]

L'instruction **INP1** permet à l'utilisateur de saisir un nombre à partir du clavier du E700, la valeur est retournée dans *dst*.

INP1 permet de saisir une valeur sans interrompre ni la tâche AUTOMAT, ni les différentes tâches parallèles en cours.

Sans paramètres, l'instruction **INP1** interrompt les instructions **INP1** éventuellement en cours dans d'autres tâches simultanées.

- *dst* contient la valeur saisie par l'utilisateur.
- *posX* De 0 à 41, ordonnée de la position désirée du curseur (numéro de colonne).
- *posY* De 0 à 7, abscisse de la position désirée du curseur (numéro de ligne).
- *fl* De 0 à 9 (fl pour left), nombre de chiffres à gauche de la virgule pour la saisie, incluant le caractère de signe « - » (exemple : saisir « -999 » ou « 999 » nécessitent fl = 4).
- *fr* De 0 à 8 (fr pour right), nombre de chiffres à droite de la virgule pour la saisie, hormis le caractère séparateur décimale « . » (exemple : saisir « .9999 » nécessite fr = 4).

  Exemples : « ±99.9999 » nécessite fl = 3 et fr = 4 ; « .999 » nécessite fl = 0 et fr = 3.
- *min* De -16 777 216 à 16 777 215, valeur minimale que l'utilisateur peut entrer.
- *max* De -16 777 216 à 16 777 215, valeur maximale que l'utilisateur peut entrer.
- *mode* contient le mode de texte et de fond pour la saisie du nombre :

  0 : mode normal (texte blanc sur fond noir).

  1 : mode inverse ou « STABILO » (texte noir sur fond blanc).
- *INPCURH* De 1 à 8, à éventuellement modifier avant l'instruction **INP1**, cette variable système permet de définir la hauteur du curseur de saisie, sa valeur par défaut est 4.
- *INPFMASK* De 0 à 255, à éventuellement modifier avant l'instruction **INP1**, cette variable système permet de définir les touches de fonctions F1 à F8 qui seront actives avec l'instruction **INP1**, en positionnant le bit b<i> de cette variable, i De 0 à 7 :
  - à 1 pour activer la touche F<i+1>.
  - à 0 pour ne pas tenir compte de la touche de fonction F<i+1>.

  Exemple : *INPFMASK* = 0x81 = 129 (bit 0 et bit 7 à 1) inhiberait toutes les touches de fonctions sauf F1 et F8.

  Sa valeur par défaut est 255 (F1 à F8 actives).
- *INPEXIT* à éventuellement lire après l'instruction **INP1**, cette variable système contient le code de la touche ayant terminé l'instruction **INP1** :

  KESC pour la touche ESCAPE, ou lorsqu'une autre tâche a effectué une instruction **INP1** sans paramètres (ce qui a eu pour effet de terminer l'instruction **INP1**).

  KUPA, KDNA, KLFA, KRTA respectivement les flèches haut, bas, gauche, droite.

  KSTOP, KPAUSE, KENTER respectivement les touches STOP, PAUSE, ENTER.

  KF1 à KF8 les touches de fonction, suivant le contenu de *INPFMASK*.

Notes :

fl et *fr* sont tels que *fl* + *fr* est inférieur ou égal à 9.

Si l'instruction **INP1** est interrompue par une instruction **INP1** sans paramètres effectuée depuis une autre tâche simultanée, alors tout se passe comme si l'utilisateur avait pressé la touche ESCAPE, et

la variable système *INPEXIT* vaut alors KESC (code de la touche ESCAPE) : la variable associée au paramètre *dst* est alors inchangée.

La valeur résultat *dst* est un nombre avec une précision effective de *fr* décimales, ce qui est différent de la précision d'affichage des valeurs (paramètre du menu accessible avec les touches MENU / F5=CONFIG / F6=OTHER / F2=USER, ligne 2 « Precision valeur reelle »).

Voir Variables systèmes INPCURH, INPEXIT, INPFMASK

Exemple :

Fichier « DISPLAY.INI » :

```
[Display0]
l2 = "Entier entre -10 000 000 et 10 000 000: "
l4 = "Reel entre -99 et 99 (5 decimales):     "
```

Fichier « MSG.INI » :

```
[Msg]
m10 = "La valeur saisie\nest egale a #N"
```

En Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
            N =                             ; Variable nombre
;
;                           CORPS DU PROGRAMME
;
            INP1    #N 5 3 8 0 -10000000 10000000 0 ; Saisie normale
                                            ; en (3,5) format -XXXXXXXX
                                            ; de -10 000 000 a 10 000 000
            MSG     10                      ; Afficher resultat m10
;
            INP1    #N 5 5 3 5 -99 99 1     ; Saisie video inverse
                                            ; en (5,5) format -XXX.XXXXX
                                            ; de -99 a 99
            MSG     10                      ; Afficher resultat m10
;
            END                             ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- L'écran utilisateur Display0 est affiché par défaut à la mise sous tension du E700.
- Déclarer une variable N.
- Demander une valeur entière à l'utilisateur.
- Afficher la valeur saisie.
- Demander une valeur décimale à l'utilisateur en vidéo inverse.
- Afficher la valeur saisie :
  - La valeur n'est pas affichée avec le nombre de décimales données lors de la saisie, mais avec le nombre de décimales défini dans la configuration du E700 et spécifié avec le menu accessible avec les touches MENU / F5=CONFIG / F6=OTHER / F2=USER (ligne 2 « Precision valeur reelle »).
- Terminer le programme.

INV dst

L'instruction **INV** inverse la valeur de la variable passée en argument *dst*

- *dst* : valeur modifiée, **dst** ← **1 ÷ dst**.

Notes :

L'erreur 14 est générée si *dst* vaut 0 (division par zéro), tous les programmes sont arrêtés, la tâche AUTOMAT est relancée après acquittement du message d'erreur par l'utilisateur.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
              VAL =                          ; Variable VAL
;
;                           INITIALISATIONS
              MOV    R0   8                  ; Init registre R0 a 8
              MOV    #VAL -0.05              ; Init VAL a -0.05
;
;                          CORPS DU PROGRAMME
; Inverser R0 et VAL:
              INV    R0                      ; R0 vaut 0.125
              INV    #VAL                    ; VAL vaut -20
;
              END                            ; Fin du programme
```

Cet exemple en Uniprog utilise le registre R0 et une variable VAL :

- On affecte 8 à R0 et -0.05 à VAL.
- On inverse R0 qui contient alors 0.125.
- On inverse VAL qui contient alors -20.

(**) ISO msg

L'instruction **ISO** est prévue pour effectuer un appel à l'ISO depuis l'Uniprog. L'exécution est comparable à celle du mode IMD. Il y a donc des limitations. Par exemple, les sous-programmes, la correction de rayon d'outil (G40, G41 et G42) ou les sauts (N, G69) ne sont pas exécutables avec l'instruction ISO. Plusieurs éléments (arcs ou segments) ne peuvent pas être exécutés sans un arrêt entre eux. Selon ce que contient l'argument *msg*, l'instruction ISO peut-être atomique ou non et modifier ou non le timer, d'où les étoiles entre parenthèses (**).

- *msg :* Numéro du message issu du fichier MSG.INI.

Notes :

L'inverse de cette fonction (appel de l'Uniprog depuis l'ISO), est réalisé par l'appel aux fonctions M.

Une partie des problèmes d'appels à l'ISO depuis l'Uniprog peut être résolue avec l'utilisation de l'instruction ***CYCLN***.

Attention, l'instruction ISO ne doit pas être exécutée depuis une tâche simultanée dont le numéro est supérieur à 4. En effet, les programmes ISO ne peuvent être exécutés que dans les tâches 0, 1, 2, 3 et 4.

Les variables (#VAR) sont autorisées dans l'instruction ISO.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m0 = "G0 X25 Y#VAR"
m1 = "G0 X0 0"
```

En Uniprog :

```
VAR         =                               ; Declaration d'une variable

;                       CORPS DU PROGRAMME

            MOV     #VAR 6.3                ; VAR <- 6.3

            ISO     0                       ; Aller en position (25; 6.3)
            WAIT    1                       ; Attente d'une seconde
            ISO     1                       ; Retour a (0; 0)
            WAIT    1                       ; Attente d'une seconde

            END                             ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Exécute la ligne ISO : G0 X25 Y#VAR avec #VAR qui vaut 6.3.
- Attendre une seconde.
- Exécute la ligne ISO : G0 X0 Y0.
- Attendre une seconde.
- Terminer le programme.

ISODEF sim msg espace [axeX [axeY [axeZ]]]

Cette instruction n'est utile qu'aux utilisateurs ayant besoin d'exécuter plusieurs programmes ISO simultanément.

L'instruction **ISODEF** définit un programme ISO comme nouvelle tâche simultanée à exécuter, cette tâche est par la suite lancée avec l'instruction ***ISORUN***. Les instructions **ISODEF** et ***ISORUN*** sont donc indissociables.

- *sim* : De 0 à 4, *sim* est le numéro de la tâche simultanée désirée pour l'exécution du programme ISO.
- *msg :* De 0 à 119, numéro de la chaîne de caractères contenant le nom du programme ISO choisi.

  Exemple: *msg* = 21, avec m21 = « PIECE10.E7I » déclaré dans le fichier « MSG.INI », désigne le programme ISO 'PIECE10.E7I'.
- *espace :* définit les axes concernés par le programme ISO.

  espace est constitué d'un code binaire sur 16 bits où le $i^{ème}$ bit « b_i » (i de 0 à 15, 16 axes physiques) est tel que :

  « b_i » vaut 1 si l'axe physique « i » est concerné,

  « b_i » vaut 0 si l'axe « i » n'est pas concerné :

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1

  Exemples :

  Si les axes concernés sont les axes 0, 1, 2 et 3, *espace* vaudra

  0000 0000 0000 1111 = 15

  Si les axes concernés sont les axes 0, 1, 2, 3, 7, 10 et 12, *espace* vaudra

  0001 0100 1000 1111 = 5263

  Si les axes concernés sont les axes 0 et 1, *espace* vaudra

  0000 0000 0000 0011 = 3

- *[axeX [axeY [axeZ]]]* : Optionnels, ces paramètres permettent de fournir au programme ISO la correspondance entre les axes utilisés sur le E700 et les axes X Y et Z de l'ISO, utile pour les fonctions ISO G17, G18, G19.

  On spécifie dans *axeX* l'axe longitudinal (axe X en ISO), dans *axeY* l'axe transversal (axe Y en ISO), et dans *axeZ* l'axe vertical (axe Z en ISO).

  Exemple :

  - si *espace* = 13 = 0000 0000 0000 1101, les axes 0, 2 et 3 sont concernés par le **ISODEF** (bits 0, 2 et 3 à 1).
  - si on donne comme paramètre optionnel 3 pour *axeX* et 2 pour *axeX*
  - si on donne comme paramètre optionnel 0 pour *axeZ*.
  - Alors:
    - Les axes 3 et 2 formeront le plan (XY en ISO).
    - L'axe 0 sera l'axe vertical (Z en ISO).

Notes :

Il y a possibilité de faire exécuter jusqu'à 5 tâches ISO simultanément (ISO multiple) :

- La tâche 0 est la tâche par défaut pour l'exécution du programme CYCLE.
- C'est aussi dans la tâche 0 que s'exécute par défaut un programme écrit en ISO.
- Le programme maître est en tâche 0 (programme CYCLE, lancé par le bouton START), il exécute l'instruction **ISODEF** pour préparer l'exécution de programmes ISO en tâches simultanées 1 à 4 (programmes esclaves).

Voir aussi Instructions ***ISORUN***, ***CYCLN*****.**

Exemple :

Fichier « DISPLAY.INI » :

```
[Display0]
l2 = "L A N C E R   P R O G R A M M E S   I S O"
f4 = "PART14"
f5 = "PART15"
f6 = "FIN"
```

Fichier « MSG.INI » :

```
[Msg]
m24 = "PART14.E7I"
m25 = "PART15.E7I"
m34 = "PROGRAMME ISO PART14.E7I EN COURS ( 5s)"
m35 = "PROGRAMME ISO PART15.E7I EN COURS (10s)"
```

En Uniprog :

```
;                          CORPS DU PROGRAMME
; Boucler sur touches F3, F5 et F6 si dans ecran utilisateur 1:
MENU0:      BRIN0   #MACRO[1] MENU0        ; Ecran uti 1? non: boucler
            SWITCH #CURKEY                 ; Tester touche pressee
              CASE  KF4 PRG14 MENUEND      ; Cas F4: PRG14 puis MENUEND
              CASE  KF5 PRG15 MENUEND      ; Cas F5: PRG15 puis MENUEND
                                           ; Autre cas (F6): continuer
MENUEND:    ENDS                           ; Fin de test
            CMP     #CURKEY KF6            ; Est-ce F6 ?
            JNE     MENU0                  ; Non: reprendre MENU0
;
; Arret des simultanes 0 a 8, sauf la tache courante:
FIN:        CMP     #PNB R0                ; R0 = tache courante ?
            JE      FIN2                   ; Oui: ne pas arreter la tache
            RSIM    R0                     ; Reveil eventuel simultane R0
            KSIM    R0                     ; Arret du simultané R0
FIN2:       INC     R0                     ; Increment numero simultane
            CMP     R0 8                   ; R0 = 8 ?
            JLE     FIN                    ; R0 different de 8, on boucle
            STOPM   -1 0                   ; Arret des mouvements
;
            END                            ; Fin du programme
;
```

```
;                              PROCEDURES
; Procedure PRG14 - Lancement programme ISO PART14.E7I:
PRG14:      WAIT1   #KEY[KF4]                ; Attendre F4 relachee
            ISODEF  1 24 7 1 2               ; programme ISO m24
                                             ; en simultane 1
                                             ; espace=7 => axes 0,1,2
                                             ; axe X=1 et axe Y=2
            ISORUN  1                        ; Lancemer ISO simultane 1
            END                              ; Fin procedure PRG14
; Procedure PRG15 - Lancement programme ISO PART15.E7I:
PRG15:      WAIT1   #KEY[KF5]                ; Attendre F5 relachee
            ISODEF  2 25 56 4 5              ; programme ISO m25
                                             ; en simultane 2
                                             ; espace=56 => axes 3,4,5
                                             ; axe X=4 et axe Y=5
            ISORUN  2                        ; Lancemer ISO simultane 2
            END                              ; Fin procedure PRG15
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Attente sur les touches de fonctions F4, F5, F6:
- si F4, appel procédure PRG14:
  - PRG14 attend que F4 soit relachee,
  - puis definit ISODEF pour lancer le programme ISO PART14.E7I en tache 1,
  - puis lance la tache 1.
- si F5, appel procédure PRG15:
  - PRG15 attend que F5 soit relachee,
  - puis definit ISODEF pour lancer le programme ISO PART15.E7I en tache 2,
  - puis lance la tache 2.
- si F6, terminer programmes ISOpuis terminer le programme.

ISORUN sim

L'instruction **ISORUN** démarre l'exécution du programme ISO préalablement défini avec l'instruction ***ISODEF***. L'instruction **ISORUN** est indissociable de l'instruction ***ISODEF***.

- *sim* : De 0 à 4, *sim* est le numéro de la tâche simultanée dont les caractéristiques ont été préalablement définies par l'instruction ***ISODEF***.

Notes :

La tâche courante qui exécute l'instruction **ISORUN** continue normalement à s'exécuter à l'instruction suivant le **ISORUN** : il y a donc exécution en parallèle des deux tâches, le programme Uniprog contenant le **ISORUN**, et le programme ISO démarré par le **ISORUN**, qui ont bien sûr deux numéros de tâches différents.

Si *sim* ne correspond pas à une définition préalablement effectuée avec l'instruction ***ISODEF***, une erreur R20 est générée (Fichier ISO invalide), tous les programmes s'interrompent, la tâche AUTOMAT est relancée après acquittement du message d'erreur par l'utilisateur.

Voir aussi Instructions ***ISODEF***, ***CYCLN*****.**

Exemple :

Se rapporter à l'exemple donné pour l'instruction ***ISODEF*** page ***86***.

JE label

L'instruction **JE** effectue un saut à l'instruction de l'étiquette *label* si la dernière comparaison avec l'instruction « ***CMP*** *src1 src2* » indique que ***src1* est égal à *src2***, et continue à l'instruction suivante sinon (**J**ump if **E**qual).

L'instruction **JE** est indissociable de l'instruction ***CMP***.

Notes :

Les instructions de saut conditionnel utilisant le résultat de l'instruction ***CMP*** sont ***JE*** (=), ***JG*** (>), ***JGE*** (≥), ***JL*** (<), ***JLE*** (≤), ***JNE*** (≠).

CMP est modale, les instructions de sauts conditionnels tiennent compte des résultats de la dernière instruction ***CMP*** qui sont conservés dans la tâche simultanée courante. Pour éviter les erreurs et bien-que ce ne soit pas obligatoire, il est recommandé que l'instruction **JE** suive l'instruction ***CMP***.

Instructions ***CMP***, ***JG***, ***JGE***, ***JL***, ***JLE***, ***JNE***.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m1  = "JE vrai => saut a l'instruction de label "
m2  = "JE faux => continuer instruction suivante"
```

En Uniprog :

```
            CMP     #IN[3] 1                ; Comparer IN[3] et 1
            JE      VRAI                    ; si IN[3] = 1, aller a VRAI
FAUX:       DISPST 2                        ; Afficher m2="JE faux"
            JMP     FIN                     ; Aller a FIN
VRAI:       DISPST 1                        ; Afficher m1="JE vrai"
FIN:        END                             ; Fin du programme
```

Ce petit exemple effectue les opérations suivantes :

- Teste l'entrée numérique 3 IN[3] en la comparant avec la valeur 1.
- En fonction du saut conditionnel avec l'instruction **JE** :
  - Si IN[3] = 1, **JE** saute au label VRAI et affiche le message m1 = "JE vrai".
  - Sinon, **JE** continue à l'instruction suivante et affiche le message m2 = "JE faux".

JG label

L'instruction **JG** effectue un saut à l'instruction de l'étiquette *label* si la dernière comparaison avec l'instruction « ***CMP*** *src1 src2* » indique que ***src1*** **est plus grand strictement que** ***src2***, et continue à l'instruction suivante sinon (**J**ump if **G**reater).

L'instruction **JG** est indissociable de l'instruction ***CMP***.

Notes :

Les instructions de saut conditionnel utilisant le résultat de l'instruction ***CMP*** sont ***JE*** (=), ***JG*** (>), ***JGE*** (≥), ***JL*** (<), ***JLE*** (≤), ***JNE*** (≠).

CMP est modale, les instructions de sauts conditionnels tiennent compte des résultats de la dernière instruction ***CMP*** qui sont conservés dans la tâche simultanée courante. Pour éviter les erreurs et bien-que ce ne soit pas obligatoire, il est recommandé que l'instruction **JG** suive l'instruction ***CMP***.

Voir Instructions ***CMP***, ***JE***, ***JGE***, ***JL***, ***JLE***, ***JNE***.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m1  = "JG vrai => saut a l'instruction de label "
m2  = "JG faux => continuer instruction suivante"
```

En Uniprog :

```
            CMP     #SEC 30                 ; Comparer SECONDES et 30
            JG      VRAI                    ; si SEC > 30, aller a VRAI
FAUX:       DISPST 2                        ; Afficher m2="JG faux"
            JMP     FIN                     ; Aller a FIN
VRAI:       DISPST 1                        ; Afficher m1="JG vrai"
FIN:        END                             ; Fin du programme
```

Ce petit exemple effectue les opérations suivantes :

- Teste la variable système SEC des secondes en la comparant avec la valeur 30.
- En fonction du saut conditionnel avec l'instruction **JG** :
  - Si SEC > 30, **JG** saute au label VRAI et affiche m1 = "JG vrai".
  - Sinon, **JG** continue à l'instruction suivante et affiche m2 = "JG faux".

JGE label

L'instruction **JGE** effectue un saut à l'instruction de l'étiquette *label* si la dernière comparaison avec l'instruction « ***CMP*** *src1 src2* » indique que ***src1* est plus grand ou égal à *src2***, et continue à l'instruction suivante sinon (**J**ump if **G**reater or **E**qual).

L'instruction **JGE** est indissociable de l'instruction ***CMP***.

Notes :

Les instructions de saut conditionnel utilisant le résultat de l'instruction ***CMP*** sont ***JE*** (=), ***JG*** (>), ***JGE*** (≥), ***JL*** (<), ***JLE*** (≤), ***JNE*** (≠).

CMP est modale, les instructions de sauts conditionnels tiennent compte des résultats de la dernière instruction ***CMP*** qui sont conservés dans la tâche simultanée courante. Pour éviter les erreurs et bien-que ce ne soit pas obligatoire, il est recommandé que l'instruction **JGE** suive l'instruction ***CMP***.

Voir aussi Instructions ***CMP***, ***JE***, ***JG***, ***JL***, ***JLE***, ***JNE***.

Exemple :

Soient dans le fichier « MSG.INI », les lignes :

```
[Msg]
m1  = "JGE vrai => saut a l'instruction de label "
m2  = "JGE faux => continuer instruction suivante"
```

En Uniprog :

```
            CMP     #SEC 30                 ; Comparer SECONDES et 30
            JGE     VRAI                    ; si SEC >= 30, aller a VRAI
FAUX:       DISPST 2                        ; Afficher m2="JGE faux"
            JMP     FIN                     ; Aller a FIN
VRAI:       DISPST 1                        ; Afficher m1="JGE vrai"
FIN:        END                             ; Fin du programme
```

Ce petit exemple effectue les opérations suivantes :

- Teste la variable système SEC des secondes en la comparant avec la valeur 30.
- En fonction du saut conditionnel avec l'instruction **JGE** :
  - Si SEC ≥ 30, **JGE** saute au label VRAI et affiche m1 = "JGE vrai".
  - Sinon, **JGE** continue à l'instruction suivante et affiche m2 = "JGE faux".

JL label

L'instruction **JL** effectue un saut à l'instruction de l'étiquette *label* si la dernière comparaison avec l'instruction « ***CMP*** *src1 src2* » indique que ***src1*** **est plus petit strictement que** ***src2***, et continue à l'instruction suivante sinon (**J**ump if **L**ess).

L'instruction **JL** est indissociable de l'instruction ***CMP***.

Notes :

Les instructions de saut conditionnel utilisant le résultat de l'instruction ***CMP*** sont ***JE*** (=), ***JG*** (>), ***JGE*** (≥), ***JL*** (<), ***JLE*** (≤), ***JNE*** (≠).

CMP est modale, les instructions de sauts conditionnels tiennent compte des résultats de la dernière instruction ***CMP*** qui sont conservés dans la tâche simultanée courante. Pour éviter les erreurs et bien-que ce ne soit pas obligatoire, il est recommandé que l'instruction **JL** suive l'instruction ***CMP***.

Voir aussi Instructions ***CMP***, ***JE***, ***JG***, ***JGE***, ***JLE***, ***JNE***.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m1  = "JL vrai => saut a l'instruction de label "
m2  = "JL faux => continuer instruction suivante"
```

En Uniprog :

```
            CMP     #SEC 30                 ; Comparer SECONDES et 30
            JL      VRAI                    ; si SEC < 30, aller a VRAI
FAUX:       DISPST 2                        ; Afficher m2="JL faux"
            JMP     FIN                     ; Aller a FIN
VRAI:       DISPST 1                        ; Afficher m1="JL vrai"
FIN:        END                             ; Fin du programme
```

Ce petit exemple effectue les opérations suivantes :

- Teste la variable système SEC des secondes en la comparant avec la valeur 30.
- En fonction du saut conditionnel avec l'instruction **JL** :
  - Si SEC < 30, **JL** saute au label VRAI et affiche le message m1 = "JL vrai".
  - Sinon, **JL** continue à l'instruction suivante et affiche le message m2 = "JL faux".

JLE label

L'instruction **JLE** effectue un saut à l'instruction de l'étiquette *label* si la dernière comparaison avec l'instruction « ***CMP*** *src1 src2* » indique que ***src1*** **est plus petit ou égal à** ***src2***, et continue à l'instruction suivante sinon (**J**ump if **L**ess or **E**qual).

L'instruction **JLE** est indissociable de l'instruction ***CMP***.

Notes :

Les instructions de saut conditionnel utilisant le résultat de l'instruction ***CMP*** sont ***JE*** (=), ***JG*** (>), ***JGE*** (≥), ***JL*** (<), ***JLE*** (≤), ***JNE*** (≠).

CMP est modale, les instructions de sauts conditionnels tiennent compte des résultats de la dernière instruction ***CMP*** qui sont conservés dans la tâche simultanée courante. Pour éviter les erreurs et bien-que ce ne soit pas obligatoire, il est recommandé que l'instruction **JE** suive l'instruction ***CMP***.

Voir aussi Instructions ***CMP***, ***JE***, ***JG***, ***JGE***, ***JL***, ***JNE***.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m1  = "JLE vrai => saut a l'instruction de label "
m2  = "JLE faux => continuer instruction suivante"
```

En Uniprog :

```
            CMP     #SEC 30                 ; Comparer SECONDES et 30
            JLE     VRAI                    ; si SEC <= 30, aller a VRAI
FAUX:       DISPST 2                        ; Afficher m2="JLE faux"
            JMP     FIN                     ; Aller a FIN
VRAI:       DISPST 1                        ; Afficher m1="JLE vrai"
FIN:        END                             ; Fin du programme
```

Ce petit exemple effectue les opérations suivantes :

- Teste la variable système SEC des secondes en la comparant avec la valeur 30.
- En fonction du saut conditionnel avec l'instruction **JLE** :
  - Si SEC < 30, **JLE** saute au label VRAI et affiche m1 = "JLE vrai".
  - Sinon, **JLE** continue à l'instruction suivante et affiche m2 = "JLE faux".

JMP label

L'instruction **JMP** effectue un saut à l'instruction de l'étiquette *label* (**J**u**MP**).

Notes :

JMP continue simplement le programme à l'étiquette *label* (pas de mémorisation d'adresse en pile).

.

Exemple :

Soient dans le fichier « MSG.INI », les lignes :

```
[Msg]
m1  = "ALARME: IN[3] vaut 0"
```

En Uniprog :

```
BOUCLE:     WAIT   0.01                  ; Attendre 10 ms
            CMP    #IN[3] 0             ; Comparer IN[3] et 0
            JE     ALARME               ; si IN[3]=0, aller a ALARME
;
            JMP    BOUCLE               ; Boucler pour surveiller
;
ALARME:     DISPC  7                    ; Emettre un Beep
            MSG    1                    ; Afficher m1
;
FIN:        END                         ; Fin du programme
```

Ce petit exemple effectue les opérations suivantes :

- Surveiller en boucle l'entrée numérique IN[3] :
  - Attendre 10 ms dans cette boucle de surveillance pour ne pas charger inutilement le processeur au détriement des autres tâches système et utilisateur.
  - Comparer IN[3] et la valeur 0.
    - Si IN[3] est égal à 0 :
      - aller au label ALARME
      - générer un Beep
      - écrire un message d'alarme m1 = « ALARME: IN[3] vaut 0 »
      - terminer le programme.
    - Sinon, reprendre la boucle.

Note : L'intérêt de cet exemple est d'illustrer une utilisation de l'instruction JMP. Il est évident qu'en pratique, on préférerait programmer :

```
            WAIT1  #IN[3]                ; Attendre 10 ms
            DISPC  7                     ; Emettre un Beep
            MSG    1                     ; Afficher m1
            END                          ; Fin du programme
```

JNE label

L'instruction **JNE** effectue un saut à l'instruction de l'étiquette *label* si la dernière comparaison avec l'instruction « ***CMP*** *src1 src2* » indique que ***src1* est différent de *src2***, et continue à l'instruction suivante sinon (**J**ump if **N**ot **E**qual).

L'instruction **JNE** est indissociable de l'instruction ***CMP***.

Notes :

Les instructions de saut conditionnel utilisant le résultat de l'instruction ***CMP*** sont ***JE*** (=), ***JG*** (>), ***JGE*** (≥), ***JL*** (<), ***JLE*** (≤), ***JNE*** (≠).

CMP est modale, les instructions de sauts conditionnels tiennent compte des résultats de la dernière instruction ***CMP*** qui sont conservés dans la tâche simultanée courante. Pour éviter les erreurs et bien-que ce ne soit pas obligatoire, il est recommandé que l'instruction **JNE** suive l'instruction ***CMP***.

Voir aussi Instructions ***CMP***, ***JE***, ***JG***, ***JGE***, ***JL***, ***JLE***.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m1  = "JNE vrai => saut a l'instruction de label "
m2  = "JNE faux => continuer instruction suivante"
```

En Uniprog :

```
            CMP    #IN[3] 1              ; Comparer IN[3] et 1
            JNE    VRAI                  ; si IN[3] <> 1, aller a VRAI
FAUX:       DISPST 2                     ; Afficher m2="JNE faux"
            JMP    FIN                   ; Aller a FIN
VRAI:       DISPST 1                     ; Afficher m1="JNE vrai"
FIN:        END                          ; Fin du programme
```

Ce petit exemple effectue les opérations suivantes :

- Teste l'entrée numérique 3 IN[3] en la comparant avec la valeur 1.
- En fonction du saut conditionnel avec l'instruction **JNE** :
  - Si IN[3] ≠ 1, **JNE** saute au label VRAI et affiche m1 = "JNE vrai".
  - Sinon, **JNE** continue à l'instruction suivante et affiche m2 = "JNE faux".

KSIM sim

L'instruction **KSIM** termine le programme en tâche simultanée *sim* en l'interrompant (**K**ill **SIM**ultaneous task).

- *sim :* De 0 à 9, est le numéro de la tâche simultanée à terminer.

Notes :

Une tâche peut se terminer elle-même (commande « **KSIM** #PNB » par exemple).

Si la tâche *sim* n'est pas présente en exécution, l'instruction **KSIM** est sans effet.

ATTENTION : une tâche ne peut être terminée que si elle est active, il est donc nécessaire de faire un ***RSIM*** avant de faire un **KSIM** sur une tâche mise en pause avec ***PSIM*** (par sécurité, il est préférable de faire systématiquement ***RSIM*** suivi de **KSIM**).

Voir aussi *Erreur ! Source du renvoi introuvable. Erreur ! Source du renvoi introuvable.* page *Erreur ! Signet non défini.*, Instructions ***ASIM***, ***PSIM***, ***RSIM*.**

Exemple :

```
KILLALL:    REP     9
             KSIM   #REPI                ; Arret du simultane 8..0
            ENDRP
            END
```

Cet exemple en Uniprog arrête toutes les tâches, sauf la 9 (AUTOMATE). Peut être utilisé dans le cas d'un arrêt d'urgence par exemple.

*LINA axe val mode

L'instruction **LINA** réalise une interpolation linéaire à partir de coordonnées en mode absolu (réalisation d'une droite, cf. les fonctions ISO G90 puis G01).

L'instruction **LINA** est le plus souvent remplacée par l'instruction ***LINA2*** qui permet de définir en une seule instruction les coordonnées à atteindre sur plusieurs axes.

LINA est utile pour la compatibilité avec l'ancien Uniprog ainsi que pour combiner des interpolations linéaires et circulaires (hélice par exemple).

Deux modes de fonctionnement sont utilisés :

- Le mode préparatoire (avec *mode* = 0) permet de donner les coordonnées à atteindre pour chacun des axes, avec une instruction **LINA** par axe.
- Le mode exécution (avec *mode* <> 0) lance l'interpolation avec les critères donnés par les différentes instructions **LINA** en mode préparatoire et par cette instruction (cas de la définition du dernier axe).

- *axe* : Numéro ou nom de l'axe pour lequel on détermine la position *val* à atteindre sur cet axe.
- *val :* valeur donnant la coordonnée absolue à atteindre sur l'axe *axe*, en unités de longueur.
- *mode* contient le mode opératoire de l'instruction **LINA** :

  0 : mode préparatoire (cas de plusieurs axes à définir avant mise en œuvre).

  Non 0 : mode exécution (cas d'un seul axe, ou pour le dernier des axes).

Notes :

L'instruction **LINA** doit obligatoirement se trouver insérée entre les instructions ***DPATH*** et ***ENDP***.

L'instruction **LINA** en mode préparatoire permet de combiner une interpolation circulaire avec une interpolation linéaire sur un autre axe.

Exemple : en combinant une interpolation linéaire en Z avec **LINA** en mode préparatoire, et une interpolation circulaire en X et Y avec ***CIRA***, on obtient une hélice (exemple « Exemples Uniprog\LINA\ELICEA.E7U ») :

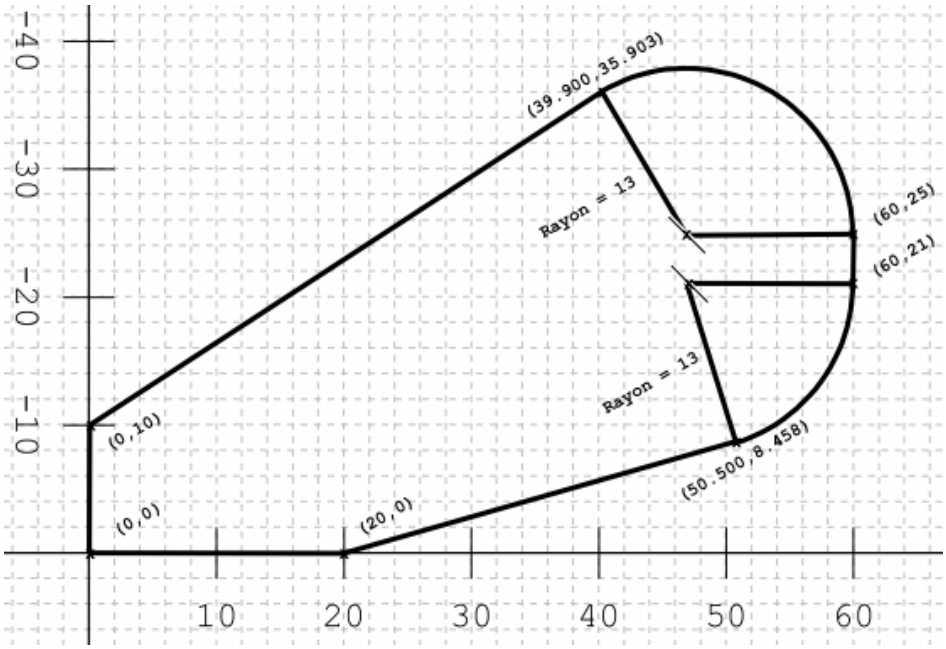
```
POSA   X #VMAX[X]   3 0        ; Aller en X=+03 attente
POSA   Y #VMAX[Y] -10 0        ; Aller en Y=-10 attente
POSA   Z #VMAX[Z]  10 2        ; Aller en Z=+10 go
DPATH  0 7 1                   ; Interpolation espace 0
                               ; axes 0=X 1=Y 2=Z a 1 m/min
 LINA   Z 15 0                 ; PreparerDroite vers Z=+15
 CIRA   X 13 Y -10  5 0 1      ; Cercle vers X=+13 Y=-10
 LINA   Z 20 0                 ; PreparerDroite vers Z=+20
 CIRA   X  3 Y -10 -5 0 1      ; Cercle vers X=+03 Y=-10
ENDP                           ; Terminer le contour
```

Voir aussi Instructions ***DPATH***, ***ENDP***, ***LINA2***, ***LINR***, ***LINR2***, ***RAD***, ***CIRA***, ***CIRR***.

Exemple :

```
                POSA   X #VMAX[X]  0 0          ; Aller en X=+00 attente
                POSA   Y #VMAX[Y]  0 2          ; Aller en Y=+00 go
;
                DPATH  0 3 1                    ; Interpolation espace 0
                                                ; axes 0=X 1=Y a 1 m/min
                 LINA   X 20  0                 ; Droite vers X=+20 preparer
                 LINA   Y 00  1                 ; Droite vers Y=+00 lancer
                 LINA   X 50.500  0             ; Droite vers X=50.5 preparer
                 LINA   Y 8.458   1             ; Droite vers Y=8.458 lancer
                 RAD    13 0                    ; Rayon 13 sens anti-horaire
                 CIRA   X 60 Y 21               ; Cercle vers X=+60 Y=+21
                 LINA   X 60 0                  ; Droite vers X=+60 preparer
                 LINA   Y 25 1                  ; Droite vers Y=+25 lancer
                 CIRA   X 39.900 Y 35.903       ; Cercle vers X=39.9 Y=35.903
                 LINA   X 0  0                  ; Droite vers X=+00 preparer
                 LINA   Y 10 1                  ; Droite vers Y=+10 lancer
                 LINA   X 0  0                  ; Droite vers X=+00 preparer
                 LINA   Y 0  1                  ; Droite vers Y=+00 lancer
                ENDP                            ; Terminer le contour
```

Cet exemple en Uniprog effectue le contour suivant :



- Définit une interpolation entre les axes X et Y:
- Décrire un contour avec droites et arcs de cercles.
- Utilisation de deux lignes pour les interpolations linéaires :
  - la première définit la coordonnée à atteindre en X et en mode préparatoire,
  - la deuxième définit la coordonnée à atteindre en Y et en mode exécution.
- Finaliser le contour.

L'exemple « LINA.E7U » livré avec la documentation (dossiers « Exemples Uniprog/Fichiers E700 communs aux exemples » et « Exemples Uniprog\LINA »), permet de réaliser l'exemple sur un E700 et de visualiser le résultat du contour sur l'écran du E700.

*LINA2 axe val { axe val }

L'instruction **LINA2** réalise une interpolation linéaire à partir de coordonnées en mode absolu (réalisation d'une droite, cf. les fonctions ISO G90 puis G01).

Les accolades « { } » autour de « *axe val* » signifient que ces paramètres vont par paire (associer un axe et une valeur), les crochets « [ ] » et les points de suspensions « . . . » signifient que l'on peut mettre autant de paires de paramètres « *axe val* » que l'on veut (parmi les axes de l'espace d'interpolation).

- *axe* : Numéro ou nom de l'axe pour lequel on détermine la position *val* à atteindre sur cet axe.
- *val :* valeur donnant la coordonnée absolue à atteindre sur l'axe *axe*, en unités de longueur.

Notes :

L'instruction **LINA2** doit obligatoirement se trouver insérée entre les instructions ***DPATH*** et ***ENDP***.

L'instruction **LINA2** réalise un segment de n'importe quelle dimension en une seule instruction.

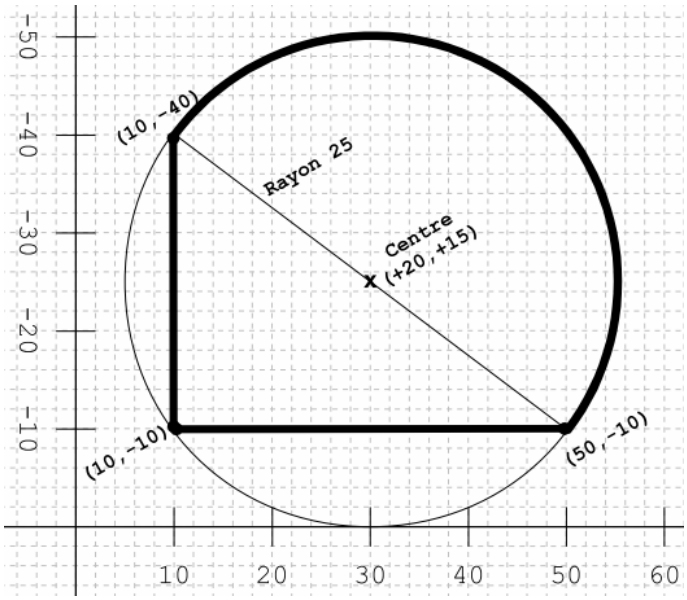
Si besoin est, l'instruction ***LINA*** en mode préparatoire permet de combiner une interpolation circulaire avec une interpolation linéaire sur un autre axe (cf. exemple donné dans les notes de l'instruction ***LINA***). Dans ce cas particulier, LINA2 n'est pas utilisable.

Voir aussi Instructions ***DPATH***, ***ENDP***, ***LINA***, ***LINR***, ***LINR2***, ***RAD***, ***CIRA***, ***CIRR***.

Exemple :

```
                POSA   X #VMAX[X]  10 0        ; Aller en X=+10 attente
                POSA   Y #VMAX[Y] -10 2        ; Aller en Y=-10 go
;
                DPATH  0 3 1                   ; Interpolation espace 0
                                               ; axes 0=X 1=Y a 1 m/min
                 LINA2  X 10 Y -40             ; Droite vers X=+10 Y=-40
                 CIRA   X 50 Y -10 20 15 0     ; Cercle vers X=+50 Y=-10
                                               ; centre (+20,+15)
                                               ; sens anti-horaire
                 LINA2  X 10 Y -10             ; Droite vers X=+10 Y=-10
                ENDP                           ; Terminer le contour
;
                END                            ; Fin du programme
```

Cet exemple en Uniprog effectue le contour suivant :



- Définit une interpolation entre les axes X et Y:
- Effectue un contour avec droites et arcs de cercles.
- Finaliser le contour.

L'exemple « LINA2.E7U » livré avec la documentation (dossiers « Exemples Uniprog/Fichiers E700 communs aux exemples » et « Exemples Uniprog\LINA2 »), permet de réaliser l'exemple sur un E700 et de visualiser le résultat du contour sur l'écran du E700.

*LINR axe val mode

L'instruction **LINR** réalise une interpolation linéaire à partir de coordonnées en mode relatif à la position courante (réalisation d'une droite, cf. les fonctions ISO G91 puis G01).

Deux modes de fonctionnement sont utilisés :

- Le mode préparatoire (avec *mode* = 0) permet de donner les coordonnées à atteindre pour chacun des axes, avec une instruction **LINR** par axe.
- Le mode exécution (avec *mode* = 1) lance l'interpolation avec les critères donnés par les différentes instructions **LINR** en mode préparatoire et par cette instruction (cas de la définition du dernier axe).

- *axe* : Numéro ou nom de l'axe pour lequel on détermine la position *val* à atteindre sur cet axe.
- *val :* valeur donnant la coordonnée à atteindre sur l'axe *axe* relativement à la position actuelle, en unités de longueur.
- *mode* contient le mode opératoire de l'instruction **LINR** :

  0 : mode préparatoire (cas de plusieurs axes à définir avant mise en œuvre).

  Autre : mode exécution (cas d'un seul axe, ou pour le dernier des axes).

Notes :

L'instruction **LINR** doit obligatoirement se trouver insérée entre les instructions ***DPATH*** et ***ENDP***.

L'instruction **LINR** est le plus souvent remplacée par l'instruction ***LINR2*** qui permet de définir en une seule instruction les coordonnées à atteindre sur plusieurs axes.

Par contre, l'instruction **LINR** en mode préparatoire permet de combiner une interpolation circulaire avec une interpolation linéaire sur un autre axe.

- Exemple : en combinant une interpolation linéaire en Z avec **LINR** en mode préparatoire, et une interpolation circulaire en X et Y avec ***CIRR***, on obtient une hélice (voir fichier programme exemple « Exemples Uniprog\LINR\ELICER.E7U ») :

```
POSA   X #VMAX[X]   3 0        ; Aller en X=+03 attente
POSA   Y #VMAX[Y] -10 0        ; Aller en Y=-10 attente
POSA   Z #VMAX[Z]  10 2        ; Aller en Z=+10 go
DPATH  0 7 1                   ; Interpolation espace 0
                               ; axes 0=X 1=Y 2=Z a 1 m/min
 LINR   Z  5   0               ; PreparerDroite vers Z=+05
 CIRR   X  10  Y 0  5 0 1      ; Cercle vers X=+10 Y=+00
 LINR   Z  5   0               ; PreparerDroite vers Z=+05
 CIRR   X -10  Y 0 -5 0 1      ; Cercle vers X=-10 Y=+00
ENDP                           ; Terminer le contour
```

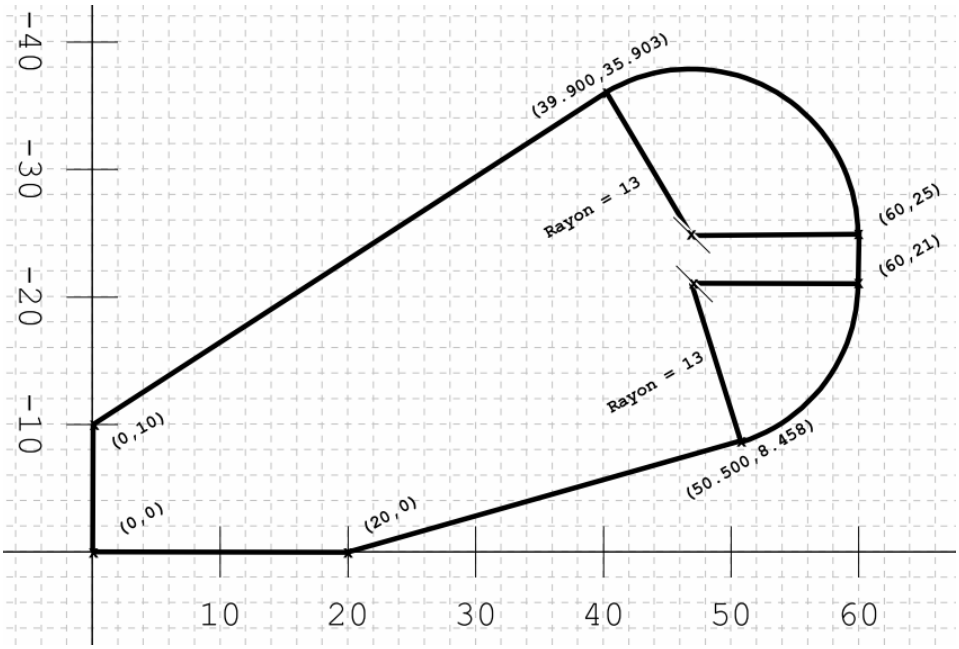
Voir aussi Instructions ***DPATH***, ***ENDP***, ***LINA***, ***LINA2***, ***LINR2***, ***RAD***, ***CIRA***, ***CIRR***.

Exemple :

En reprenant l'exemple de l'instruction ***LINA***, nous avons les instructions Uniprog suivantes :

```
            POSA   X #VMAX[X]  0 0          ; Aller en X=+00 attente
            POSA   Y #VMAX[Y]  0 2          ; Aller en Y=+00 go
;
            DPATH  0 3 1                    ; Interpolation espace 0
                                            ; axes 0=X 1=Y a 1 m/min
             LINR   X 20  0                 ; Droite vers X=+20 preparer
             LINR   Y 00  1                 ; Droite vers Y=+00 lancer
             LINR   X 30.500  0             ; Droite vers X=+30.5 preparer
             LINR   Y 8.458   1             ; Droite vers Y=+8.458 lancer
              RAD    13 0                   ; Rayon 13 sens anti-horaire
              CIRR   X 9.500 Y 12.542       ; Cercle vers X=+9.5 Y=+12.542
             LINR   X 0  0                  ; Droite vers X=+00 preparer
             LINR   Y 4  1                  ; Droite vers Y=+04 lancer
              CIRR   X -20.100 Y 10.903     ; Cercle vers X=-20.1 Y=10.903
             LINR   X -39.900  0            ; Droite vers X=-39.9 preparer
             LINR   Y -25.903  1            ; Droite vers Y=-25.903 lancer
             LINR   X 0    0                ; Droite vers X=+00 preparer
             LINR   Y -10  1                ; Droite vers Y=-10 lancer
            ENDP                            ; Terminer le contour
```

Cet exemple en Uniprog effectue le contour suivant (en rouge) :



- Définit une interpolation entre les axes X et Y:
- Effectue un contour particulier avec droites et arcs de cercles.
- Utilisation de deux lignes pour les interpolations linéaires :
  - la première définit la coordonnée à atteindre en X et en mode préparatoire,
  - la deuxième définit la coordonnée à atteindre en Y et en mode exécution.
- Finaliser le contour.

L'exemple « LINR.E7U » livré avec la documentation (dossiers « Exemples Uniprog/Fichiers E700 communs aux exemples » et « Exemples Uniprog\LINR »), permet de réaliser l'exemple sur un E700 et de visualiser le résultat du contour sur l'écran du E700.

*LINR2 axe val { axe val }

L'instruction **LINR2** réalise une interpolation linéaire à partir de coordonnées en mode relatif à la position courante (réalisation d'une droite, cf. les fonctions ISO G91 puis G01).

Les accolades « { } » autour de « *axe val* » signifient que ces paramètres vont par paire (associer un axe et une valeur), les crochets « [ ] » et les points de suspensions « . . . » signifient que l'on peut mettre autant de paires de paramètres « *axe val* » que l'on veut (parmi les axes de l'espace d'interpolation).

- *axe* : Numéro ou nom de l'axe pour lequel on détermine la position *val* à atteindre sur cet axe.
- *val :* valeur donnant la coordonnée à atteindre sur l'axe *axe* relativement à la position actuelle, en unités de longueur.

Notes :

L'instruction **LINR2** doit obligatoirement se trouver insérée entre les instructions ***DPATH*** et ***ENDP***.

L'instruction **LINR2** réalise un segment de n'importe quelle dimension en une seule instruction.

Si besoin est, l'instruction ***LINR*** en mode préparatoire permet de combiner une interpolation circulaire avec une interpolation linéaire sur un autre axe (cf. exemple donné dans les notes de l'instruction ***LINR***). Ceci ne pouvant pas être réalisé avec LINR2.

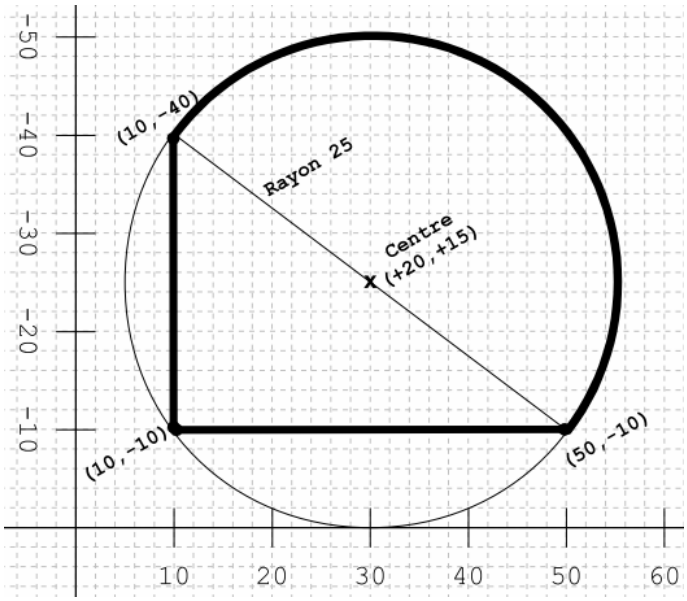
Voir aussi Instructions ***DPATH***, ***ENDP***, ***LINA***, ***LINA2***, ***LINR***, ***RAD***, ***CIRA***, ***CIRR***.

Exemple :

En reprenant l'exemple de l'instruction ***LINA2***, nous avons les instructions Uniprog suivantes :

```
            POSA   X #VMAX[X]  10 0        ; Aller en X=+10 attente
            POSA   Y #VMAX[Y] -10 2        ; Aller en Y=-10 go
            CALL   RECORDON                ; Memoriser chemin contour
            DPATH  0 3 1                   ; Interpolation espace 0
                                           ; axes 0=X 1=Y a 1 m/min
             LINR2  X 0   Y -30            ; Droite vers X=+00 Y=-30
              CIRR   X 40  Y  30  20 15 0  ; Cercle vers X=+40 Y=+30
; centre (+20,+15)
; sens anti-horaire
             LINR2  X -40 Y  0             ; Droite vers X=-40 Y=+00
            ENDP                           ; Terminer le contour
;
            CALL   RECORDOFF               ; Fin memoriser contour
            CALL   ECRPLOT                 ; Afficher le contour
            CALIN1 #OVERSTRK BUTEES        ; Faire Fonction BUTEES
                                           ; si course axe depassee
            END                            ; Fin du programme
```

Cet exemple en Uniprog effectue le contour suivant :



- Définit une interpolation entre les axes X et Y:
  - Effectue un contour avec droites et arcs de cercles.
- Finaliser le contour.

L'exemple « LINR2.E7U » livré avec la documentation (dossiers « Exemples Uniprog/Fichiers E700 communs aux exemples » et « Exemples Uniprog\LINR2 »), permet de réaliser l'exemple sur un E700 et de visualiser le résultat du contour sur l'écran du E700.

LOAD src

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **LOAD** charge l'accumulateur de la tâche courant*e accum[PNB]* avec *src*.

- *src* : nombre à charger dans l'accumulateur, **accum[PNB] ← *src*.**
- *accum[PNB]* : représente la mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, non utilisable directement en Uniprog).

  Est modifié par l'instruction pour être initialisée avec *src*.

Note :

Attention: dans l'ancien Uniprog, cette instruction s'appelait **LOADD**.

Exemple:

Fichier « MSG.INI » :

```
[Msg]
m10 = "Utilisation accumulateur:\nVAL vaut (IN[3] x Pi x 1000) = #VAL"
```

En Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
             PI  = 3.141593                ; Constante Pi
             VAL =                         ; Variable
;
;                          CORPS DU PROGRAMME
             LOAD   10                     ; Ici accum[PNB]=10
             MULD   #IN[3]                 ; Ici accum[PNB]=10
                                           ; ou =0 si IN[3]=0
             STORE  #VAL                   ; VAL=accum[PNB]=10 ou 0
             LOAD   PI                     ; Ici accum[PNB]=3.141593
             MULD   #VAL                   ; Ici accum[PNB]=31.41593 ou 0
             STORE  R0                     ; R0=accum[PNB]=31.41593 ou 0
             MUL    R0 10                  ; Ici R0 = 314.1593 ou 0
             LOAD   R0                     ; Ici accum[PNB]=314.1593 ou 0
             STORE  #VAL                   ; VAL=accum[PNB]=314.1593 ou 0
             MUL    #VAL 10                ; Ici VAL=3141.593 ou 0
             LOAD   #VAL                   ; Ici accum[PNB]=3141.593 ou 0
             STORE  #VAL                   ; VAL=accum[PNB]=3141.593
                                           ; ou =0 si IN[3]=0
             MSG    10                     ; Afficher resultat
;
             END                           ; Fin du programme
```

Cet exemple en Uniprog donne un aperçu des différentes possibilités d'emploi de l'instruction **LOAD**, et différents cas de figures au fil des instructions pour les calculs avec accumulateur, sans accumulateur, pour l'utilisation des variables, des constantes et des registres …

LOG [src]

L'instruction **LOG** permet de mémoriser une information dans l'historique User du E700.

Cette information appelée LOG ou trace, est une ligne sous la forme « **Sim : *sim* – Line : *src*, *temps*** » :

- *sim* est le numéro de la tâche simultanée courante (de 0 à 9),
- *src* est l'entier fourni à l'instruction **LOG**,
- si *src* est omis, *src* contient le numéro de la ligne du programme en cours,
- *temps* est la date de l'événement en temps absolu d'utilisation du E700, unité 200 millisecondes.
- *src* : paramètre optionnel, entier à mémoriser dans l'historique. C'est donc au programmeur de gérer la signification des valeurs de *src*. Si *src* est omis, l'historique contient le numéro de la ligne Uniprog en cours d'exécution.

Notes :

La capacité de l'historique est de 30 valeurs. Au-delà, une nouvelle instruction **LOG** va effacer la ligne d'historique la plus ancienne, ce qui permet de conserver les 30 informations les plus récentes dans l'historique User.

Etant donné que seul 30 événements peuvent être mémorisés, **LOG** peut être utilisée pour surveiller des endroits sensibles du programme, et il faudrait donc éviter d'appeler trop souvent cette instruction.

Le menu avec les touches MENU / F6 = OTHER / F3 = LOGS / F4 = USER :

- Permet de visualiser les informations mémorisées avec l'instruction **LOG**.
- Le temps affiche le temps écoulé depuis l'événement, en secondes.

Le menu avec les touches MENU / F6 = OTHER / F3 = LOGS / F6 = WRFILE :

- Permet de transférer l'ensemble des historiques (System, Error, Uniprog, User) au format texte ASCII dans le fichier « E700.LOG » sur le E700.
- Il est possible ensuite d'éditer ce fichier sur le E700 avec le menu MEM, puis les flèches haut et bas jusqu'à sélectionner « E700.LOG », puis menu EDIT.
- Il est encore possible de transférer ce fichier sur un ordinateur pour le visualiser.
- La partie de l'historique User se trouve à partir de la ligne 95 jusqu'à la fin du fichier.
- Le temps qui figure en fin de ligne de chaque historique est donc en cinquième de secondes (200 millisecondes d'unité) depuis la mise en service du E700.

Exemple :

```
;                          CORPS DU PROGRAMME
            LOG                                ; Inscription historique
                                               ; avec numero ligne courante
            LOG    10150                       ; Inscription historique
                                               ; avec numero 10150
             CALL   ECRLOG                     ; Afficher LOGS User
             END                               ; Fin du programme
;
;                               PROCEDURES
;  Procedure retour Ecran Principal, puis affichage historique User:
ECRLOG:      SKS     KMENU                     ; Afficher MENU
ECRLOG_:     SKS     KESC                      ; Simuler touche ESC
             BRIN0   #MACRO[0] ECRLOG_         ; Est-on dans l'ecran 0
             SKS     KMENU                     ; Afficher MENU
             SKS     KF6                       ; MENU/OTHER
             SKS     KF3                       ; MENU/OTHER/LOGS
             SKS     KF4                       ; MENU/OTHER/LOGS/USER
             END                               ; Fin procedure
```

Le résultat des lignes d'historiques écrites dans le fichier « E700.LOG », à la rubrique User en ligne 95 sont les suivantes :

```
Ligne 95 :  User
Ligne 96:          Sim: 0 - Line: 10150,1649028
Ligne 97:          Sim: 0 - Line: 25,1649015
```

Explications de l'exemple en Uniprog et des résultats :

- Deux instructions **LOG** sont effectuées l'une à la suite de l'autre.
- La première instruction **LOG** :
  - se trouve à la ligne 25 dans notre exemple fourni dans le fichier « Exemples Uniprog\LOG\LOG.E7U »,
  - inscrit l'historique que l'on voit à la ligne 97 du fichier « E700.LOG »,
  - on retrouve le simultané 0 dans lequel s'est exécuté le programme Uniprog, la ligne 25 de l'instruction **LOG**, et le temps en 200 *ième* de millisecondes depuis la mise en fonctionnement du E700 sur lequel s'est exécuté le programme.
- La deuxième instruction **LOG** :
  - fournit en paramètre *src* = 10150,
  - on voit 10150 à la ligne 96 du fichier « E700.LOG », ligne historique la plus récente.
  - l'historisation apparait 1 649 028 – 1 649 015 = 13 unités de temps après le premier LOG (13 x 200 ms = 2.6 secondes d'écart, dûes à d'autres instructions insérées entre les deux **LOG** dans le fichier exemple).
  - Le numéro de la ligne du programme n'apparaît pas dans ce cas.
- La procédure ECRLOG permet de simuler l'appuie de touches sur le E700 pour arriver au menu MENU/OTHER/LOGS/USER qui affiche les LOGs User :
  - Cette procédure commence par faire des ESCAPE pour retourner au menu principal, puis effectue la séquence de touches voulue.
  - dans ce menu, on voit sur le E700 le temps en secondes ou en heure depuis l'événement, et non en date absolue comme dans le fichier « E700.LOG ».

MEMR type dst [1]

Instruction système
à ne pas utiliser.

MEMW type src [1]

Instruction système
à ne pas utiliser.

MENU fct msg [scr]

L'instruction **MENU** permet de modifier le libellé des touches de fonctions F1 à F6 d'un menu utilisateur.

- *fct* : De 1 à 6, désigne la touche de fonction concernée, de *fct* = 1 pour F1 à *fct* = 6 pour F6.
- *msg* : numéro du message utilisateur contenant le libellé de la touche de fonction concernée.

  -1 : la touche de fonction *fct* sera inopérante pour le menu Display concerné.

  de 0 à 119 : la chaîne de caractères « m<*msg*> » de 1 à 6 caractères, sera affichée au dessus de la touche de fonction *fct* du Display concerné (« m<*msg*> » définie dans le fichier « MSG.INI »).

  Exemples, utilisation *msg* = 10 et *msg* = 11, déclarations dans « MSG.INI »:

```
[Msg]
m10 = "ARRET"
m11 = "MARCHE"
```

- *scr* : De 0 à 9, est optionnel, spécifie le numéro de l'écran utilisateur Display<*scr*> concerné.

  Si *scr* est omis, c'est l'écran utilisateur en cours d'utilisation qui est modifié (Display0 à la mise sous tension du E700, s'il est défini dans « DISPLAY.INI », ou le Display choisi par la dernière instruction ***DISPS***).

Notes :

Si *msg* = -1, la touche de fonction n'aura pas d'effet, l'instruction ***GETKF*** par exemple ne serait pas alors exécutée avec cette touche de fonction, et elle en attendrait une autre valide (comportant un libellé de 1 à 6 caractères). Un nouvel appel de **MENU** pour cette touche de fonction avec un libellé la réactiverait, et elle serait alors active pour l'instruction ***GETKF*** par exemple.

Exemple :

Fichier « DISPLAY.INI » :

```
[Display0]
l0 = "EIP SA - Exemple instruction MENU:       "
f6  = "FIN"
```

Fichier « MSG.INI » :

```
[Msg]
m11 = "START"
m12 = "STOP"
m20 = "Traitement en cours ..."
```

En Uniprog :

```
;                          INITIALISATIONS
            CALL   FINTRAIT                ; Effacer "Traitement"
;
;                        CORPS DU PROGRAMME
;
BOUCLE:     GETKF  R0                      ; Saisie touche F1, F2, F6
            SWITCH R0                      ; En fonction touche:
              CASE 1  DEBTRAIT  FINSWI     ; Cas F1, lancer DEBTRAIT
              CASE 2  FINTRAIT  FINSWI     ; Cas F2, lancer FINTRAIT
                                           ; Sinon,  F6 = Fin
FINSWI:     ENDS                           ; Fin SWITCH
;
; Si F6 est pressee, effacer "Traitement" puis terminer le programme:
            CMP    R0 6                    ; F6 pressee ?
            JNE    BOUCLE                  ; Non: Boucle attendre touche
            CALL   FINTRAIT                ; Oui: Effacer "Traitement"
            END                            ; Fin du programme
;
;                              PROCEDURES
;
; Procedure DEBTRAIT, affiche "Traitement":
DEBTRAIT:   GTXY   1 3                     ; Curseur en (3,1)
            DISPST 20                      ; Afficher m10
            MENU   1 -1  1                 ; Definir F1 inactive
            MENU   2 12  1                 ; Definir F2 = m12 = "STOP"
            END                            ; Fin procedure DEBTRAIT
;
; Procedure FINTRAIT, efface "Traitement":
FINTRAIT:   GTXY   0 3                     ; Curseur ligne 3
            DISPC  1                       ; Effacer ligne
            MENU   1 11  1                 ; Definir F1 = m11 = "START"
            MENU   2 -1  1                 ; Definir F2 inactive
            END                            ; Fin procedure FINTRAIT
```

Cet exemple en Uniprog utilise le menu et les actions suivantes:

- F1 = « START » :
  - afficher la chaîne de caractères « Traitement » dans m20.
  - inhiber F1 avec l'instruction **MENU** et *msg* = -1
  - définir F2 = « STOP » avec l'instruction **MENU** et *msg* = 12
- F2 = « STOP »
  - effacer la ligne « Traitement »
  - définir F1 = « START » avec l'instruction **MENU** et *msg* = 11
  - inhiber F2 avec l'instruction **MENU** et *msg* = -1
- F6 = « FIN »:
  - effacer la ligne « Traitement »
  - terminer le programme.

(MFILE numéro , Ne pas utiliser)

Ne pas utiliser cette instruction !

.

MOD dst src

L'instruction **MOD** retourne le reste de la division entière de *dst* par *src* (**MOD**ulo).

- *dst* : valeur modifiée, **dst ← dst Modulo src**
- *src* : diviseur, entier positif et non nul.

  L'erreur 14 est générée si *src* vaut 0 (division par zéro), tous les programmes sont arrêtés, la tâche AUTOMAT est relancée après acquittement du message d'erreur par l'utilisateur.

Notes :

Si *dst* est négatif, le résultat de l'instruction **MOD** est négatif :

(-(*dst*) Modulo *src*) = -(*dst* Modulo *src*).

Si *src* est négatif, le résultat de l'instruction **MOD** est négatif :

(*dst* Modulo (-*src*)) = -(*dst* Modulo *src*).

Si *dst* ou *src* ont une partie décimale, il n'en est pas tenu compte :

(*dst* Modulo *src*) = (Int(*dst*) Modulo Int(*src*)), où Int() est l'arrondi à l'entier le plus proche.

.

Exemple :

```
;              DECLARATION CONSTANTES ET VARIABLES
             DST =                          ; Variable DST
;
;                          INITIALISATIONS
             MOV    #DST 3.14159            ; Init DST a 3.14159
             MOV    R0   257                ; Init registre R0 a 257
             MOV    R1   2                  ; Init registre R1 a 2
;
;                         CORPS DU PROGRAMME
             MOD    #DST  R1                ; DST = (3.14159 Modulo 2) = 1
             MOD    R0 3                    ; R0  = (257 Modulo 3) = 2
             END                            ; Fin du programme
```

Cet exemple en Uniprog effectue les actions suivantes :

- Initialiser la variable DST, les registres R0 et R1.
- Calculer (DST Modulo R1).
- Calculer (R0 Modulo 3).

Les fichiers exemples joints dans le répertoire « Exemples Uniprog\MOD », permettent de mettre en œuvre cet exemple sur un E700, sans danger pour l'équipement, et aussi de saisir dividende et diviseur et visualiser le résultat de l'instruction **MOD**.

MOV dst src

L'instruction **MOV** transfert le contenu de *src* dans *dst* (**MOV**e).

- *dst* : variable, registre, ou élément de tableau recevant la valeur de *src*, **dst ← src**
- *src* : nombre source à affecter à *dst*.

  Peut être une valeur littérale (exemple : 3.14159), une constante, une variable, un registre, un élément de tableau.

Notes :

L'erreur R3 est générée (Destination invalide) si *dst* est une valeur littérale ou une constante, tous les programmes sont arrêtés, la tâche AUTOMAT est relancée après acquittement du message d'erreur par l'utilisateur.

Voir aussi :

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
              CNS = 3                          ; Constante CNS vaut 3
              VAL =                            ; Variable VAL
;
;                           CORPS DU PROGRAMME
              MOV    #VAL 3.14159              ; VAL vaut 3.14159
              MOV    R0   #VAL                 ; R0 vaut 3.14159
              MOV    R0   CNS                  ; R0 vaut 3
              MOV    R1   #IN[R0]              ; R1 vaut valeur de IN[3]
              MOV    R1   KESC                 ; R1=Code Touche ESCAPE
              END                              ; Fin du programme
```

Cet exemple en Uniprog montre différentes possibilités dans les types d'arguments src et dst pour l'instruction **MOV** :

- Utilisation de la constante CNS.
- Utilisation de la variable VAL.
- Utilisation des registres R0 et R1.
- Utilisation de l'élément de tableau IN[3], entrée numérique du E700.

*MSG msg

L'instruction **MSG** affiche à l'écran du E700 la chaîne de caractères numéro *msg*, avec demande de confirmation de la part de l'utilisateur (message d'information).

- *msg* : De 0 à 119, numéro du message utilisateur à afficher.

  La chaîne de caractères « m<*msg*> » aura été définie dans le fichier « MSG.INI » (exemple: *msg* = 10, définition « m10 = "Texte exemple" » dans « MSG.INI »).

Notes :

Si la définition d'une variable utilisateur est omise et que le m<*msg*> n'existe pas, il est affiché le message d'erreur R17 « Msg inexistant », tous les programmes sont arrêtés, la tâche AUTOMAT est relancée après acquittement du message d'erreur par l'utilisateur (exemple: si *msg* = 3 et m3 non défini dans « MSG.INI », alors affichage du message d'erreur R17 « Msg inexistant »).

L'instruction **MSG** fonctionne comme l'instruction ***ERROR***, mis à part que pour **MSG** toutes les tâches Uniprog et les mouvements continuent. C'est le programmeur qui gère les actions à exécuter lors de l'affichage puis de la quittance de ce type de message par l'utilisateur.

L'instruction MSG affiche le message à partir de la ligne 1 colonne 2 ((1 , 2) et non (0 , 0)).

Le texte du message peut comporter des sauts de lignes (caractère « \n »), ou des noms de variables en les préfixant du caractère « # ».

Exemple : m3 = "Titre (ligne 1)\nVariable Val = #VAL (ligne 2)", et si VAL vaut 10, « ***MSG*** 3 » afficherait deux lignes avec la valeur de VAL à la place des caractères « #VAL »:

```
Titre (ligne 1)
Variable Val = 10 (ligne 2)
```

Voir aussi Instruction ***ERROR***.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m1  = "La vitesse en X a depassee 0.5\n   et vaut #FVACT[X]"
```

En Uniprog :

```
; Attendre vitesse axe X superieur a 0.5, et afficher message m1:
BOUCLE:     CMP     #FVACT[X] 0.5          ; Vitesse X >= 0.5 ?
            JL      BOUCLE                 ; non: boucler a BOUCLE
            MSG     1                      ; oui: afficher m1
;
            END                            ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Boucle d'attente pour que la vitesse de l'axe 0 (axe X) soit plus grande que 0.5 unités.
- Lorsque l'axe X a atteint ou dépassé la vitesse 0.5 unités, afficher la chaîne de caractères m1 qui informe l'utilisateur.

MUL dst src

L'instruction **MUL** effectue la multiplication de ses deux arguments et retourne le résultat dans le premier argument.

- *dst* : valeur modifiée, **dst ← dst x src**
- *src* : valeur à multiplier avec *dst*.

Exemple :

```
;              DECLARATION CONSTANTES ET VARIABLES
               CNS = 12                        ; Constante CNS vaut 12
               VAL =                           ; Variable VAL
;
;                           INITIALISATIONS
               MOV    R0   1050                ; R0 vaut 1050
               MOV    #VAL 0.027               ; VAL vaut 0.027
;
;                           CORPS DU PROGRAMME
               MUL    R0   #VAL                ; R0 <-R0 x VAL = 28.35
               MUL    R0   CNS                 ; R0 <-R0 x CNS = 340.2
               MUL    #VAL 3.14159             ; R0 <-R0 x VAL = 0.08482293
               MUL    R0   #IN[3]              ; R0 <-R0 x IN[3]:
                                               ; = 240.2 si entree IN[3]=1
                                               ; = 0     si entree IN[3]=0
;
               END                             ; Fin du programme
```

Cet exemple en Uniprog utilise le registre R0 et une variable VAL :

- On multiplie le registre R0 par la variable VAL (préfixée par le caractère « # »).
- On multiplie le registre R0 par la constante CNS (non préfixée par le caractère « # »).
- On multiplie la variable VAL par la valeur immédiate 3.14159.
- On multiplie le registre R0 par l'élément de tableau IN[3] (entrée numérique 3, préfixée par le caractère « # »).

MULD src

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **MULD** multiplie l'accumulateur de la tâche courant*e accum[PNB]* par *src* et met le résultat dans l'accumulateur de la tâche courante (**MUL**tiplication **D**irecte).

- *src* : nombre multiplicateur, **accum[PNB] ← accum[PNB] x src**.
- *accum[PNB]* : représente la mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, non utilisable directement en Uniprog).

  Est modifié par l'instruction pour contenir le résultat.

Exemple:

```
;               DECLARATION CONSTANTES ET VARIABLES
              VAL =                          ; Variable
;
;                           CORPS DU PROGRAMME
              MOV     #VAL 30                ; VAL <- 30
              LOAD    #VAL                   ; Ici accum[PNB] = 30
              MULD    3                      ; Ici accum[PNB] = 30x3 = 90
              MOV     #VAL 5                 ; VAL <- 5
              MULD    #VAL                   ; Ici accum[PNB] = 90x5 = 450
              STORE   #VAL                   ; VAL<-accum[PNB] (VAL = 450)
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Initialiser la variable VAL avec 30
- Effectuer les opérations VAL <- ((VAL * 3) * 5) en utilisant l'accumulateur.

NEG dst

L'instruction **NEG** inverse le signe de la variable passée en argument *dst* (**NEG**ate).

- *dst* : valeur modifiée, **dst ← - dst**.

Exemple :

```
;              DECLARATION CONSTANTES ET VARIABLES
             VAL =                          ; Variable VAL
;
;                          INITIALISATIONS
             MOV    R0   8                  ; Init registre R0 a 8
             MOV    #VAL -0.05              ; Init VAL a -0.05
;
;                         CORPS DU PROGRAMME
; Inverser les signes de R0 et VAL:
             NEG    R0                      ; R0  vaut -8
             NEG    #VAL                    ; VAL vaut 0.05
;
             END                            ; Fin du programme
```

Cet exemple en Uniprog utilise le registre R0 et une variable VAL :

- On affecte 8 à R0 et -0.05 à VAL.
- On inverse le signe de R0 qui contient alors -8.
- On inverse le signe de VAL qui contient alors 0.05.

NOP

L'instruction **NOP** n'effectue aucune opération, et passe à l'instruction suivante (**N**o **OP**eration).

Notes :

La présence de l'instruction **NOP** est générale à la plupart des langages de programmation.

Dans le cas où on privilégie la lisibilité des programmes, l'instruction **NOP** peut être employée en toute liberté, pour mettre en valeur un label ou le début d'une fonction :

```
MONLABEL:   NOP                         ; Debut bloc instructions MONLABEL
```

NOP peut être utilisée dans le cas de boucles sans instructions ou pour attentes : l'instruction ***WAIT*** est mieux adaptée à ce type de fonctionnement, sauf si l'on veut rendre volontairement le processeur actif (cas de tests ou autres …).

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
;
;                         INITIALISATIONS
;
;                        CORPS DU PROGRAMME
;
            CALL   BOUCLENOP                ; Lancer procedure BOUCLENOP
;
            END                             ; Fin du programme
;
;                             PROCEDURES
;
BOUCLENOP:  NOP                             ; Debut procedure BOUCLENOP
            REP    10000                    ; Repeter 10 000 fois
              NOP                           ; Pas d'action
            ENDRP                           ; Fin repetition
            END                             ; Fin procedure BOUCLENOP
```

Cette exemple en Uniprog utilise la procédure BOUCLENOP contenant une boucle avec l'instruction **NOP**, cette procédure a pour seule activité d'utiliser le processeur.

NOT dst

L'instruction **NOT** inverse chacun des bits de la valeur entière de l'argument *dst* (notée aussi ~).

Le not logique est tel qu'un bit a pour résultat 1 si la source vaut 0, et pour résultat 0 si la source vaut 1 :

- not 0 = 1
- not 1 = 0

- *dst* : valeur modifiée avec son complément bit à bit, **dst** ← ~ **dst**.

Exemple en logique :

~ 3.14159 = ~ 3 (on prend la valeur entière arrondie)

= ~ 0000 0000 0000 0000 0000 0000 0000 0011

= 1111 1111 1111 1111 1111 1111 1111 1100

= -4

Donc ~ 3.14159 = -4

Exemple en Uniprog :

```
MOV    R0 #IN[0]
NOT    R0
MOV    #OUT[0]
END
```

Cet exemple charge dans la sortie 0 le contraire de l'entrée 0. Si IN[0] vaut 0, alors OUT[0] vaudra 1 et si IN[0] vaut 1, alors OUT[0] vaudra 0.

OFF dst

L'instruction **OFF** attribue la valeur 0 à son argument *dst*.

- *dst* : variable remise à zéro, **dst ← 0**.

Voir aussi Instruction ***ON***,

Exemple :

Pour remettre à zéro chacune des 8 sorties numérique OUT[ ], il suffirait d'écrire :

```
REP    8                        ; Repeter 8 fois
  OFF  #OUT[R9]                 ; Remise a zero sortie OUT[i]
ENDRP                           ; Fin boucle
```

ON dst

L'instruction **ON** attribue la valeur 1 à son argument *dst*.

- *dst* : variable positionnée à 1, **dst ← 1**.

Voir aussi Instructions ***OFF***.

Exemple :

Pour remettre à un chacune des 8 sorties numérique OUT[ ], il suffirait d'écrire :

```
REP     8                       ; Repeter 8 fois
  ON    #OUT[R9]                ; Mise a 1 sortie OUT[i]
ENDRP                           ; Fin boucle
```

OR dst src

L'instruction **OR** effectue le OU logique de ses deux arguments (**OR**, noté aussi « | »), et retourne le résultat dans le premier de ces arguments.

- *dst* : valeur modifiée, **dst ← dst OR src**
- *src* : autre valeur argument du **OR**.

Exemple en logique :

```
3.14159 | -589.545      =     3 | -590 (valeurs entières arrondies)

                         =     0000 0000  0000 0000  0000 0000  0000 0011
                         |     1111 1111  1111 1111  1111 1101  1011 0010

                         =     1111 1111  1111 1111  1111 1101  1011 0011

                         =     -589
```

Donc 3.14159 | -590.545 = -589

Exemple en Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
              CNS = 11                        ; Constante CNS vaut 11
              VAL =                           ; Variable VAL
;
;                           INITIALISATIONS
              MOV     R0    3.14159           ; R0 vaut 3.14159
              MOV     #VAL -589.545           ; VAL vaut -589.545
;
;                          CORPS DU PROGRAMME
              OR      R0    #VAL              ; R0  <-    3 | -590 = -589
              OR      R0    CNS               ; R0  <- -589 | 11   = -581
              OR      #VAL #IN[3]             ; VAL <- -590 | IN[3]:
                                              ; = -589 si entree IN[3]=1
                                              ; = -590 si entree IN[3]=0
              OR      #VAL 2.71828            ; VAL <- VAL | 3 = -589
                                              ; quelle que soit IN[3]
              END                             ; Fin du programme
```

Cet exemple montre les différentes opérations logiques avec valeurs immédiates, constantes, registres, variables, élément de tableau correspondant à une entrée numérique.

Pour surveiller si l'une des entrées numériques IN[0] et IN[1] est à 1 (deux portes de sécurité par exemple), il suffirait d'écrire :

```
BOUCLE:       MOV     R0 #IN[0]               ; Lire entree IN[0]
              OR      R0 #IN[1]               ; OR avec entrée IN[1]
              BRIN0   R0 BOUCLE               ; IN[0] et IN[3] a 0: boucler
;
              ERROR   ...                     ; Gestion du cas
```

**PECK mode axe prof tempo passe garde vitesse

L'instruction **PECK** effectue des cycles de perçages, quatre types sont possibles :

- le perçage simple : avec remontée rapide ;
- l'alésage : avec remontée à vitesse de perçage ;
- le perçage avec débourrage : perçage en plusieurs passes avec remontée rapide à la surface entre deux passes ;
- le perçage avec brise-copeaux : perçage en plusieurs passes avec petit dégagement entre chaque passe.

Ces quatre types de perçages sont expliqués dans une documentation qui décrit les cycles de perçage, en annexe 3 de la documentation de l'ISO sur le E700.

- *mode :* Cet argument définit le mode de perçage à mettre en œuvre :

  81 pour un perçage simple ou pour un perçage avec débourrage (G81 en ISO).

  82 pour un alésage (G82 en ISO).

  83 pour un perçage avec brise-copeaux (G83 en ISO).

- *axe :* Numéro ou nom de l'axe de perçage.

- *prof :* Profondeur de perçage, valeur en position absolue et signée.

  L'origine du référentiel sur l'axe axe doit correspondre à la surface de la pièce à usiner (début de perçage) : L'emploi de l'instruction ***TOOL*** pour les décalages d'origines, ou de l'instruction ***G5358***, permettent d'adapter les référentiels d'origine avant l'instruction **PECK** pour que le zéro corresponde au début de la surface de perçage.

- *tempo :* Temps d'attente au fond du trou, en secondes.

- *passe :* Profondeur pour chacune des passes, valeur en position relative et non signée (positive).

  Mettre *passe* = *prof* pour un perçage simple (sans débourrage) ou pour un alésage.

  Contient la profondeur d'une passe pour les perçages avec débourrage ou avec brise-copeaux : dans ce cas, c'est le système du E700 qui calcule automatiquement le nombre de passes.

- *garde :* Définit la garde avant d'atteindre la surface de perçage et à partir de laquelle l'outil prend la vitesse de perçage et non la vitesse de positionnement, valeur en position relative et non signée (positive).

  Mettre *garde* = 0 dans les case de perçage simple (sans débourrage) ou de perçage avec un alésage.

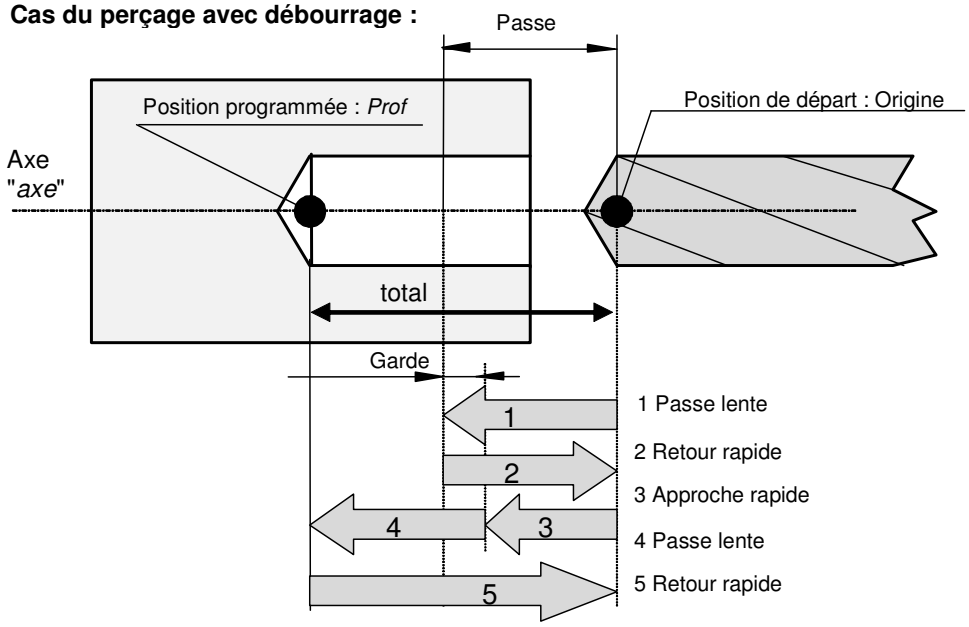
- *vitesse :* Vitesse de plongée selon l'axe *axe* (appelée aussi vitesse de travail ou de perçage). La remontée se fait à vitesse maximum pour l'axe *axe*, celle décrite dans la configuration.

Notes :

Le cycle de perçage est le suivant, en considérant les mouvements de l'outil sur l'axe *axe*, et sachant que l'origine du référentiel sur l'axe *axe* correspond à la surface de la pièce à usiner (début de perçage) :

- Au début du cycle, positionner l'outil à vitesse maximale de sa position actuelle jusqu'à l'origine (zéro de l'axe *axe*) plus la garde *garde*.
- Le perçage débute à la vitesse de travail *vitesse*.
- Dans le cas d'un perçage avec débourrage ou avec brise-copeaux, plusieurs phases intermédiaires s'enchaînent :

Cas du perçage avec débourrage :



  - Perçage à vitesse lente d'une profondeur de *passe* unités (phase 1).
  - Interruption du perçage avec remontée à vitesse maximale :
    - A l'origine plus *garde* pour le perçage avec débourrage (phase 2).
    - D'une hauteur relative de *garde* pour le perçage brise-copeaux ;
  - Puis retour à la dernière profondeur moins *garde* à vitesse maximale (phase 3 pour le débourrage, on est en position pour le brise-copeaux).
  - Reprise du perçage à vitesse de travail pour une nouvelle profondeur relative de *passe* unités (phase 4).
  - On reprend les Phases 1 à 4 par profondeurs successives de perçage de *passe* unités, jusqu'à atteindre la profondeur désirée *prof*.
- Lorsque la profondeur de perçage est atteinte, attendre *tempo* secondes au fond du trou.
- Remonter à la position de l'origine (zéro de l'axe *axe*) plus *garde* :
  - À vitesse de travail pour le perçage alésage ;
  - À vitesse maximale pour les autres perçages (Phase 5 pour les cas de débourrage ou brise-copeaux).
- Remonter à la position au départ de l'instruction **PECK** à la vitesse maximale.

Exemple :

En Uniprog :

```
;              DECLARATION CONSTANTES ET VARIABLES
               Z_MVT   =  10                    ; Position Z en mouvements
               V_USI   =  0.2                   ; Vitesse usinage
               PROF    = -10                    ; Profondeur usinage
               TEMPO   =  3                     ; Attente au fond
               PASSE   =  4                     ; Passe percage
               GARDE   =  2                     ; Garge percage
;
;                          CORPS DU PROGRAMME
               POSA    Z #VMAX[Z] Z_MVT 1       ; Aller en Z=+10 go
               POSR    X #VMAX[X]  10 0         ; Aller en X=+10 attente
               POSR    Y #VMAX[Y] -10 2         ; Aller en Y=-10 go
               POSA    Z #VMAX[Z] GARDE 1       ; Aller en Z=+02 go
               PECK    81 Z PROF TEMPO PROF 0 V_USI
                                                ; Percage en Z=-20
                                                ; 3 s fond trou, vitesse 0.5
                                                ; passe -22, garde 0
               POSA    Z #VMAX[Z] Z_MVT 1       ; Aller en Z=+10 go
               POSR    X #VMAX[X]  10 0         ; Aller en X=+10 attente
               POSR    Y #VMAX[Y] -10 2         ; Aller en Y=-10 go
               POSA    Z #VMAX[Z] GARDE 1       ; Aller en Z=+02 go
               PECK    82 Z PROF TEMPO PROF 0 V_USI
                                                ; Alesage en Z=-20
                                                ; 3 sec. puit, vitesse 0.5
                                                ; passe -22, garde 0
                                                ; vitesse 0.5
               POSA    Z #VMAX[Z] Z_MVT 1       ; Aller en Z=+10 go
               POSR    X #VMAX[X]  10 0         ; Aller en X=+10 attente
               POSR    Y #VMAX[Y] -10 2         ; Aller en Y=-10 go
               POSA    Z #VMAX[Z] GARDE 1       ; Aller en Z=+02 go
               PECK    83 Z PROF TEMPO PASSE GARDE V_USI
                                                ; Debourrage en Z=-20
                                                ; 3 sec. puit, vitesse 0.5
                                    ; passe -22, garde 0
               POSA    Z #VMAX[Z] Z_MVT 1       ; Aller en Z=+10 go
               POSR    X #VMAX[X]  10 0         ; Aller en X=+10 attente
               POSR    Y #VMAX[Y] -10 2         ; Aller en Y=-10 go
               POSA    Z #VMAX[Z] GARDE 1       ; Aller en Z=+02 go
               PECK    81 Z PROF TEMPO PASSE GARDE V_USI
                                                ; Brise-Copeaux en Z=-20
                                                ; 3 s fond trou, vitesse 0.5
                                                ; passe -22, garde 0
;
               END                              ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Initialise les axes comme référencés et sans mouvements.
- Effectue ensuite 4 perçages : simple, alésage, avec débourrage, et avec brise-copeaux.

POP dst

L'instruction **POP** dépile la dernière valeur empilée dans la pile de la tâche simultanée courante, et la retourne dans l'argument *dst*.

- *dst* : contient la valeur dépilée.

Notes :

La valeur a été préalablement empilée à l'issue de l'instruction ***PUSH***, ou bien à l'issue d'une instruction d'appel de sous-programme ***CALL***, ***CALIN0***, ***CALIN1***.

Si la pile est vide pour le simultané courant, l'instruction **POP** génère une erreur d'exécution numéro 10 « Stack Underflow » (Pile vide), tous les programmes s'interrompent, la tâche AUTOMAT est relancée après acquittement du message d'erreur par l'utilisateur.

Voir aussi Variables systèmes STACK, SP, Instruction ***PUSH***.

Exemple :

```
;                DECLARATION CONSTANTES ET VARIABLES
;
;                          CORPS DU PROGRAMME
             MOV     R0 3.14159              ; R0 vaut 3.14159
;
             CALL    EXEMPLE                 ; Lancer procedure EXEMPLE
                                             ; Ici on retrouve:
                                             ;   R0 = 3.14159
;
             END                             ; Fin du programme
;
;                              PROCEDURES
; Procedure EXEMPLE qui utilise PUSH et POP sur R0 et R1
EXEMPLE:     PUSH    R0                      ; Sauvegarde R0
             PUSH    R1                      ; Sauvegarde R1
;
             MOV     R0 2.71828              ; R0 vaut 2.71828
             MOV     #VAL[R8] 10             ; VAL[R8] vaut 10
             MOV     #LIGNE 4                ; Affichage ligne 4
             CALL    AFFICHE                 ; Afficher R0

             POP     R1                      ; restituer R1
             POP     R0                      ; restituer R0
             END                             ; Fin procedure EXEMPLE
```

Afin de pouvoir appeler cette procédure sans se soucier de l'utilisation ou non de la valeur du registre R0, on sauvegarde (PUSH) cette valeur en début de procédure, on la modifie pendant l'exécution de la procédure puis on la restitue (POP) en fin de procédure.

Dans cet exemple, la sauvegarde (PUSH) et la restitution (POP) de R1 est inutile puisque la valeur de R1 n'y est pas modifiée. La volonté de mettre R1 ici est de montrer le fonctionnement de la pile du type FIFO (first in first out ou premier entré, premier sorti). On voit que si on empile R**0** puis R**1**, il faudra dépiler d'abord R**1** puis ensuite R**0**.

*POSA axe vitesse pos mode

L'instruction **POSA** effectue un positionnement en coordonnées absolues sur un axe (cf. les fonctions ISO G90 puis G0).

- *axe* : Numéro ou nom de l'axe pour lequel on détermine la position *val* à atteindre sur cet axe.
- *vitesse :* consigne de vitesse pour le déplacement en unités de vitesse.

  On peut donner à *vitesse* la valeur de la variable système VMAX pour que l'axe *axe* considéré (exemple : *vitesse* = #VMAX[X] pour déplacement à vitesse maximale sur l'axe X). Ce qui correspond à un G0 en ISO.
- *pos :* valeur donnant la coordonnée absolue à atteindre sur l'axe *axe*, en unités de longueur, et relativement aux référentiels d'origines actuellement utilisés.
- *mode* contient le mode opératoire de l'instruction **POSA** :

  0 : Mode préparatoire, le mouvement n'est pas lancé (cas de plusieurs axes à définir avant mise en œuvre). Le mode 0 est donc toujours accompagné d'éventuellement d'autres modes 0 et d'exactement un mode 2.

  1 : Mode exécution, déplacement individuel sur un axe (déplacement lancé).

  2 : Mode positionnement simultané, le mouvement est lancé pour tous les axes définis en mode préparatoire et pour l'axe défini par ce mode simultané (mouvements lancés, chacun des axes va à sa vitesse et indépendamment les uns des autres). Le mode 2 est donc toujours précédé d'un ou plusieurs modes 0.

  3 : Mode positionnement sans attente, déplacement individuel avec lancement du mouvement, l'instruction **POSA** n'attend pas la fin du mouvement pour passer à l'instruction suivante (contrairement aux modes 1 et 2 pour lesquels l'instruction qui suit le **POSA** ne sera effectuée qu'à la fin du positionnement demandé)

Notes :

L'instruction **POSA** en mode préparatoire permet de lancer un positionnement simultané sur plusieurs axes :

- une instruction **POSA** par axe en mode préparatoire (mode 0) ;
- l'instruction **POSA** pour le dernier axe en mode positionnement simultané (mode 2).

Dans le cas d'un positionnement simultané, les axes vont chacun à leur rythme : si l'on veut s'assurer de trajectoires d'outils spécifiques (éviter des obstacles, suivre une gorge, …), il est donc nécessaire d'utiliser les instructions d'interpolation pour les déplacements.

Le mode 1 sur plusieurs axes définit un déplacement séquentiel (un axe après l'autre), et le mode 3 permet un déplacement à la demande qui se traduit par un déplacement simultané si un nouveau **POSA** en mode 3 est effectué alors qu'un précédent sur un autre axe n'est pas encore terminé.

Voir aussi Instructions ***POSO***, ***POSR***.

Exemple :

En Uniprog :

```
;                          CORPS DU PROGRAMME
;
                POSA   X #VMAX[X]  10 1          ; G0 X10
                POSA   Y #VMAX[Y] -10 1          ; G0 Y-10
                DISPC  7                         ; BEEP en fin de mouvement
;
                POSA   X #VMAX[X]  10 0          ;
                POSA   Y #VMAX[Y]  10 2          ; G0 X10 Y10
                DISPC  7                         ; BEEP en fin de mouvement
;
                POSA   X #VMAX[X] -10 3          ;
                POSA   Y #VMAX[Y] -10 3          ; G0 X-10 Y-10 sans attente
LOOP:           DISPC  7                         ; BEEPs pendant le mouvement
                BRIN1  #RUN[X] LOOP
                END                              ; Fin du programme
```

*POSO axe vitesse pos mode

L'instruction **POSO** effectue un positionnement en coordonnées absolues sur un axe, par rapport aux prises de références machines (instruction ***REF***) et sans tenir compte des référentiels d'origines actuellement utilisés.

- *axe* : Numéro ou nom de l'axe pour lequel on détermine la position *val* à atteindre sur cet axe.
- *vitesse :* consigne de vitesse pour le déplacement en unités de vitesse.

  Le plus souvent, on donne a *vitesse* la valeur de la variable système VMAX pour l'axe *axe* considéré (exemple : *vitesse* = #VMAX[0] pour la vitesse maximale de déplacement sur l'axe 0). Ce qui correspond à un G0 en ISO.
- *pos :* valeur donnant la coordonnée absolue à atteindre sur l'axe *axe*, en unités de longueur, et sans tenir compte des déplacements d'origines actuellement utilisés.
- *mode* contient le mode opératoire de l'instruction **POSO** :

  0 : Mode préparatoire, le mouvement n'est pas lancé (cas de plusieurs axes à définir avant mise en œuvre).

  1 : Mode exécution, déplacement individuel sur un axe (déplacement lancé).

  2 : Mode positionnement simultané, le mouvement est lancé pour tous les axes définis en mode préparatoire et pour l'axe défini par ce mode simultané (mouvements lancés, chacun des axes va à sa vitesse et indépendamment les uns des autres).

  3 : Mode positionnement sans attente, déplacement individuel avec lancement du mouvement, l'instruction **POSO** n'attend pas la fin du mouvement pour passer à l'instruction suivante (contrairement aux modes 1 et 2 pour lesquels l'instruction qui suit le **POSO** ne sera effectuée qu'à la fin du positionnement demandé)

Notes :

L'instruction **POSO** en mode préparatoire permet de lancer un positionnement simultané sur plusieurs axes :

- une instruction **POSO** par axe en mode préparatoire (mode 0) ;
- l'instruction **POSO** pour le dernier axe en mode positionnement simultané (mode 2).

Dans le cas d'un positionnement simultané, les axes vont chacun à leur rythme : si l'on veut s'assurer de trajectoires d'outils spécifiques (éviter des obstacles, suivre une gorge, …), il est donc nécessaire d'utiliser les instructions d'interpolation pour les déplacements.

Le mode 1 sur plusieurs axes définit un déplacement séquentiel (un axe après l'autre), et le mode 3 permet un déplacement à la demande qui se traduit par un déplacement simultané si un nouveau **POSO** en mode 3 est effectué alors qu'un précédent sur un autre axe n'est pas encore terminé.

Voir aussi Instructions ***POSA***, ***POSR***.

Exemple :

```
;                          CORPS DU PROGRAMME
;
            REF    X                          ; Prise de reference axe X
            REF    Y                          ; Prise de reference axe Y

            POSO   X #VMAX[X]   50  0         ;
            POSO   Y #VMAX[Y]  -50  2         ; G0 X50 Y-50

            POSO   X #VMAX[X] 0  1            ; G0 X0
            POSO   Y #VMAX[Y] 0  1            ; Y0


            G5358  54                         ; G54
;
; Positionnement degagement en X et Y sans tenir compte des origines:

            POSO   X #VMAX[X]   50  0         ; G53
            POSO   Y #VMAX[Y]  -50  2         ; G0 X50 Y-50
;
            END                               ; Fin du programme
```

On peut voir dans cet exemple que les mouvements effectués avant ou après le déplacement d'origine G54 sont les mêmes. Que l'on applique ou pas G54, POSO se positionne toujours par rapport à la cote de référence machine.

*POSR axe vitesse pos mode

L'instruction **POSR** effectue un positionnement en coordonnées relatives sur un axe (cf. les fonctions ISO G91 puis G0).

- *axe* : Numéro ou nom de l'axe pour lequel on détermine la position *val* à atteindre sur cet axe.
- *vitesse :* consigne de vitesse pour le déplacement en unités de vitesse.

  Le plus souvent, on donne a *vitesse* la valeur de la variable système VMAX pour l'axe *axe* considéré (exemple : *vitesse* = #VMAX[0] pour la vitesse maximale de déplacement sur l'axe 0). G0 en ISO.
- *pos :* valeur donnant la coordonnée à atteindre sur l'axe *axe* relativement à la position actuelle, en unités de longueur.
- *mode* contient le mode opératoire de l'instruction **POSR** :

  0 : Mode préparatoire, le mouvement n'est pas lancé (cas de plusieurs axes à définir avant mise en œuvre).

  1 : Mode exécution, déplacement individuel sur un axe (déplacement lancé).

  2 : Mode positionnement simultané, le mouvement est lancé pour tous les axes définis en mode préparatoire et pour l'axe défini par ce mode simultané (mouvements lancés, chacun des axes va à sa vitesse et indépendamment les uns des autres).

  3 : Mode positionnement sans attente, déplacement individuel avec lancement du mouvement, l'instruction **POSR** n'attend pas la fin du mouvement pour passer à l'instruction suivante (contrairement aux modes 1 et 2 pour lesquels l'instruction qui suit le **POSR** ne sera effectuée qu'à la fin du positionnement demandé)

Notes :

L'instruction **POSR** en mode préparatoire permet de lancer un positionnement simultané sur plusieurs axes :

- une instruction **POSR** par axe en mode préparatoire (mode 0) ;
- l'instruction **POSR** pour le dernier axe en mode positionnement simultané (mode 2).

Dans le cas d'un positionnement simultané, les axes vont chacun à leur rythme : si l'on veut s'assurer de trajectoires d'outils spécifiques (éviter des obstacles, suivre une gorge, …), il est donc nécessaire d'utiliser les instructions d'interpolation pour les déplacements.

Le mode 1 sur plusieurs axes définit un déplacement séquentiel (un axe après l'autre), et le mode 3 permet un déplacement à la demande qui se traduit par un déplacement simultané si un nouveau **POSR** en mode 3 est effectué alors qu'un précédent sur un autre axe n'est pas encore terminé.

Voir aussi Instructions ***POSA***, ***POSO***.

Exemple :

Soit le fichier « DISPLAY.INI » suivant :

```
[Display0]
l0 = "EIP SA - Exemple instruction POSR:       "
l2 = "  POSITIONNEMENTS RELATIFS SUCCESSIFS    "
l3 = "    ( presser START pour commencer)      "
f1 = "SUITE"
```

En Uniprog :

```
;                          CORPS DU PROGRAMME
;
              POSR   X #VMAX[X]  10 0        ; Decaler X de +10 attente
              POSR   Y #VMAX[Y] -10 2        ; Decaler Y de -10 go
              WAITK  R0                      ; Attente touche F1=SUITE
;
              POSR   X #VMAX[X]  10 0        ; Decaler X de +10 attente
              POSR   Y #VMAX[Y] -10 2        ; Decaler Y de -10 go
              WAITK  R0                      ; Attente touche F1=SUITE
;
              POSR   X #VMAX[X]  10 0        ; Decaler X de  10 attente
              POSR   Y #VMAX[Y] -10 2        ; Decaler Y de -10 go
              WAITK  R0                      ; Attente touche F1=SUITE
;
              POSR   X #VMAX[X] -30 0        ;
              POSR   Y #VMAX[Y]  30 2        ; Retour au depart
              WAITK  R0                      ; Attente touche F1=SUITE
;
              END                            ; Fin du programme
```

Cet exemple effectue successivement 4 fois les opérations suivantes, à des adresses en X et en Y différentes :

- Positionnement relatif en X à vitesse maximale et attente avant mise en œuvre.
- Positionnement relatif en Y à vitesse maximale et lancement du positionnement (en X et Y).
- Attente de la touche F1 = SUITE pour passer au positionnement suivant.

PSIM [sim]

L'instruction **PSIM** met en pause la tâche simultanée *sim*, ou si *sim* est omis, met en pause l'ensemble des tâches simultanées en cours à l'exception de la tâche courante (qui exécute **PSIM**) et de la tâche 9 AUTOMAT. L'instruction ***RSIM*** réactive les tâches simultanées en pause.

- *sim :* De 0 à 8, est le numéro de la tâche simultanée à mettre en pause.

  Si *sim* est omis, les tâches simultanées hors tâche courante et 9 sont mises en pause.

Notes :

Si aucune autre tâche simultanée n'est présente en exécution en dehors de l'AUTOMAT et de la tâche courante, l'instruction **PSIM** est sans effet.

Voir aussi Instructions ***RSIM*.**

Exemple :

```
            REP     5                       ; Pour LED i, i de 1 a 6
             ASIM   SIM                     ; Lancer tache parallele SIM
            ENDRP                           ; Fin boucle repetition
;
; Figer les taches paralleles si IN[0] vaut 0, les liberer sinon:
BOUCLE:     WAIT0   #IN[0]                  ; Attente si IN[0] = 0
            PSIM                            ; Figer autres taches
            WAIT1   #IN[0]                  ; Attente si IN[0] = 1
            RSIM                            ; Liberer autres taches
            JMP     BOUCLE
;
; Tache SIM : fait clignoter les sorties
SIM:        CPL     #OUT[R8]
            WAIT    0.5                     ; Clignotement de 1 seconde
            JMP     SIM                     ; Boucler sans fin
```

Cet exemple en Uniprog fait clignoter les sorties. L'entrée IN[0] permet de figer l'ensemble des clignotements (si IN[0] est à 1) ou de réactiver les clignottements si IN[0] repasse à 0.

PUSH src

L'instruction **PUSH** empile la valeur désignée par *src* dans la pile de la tâche simultanée courante.

- *src* : contient la valeur à empiler.

Notes :

La valeur sera ensuite dépilée avec l'instruction ***POP***.

Si la pile est pleine pour le simultané courant, l'instruction **PUSH** génère une erreur d'exécution numéro 6 « Stack Overflow » (pile pleine), tous les programmes s'interrompent, la tâche AUTOMAT est relancée après acquittement du message d'erreur par l'utilisateur.

Voir aussi Variables systèmes STACK, SP, Instruction ***POP***.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
              VAL   =                         ; Variable VAL
;
;                          CORPS DU PROGRAMME
              MOV     R0    3.14159           ; R0 vaut 3.14159
              MOV     #VAL -589.545           ; VAL vaut -589.545
;
              CALL    EXEMPLE                 ; Lancer procedure EXEMPLE
                                              ; Ici on retrouve R0 = 3.14159
;
              END                             ; Fin du programme
;
;                               PROCEDURES
; Procedure EXEMPLE qui utilise PUSH et POP sur R0 et R1:
EXEMPLE:      PUSH    R0                      ; Sauvegarde R0
              PUSH    R1                      ; Sauvegarde R1
;
              MOV     R0   2.71828            ; R0 vaut 2.71828
              MOV     R1   10                 ; R1 vaut 10
              MOV     #LIGNE   4              ; Affichage ligne 4
              CALL    AFFICHE                 ; Afficher R0 et VAL
;
              POP     R1                      ; Restituer R1
              POP     R0                      ; Restituer R0
              END                             ; Fin procedure EXEMPLE
```

Cet exemple en Uniprog utilise une variable VAR et le registre R0 dans le programme principal.

La procédure EXEMPLE utilise elle aussi le registre R0, mais sauvegarde sa valeur au tout début avec l'instruction **PUSH** pour ensuite travailler librement avec ce registre. À la fin de la procédure, il y a restitution de R0 avec l'instruction ***POP*** : R0 reprend la valeur qu'il avait avant l'instruction **PUSH** au début de EXEMPLE.

De retour dans le programme principal après l'appel à EXEMPLE, R0 a conservé sa valeur.

*QREF axe

L'instruction **QREF** (Quick reference) effectue la prise de références simplifiée sur l'axe *axe*. Cette prise de référence ne se fait que sur l'index du moteur, sans passer par le détecteur de proximité. Elle ne fonctionne que pour des axes servos montés sur des cartes E700-AE2 ou E700-AE4.

- *axe* : Numéro ou nom de l'axe sur lequel faire les prises de références simplifiées.

Notes :

Les prises de références se font avec les paramètres définis dans la configuration du E700 et accessibles par les touches MENU / F5 = CONFIG / F3 = AXES / F4 = REF, le choix de l'axe se fait par sa lettre ou avec les touches F3 = AXE- et F4 = AXE+.

Les opérations effectuées lors d'une prise de référence se font en trois phases, avec les vitesses rapides (vitesse max G0) et lentes et les directions définies dans la configuration des prises de références de l'axe :

- Phase 1 : Aller à 180 degrés de l'index en vitesse rapide (vitesse max ou G0). Le sens pour y aller est indéterminé. Si cette position est trop proche des limites de course, on s'en éloigne de deux tours. Puis on s'approche dans le sens de la prise de référence normale (voir REF) à vitesse rapide à 90 degrés de l'index.
- Phase 2 : La bascule est réarmée de manière logicielle. On recherche l'index à vitesse lente dans le même sens que celui de la référence habituelle (voir REF).
- Phase 3 : Arrêt instantané dès que l'index du moteur est atteint.

Le potentiomètre FEED est actif pour les vitesses rapides, mais les vitesses lentes ne tiennent pas compte de FEED (répétabilité des prises de références).

Il est bien clair que cette référence simplifiée ne remplace pas la référence habituelle. L'idée d'une référence simplifiée est que si en cas d'urgence, les moteurs doivent être coupés pour des raisons de sécurité et que ceux-ci ne bougent pas ou presque durant cet arrêt, alors, par une référence simplifiée, on peut se recaler sur l'index sans retourner au détecteur de proximité et ainsi gagner du temps. Cette méthode a ses limites, notamment si le moteur bouge trop lors de sa mise hors tension. La méthode de recherche de l'index est rigoureusement identique à celle utilisée dans l'instruction REF.

Voir aussi Variables systèmes REFEN, REFDONE, NOMANREF, DQREF. Instruction ***REF***.

Exemple :

Soient les instructions Uniprog suivantes du programme « REF.E7U » ou « PON.E7U » :

```
QREF   X          ; Reference rapide axe X
QREF   Y          ; Reference rapide axe Y
END               ; Fin programme PON.E7U
```

Cet exemple en Uniprog effectue les prises de références simplifiées sur l'axe X, puis sur l'axe Y.

RAD rayon mode

L'instruction **RAD** définit le rayon et le sens de rotation pour les interpolations circulaires qui suivront (instructions ***CIRA*** et ***CIRR***).

- *rayon* : Rayon en unités de longueur.
- *mode* : Sens de rotation :
  - 1 : Rotations dans le sens horaire.
  - 0 : Rotations dans le sens anti-horaire.

Notes :

Le rayon et le sens sont modaux, à savoir que si l'on fait un arc de cercle en plusieurs morceaux, ou que l'on fait d'autres arcs de même rayon et de même sens, il n'y a pas besoin de répéter l'instruction **RAD** (la dernière est prise en compte).

Voir aussi Instructions ***DPATH***, ***ENDP***, ***CIRA***, ***CIRR***.

Exemples :

Les exemples des instructions ***CIRA*** et ***CIRR*** en pages ***45*** et ***50*** donnent des précisions sur la mise en œuvre de l'instruction **RAD**. En reprenant l'exemple de l'instruction ***CIRA***, nous avons en Uniprog :

```
;                          CORPS DU PROGRAMME
              POSA   X #VMAX[X] 25 0          ; Positionnement de X ...
              POSA   Y #VMAX[Y] 15 2          ; et de Y a vitesse rapide
              DPATH  0 3 #VMAX[X]             ; Def espace interpolation
              RAD    10 1                     ; Rayon 10 et sens horaire
               CIRA  X 15 Y  5                ; Arc de cercle etape 1
               CIRA  X  8 Y 22                ; Arc de cercle etape 2
              ENDP                            ; Fin contour: execution
;
              END                             ; Fin du programme
```

Ce programme effectue les opérations suivantes :

- Positionnement depuis l'origine au point de depart en X = 25 et Y = 15.
- Définir une interpolation dans l'espace 0 pour les axes X et Y :
  - Définir un rayon de 10 et un sens de rotation horaire.
  - Decrire un arc de cercle en deux étapes et aller au point d'arrivee (8,22). Comme il l'est expliqué dans l'exemple de l'instruction ***CIRA*** en page ***45***, une seule étape donne deux possibilités d'arc de cercle, les deux étapes permettent de définir quel arc de cercle nous voulons parmi les deux.
  - Terminer le contour.
- Terminer le programme:

**REF axe

L'instruction **REF** effectue la prise de références sur l'axe *axe*.

- *axe* : Numéro ou nom de l'axe sur lequel faire les prises de références.

Notes :

Les prises de références se font avec les paramètres définis dans la configuration du E700 et accessibles par les touches MENU / F5 = CONFIG / F3 = AXES / F4 = REF, le choix de l'axe se fait par sa lettre ou avec les touches F3 = AXE- et F4 = AXE+.

Les opérations effectuées lors d'une prise de référence se font en trois phases, avec les vitesses nominales et lentes et les directions définies dans la configuration des prises de références de l'axe :

- Phase 1 : Cette phase de la prise de référence est exécutée uniquement si le contact de référence est déjà actif. Sortie du contact de référence à la vitesse nominale, avec décélération en sortie du contact de référence.
- Phase 2 : Déplacement vers le contact de référence à vitesse nominale avec décélération sur contact de référence actif.
- Phase 3 : Déplacement en sens opposé à vitesse lente et arrêt instantané dès que le contact de référence n'est plus actif, ou de rencontre de l'index moteur s'il est ainsi configuré.

Le contacteur de référence a deux états possible en fonction de la position sur l'axe et ne devrait pas pouvoir être traversant (1 d'un côté du capteur, 0 de l'autre).

Le potentiomètre FEED est actif pour les vitesses nominales, mais les vitesses lentes ne tiennent pas compte de FEED (répétabilité des prises de références).

Un offset peut être attribué à chaque axe, permettant de définir la position du zéro machine par rapport à la position du capteur de prise de références :

Touches MENU / F5 = CONFIG / F3 = AXES / F2 = MOTION, à la ligne OFFSET.

Dans le menu TOOLPOS, une « * » après le nom de l'axe signifie que l'axe n'a pas été référencé.

Voir aussi Variables systèmes REFEN, REFDONE, NOMANREF. Instructions ***REFR, QREF***.

Exemple :

Soient les instructions Uniprog suivantes du programme « REF.E7U » ou « PON.E7U » :

```
REF     X          ; Prendre references axe X
REF     Y          ; Prendre references axe Y
END                ; Fin programme PON.E7U
```

Cet exemple en Uniprog effectue les prises de références sur l'axe X, puis sur l'axe Y, et est généralement défini comme programme de prise de références P.ON (touches MEM / F1 = P.ON).

*REFR axe retour

Cette instruction n'est utile que si l'axe sur lequel on prend la référence est muni d'une règle incrémentale pour le contrôle de position. REFR : REF avec **R**ègle.

L'instruction **REFR** effectue la prise de références sur l'axe *axe*, avec retour rapide vers l'index.

- *axe* : Numéro ou nom de l'axe sur lequel effectuer la prise de référence.
- *retour* : Distance à parcourir à vitesse rapide au moment du retour entre le contacteur de référence et l'index de référence (index du moteur ou de la règle).

  NB : si *retour* est trop grand de telle façon que l'index est passé pendant l'avance rapide, la prise de référence échoue et l'axe va jusqu'à la butée mécanique.

Notes :

L'instruction **REFR** effectue les mêmes opérations que l'instruction ***REF*** décrite en page ***139***.

La différence entre ***REF*** et **REFR** est dans la phase 3 :

- Dans le cas de la présence d'un contacteur de référence et d'un index moteur, la phase 3 décrite dans l'instruction ***REF*** dure moins d'un tour moteur à vitesse lente.
- Dans le cas d'un contacteur de référence avec un index sur une règle, le contacteur et l'index peuvent être suffisamment éloignés pour que le positionnement à vitesse lente vers l'index soit très long : l'instruction **REFR** permet de définir une distance de retour rapide entre le contacteur et l'index avant que la vitesse d'approche de l'index soit à nouveau lente.
- Il est bien clair que même si un axe est muni d'une règle, une prise de référence avec REF reste possible. Elle sera plus lente, mais fonctionnera tout de même correctement.

Se reporter à l'instruction ***REF*** pour les autres caractéristiques.

Voir aussi REFEN, REFDONE, Instruction ***REF***.

Exemple :

```
REFR   X 30          ; Prendre references axe X
REFR   Y 20          ; Prendre references axe Y
END                  ; Fin du programme
```

Cet exemple en Uniprog effectue les prises de références sur l'axe X puis sur l'axe Y avec un retour rapide entre capteur et index (30 unités de longueurs pour X, 20 unités de longueurs pour Y).

REP n

L'instruction **REP** (REPEAT) définit une boucle de répétition, le bloc d'instructions se trouvant entre les instructions **REP** et ***ENDRP*** sont répétées *n* fois.

- *n* : entier naturel, contient le nombre de répétitions à effectuer.

Notes :

Les instructions du bloc répété ont à disposition l'indice ou numéro de passage dans la boucle :

- Le registre *R9* représente l'indice de la boucle et croît de 0 (valeur de *R9* à la première boucle) à *n* - 1 (valeur de *R9* à la dernière des *n* boucles).
- La variable système *#REPI* contient aussi l'indice de la boucle, mais décroît de *n* – 1 (valeur de *#REPI* à la première boucle) à 0 (valeur de *#REPI* à la dernière boucle).
- On pourra utiliser le registre *R9* comme indice d'un tableau, et la variable *#REPI* comme variable (une variable ne peut être utilisée comme indice d'un tableau).

Comme plusieurs boucles **REP** - ***ENDRP*** peuvent être imbriquées, l'instruction ***ENDRP*** correspond au dernier **REP** rencontré n'ayant pas été déjà associé à un ***ENDRP***.

Exemple :

```
Ligne 10 : REP #VAR1
"instructions eventuelles" (REP ligne 10, ENDP ligne 90)
Ligne 20:   REP #VAR2
               "instructions eventuelles" (REP ligne 20, ENDP ligne 80)
Ligne 30:     REP #VAR3
                 "instructions eventuelles" (REP ligne 30, ENDP ligne 70)
Ligne 70:     ENDRP
                "instructions eventuelles" (REP ligne 20, ENDP ligne 80)
Ligne 80:   ENDRP
             "instructions eventuelles" (REP ligne 10, ENDP ligne 90)
Ligne 90: ENDRP
```

- Les instructions (y compris les répétitions) comprises entre le **REP** de la ligne 10 et le ***ENDRP*** de la ligne 90 sont répétées VAR1 fois.
- À leur tour, et à chacun des passages dans la boucle **REP** ligne 10 et ***ENDRP*** ligne 90, les instructions (y compris les répétitions) comprises entre le **REP** de la ligne 20 et le ***ENDRP*** de la ligne 80 sont répétées VAR2 fois.
- Et à leur tour, les instructions comprises entre le **REP** de la ligne 30 et le ***ENDRP*** de la ligne 70 sont répétées VAR3 fois.
- Au total :
  - les instructions entre les lignes 10 et 20 et entre les lignes 80 et 90 seront répétées VAR1 fois
  - les instructions entre les lignes 20 et 30 et entre les lignes 70 et 80 seront répétées VAR1 x VAR2 fois
  - les instructions entre les lignes 30 et 70 seront répétées VAR1 x VAR2 x VAR3 fois.

À l'issue de l'instruction ***ENDRP***, l'index REPI et le registre R9 qui donnent l'indice dans la boucle **REP** - ***ENDRP***, reprennent la signification de la boucle supérieur, ou si la boucle **REP** - ***ENDRP*** était au niveau principal, l'index REPI et le registre R9 n'ont alors plus de valeurs significatives.

Voir aussi Variable système REPI, registre R9, Instruction ***ENDRP***.

Exemple :

```
;                          INITIALISATIONS
              OFF       #OUT[1]                ; Declencher OUT 1
              OFF       #OUT[2]                ; Declencher OUT 2
;
;                       CORPS DU PROGRAMME
              REP       10                     ; Boucler 10 fois
                ON      #OUT[1]                ;  Enclencher OUT 1
                REP     4                      ;  Boucler 4 fois
                  ON    #OUT[2]                ;   Enclencher OUT 2
                  WAIT  0.2                    ;   Pause de 0.2 secondes
                  OFF   #OUT[1]                ;   Declencher OUT 1
                  OFF   #OUT[2]                ;   Declencher OUT 2
                  WAIT  0.8                    ;   Pause de 0.8 secondes
                ENDRP                          ;  Fin boucle REP
              ENDRP                            ; Fin boucle REP;
              END                              ; Fin du programme
```

Cet exemple en Uniprog utilise deux boucles **REP** - ***ENDRP*** imbriquées pour faire clignoter les sorties 1 et 2 :

- Faire 10 fois une boucle pour faire clignoter la sortie 1 :
- À chacune des boucles, fait une autre boucle pour faire clignoter 4 fois la sortie 2.

RSIM [sim]

L'instruction **RSIM** réactive la tâche simultanée *sim*, ou si *sim* est omis, réactive l'ensemble des tâches simultanées suspendues par l'instruction ***PSIM***. Les tâches réactivées reprennent là où elles en étaient.

- *sim :* De 0 à 8, est le numéro de la tâche simultanée à réactiver.

  Si *sim* est omis, toutes les tâches simultanées suspendues sont réactivées.

Notes :

Si *sim* est omis et si aucune tâche simultanée n'est suspendue, ou si *sim* est spécifié et si la tâche *sim* n'est pas présente ou n'est pas suspendue, l'instruction **RSIM** est sans effet.

Se rapporter à l'instruction ***PSIM*** en page ***135*** pour les autres précisions.

Voir aussi SIMPTR, SIMACT, Instruction ***PSIM***.

Un exemple est donné dans l'explication de l'instruction PSIM.

RX

Cette instruction n'est pas implémentée à l'heure actuelle.

**RX232 *l/c adresse*

L'instruction **RX232** permet de recevoir des données de la liaison série RS232, et les stocke dans un tableau se trouvant à l'adresse *adresse*.

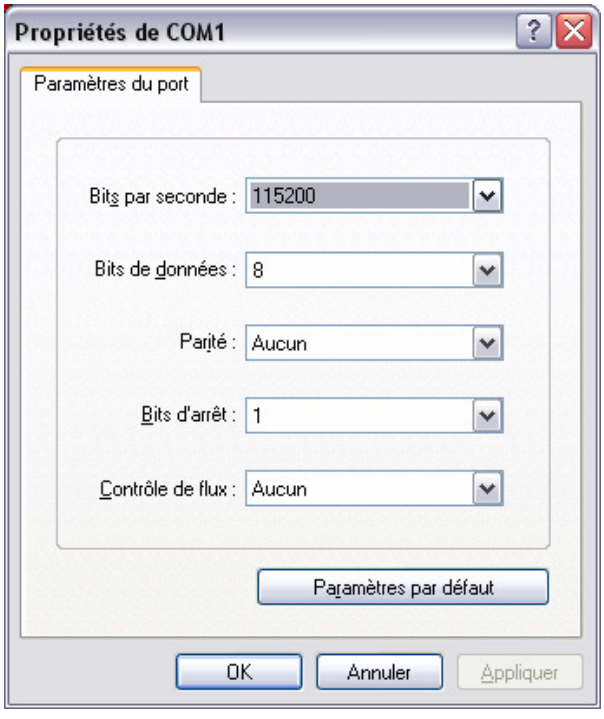
- *l/c* : De 0 à 254, définit soit la longueur du message à recevoir, soit le code du caractère terminal qui termine la réception (choix déterminé par la variable système RXOPT).
- *adresse* : Adresse du tableau de valeurs accueillant les données reçues de la liaison série RS232 (comme c'est une adresse, pas de « # » devant le nom du tableau !).
- *RXOPT :* Variable système qui définit le type de réception pour l'instruction **RX232**:
  - 0 : Valeur par défaut. Réception jusqu'au caractère fin défini par *l/c* (exemple : *l/c* = 13 pour recevoir les éléments jusqu'à réception du caractère carriage RETURN avant de terminer l'instruction **RX232**).
  - 1 : Réception d'une longueur de message donnée par *l/c* (exemple : *l/c* = 5 pour recevoir 5 éléments avant de terminer l'instruction **RX232**).
- *XTOUT :* Variable système qui définit le timeout de réception de la ligne RS232, en secondes, 0 est la valeur par défaut.
- *XRECRX* : Variable système qui contient le nombre d'éléments reçus après l'instruction **RX232**.
- *XERR :* Variable système qui donne le statut de la liaison RS232 :
  - 2 : Ok.
  - - 4 : Erreur séquence d'initialisation.
  - - 3 : Données perdues en réception.
  - - 2 : Problème réception.
  - - 1 : Interrompu.
  - 3 : Timeout.
  - 0 : Pas lancé.
  - 1 : En cours.

Notes :

La configuration de la ligne série RS232 est la suivante :

La vitesse peut être :

- 9600 bits par secondes.
- 19200 bits par secondes.
- 38400 bits par secondes.
- 57600 bits par secondes.
- 115200 bits par secondes.



Les instructions **RX232** et ***TX232*** s'appliquent dans le cas où le DIP switch 5 de la carte CPU du E700 est sur ON (cas où la liaison série RS232 est dédiée à l'utilisation par les programmes Uniprog).

Dans le cas où RXOPT = 0, la taille et la fin de réception dépend de l'équipement qui émet vers le E700 :

- bien veiller à ce que la taille du tableau défini par *adresse* ait au minimum 254 éléments, et puisse recevoir l'ensemble des données susceptibles d'être reçues.
- Si le message reçu excédait la taille du tableau, le système écrirait à des emplacements mémoires inopportuns, ce qui génèrerait certainement des incohérences systèmes ou des dysfonctionnements.

Les instructions **RX232** et ***TX232*** sont exclusives et ne peuvent pas être employées simultanément dans deux tâches en parallèle.

Par contre, le E700 peut recevoir en même temps qu'il émet (instructions **RX232** et ***TX232*** utilisées alternativement dans une même tâche), à concurrence de 254 éléments reçus sans avoir été traités par une instruction **RX232** : au-delà, l'erreur -3 est rencontrée en réception (Données Perdues), ce qui nécessite alors d'exécuter l'instruction ***CLR232***.

Les instructions **RX232** et ***TX232*** peuvent à priori être employées pour les échanges avec tout autre équipement RS232 configurable, y compris d'autres E700.

Voir aussi Variables systèmes RXOPT, TXOPT, Instructions ***TX232***, ***CLR232***.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
             MESSAGE[256] =                  ; Tableau elements RS232
;
;                          INITIALISATIONS
             MOV     #XTOUT 15               ; Timeout RS232 15 secondes
             CLR232                          ; Init reception RS232
;
;                         CORPS DU PROGRAMME
; Recevoir 5 caracteres depuis liaison serie RS232 et les afficher:
             ON      #RXOPT                  ; Mode RS232 longueur message
             RX232   5 MESSAGE               ; Recevoir 5 elements RS232
             DISPN   #XERR 2 0               ; Affiche statut XERR
             CMP     #XERR 2                 ; Reception ok ?
             JNE     SUITE3                  ; Non: ne pas afficher message
SUITE2:      WAIT    0.3                     ; Pause de 0.3 seconde
             ENDRP                           ; Fin de repetition
;
; Recevoir depuis liaison serie RS232 jusqu'au RETURN:
SUITE3:      OFF     #RXOPT                  ; Mode RS232 car. terminal
             RX232   13 MESSAGE              ; Recevoir RS232 jusque RETURN
             DISPN   #XERR 2 0               ; Affiche statut XERR
             CMP     #XERR 2                 ; Reception ok ?
             JNE     SUITE7                  ; Non: ne pas afficher message
;
SUITE7:      WAIT    1                       ; Pause de 1 seconde
             END                             ; Fin du programme
```

Ce programme en Uniprog effectue les opérations suivantes:

- Positionner le timeout pour l'attente des échanges via RS232 à 15 secondes.
- Recevoir 5 caractères depuis la liaison série RS232, puis afficher le statut XERR.
- Recevoir des caractères depuis la liaison série RS232 jusqu'au caractère 13=RETURN, puis afficher le statut XERR.
- Faire une pause de 1 seconde et terminer le programme.

Le fichier exemple fourni « Exemples Uniprog\RX232\RX232.E7U », permet de visualiser les éléments reçus. Utiliser par exemple l'application « Hyper Terminal » (sous Microsoft Windows, dans Tous Les Programmes / Accessoires / Communications) pour échanger les éléments avec le E700 via la liaison série RS232.

SAVE

L'instruction **SAVE** permet de déclencher une sauvegarde des paramètres utilisateurs dans les fichiers INI.

Notes :

Cette instruction est à utiliser avec parcimonie pour éviter un vieillissement prématuré de la flash parallèle.

A chaque START du programme CYCLE (START par pression sur le bouton START ou START soft avec la variable système STARTFLG), les sauvegardes nécessaires se font pour :

- Les données et paramètres système (fichier « E700.INI »).
- Les paramètres des origines et les paramètres d'outils (fichiers « ORIGIN.INI » et « TOOL.INI »).
- Les paramètres utilisateurs (fichier « PUSER.INI »).

L'instruction **SAVE** peut être utile si l'on désire sauvegarder les configurations en fin de programme (exemple : dans la fonction M30), pour s'assurer de la sauvegarde avant extinction du E700.

L'emploi de la **FRAM** permet aussi de sauvegarder de manière permanente les données utilisateur. L'avantage de la FRAM est qu'elle est inusable, contrairement à la flash dont la durée de vie diminue en fonction du nombre d'écritures effectuées.

Nous ne donnons volontairement pas d'exemple d'utilisation afin de décourager le programmeur à utiliser cette instruction.

*SCUT src1 . . . { srci }

L'instruction **SCUT** permet de visualiser directement un des écrans prédéfinis comme accès direct (**S**hort **CUT**).

- *src1 à srci* : Le premier paramètre *src1* désigne l'écran désiré sous la forme d'un nombre :
  - *src1* = « NuméroPage+Touche F1-F6+ToucheF1-F6+ToucheF1-F6 »,
  - où NuméroPage vaut 1 pour TOOLPOS, 2 pour EDIT, 3 pour MEM, 4 pour TRACE, ou 5 pour MENU.

  Exemple : *src1* = 5110 pour MainMenu=5, User=1, PUserA=1, Default=0.

  La liste des écrans accessibles directement par **SCUT** est actuellement la suivante :
  - 5110: MENU / USER / PUSER A / DEFAULT.
  - 5120: MENU / USER / PUSER B / DEFAULT.
  - 5130: MENU / USER / PUSER C / DEFAULT.
  - 5140: MENU / USER / PUSER D / DEFAULT.
  - 5200: MENU / ORIGIN. Dans ce cas, les paramètres optionnels sont nécessaires :
    - *src2* désigne les référentiels concernés :
      - *src2* = 255 pour ne désigner aucun référentiel ni axe, écran ORIGIN par défaut (*src3* et *src4* non significatifs).
      - *src2* = 0 pour désigner G60.
      - *src2* = 1 pour désigner G54 à G58.
    - *src3* désigne le référentiel concerné :
      - si *src2* = 0, alors *src3* de 0 à 63 pour désigner le G60 concerné.
      - si *src2* = 1, alors *src3* va de *src3* = 0 pour G54 à *src3* = 4 pour G58.
    - *src4* désigne le numéro d'axe concerné, *src4* de 0 à 15.
  - 5300: MENU / TOOL. Dans ce cas, les paramètres optionnels sont necessaries :
    - *src2* désigne le numéro de l'outil concerné, *src2* de 0 à 100.
    - *src3* désigne le paramètre concerné :
      - *src3* = 1 pour la longueur de l'outil.
      - *src3* = 2 pour le rayon de l'outil.
  - 5420: MENU / COM / USB. (Commande ASCII pour sortir : **QUIT**)
  - 5460: MENU / COM / RS232. (Commande ASCII pour sortir : **QUIT**)

Notes :

Une fois l'instruction **SCUT** exécutée, l'utilisateur ne peut ressortir de l'écran affiché que par l'appui de la touche ESCAPE : il revient alors à l'écran précédant l'instruction **SCUT**.

Exemple :

Soient dans le fichier « DISPLAY.INI », les lignes :

```
[Display0]
l0 = "EIP SA - Exemple instruction SCUT:      "
l2 = "Utilisation des raccourcis ecrans:      "
l3 = "      Selectionner un acces direct      "
l4 = "              avec F1 a F5              "
f1 = "5110"
f2 = "5120"
f3 = "5130"
f4 = "5200"
f5 = "5300"
f6 = "FIN"
```

En Uniprog :

```
;                          CORPS DU PROGRAMME
DEBUT:      WAITK   R0                      ; Attendre touche pressee
            SWITCH  R0                      ; Suivant touche pressee
              CASE    1 E5110 FINSWI        ; F1: procedure E5110
              CASE    2 E5120 FINSWI        ; F2: procedure E5120
              CASE    3 E5130 FINSWI        ; F3: procedure E5130
              CASE    4 E5200 FINSWI        ; F4: procedure E5200
              CASE    5 E5300 FINSWI        ; F5: procedure E5300
                                            ; Sinon F6: continuer
FINSWI:     ENDS                            ; Fin SWITCH
;
            CMP    R0 6                     ; F6 pressee ?
            JNE    DEBUT                    ; Non: Boucle attendre touche
            END                             ; Fin du programme
;
;                               PROCEDURES
; Procedure E5110 affiche MENU / USER / PUSER A / DEFAULT:
E5110:      SCUT    5110                    ; Raccourci ecran desire
            END                             ; Fin procedure E5110
;
; Procedure E5120 affiche MENU / USER / PUSER B / DEFAULT:
E5120:      SCUT    5120                    ; Raccourci ecran desire
            END                             ; Fin procedure E5120
;
; Procedure E5130 affiche MENU / USER / PUSER C / DEFAULT:
E5130:      SCUT    5130                    ; Raccourci ecran desire
            END                             ; Fin procedure E5130
;
; Procedure E5200 affiche MENU / ORIGIN:
E5200:      SCUT    5200 1 4 1              ; Raccourci ecran desire
            END                             ; Fin procedure E5200
;
; Procedure E5300 affiche MENU / TOOL:
E5300:      SCUT    5300 0 1                ; Raccourci ecran desire
            END                             ; Fin procedure E5300
```

Cet exemple en Uniprog effectue les opérations suivantes:

- Attendre une touche de fonction F1 à F6 (cf. Display0).
- Si F6 pressée, terminer le programme.
- Sinon, faire raccourci suivant touche Fi pressée.
- Reprendre l'attente d'une touche F1 à F6.

SHL dst décalage

L'instruction **SHL** retourne son argument *dst* dont les bits ont été décalés de *décalage* rangs sur la gauche (**SH**ift **L**eft).

- *dst* : valeur modifiée, *dst* est d'abord arrondi à l'entier le plus proche, puis ses bits sont décalés sur la gauche : **dst ← Round(dst) << décalage**
- *décalage* : De 0 à 24, Nombre de décalages des bits sur la gauche.

  À chaque décalage d'un bit sur la gauche, le bit de poids faible prend la valeur 0.

  Si *décalage* est négatif ou plus grand que 24, le résultat est indéfini.

Notes :

Cette opération revient à la multiplication entière de *dst* par $2^{décalage}$. L'argument est d'abord arrondi à l'entier le plus proche.

Exemple : **SHL** 73 5 = 73 x 2^5 = 2 336.

Exemple en logique :

73 << 5 = 0000 0000 0000 0000 0000 0000 0100 1001 << 5

= 0000 0000 0000 0000 0000 1001 0010 0000

= 2 336

Donc 73 << 5 = 2 336.

Exemple en Uniprog :

L'instruction **SHL** avec variables, registres, constantes, tableaux d'entrées, valeurs immédiates :

```
;              DECLARATION CONSTANTES ET VARIABLES
               CNS = 7                          ; Constante CNS vaut 7
               VAL =                            ; Variable VAL
;
;                          INITIALISATIONS
               MOV    R0   73                   ; R0 vaut 73
               MOV    #VAL 5                    ; VAL vaut 5
;
;                        CORPS DU PROGRAMME
               SHL    R0   #VAL                 ; R0 <- 73 << 5 = 2336
;
               SHL    #VAL CNS                  ; VAL <- 5 << 7 = 640
;
               MOV    R0  #IN[3]                ; R0 <- IN[3]
               SHL    R0   8                    ; R0 <- IN[3] << 8:
.
               SHL    #VAL -2.71828             ; (VAL << -2) indefini !
;
               END                              ; Fin du programme
```

SHR dst décalage

L'instruction **SHR** retourne son argument *dst* dont les bits ont été décalés de *décalage* rangs sur la droite (**SH**ift **R**ight).

- *dst* : valeur modifiée, *dst* est d'abord arrondi à l'entier le plus proche, puis ses bits sont décalés sur la droite : **dst ← Round(dst) >> décalage**
- *décalage* : De 0 à 24, Nombre de décalages des bits sur la droite.

  À chaque décalage d'un bit sur la droite, le bit de poids fort prend la valeur suivante :

  - 0 si *dst* est positif.
  - 1 si *dst* est négatif.

  Si *décalage* est négatif ou plus grand que 24, le résultat est indéfini.

Notes :

Cette opération revient à la division entière de *dst* par $2^{décalage}$.

Exemple : **SHR** 2339 5 = $2339 \div 2^5 = 73$.

Le signe du résultat est conservé, et le décalage à droite d'un nombre négatif donne un nombre négatif ou nul.

Exemples en logique :

2 339 >> 5 = 0000 0000 0000 0000 0000 1001 0010 0011 >> 5

= 0000 0000 0000 0000 0000 0000 0100 1001

= 73

Donc 2 339 >> 5 = 73.

- 2 339 >> 5 = 1111 1111 1111 1111 1111 0110 1101 1101 >> 5

= 1111 1111 1111 1111 1111 1111 1011 0110

= - 74

Donc - 2 339 >> 5 = - 74.

Exemple en Uniprog :

L'instruction **SHR** avec variables, registres, constantes, valeurs immédiates :

```
;               DECLARATION CONSTANTES ET VARIABLES
                CNS = 2                         ; Constante CNS vaut 2
                VAL =                           ; Variable VAL
;
;                           INITIALISATIONS
                MOV     R0    -2339             ; R0 vaut -2339
                MOV     #VAL 5                  ; VAL vaut 5
;
;                          CORPS DU PROGRAMME
                SHR     R0    #VAL              ; R0 <- -2339 >> 5 = -74
;
                SHR     #VAL CNS                ; VAL <- 5 >> 2 = 1
;
                MOV     R0   #VMAX[0]           ; R0 <- VMAX[0]
                MUL     R0   1000              ; R0 <- 1000 x VMAX[0]
                SHR     R0    8                 ; R0 <- (1000 VMAX[0]) >> 8
;
                SHR     R0   -2.71828           ; (R0 >> -2) indefini !
;
                END                             ; Fin du programme
```

SIN dst

Retourne le sinus de son argument *dst*.

- *dst* : valeur modifiée, **dst ← SIN (dst)**

  Après l'exécution de **SIN**, l'argument *dst* contient une valeur comprise entre –1.0 et 1.0.

Notes :

Les unités de mesure dans lesquelles travaillent les fonctions trigonométriques sont définies dans la variable système # DEG :

Avant l'instruction **SIN**, on affectera donc la variable DEG avec l'une des instructions ***ON***, ***OFF*** de la valeur :

- ON pour travailler en degrés (valeur 1, par défaut au démarrage du E700).
- OFF pour travailler en radians (valeur 0).

Si on met DEG à 0, alors l'argument dst doit contenir une valeur d'angle en radians.

Voir aussi Variable système DEG, Instructions ***ATN***, ***COS***, ***TAN***.

Exemple :

```
PI                = 3.1415927      ; Constante Pi
;
;                         CORPS DU PROGRAMME
; Par defaut, on travaille en degres
             MOV    R0 30           ; Ici R0 vaut 30 degres
             SIN    R0              ; Ici R0 vaut Sin(30) = 0.5

             OFF    #DEG            ; On travaille maintenant en radians
             MOV    R0 PI           ; Ici, R0 vaut Pi
             DIV    R0 6            ; Ici, R0 vaut Pi / 6 radians
             SIN    R0              ; Ici R0 vaut Sin(Pi/6) = 0.5
;
             END                    ; Fin du programme
```

Cet exemple cacul un sinus en degré, puis un sinus en radian.

*SKS src1 { srci }

L'instruction **SKS** permet de simuler des pressions de touches sur le clavier du E700 depuis un programme (**S**hort **K**ey **S**equence).

- *src1* à *srci* : Suite des touches à simuler. Chacun des *srci* est une constante définie dans le fichier système « E700KEY.E7M » :

KSTART	touche **START**	**KEYX**	touche **X**	**KEY0**	touche **0**
KSTOP	touche **STOP**	**KEYY**	touche **Y**	**KEY1**	touche **1**
		KEYZ	touche **Z**	**KEY2**	touche **2**
KF1	touche **F1**	**KEYA**	touche **A**	**KEY3**	touche **3**
KF2	touche **F2**	**KEYB**	touche **B**	**KEY4**	touche **4**
KF3	touche **F3**	**KEYC**	touche **C**	**KEY5**	touche **5**
KF4	touche **F4**	**KEYD**	touche **D**	**KEY6**	touche **6**
KF5	touche **F5**	**KEYE**	touche **E**	**KEY7**	touche **7**
KF6	touche **F6**	**KEYF**	touche **F**	**KEY8**	touche **8**
KF7	touche **F7**	**KEYG**	touche **G**	**KEY9**	touche **9**
KF8	touche **F8**	**KEYH**	touche **H**		
		KEYI	touche **I**		
KAUTO	touche **AUTO**	**KEYJ**	touche **J**		
KSTEP	touche **STEP**	**KEYK**	touche **K**	**KUPA**	flèche **haut**
KSAT	touche **SAT**	**KEYL**	touche **L**	**KDNA**	flèche **bas**
KPAUSE	touche **PAUSE**	**KEYM**	touche **M**	**KLFA**	flèche **gauche**
		KEYN	touche **N**	**KRTA**	flèche **droite**
KJOGM	touche **JOGGING -**	**KEYO**	touche **O**		
KJOGP	touche **JOGGING +**				
		KMINUS	touche **-**		
KTOOLP	touche **TOOLP**	**KPOINT**	touche **.**	**KESC**	touche **ESC**
KEDIT	touche **EDIT**	**KSPACE**	touche **SPACE**	**KCLR**	touche **CLR**
KMEM	touche **MEM**	**KMISC**	touche **MISC**	**KGOTO**	touche **GOTO**
KTRACE	touche **TRACE**	**KALPHA**	touche **ALPHA**	**KINS**	touche **INS**
KMENU	touche **MENU**	**KENTER**	touche **ENTER**	**KDEL**	touche **DEL**

- *SKSDELAY* : Variable système, définit l'intervalle de temps entre deux touches *src*i successives, en multiple de 4 millisecondes. Sa valeur par défaut est 10 ms. Exemple, si SKSDELAY = 50 et que l'on spécifie comme paramètres de l'instruction **SKS**, les deux touches *src1* = KF1 et *src2* = KF2, tout se passe comme si l'utilisateur appuie sur la touche F1, puis 50 x 4 = 200 ms plus tard, appuie sur la touche F2.

Notes :

Si l'on ne connait pas le temps nécessaire à la réalisation d'une opération consécutive à l'appui d'une touche, et que l'on souhaite enchaîner plusieurs actions de touches, il est préférable de faire plusieurs instructions **SKS** individuelles successives avec chacune une touche, car l'instruction **SKS** se termine lorsque la dernière touche a terminée son action. Exemple : pour visualiser l'écran des LOGs utilisateur :

```
SKS     KMENU          ; Afficher MENU
SKS     KF6            ; MENU/OTHER
SKS     KF3            ; MENU/OTHER/LOGS
SKS     KF4            ; MENU/OTHER/LOGS/USER
```

Est préférable à :

```
SKS     KMENU KF6 KF3 KF4      ; MENU/OTHER/LOGS/USER
```

Voir aussi Variable système SKSDELAY, Instructions ***SCUT***.

Exemple :

```
;                          CORPS DU PROGRAMME
;
ECRLOG:      SKS      KMENU                 ; Afficher MENU
ECRLOG1:     BRIN1    #MACRO[0] ECRLOG2     ; Ecran principal 0 ? Oui: suite
                                            ; Non: atteindre ecran principal
             BRIN1    #MACRO[1] ECRLOG2     ; Ecran principal 1 ? Oui: suite
                                            ; Non: atteindre ecran principal
             SKS      KESC                  ; Simuler ESC (remonter ecran)
             JMP      ECRLOG1               ; Retour tests ecrans principaux
;
ECRLOG2:     SKS      KMENU KF6 KF3 KF4     ; MENU/OTHER/LOGS/USER
;
             END                            ; Fin du programme
```

Ou encore ce qui est équivalent à partir de l'étiquette ECRLOG2:

```
ECRLOG2:     SKS      KMENU                 ; Afficher MENU
             SKS      KF6                   ; MENU/OTHER
             SKS      KF3                   ; MENU/OTHER/LOGS
             SKS      KF4                   ; MENU/OTHER/LOGS/USER
;
             END                               ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Afficher le menu des LOGs utilisateurs en simulant l'appui des touches correspondantes:
  - Affiche MENU et revient à l'écran principal, pour être sûr d'être au niveau principal.
  - Affiche MENU (on est au niveau de base) puis le sous-menu OTHER/LOGS/USER.
- Terminer le programme.

SPEC src par_1 par_2 … par_n

L'instruction **SPEC** permet d'étendre le jeu des instructions Uniprog, en proposant des fonctions spécifiques, rapidement mises en place par l'équipe de développement, et qui répondent à une demande particulière.

- *src* : Numéro de la fonction spéciale à mettre en œuvre.

Notes :

Les fonctions spéciales sont :

SPEC 0 : Lorsque l'étage de puissance d'un moteur servo permet de passer du mode consigne de position au mode consigne de vitesse, lors du retour au mode consigne de position, cette fonction permet de retrouver la position du moteur sans faire une nouvelle prise de référence.

SPEC 1 : Permet de mettre en surbrillance des textes à l'écran.

SPEC 2 : Permet d'afficher le nom du programme CYCLE.

SPEC 3 : Permet de déterminer la présence ou l'absence de la carte Ethernet qui est en option.

SPEC 4 : Offre certaines fonctions sur la carte Ethernet qui est en option.

SPEC 5 : Offre certaines fonctions sur la carte Ethernet qui est en option.

SPEC 6 : Permet de lire la première ligne du fichier dans lequel se trouve le programme CYCLE.

SPEC 7 : Permet de copier des valeurs stockées en mémoire utilisateur FRAM vers un fichier.

SPEC 8 : Permet de copier des valeurs stockées dans un fichier vers la mémoire utilisateur FRAM.

SPEC 9 : Permet de modifier la liste des axes affichée dans l'écran TOOL POS.

SPEC 10 : Ne fait rien. Prévue pour calculer les temps d'accès aux arguments en lecture.

SPEC 11 : Ne fait rien. Prévue pour calculer les temps d'accès aux arguments en écriture.

SPEC 12 : Permet de tester la communication entre une nouvelle carte AE et un étage de puissance de type Yaskawa.

SPEC 13 : Envoi de messages sur Ethernet.

SPEC 14 : Ouvrir ou fermer le canal Eternet pour SPEC 13.

SPEC 15 : Modifier FPABS ainsi que les STROKEs (butées soft).

SPEC 16 : Echange de deux noms d'axes

SPEC 17 : Ecrire UNIVS[0]..UNIVS[49] dans un fichier nnn.TXT. nnn = #FILENAME.

SPEC 18 : Lecture (opération inverse de SPEC 16).

SPEC 19 : TouchScreen : Modifier la couleur d'un bouton.

SPEC 20 : Entrée ou sortie du mode de communication non-sécurisé

SPINDL mode [axe]

L'instruction **SPINDL** détermine l'affectation du bouton SPINDLE (panneau avant du E700).

- *mode* : appartient à {0 ; 1 ; 2 ; 4 ; 5 ; 6 ; 7}, définit le type d'opération associé au bouton SPINDL :

  0 : Affecte le bouton SPINDLE à la sortie analogique #DAC[0] (pour piloter la vitesse d'une broche).

  1 : Affecte le bouton SPINDLE à la sortie analogique #DAC[1] (pour piloter la vitesse d'une broche).

  2 : Affecte le bouton SPINDLE aux sorties analogiques #DAC[0] et #DAC[1]. C'est la valeur de #CURDAC[0] (G75/G76) qui fait foi dans le calcul (valeur ← DACVAL * MAXRPM / 255)

  7 : Affecte le bouton SPINDLE à la variable système #SPOT **dans l'échelle de 0 à 255**. La gestion des actions est alors faite par programme Uniprog en lisant la variable #SPOT, et en gérant ensuite la variable #SPLI si nécessaire.

  5 : Affecte le bouton SPINDLE à la variable système SPLI [*axe*] pour piloter l'axe *axe* en vitesse, et le bouton FEED n'est plus associé à l'axe *axe*. Est utilisé dans le cas d'un axe C (broche pilotée en position) pour piloter la vitesse de l'axe C avec le bouton SPINDLE plutôt qu'avec le bouton FEED (cas d'une broche et non d'un axe).

  6 : Supprime l'affectation du bouton SPINDLE à la variable système SPLI [*axe*], et le bouton FEED est réassocié à l'axe *axe* pour le piloter en vitesse.

  4 : Pour désactiver le bouton SPINDLE (aucune action associée).

- *[axe]* : Numéro ou nom de l'axe concerné par l'affectation du bouton SPINDLE.

  Optionnel, *axe* est nécessaire si *mode* =5 ou *mode* = 6.

Variables systèmes SPOT, SPINRDY, DACVAL, MAXRPM, PFEED.

Exemple :

Code UNIPROG typique d'une fonction M03 (enclenchement de la broche) :

Code à stocker dans le fichier FCTM3.E7M

a) Avec potentiomètre SPINDLE actif :

```
M3:  SPINDL   #CURDAC[0]     ; FIXE PAR G75 OU G76
     ON       #OUT[BRON]     ; Avec BRON = n (n : numero de la sortie
                             ;              numerique correspondante)
     END
```

b) Potentiomètre SPINDLE inactif :

```
M3:  SPINDL   4              ; ARRET DE LA BROCHE
     PUSH     R0             ; SAUVEGARDE DES REGISTRES
     PUSH     R1
     MOV      R0 #CURDAC[0]  ; DAC COURANT
     MOV      R1 #DACVAL[R0] ; R1  := Sxx
     DIV      R1 #MAXRPM[R0] ; R1  := R1 / MAX RPM
     MUL      R1 255         ; R1  := R1 * 255
     MOV      #DAC[R0] R1    ; DAC := 255 * Sxx / MAX RPM

     ON       #OUT[BRON]     ; Avec BRON = n (n : numero de la sortie
                             ;                numerique correspondante)
     POP      R1             ; RESTAURATION DES REGISTRES
     POP      R0
     END
```

Code UNIPROG typique d'une fonction M05 (arrêt de la broche) :

Code à stocker dans le fichier FCTM5.E7M

```
M5:  SPINDL   4              ; ARRET DE LA BROCHE
     OFF      #OUT[BRON]     ; Avec BRON = n (n : numero de la sortie
                             ;                numerique correspondante)
     PUSH     R0             ; SAUVEGARDE DES REGISTRES
     MOV      R0 #CURDAC[0]  ; R0          := DAC COURANT
     MOV      #DAC[R0] 0     ; DAC COURANT := 0
     POP      R0             ; RESTAURATION DES REGISTRES
     END
```

En supposant que la sortie numérique sur laquelle est connectée la broche soit la sortie interne 6 par exemple, déclarer :

```
BRON            = 6
```

SQR dst

L'instruction **SQR** modifie une valeur en le carré de cette valeur (**SQ**ua**R**e).

- *dst* : De - 16 777 216 à 16 777 215, valeur modifiée, **dst** ← **dst** ^{2}

Exemple :

```
;              DECLARATION CONSTANTES ET VARIABLES
               VAL =                          ; Variable VAL
;
;                           INITIALISATIONS
               MOV    R0   -1050              ; R0 vaut -1050
               GTXY   10   2                  ; Curseur en (2,10)
               DISPN  R0   4 4                ; Afficher R0: XXXX.XXXX
               MOV    #VAL 0.027              ; VAL vaut 0.027
               GTXY   30   2                  ; Curseur en (2,30)
               DISPN  #VAL 4 4                ; Afficher R0: XXXX.XXXX
;
;                         CORPS DU PROGRAMME
               SQR    R0                      ; R0 <- R0^2 = 1102500
               GTXY   10   4                  ; Curseur en (4,10)
               DISPN  R0   7 1                ; Afficher R0: XXXXXXX.X
;
               SQR    #VAL                    ; VAL <- VAL^2 = 0.000729
               GTXY   10   5                  ; Curseur en (5,10)
               DISPN  #VAL 2 6                ; Afficher VAL: XX.XXXXXX
;
               END                            ; Fin du programme
```

Cet exemple en Uniprog utilise le registre R0 et une variable VAL :

- On élève le registre R0 au carré, puis on affiche le résultat.
- On élève la variable VAL au carré, puis on affiche le résultat.
- On termine le programme.

SQRT dst

L'instruction **SQRT** modifie une valeur en la racine carrée de cette valeur (**SQ**uare **R**oo**T**).

- *dst* : De 0 à 16 777 215, valeur modifiée, **dst** ← **dst** $^{1/2}$

Note :

Si *dst* est négatif, l'instruction **SQRT** provoque l'affichage de l'erreur 13 et arrêt des programmes du E700.

Exemple :

```
;              DECLARATION CONSTANTES ET VARIABLES
              VAL =                          ; Variable VAL
;
;                           INITIALISATIONS
              MOV    R0   1050               ; R0 vaut 1050
              GTXY   10   2                  ; Curseur en (2,10)
              DISPN  R0   4 4                ; Afficher R0: XXXX.XXXX
              MOV    #VAL 0.027              ; VAL vaut 0.027
              GTXY   30   2                  ; Curseur en (2,30)
              DISPN  #VAL 4 4                ; Afficher R0: XXXX.XXXX
;
;                         CORPS DU PROGRAMME
              SQRT   R0                      ; R0 <- R0^1/2 = 32.40370
              GTXY   10   4                  ; Curseur en (4,10)
              DISPN  R0   3 5                ; Afficher R0: XXX.XXXXX
;
              SQRT   #VAL                    ; VAL <- VAL^1/2 = 0.164317
              GTXY   10   5                  ; Curseur en (5,10)
              DISPN  #VAL 2 6                ; Afficher VAL: XX.XXXXXX;
              END                            ; Fin du programme
```

Cet exemple en Uniprog utilise le registre R0 et une variable VAL :

- On calcul la racine carrée du registre R0, puis on affiche le résultat.
- On calcul la racine carrée de la variable VAL, puis on affiche le résultat.
- On termine le programme.

START dst

L'instruction **START** récupère l'état du START (Bouton START sur le panneau, START externe, ou START à deux mains).

- *dst* : De 0 à 1, contient en retour l'état du bouton START.

  0 : Bouton START au repos.

  1 : Bouton START pressé (START du panneau, ou START externe, ou START à deux mains).

Notes :

Si l'on souhaite récupérer seulement l'état du bouton START du panneau avant, il suffit de lire la variable système KEY[KSTART], mais dans ce cas on ne connait pas l'état du bouton START externe ou du START à deux mains s'ils sont configurés actifs.

Par contre, la variable système BSTART donne l'état du bouton START incluant le bouton START du panneau, le START externe, et le START à deux mains.

L'instruction **START est utilisable uniquement dans la tâche AUTOMAT** pour gérer le démarrage des programmes P.ON (prises de références) et CYCLE, et avec la variable système STARTFLG :

- L'instruction **START** ne retourne 1 que si toutes les tâches hors AUTOMAT sont inactives, et si l'un des boutons START est enclenché.
- L'instruction **START** retourne 0 si un programme autre que la tâche AUTOMAT est en cours d'exécution, quelle que soit l'action sur l'un des boutons START.

Lors du lancement du programme P.ON, il est préférable de tester en fin du programme P.ON que le bouton START est bien relâché, afin que le programme CYCLE ne s'enchaîne pas directement (cas où l'utilisateur maintient le START appuyé longtemps, et où la prise de référence est très rapide). Le programme P.ON peut contenir pour cela les instructions suivantes en fin de programme :

```
;                          FIN PROGRAMME P.ON
ENDPON:     WAIT1  #BSTART                  ; Attente START relache
            END                             ; Fin programme P.ON
```

Le programme AUTOMAT le plus souvent employé a la structure suivante :

```
;                          INITIALISATIONS
STARTINIT:  <instructions d'initialisations , ICI …>
;
;                  BOUCLE PRINCIPALE TACHE AUTOMAT
AUTO:       START  #STARTFLG                ; Lecture etat START
;
            <instructions surveillance , ICI …>
;
            CALIN1 #LED[LSTART]             ; Si CYCLE en cours, CYCLEON
;
            JMP    AUTO                     ; Boucle tache AUTOMAT
;
;                             PROCEDURES
; Procedure CYCLEON, Surveillance pendant CYCLE:
CYCLEON:    <instructions de surveillance pendant CYCLE , ICI …>
            END                             ; Fin procedure CYCLEON
```

Cette structure est adaptée à la majeure partie des cas de figures, la tâche AUTOMAT est indépendante du programme CYCLE.

Voir aussi Variables systèmes BSTART, STARTFLG, STARTFAIL, SIMPTR, BSTOP.

STOPM axe decel

L'instruction **STOPM** arrête les mouvements sur un axe ou sur l'ensemble des axes, avec décélération ou d'une manière brutale.

- *axe* : Numéro ou nom de l'axe dont on souhaite stopper les mouvements.

  axe < 0 : tous les axes seront stoppés.

  axe ≥ 0 seul l'axe *axe* sera stoppé.

- *decel :* De 0 à 1, désigne le mode d'arrêt des mouvements.

  1 : Arrêt brusque, sans rampes de décélération. L'axe (ou les axes) perdent leur(s) référence(s).

  0 : Arrêt normal, avec rampes de décélération.

Exemple :

```
; Procedure STOPALL, Arrete les taches 0 a 8 et Arret mouvements:
STOPALL:    REP     9                       ; Pour chaque tache
              CMP    #REPI #PNB             ; Est-ce cette tache ?
              JE     STOPALL1               ; Oui: ne pas terminer
              KSIM   #REPI                  ; Terminer tache 0 a 8
STOPALL1:   ENDRP                           ; Fin arret taches 0 a 8
            STOPM   -1 0                    ; Arreter axes avec rampes
            OFF     #REFDONE                ; Perte references axes:
                                            ; allume bouton STOP si P.ON
            KSIM    #PNB                    ; Terminer tache courante
            END                             ; Fin procedure STOPALL
```

Cet exemple en Uniprog représente une procédure qui pourrait se trouver dans la tâche AUTOMAT ou dans un programme :

- Pour chacune des tâches de 0 à 8, hors la tâche courante qui exécute cette procédure, arrêt de la tâche.
- Arrêter les mouvements sur l'ensemble des axes, avec rampe de décélération (mode 0).
- Supprimer les références, le bouton STOP s'allume si un programme P.ON est défini.
- Terminer la tâche courante.

Le fichier exemple joint « Exemples Uniprog\STOPM\STOPM.E7M » utilise cette procédure en traçant un contour, et en surveillant en parallèle la vitesse de l'axe X : lorsque cette vitesse dépasse 49.7 % de la vitesse max sur l'axe X, cette procédure interrompt les tâches et les mouvements. On visualise ensuite le tracé des vitesses en fonction du temps sur l'écran du E700.

STORE dst

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **STORE** sauvegarde l'accumulateur de la tâche courant*e accum[PNB]* dans *dst*.

- *dst* : registre, variable, ou élément de tableau recevant la valeur de l'accumulateur,

  dst ← accum[PNB].

- *accum[PNB]* : mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, en lecture et écriture).

  est lu par l'instruction pour récupérer sa valeur.

Voir aussi instruction ***LOAD***.

Attention : Dans l'ancien Uniprog, l'instruction était STORED.

Exemple:

En Uniprog :

```
LOAD   5                    ; acc <- 5
ADDD   2                    ; acc <- acc + 2 = 5 + 2 = 7
STORE  R0                   ; R0 <- acc = 7
END                         ; Fin du programme
```

SUB dst src

L'instruction **SUB** effectue la soustraction de ses deux arguments et retourne le résultat dans le premier argument (**SUB**stract).

- *dst* : valeur modifiée, **dst ← dst - src**
- *src* : nombre à soustraire de *dst*.

Voir aussi :

Instructions ***ADD***.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
              PI  =  3.141593                ; Constante Pi
              VAL =                          ; Variable VAL
;
;                          INITIALISATIONS
              MOV    R0   8                  ; Init registre R0 a 8
              MOV    #VAL -0.7               ; Init VAL a -0.7
;
;                        CORPS DU PROGRAMME
              SUB    R0   PI                 ; R0 vaut 8 - Pi =  4.85841
              SUB    #VAL R0                 ; VAL vaut -0.7 - (8 - Pi)
                                             ;                = -5.55841
;
              END                            ; Fin du programme
```

Cet exemple effectue des calculs avec des registres, des variables et constantes.

SUBD src

Cette instruction ne devrait pas être utilisée,
elle n'a de sens que pour des questions de compatibilité avec l'ancien Uniprog.

L'instruction **SUBD** effectue l'addition de la valeur de *src* dans l'accumulateur (**SUB**stract **D**irect).

- *src* : nombre à retrancher de l'accumulateur, **accum[PNB] ← accum[PNB] - *src*.**
- *accum[PNB]* : représente la mémoire de stockage accumulateur de la tâche courante *PNB* (argument implicite, non utilisable directement en Uniprog).

  Est modifié par l'instruction pour contenir le résultat.

Exemple :

```
;               DECLARATION CONSTANTES ET VARIABLES
                VAL =                           ; Variable VAL
;
;                           INITIALISATIONS
                MOV     #VAL 53                 ; Init VAL a 53
;
;                         CORPS DU PROGRAMME
                LOAD    #VAL                    ; Charge le contenu de VAL
                                                ; dans l'accumulateur
                                                ; Accumulateur vaut ici 53
                SUBD    26                      ; accum vaut accum + 26
                                                ; = 53 - 26 = 27
                                                ; Accumulateur vaut 27
                STORE   #VAL                    ; Stocke l'accumulateur
                                                ; dans VAL, VAL vaut ici 27
;
                END                             ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Initialiser la variable VAL avec 53.
- Effectuer les opérations VAL <- (VAL - 26) en utilisant l'accumulateur.

SWITCH src

L'instruction **SWITCH** initie un bloc de tests multiples terminé par l'instruction ***ENDS***, et contenant une ou plusieurs instructions ***CASE***, permettant d'effectuer des ensembles choisis d'instructions suivant la valeur de *src.*

Les trois instructions de test multiple **SWITCH**, ***CASE*** et ***ENDS*** s'utilisent avec la syntaxe suivante :

```
              SWITCH src
                CASE src1a label1b labelFin
                CASE src2a label2b labelFin
                CASE src3a label3b labelFin
                              . . .
                CASE src(n-1)a label(n-1)b labelFin
                CASE srcna labelnb labelFin
labelFin:     ENDS
```

La signification est la suivante :

- Pour tout i, tel que 1 ≤ i ≤ n, si ***src*** = $\boldsymbol{src_{ia}}$, alors appeler la procédure au label $\boldsymbol{label_{ib}}$ puis, au retour, sauter au label ***labelFin*** spécifié dans le ***CASE***.
- Le code de la procédure au label $\boldsymbol{label_{ib}}$ peut contenir lui-même aussi un **SWITCH** - ***CASE*** - ***ENDS*** (on dit alors qu'ils sont imbriqués, « nested » en Anglais).

L'instruction ***ENDS*** doit être exécutée pour terminer le bloc **SWITCH** - ***CASE*** - ***ENDS***.

- *src* : valeur de référence pour les tests conditionnels des instructions ***CASE*** qui se trouvent entre le **SWITCH** et le ***ENDS***.

Notes :

Comme plusieurs blocs d'instructions **SWITCH** - ***CASE*** - ***ENDS*** peuvent être imbriqués, l'instruction ***ENDS*** correspond au dernier **SWITCH** rencontré.

À l'intérieur d'un bloc d'instructions **SWITCH** - ***ENDS***, la première instruction ***CASE*** remplissant les conditions est prise en compte, les instructions suivantes jusqu'au ***ENDS*** correspondant ne sont alors pas exécutées : de ce fait, bien faire attention aux différentes imbrications.

Les imbrications peuvent être issues de **SWITCH** - ***CASE*** - ***ENDS*** dans un sous-programme (fonction ou procédure), et donc pas uniquement au niveau du programme principal.

Exemple :

Cas d'un **SWITCH** ***ENDS*** qui contient un ***CASE*** faisant appel à un sous-programme qui contient lui-même un **SWITCH** - ***CASE*** - ***ENDS***, et ainsi de suite : dans ce cas, les **SWITCH** sont imbriqués.

À l'issue de l'instruction ***ENDS*** qui termine le **SWITCH** principal, une instruction ***CASE*** serait alors inadaptée (blocage du programme).

Voir aussi Instructions ***CASE***, ***ENDS***.

Exemple : Voir instruction CASE.

TAN dst

L'instruction **TAN** retourne la tangente de son argument *dst*.

- *dst* : valeur modifiée, **dst ← TAN (dst)**

  dst = π / 2 rad. (en mode radian), ou *dst* = 90° (en mode degré) provoque une erreur d'exécution et l'arrêt des programmes du E700.

Notes :

Les unités de mesure dans lesquelles travaillent les fonctions trigonométriques sont définies dans la variable système # DEG :

Avant l'instruction **TAN**, on affectera donc la variable DEG avec l'une des instructions ***ON***, ***OFF*** de la valeur :

- ON pour travailler en degrés (valeur 1, par défaut au démarrage du E700).
- OFF pour travailler en radians (valeur 0).

Si on met DEG à 0, alors l'argument dst doit contenir une valeur d'angle en radians.

Voir aussi Variable système DEG, Instructions ***ATN***, ***COS***, ***SIN***.

Exemple :

```
PI                  = 3.1415927      ; Constante Pi
;
;                         CORPS DU PROGRAMME
; Par defaut, on travaille en degres
             MOV    R0 45            ; Ici R0 vaut 45 degres
             TAN    R0               ; Ici R0 vaut Tan(45) = 1

             OFF    #DEG             ; On travaille maintenant en radians
             MOV    R0 PI            ; Ici, R0 vaut Pi
             DIV    R0 4             ; Ici, R0 vaut Pi / 4 radians
             TAN    R0               ; Ici R0 vaut Tan(Pi/4) = 1
;
; Generer une erreur avec le calcul de Tan(Pi/2 rad.):
             ON     #DEG             ; Travaille en degres
             MOV    R0 90            ; Ici, R0 vaut 90
             TAN    R0               ; Tan(90) => Erreur execution
                                     ; Les instructions suivantes
                                     ; ne sont pas effectuees ...
;
             END                     ; Fin du programme
```

*THREAD pas pro prf ang fin n

L'instruction **THREAD** exécute un filetage par peignage (multi-passes). Elle nécessite un codeur de position de 1024 divisions monté avec un rapport 1:1 sur la broche. Le codeur de position est utilisé uniquement pour effectuer le filetage. il ne permet pas le positionnement de broche.

- *pas* : Pas de filetage en [mm] ou unités de longueur de l'axe Z.
- *pro* : Déplacement relatif total en X, signé, depuis la position de départ jusqu'à la passe de finition. Si la position de départ coïncide avec le rayon du cylindre ébauche, alors *pro* est la profondeur du filet en [mm] ou unités de longueur de l'axe X.
- *prf* : Profondeur de la passe de finition en [mm] ou unités de longueur de l'axe X.
- *ang* : Angle de progression du peignage [degrés].
- *fin* : Position, signée, selon l'axe Z de la fin du filetage. Attention, il faut prévoir de la marge pour le temps de réaction ainsi que la décélération.
- *n* : Nombre de passes. La passe de finition n'est pas comprise dans ce nombre.

Notes :

La fonction ISO qui appelle le filetage est G39.

L'index du codeur est utilisé pour le filetage. Cela signifie que cet index n'est plus disponible pour une éventuelle autre utilisation. Cela signifie également que la commande E700 doit être prévue pour le filetage. Si ce n'est pas le cas, une petite intervention doit être exécutée par le personnel technique de EIP.

Voir aussi Variables système THLEN (130), THRERR, THUSRLEN, THTOLER et THSPCTRL (de 196 à 200).

TOOL numéro [sim]

L'instruction **TOOL** spécifie l'utilisation de l'outil « TOOL Uniprog » numéro *numéro*.

Il n'y a **pas de prise en charge de la correction d'outil en Uniprog** :

l'instruction TOOL en Uniprog est donc un déplacement d'origines équivalent à la commande ISO « G60 D*numéro* ».

A la suite de l'instruction **TOOL**, les programmes Uniprog travaillant dans l'espace d'interpolation *sim* prendront en compte les déplacements d'origines de l'outil, et les programmes ISO travaillant dans l'espace d'interpolation *sim* prendront en compte à la fois les déplacements d'origines et le rayon associés à l'outil.

- *numéro* : De -1 à 63, numéro de l'un des 63 outils TOOL Uniprog à utiliser.

  Si *numéro* = -1, désactive l'utilisation d'un outil TOOL Uniprog (annule le décalage d'origine G60).

- *[sim]* : De 0 à 4, optionnel, 0 par défaut, numéro de l'espace d'interpolation concerné pour l'utilisation de l'outil *numéro* (cf. instruction ***ISODEF***).

Notes :

Par défaut, il existe 64 TOOL Uniprog utilisables (de 0 à 63) :

- Leurs caractéristiques sont définies avec le menu MENU / F2 = ORIGIN.
- Il est possible de configurer les déplacements d'origines en mode TEACH avec le menu TOOLPOS / F3 = TEACH puis F2 = G60, en ayant auparavant mis en service le « G60 D*numéroOutil* » désiré avec la commande TRACE / F2 = MDI (puis taper « G60 D10 » par exemple pour l'outil 10, puis ENTER, puis F1 = EXEC).

Exemple :

```
            POSA   X #VMAX[X]  10 0        ; Aller en X=+10 attente
            POSA   Y #VMAX[Y] -10 2        ; Aller en Y=-10 go
;
; Initialiser sans utilisation d'outil:
            TOOL   -1                      ; Pas utilisation TOOL Uniprog
            TOOLI  -1                      ; Pas utilisation TOOL
;
            DPATH  0 3 1                   ; Interpolation espace 0
                                           ; axes 0=X 1=Y a 1 m/min
              LINR2  X 0   Y -30           ; Droite vers X=+00 Y=-30
              CIRR   X 40  Y  30  20 15 0  ; Cercle vers X=+40 Y=+30
                                           ; centre (+20,+15)
                                           ; sens anti-horaire
              LINR2  X -40 Y  0            ; Droite vers X=-40 Y=+00
            ENDP                           ; Terminer le contour
;
; Initialiser et utiliser l'outil TOOL Uniprog 1:
            TOOL   1                       ; Utilisation TOOL Uniprog 1
;
            POSA   X #VMAX[X]  10 0        ; Aller en X=+10 attente
            POSA   Y #VMAX[Y] -10 2        ; Aller en Y=-10 go
;
            DPATH  0 3 1                   ; Interpolation espace 0
                                           ; axes 0=X 1=Y a 1 m/min
              LINR2  X 0   Y -30           ; Droite vers X=+00 Y=-30
              CIRR   X 40  Y  30  20 15 0  ; Cercle vers X=+40 Y=+30
                                           ; centre (+20,+15)
                                           ; sens anti-horaire
              LINR2  X -40 Y  0            ; Droite vers X=-40 Y=+00
            ENDP                           ; Terminer le contour
;
            END                            ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Se positionne sur les axes X et Y en (10,-10).
- Supprime les éventuels outils précédemment appliqués.
- Définit une interpolation en X et Y:
  - Effectue un contour particulier avec droites et arcs de cercles (cf. document ci-joint « Exemples Uniprog\TOOL\Graphe TOOL.pdf »).
  - Finalise le contour.
- Utilise l'outil TOOL Uniprog 1 qui effectue un décalage d'origines.
- Définit une interpolation en X et Y :
  - Reprend le même contour que précédemment.
  - Finalise le contour.

Le fichier programme exemple joint « Exemples Uniprog\TOOL\TOOL.E7U » effectue les contours de cet exemple et affiche le résultat sur l'écran du E700.

TOOLI src [sim]

L'instruction **TOOLI** spécifie l'utilisation de l'outil « TOOL » numéro *src* (cf. commande ISO T).

Il n'y a **pas de prise en charge de la correction d'outil en Uniprog** :

> **l'instruction TOOLI en Uniprog est donc un déplacement d'origine de la longueur de l'outil sur l'axe de l'outil (axe Z en général), ainsi que la définition d'un rayon.**

- *src* : De -1 à 100, numéro de l'un des 100 outils TOOL à utiliser.

  Si *src* = -1, annule le précédent **TOOLI**.

- *[sim]* : De 0 à 4, optionnel, 0 par défaut, numéro de l'espace d'interpolation concerné pour l'utilisation de l'outil *src* en cas d'interpolations multiples (cf. instruction ***ISODEF***).

Notes :

Il existe 100 TOOL utilisables (de 0 à 99) :

- Leurs caractéristiques sont définies avec le menu MENU / F3 = TOOL.
- Il est possible de configurer les longueurs en mode TEACH avec le menu TOOLPOS / F3 = TEACH puis F3 = T LEN, en ayant auparavant mis en service le « T*numéroOutil* » désiré avec la commande TRACE / F2 = MDI (puis taper « T12 » par exemple pour l'outil 12, puis ENTER, puis F1 = EXEC).

Voir aussi Variables systèmes TOOLLEN, TTOOLRAD, VTOOL, TOOLRAD, OFST, Instruction ***TOOL***.

Exemple :

```
            POSA   X #VMAX[X]  10 0        ; Aller en X=+10 attente
            POSA   Z #VMAX[Z] -10 2        ; Aller en Z=-10 go
;
; Initialiser sans utilisation d'outil:
            TOOLI  -1                      ; Pas d'utilisation de TOOL
;
            DPATH  0 5 1                   ; Interpolation espace 0
                                           ; axes 0=X 2=Z a 1 m/min
              LINR2  X 0   Z -30           ; Droite vers X=+00 Z=-30
              CIRR   X 40  Z  30  20 15 0  ; Cercle vers X=+40 Z=+30
                                           ; centre (+20,+15)
                                           ; sens anti-horaire
              LINR2  X -40 Z  0            ; Droite vers X=-40 Z=+00
            ENDP                           ; Terminer le contour
;
; Initialiser et utiliser l'outil TOOL 1:
            TOOLI  1                       ; Utilisation TOOL 1
;
            POSA   X #VMAX[X]  10 0        ; Aller en X=+10 attente
            POSA   Z #VMAX[Z] -10 2        ; Aller en Z=-10 go
;
            DPATH  0 5 1                   ; Interpolation espace 0
                                           ; axes 0=X 2=Z a 1 m/min
              LINR2  X 0   Z -30           ; Droite vers X=+00 Z=-30
              CIRR   X 40  Z  30  20 15 0  ; Cercle vers X=+40 Z=+30
                                           ; centre (+20,+15)
                                           ; sens anti-horaire
              LINR2  X -40 Z  0            ; Droite vers X=-40 Z=+00
            ENDP                           ; Terminer le contour
;
            END                            ; Fin du programme
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Se positionne sur les axes X et Z en (10,-10).
- Définit de ne pas utiliser d'outil TOOL.
- Définit une interpolation en X et Z:
  - Effectue un contour particulier avec droites et arcs de cercles (cf. document ci-joint « Exemples Uniprog\TOOL\Graphe TOOL.pdf »).
  - Finalise le contour.
- Utilise l'outil TOOLI 1 qui effectue un décalage d'origine en Z.
- Définit une interpolation en X et Z :
  - Reprend le même contour que précédemment.
  - Finalise le contour.

Le fichier programme exemple joint « Exemples Uniprog\TOOLI\TOOLI.E7U » effectue les contours de cet exemple et affiche le résultat sur l'écran du E700. Lors de la réalisation du contour et de l'affichage des coordonnées avec TOOLPOS, on remarquera les coordonnées différentes en Z entre le premier et le deuxième contour.

*TPING axe pas coord sortie [S2]

L'instruction **TPING** effectue un taraudage sur porte outil souple.

- *axe* : Numéro ou nom de l'axe du taraudage.
- *pas :* Pas du taraudage en mm.
- *coord :* Valeur donnant la coordonnée absolue à atteindre sur l'axe *axe*, en unités de longueur, et relativement aux référentiels d'origines actuellement utilisés.
- *sortie* Sortie numérique pour inverser le sens de rotation de la broche lorsque la position finale *coord* sur l'axe d'avance est atteinte.

  sortie contient l'état de la sortie en fin de taraudage, ***sortie* ne doit donc pas être une constante**.
- *S2* Seconde sortie numérique optionnelle pour inverser le sens de rotation de la broche lorsque la position finale *coord* sur l'axe d'avance est atteinte.

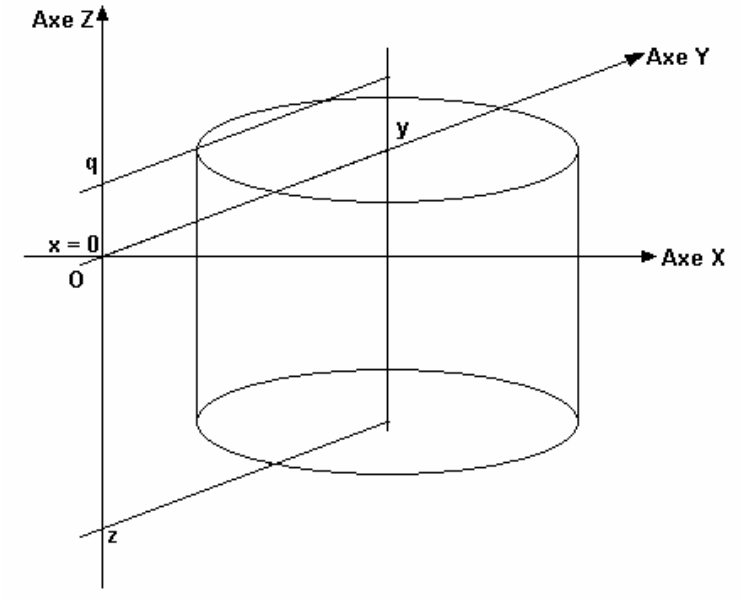
  *S2 c*ontient l'état de de la sortie en fin de taraudage, ***S2* ne doit donc pas être une constante**. Elle est utilisée dans les cas où la broche nécessite deux sorties pour changer de sens

ATTENTION : L'avance est déterminée ici sans être asservie à la vitesse de rotation de la broche. Ceci exige donc l'usage d'un porte-outil avec <u>compensation de longueur</u> pour absorber les variations de synchronisation.

Taraudage en ISO : **G36 Xx Yy Zz Pp Qq Dd**

Avec :

(x; y) : Coordonnées (absolues ou relatives) du centre du trou à tarauder.
z : Profondeur du trou.
p : Le pas du taraudage.
q : Position initiale de Z (Garde).
d: Numéro de la fonction M associée.



La fonction G36 de taraudage calcule automatiquement la vitesse à appliquer sur Z en fonction de la vitesse de rotation de la broche et du pas. Si la vitesse en Z ainsi calculée est trop grande, la vitesse de broche sera automatiquement diminuée en conséquence.

Attention : Pendant l'exécution du taraudage, le potentiomètre FEED est désactivé pour l'axe de plongée. La désactivation du potentiomètre SPINDLE (si celui-ci est actif) est la tâche du programmeur. Le programmeur doit aussi gérer l'arrêt de la broche en cas de pression sur le bouton STOP. **Les boutons PAUSE et SAT sont actifs pendant le taraudage. Une pression sur l'un d'eux va arrêter (ou respectivement modifier la vitesse de) l'axe Z, ce qui aura pour conséquence un probable bris du taraud !**

En ISO, la vitesse de broche est déterminée par la commande S. La broche est pilotée par un conver-tisseur de fréquences. En Uniprog, la séquence suivante permet de fixer une vitesse de broche :

```
ISO_S:  PUSH    R1
        MOV     R1, #CURDAC
        MOV     #DACVAL[R1] ttttt   ; t/min
        POP     R1
        END
```

CURDAC est la sortie analogique courante (0 ou 1). Puis on fixe une valeur en tours par minutes (ttttt) dans le registre de la sortie analogique courante. Ce registre s'appelle DACVAL.

En ISO, le pas est fixé dans le paramètre P de la fonction G36. En Uniprog, le pas est un paramètre de la commande TPING.

Il faut encore écrire une fonction M en Uniprog. Cette fonction est appelée directement par la fonction G36 et elle détermine la sortie à inverser lorsqu'on atteint le fond du trou, juste avant la remontée.

La fonction M peut être n'importe quel numéro. Cependant, on peut s'en tenir à la règle qui dit que cette fonction est M36.

Exemple de fonction M36 appelée directement depuis G36 (fichier FCTM36.E7M)

```
;Taraudage

        ;       +-------------------- Axe de plongee
        ;       I  +----------------- Pas du taraud (Uniprog seul.)
        ;       I  I   +------------- Profondeur du trou (Uniprog seul.)
        ;       I  I   I     +------- Sortie a inverser
        ;       I  I   I     I
        ;       v  v   v     v
M36:    TPING   Z 1.5 -25 #OUT[0]
        CPL     #OUT[0]
        END
```

Dans cet exemple, c'est la sortie interne 0 qui va s'inverser au fond du trou. Elle devra être câblée sur l'entrée «sens» du convertisseur de fréquences.

Lorsque la fonction M36 est appelée depuis une fonction G36 de l'ISO, les paramètres *pas du taraud* et *profondeur du trou* sont fixés dans la fonction G36 elle-même. Cela signifie que les arguments *pas du taraud* et *profondeur du trou* donnés à la commande Uniprog TPING sont ignorés. Ils doivent toutefois être représentés car il n'existe qu'une seule syntaxe de la commande TPING

Donc on pourrait remplacer la ligne par `M36:  TPING  Z 0 0 #OUT[0]`, ça ne changerait rien si M36 est appelée depuis une fonction G36. Ces paramètres ne sont utiles que pour un programme de taraudage écrit en Uniprog.

Exemple de programme ISO :

```
Taraudage

%1
G53
G0  Z0
G0  X0 Y0
S500 M3
G54
G36 X0 Y0 Z-15 P2.5 Q1 D36
X2
X4
X6
Y2
Y4
Y6
G53 M5
G0  Z0
G0  X0 Y0
%
```

Séquence de taraudages en (0; 0) puis en (2; 0) puis en (4; 0) puis en (6; 0) puis en (6; 2) puis en (6; 4) et finalement en (6; 6).

Vitesse de broche : 500 t/min. Profondeur des trous : -15 mm. Pas de 2 mm. La garde est de 1 mm. Cela signifie que le taraud remonte d'un mm au-dessus du trou avant que le déplacement en X et Y soit effectué. La fonction M appelée est la fonction M36, décrite à la page précédente.

Donnons maintenant un exemple de gestion du bouton STOP pour que la broche s'arrête en cas de pression sur celui-ci. On implémente ce code dans la tâche AUTOMAT (fichier AUTOMAT.E7M).

```
;PLC Task

        BRIN1   #INITRDY AUTOMAT
        MOV     #INITRDY 1

AUTOMAT:START   #STARTFLG
        CALIN1  #STOPPRESS ASTOP ; Si bouton STOP presse, alors CALL ASTOP
        JMP     AUTOMAT

;-------------------------------
ASTOP:  CALL    M5              ; Arret de la broche
        OFF     #STOPPRESS      ; Extinction du temoin bouton STOP presse
        WAIT1   #KEY[KSTOP]     ; Attendre relache du bouton STOP
        END
```

Pour terminer et pour être tout à fait complet, on donne encore un exemple d'implémentation de fonctions M3, M4 et M5. On suppose ici que la sortie pour enclencher la broche est la sortie interne 1 et comme précédemment, la sortie interne 0 permet d'inverser le sens de rotation de la broche.

Fichier FCTM3.E7M (fonction M3). Le potentiomètre SPINDLE est inactivé dans cet exemple.

```
;Encl. Broche horaire

M3:     PUSH    R0
        PUSH    R1

        MOV     R1 #CURDAC      ; Fixe par G75 / G76 (0 par defaut)
        MOV     R0 #DACVAL[R1]  ; VALEUR S ISO
        DIV     R0 #MAXRPM[R1]  ; S/VITESSE MAX
        MUL     R0 255
        MOV     #DAC[R1] R0
        OFF     #OUT[1]         ; Sens horaire
        ON      #OUT[0]

        POP     R1
        POP     R0
        END
```

Fichier FCTM4.E7M (fonction M4). Le potentiomètre SPINDLE est inactivé dans cet exemple.
En rouge, les différences avec M3.

```
;Encl. Broche anti-horaire

M4:     PUSH    R0
        PUSH    R1

        MOV     R1 #CURDAC      ; Fixe par G75 / G76 (0 par defaut)
        MOV     R0 #DACVAL[R1]  ; VALEUR S ISO
        DIV     R0 #MAXRPM[R1]  ; S/VITESSE MAX
        MUL     R0 255
        MOV     #DAC[R1] R0
        ON      #OUT[1]         ; Sens anti-horaire
        ON      #OUT[0]

        POP     R1
        POP     R0
        END
```

Fichier FCTM5.E7M (fonction M5) :

```
;Decl. Broche

M5:     PUSH    R1

        MOV     R1 #CURDAC
        MOV     #DAC[R1] 0
        OFF     #OUT[1]
        OFF     #OUT[0]         ; Sens horaire

        POP     R1
        END
```

*TX module index valeur err

L'instruction **TX** envoie des données à un autre E700 connecté au même réseau RS485.

Chacun des E700 connecté sur un même réseau RS485 a une adresse unique sur ce réseau, et peut envoyer à l'un des autres E700 référencé par son adresse *module*, la valeur *valeur* à la position *index* du tableau de valeurs TXRXTAB [ ] du E700 cible.

Sur le E700 de destination : **TXRXTAB [*index*] ← *valeur***.

- *module* : De 0 à 15, contient le numéro du module E700 de destination du message à transmettre via le réseau RS485.
- *index* : De 0 à 19, désigne le numéro d'élément de destination dans le tableau de valeurs TXRXTAB [ ] du E700 cible.
- *valeur* : De -16 777 216 à 16 777 215, valeur à transmettre au E700 cible.
- *err* : contient le statut de l'instruction **TX**.

  0 : le message a été transmis correctement.

  255 : échec écriture vers le E700 *module*.

  254 : bus RS485 en erreur (TimeOut, Framing, …).

  253 : l'adresse de destination *module* est la propre adresse du E700 sur le bus RS485.

  252 : le bus RS485 est dans l'état Off.
- TXRXTAB[ ] : variable système, tableau de **20 éléments** contenant les valeurs reçues des autres E700 sur le réseau RS485.
- DSWITCH : les interrupteurs 1 à 4 du dip switch de la carte mère du E700 représentent l'adresse du E700 sur le bus RS485. À partir de la variable système DSWITCH, faire un ***AND*** avec la valeur 15 pour ne conserver que la valeur de l'adresse sur le bus RS485.

Notes :

C'est aux différents programmes des E700 connectés sur un même réseau RS485 qu'il incombe de gérer les échanges et synchronisations fonctionnels entre les E700, à travers l'instruction **TX** et le tableau de valeurs TXRXTAB [ ].

La gestion du bus RS485 est accessible avec le menu MENU / F4 = COM / F1 = BUS.

Voir aussi *Erreur ! Source du renvoi introuvable. Erreur ! Source du renvoi introuvable.* page *Erreur ! Signet non défini.*, Variables systèmes TXRXTAB [ ], DSWITCH.

Exemple :

Fichier « DISPLAY.INI » :

```
[Display0]
l0 = "EIP SA - Exemple instruction TX:         "
l2 = "Adresse RS485 de ce E700 =               "
l3 = "Envoyer 2.7 dans TXRXTAB[3] du E700 no 7 "
l4 = "Statut de la transmission:"
```

Fichier « MSG.INI » :

```
[Msg]
m0  = "Transmission OK"
m1  = "Bus RS485 Off"
m2  = "Adresse cible=Adresse source"
m3  = "Bus RS485 en erreur"
m4  = "Echec ecriture E700 cible"
m5  = "Erreur inconnue"
```

En Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
             ADR485 =                        ; Adresse RS485 du E700
             MSG485 =                        ; Numero message erreur
             E700CIBLE = 5                   ; Adresse RS485 autre E700
;
;                          CORPS DU PROGRAMME
             MOV     #ADR485 #DSWITCH        ; Lire dip switch carte E700
             AND     #ADR485 15              ; Adresse RS485 bits 0 a 3
             GTXY    27 2                    ; Curseur en (2,27)
             DISPN   #ADR485 2 0             ; Afficher adresse RS485
             DISPC   3                       ; Effacer fin de ligne
             TX      E700CIBLE 3 2.7 #MSG485 ; Envoyer 2.7 dans TXRXTAB[3]
                                             ; du E700 numero E700CIBLE
             SWITCH  #MSG485                 ; Tester valeur retour TX
               CASE  0   P485OK   FIN485     ; Transmission OK
               CASE  252 P485ERR1 FIN485     ; Bus RS485 Off
               CASE  253 P485ERR2 FIN485     ; Adresse cible=Adresse source
               CASE  254 P485ERR3 FIN485     ; Bus RS485 en erreur
               CASE  255 P485ERR4 FIN485     ; Echec ecriture E700 cible
               CALL      P485ERRN            ; Autre cas: Erreur inconnue
FIN485:      ENDS                            ; Fin test valeur retour TX
;
             END                             ; Fin du programme
;
;                               PROCEDURES
;
; P485xxxx , Procedures affichage statut transmission RS485:
P485OK:      MOV     #MSG485 0               ; Transmission OK
             JMP     P485AFFI                ; Afficher message
P485ERR1:    MOV     #MSG485 1               ; Bus RS485 Off
             JMP     P485AFFI                ; Afficher message
P485ERR2:    MOV     #MSG485 2               ; Adresse cible=Adresse source
             JMP     P485AFFI                ; Afficher message
P485ERR3:    MOV     #MSG485 3               ; Bus RS485 en erreur
             JMP     P485AFFI                ; Afficher message
P485ERR4:    MOV     #MSG485 4               ; Echec ecriture E700 cible
             JMP     P485AFFI                ; Afficher message
P485ERRN:    MOV     #MSG485 5               ; Erreur inconnue
P485AFFI:    GTXY    5 5                     ; Curseur en (5,5)
             DISPST  #MSG485                 ; Afficher message statut
             DISPC   3                       ; Effacer fin de ligne
             END                             ; Fin procedures P485xxxx
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Lire et afficher l'adresse RS485 du E700.
- Envoyer la valeur 2.7 dans TXRXTAB[3] du E700 no 7 du bus RS485.
- Afficher le message du statut de la transmission RS485.
- Terminer le programme.

Ce programme utilise un ensemble de procédures P485xxxx regroupées:

- Plusieurs points d'entrées pour désigner le message d'erreur à afficher : P485OK, P485ERR1, P485ERR2, P485ERR3, P485ERR4, P485ERRN.
- Un seul bloc d'instructions à l'adresse P485AFFI pour l'affichage du message de statut.

*TX232 longueur adr/msg

L'instruction **TX232** permet de transmettre des données via la liaison série RS232 à partir soit d'un tableau de données, soit d'un message utilisateur.

- *longueur* : De 0 à 254, définit la longueur du message à émettre si TXOPT = 1, et n'est pas utilisé si TXOPT = 0. Attention cet argument n'est pas optionnel. Il faut le déclarer même si TXOPT = 0.
- *adr/msg*: Contient soit une adresse, soit un numéro de message utilisateur.

  TXOPT=0 : Numéro du message utilisateur défini dans le fichier « MSG.INI » (exemple : 3 pour transmettre m3).

  TXOPT=1 : Adresse du tableau de valeurs contenant les données à émettre via la liaison série RS232 (comme c'est une adresse, pas de « # » devant le nom du tableau !).
- *TXOPT :* Variable système qui définit le type d'émission pour l'instruction **TX232** :

  0 : Transmission d'un message utilisateur défini dans le fichier « MSG.INI ».

  1 : Transmission de valeurs contenues dans un tableau.
- *XERR :* Variable système qui donne le statut de la liaison RS232 :

  2 : Ok.

  - 4 : Erreur séquence d'initialisation.

  - 1 : Interrompu.

  0 : Pas lancé.

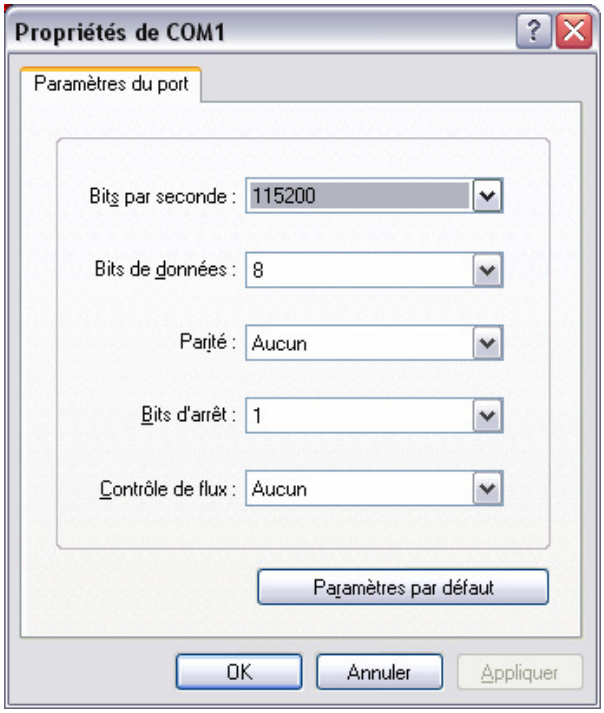
  1 : En cours.

Notes :

La configuration de la ligne série RS232 est la suivante :

La vitesse peut être :

- 9600 bits par secondes.
- 19200 bits par secondes.
- 38400 bits par secondes.
- 57600 bits par secondes.
- 115200 bits par secondes.



Les instructions ***RX232*** et **TX232** s'appliquent dans le cas où le Dip Switch 5 de la carte CPU du E700 est sur ON (cas où la liaison série RS232 est dédiée à l'utilisation par les programmes Uniprog).

Les instructions ***RX232*** et **TX232** sont exclusives et ne peuvent pas être employées simultanément dans deux tâches en parallèle.

Par contre, le E700 peut recevoir en même temps qu'il émet (instructions ***RX232*** et **TX232** utilisées alternativement dans une même tâche), à concurrence de 254 éléments reçus sans avoir été traités par une instruction ***RX232*** : au-delà, l'erreur -3 est rencontrée en réception (données perdues), ce qui nécessite alors d'exécuter l'instruction ***CLR232***.

Les instructions ***RX232*** et **TX232** peuvent a priori être employées pour les échanges avec tout autre équipement RS232 configurable, y compris d'autres E700.

Voir aussi *Erreur ! Source du renvoi introuvable. Erreur ! Source du renvoi introuvable.* page *Erreur ! Signet non défini.*, Variables systèmes TXOPT, RXOPT, Instructions ***RX232***, ***CLR232***.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m10 = "Fin de transmission du E700 via liaison RS232."
```

En Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
             MESSAGE[256] =                  ; Tableau elements RS232
;
;                          INITIALISATIONS
             MOV    #XTOUT 15               ; Timeout RS232 15 secondes
             CLR232                         ; Init reception RS232
;
;                         CORPS DU PROGRAMME
; Recevoir 13 fois 254 caracteres depuis liaison serie RS232:
DEBUT:       MOV     R0 MESSAGE             ; R0<-adresse tableau MESSAGE
SUITE1:      INC     R1                     ; Increment numero message
             ON      #RXOPT                 ; Mode RS232 longueur message
             RX232   254 R0                 ; Recevoir 254 elements RS232
             CMP     #XERR 2                ; Reception ok ?
             JE      SUITE2                 ; Oui: ne pas faire de pause
             OFF     #XERR                  ; Acquittement erreur RS232
             WAIT    3                      ; Pause de 3 secondes
             JMP     SUITE3                 ; Sur erreur, pas nouveau bloc
;
SUITE2:      ADD     R0 254                 ; 254 elements suivants
SUITE3:      CMP     R0 3200                ; Total recus <= 3200 ?
             JLE     SUITE1                 ; Oui: nouveau bloc reception
```

```
; Emettre les 13 fois 254 caracteres recus via la liaison serie RS232:
            MOV     R0 MESSAGE              ; R0<-adresse tableau MESSAGE
SUITE4:     INC     R1                      ; Increment numero message
            ON      #TXOPT                  ; Mode RS232 adresse tableau
            TX232   254 R0                  ; Transmettre 254 elements
            CMP     #XERR 2                 ; Reception ok ?
            JE      SUITE5                  ; Oui: ne pas faire de pause
            OFF     #XERR                   ; Acquittement erreur RS232
            CLR232                          ; Initialise reception RS232
            WAIT    3                       ; Pause de 3 secondes
            JMP     SUITE6                  ; Sur erreur, pas nouveau bloc
;
SUITE5:     ADD     R0 254                  ; 254 elements suivants
SUITE6:     CMP     R0 3200                 ; Total envoyes <= 3200 ?
            JLE     SUITE4                  ; Oui: nouveau bloc emission
SUITE7:     OFF     #TXOPT                  ; Mode RS232 numero message
            TX232   0 10                    ; Transmettre message m10;
;
SUITE8:     WAIT    1                       ; Pause de 1 seconde
            END                             ; Fin du programme
```

Ce programme en Uniprog effectue les opérations suivantes:

- Positionner le timeout pour l'attente des échanges via RS232 à 15 secondes.
- Recevoir 13 fois 254 caractères depuis la liaison série RS232.
- Emettre les 13 fois 254 caractères vers la liaison série RS232.
- Emettre le message m10 « FIN DE TRANSMISSION ».
- Faire une pause de 1 seconde et terminer le programme.

Le fichier exemple fournit « Exemples Uniprog\TX232\TX232.E7U », permet de recevoir le fichier « Exemples Uniprog\TX232\FICHIER.TXT », et de le réémettre, en utilisant par exemple l'application « Hyper Terminal » (sous Microsoft Windows, dans Tous Les Programmes / Accessoires / Communications) pour les échanges entre ordinateur et E700 via la liaison série RS232 :

- Commencer par définir la redirection des caractères reçus du E700 dans un fichier avec le menu « Transfert / Capturer le texte… » et en donnant le nom d'un fichier résultat.
- Lancer le programme TX232.E7U sur le E700.
- Envoyer le fichier au E700 avec le menu « Transfert / Envoyer un fichier texte… » et en sélectionnant le fichier « Exemples Uniprog\TX232\FICHIER.TXT ».
- Terminer la capture de la réception avec le menu « Transfert / Capturer le texte / Arrêter… ».
- Le fichier résultat sur l'ordinateur devrait contenir les données envoyées par le E700.

(**) UART fonction { paramètres }

L'instruction **UART** (**U**niversal **A**synchronous **R**eceiver/**T**ransmitter) gère les communications de l'UART des cartes d'axe AE2, AE4. Cette instruction est donc inutile si la commande E700 est dépourvue de telles cartes d'axe. En général, cette instruction est utilisée pour communiquer avec les amplificateurs d'axes (drivers) si ceux-ci sont pourvus d'une telle fonction de communication sérielle. Quatre différentes fonctions sont regroupées dans cette instruction unique :

UART 0 : Initialisation de la communication

****UART 1** : Envoi d'une chaîne de caractères (carte AE E700 → appareil externe)

****UART 2** : Réception d'une chaîne de caractères (appareil externe → carte AE E700)

UART 3 : Effacement du tampon de réception

UART 0 ser baudrate parité data stop (Initialisation de la communication)

- *0* : définit la fonction d'initialisation de la communication sérielle.
- *ser* : Comme plusieurs cartes AE peuvent être présentes, c'est ici que l'on les différentie. Bien qu'un seul UART gère tous les sériels d'une carte, un système de commutation permet d'utiliser le même UART pour plusieurs ports sériels.
- *baudrate* : Vitesse de communication en bits par secondes. Cette vitesse est définie par le protocole du driver avec lequel on veut communiquer.
- *parité* : 0 : pas de parité, 1 : parité impaire (odd) et 2 : parité paire (even). Cette parité est définie par le protocole du driver avec lequel on veut communiquer.
- *data* : Nombre de bits de données (5, 6, 7 ou 8). Ce nombre est défini par le protocole du driver avec lequel on veut communiquer.
- *stop* : Nombre de bits de stop (1 ou 2). Ce nombre est défini par le protocole du driver avec lequel on veut communiquer.

(*)UART 1 ser longueur adr/msg (Envoi d'une chaîne de caractères) Atomique, mais utilise le timer

- *1* : définit la fonction d'envoi.
- *ser* : Comme plusieurs cartes AE peuvent être présentes, c'est ici que l'on les différentie. Bien qu'un seul UART gère tous les sériels d'une carte, un système de commutation permet d'utiliser le même UART pour plusieurs ports sériels.
- *longueur* : De 0 à 254, définit la longueur du message à émettre si UTXOPT = 1, et n'est pas utilisé si UTXOPT = 1. Attention cet argument n'est pas optionnel. Il faut le déclarer même si UTXOPT = 1.
- *adr/msg* : Contient soit une adresse, soit un numéro de message utilisateur.

  UTXOPT=0 :Numéro du message utilisateur défini dans le fichier « MSG.INI » (exemple : 3 pour transmettre m3).

  UTXOPT=1 :Adresse du tableau de valeurs contenant les données à émettre via la liaison série RS232 (comme c'est une adresse, pas de « # » devant le nom du tableau !).

Codes d'erreur de la variable UARTERR :

7	6	5	4	3	2	1	0
Fifo global error	Axis run	Bad message	Break	Framing	Parity	Overrun	Timeout

UARTERR est un byte. Donc si UARTERR = 36 par exemple, alors 36_d = 24h = 0010 0100_b. Il y a donc un mauvais message et une erreur de parité.

****UART 2** ser l/c [adresse] (Réception d'une chaîne de caractères)

- *0* : définit la fonction d'initialisation de la communication sérielle.
- *ser* : Comme plusieurs cartes AE peuvent être présentes, c'est ici que l'on les différentie. Bien qu'un seul UART gère tous les sériels d'une carte, un système de commutation permet d'utiliser le même UART pour plusieurs ports sériels.
- *l/c* : De 0 à 254, définit soit la longueur du message à recevoir, soit le code du caractère terminal qui termine la réception (choix déterminé par la variable système URXOPT).
- *adresse* : Adresse du tableau de valeurs accueillant les données reçues de la liaison série UART (comme c'est une adresse, pas de « # » devant le nom du tableau !). Ce paramètre est optionnel. S'il est omis, alors le message reçu sera stocké dans la variable système USTRREC.

Variables système concernées :

URXOPT : Variable système qui définit le type de réception pour l'instruction **UART 2**

0 : Valeur par défaut. Réception jusqu'au caractère fin défini par *l/c* (exemple : *l/c* = 13 pour recevoir les éléments jusqu'à réception du caractère carriage RETURN avant de terminer l'instruction **UART 2**).

1 : Réception d'une longueur de message donnée par *l/c* (exemple : *l/c* = 5 pour recevoir 5 éléments avant de terminer l'instruction **UART 2**).

UXTOUT : Variable système qui définit le timeout de réception (et aussi d'envoi), en secondes, 0 est la valeur par défaut.

UARTERR : Variable système qui donne le statut de la liaison sérielle. Voir **UART 1** ci-dessus pour les différents codes.

UART 3 ser (Effacement du tampon de réception)

- *3* : définit la fonction d'effacement du tampon de réception.
- *ser* : Comme plusieurs cartes AE peuvent être présentes, c'est ici que l'on les différentie. Bien qu'un seul UART gère tous les sériels d'une carte, un système de commutation permet d'utiliser le même UART pour plusieurs ports sériels.

Notes :

ATTENTION : l'instruction **UART** est exclusive et ne peut pas être utilisée simultanément dans plusieurs tâches parallèles. Dans la commande E700, version compacte, *ser* (numéro de port sériel) et axe (numéro d'axe) sont identiques. Ce n'est pas forcément le cas dans la version CPU.

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m0 = "W0001000000FF"
```

En Uniprog :

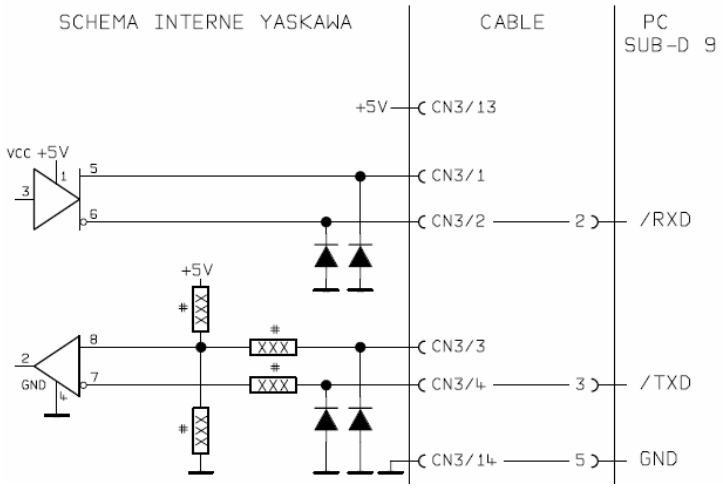
```
U_INI           =       0
U_TXD           =       1
U_RXD           =       2
U_CLR           =       3

SERY            =       0               ; Axe Yaskawa
CR              =
```

```
        MOV     #UXTOUT 1               ; 1 seconde de timeout

        MOV     #UTXOPT 0               ; Static (MSG.INI)
        MOV     #URXOPT 0               ; Control (<CR>)

        UART    U_INI SERY 19200 2 7 1  ; Init serial SERY : 19200,N,8,1
        UART    U_CLR SERY              ; Clear fifo serial SERY
        UART    U_TXD SERY 13 0         ; Send MSG 0 to serial SERY
        BRIN1   #UARTERR LABEL
        CALL    TXCR
        BRIN1   #UARTERR LABEL
        UART    U_RXD SERY 13           ; Receive answer from serial SERY
        BRIN1   #UARTERR LABEL
        ...

LABEL:  ...


; ---------------------------------------------------------------------
;  Envoi d'un <CR>
; ---------------------------------------------------------------------

TXCR:   PUSH    #UTXOPT

        MOV     #UTXOPT 1
        MOV     #CR 13
        UART    U_TXD SERY 1 CR         ; Send CR[0] to serial SERY

        POP     #UTXOPT
        END
```

Cet exemple en Uniprog permet de lire le paramètre 100 Pn100 sur un amplificateur Yaskawa de type XTRADRIVE Cet exemple pourrait aussi être utilisé avec un SIGMA II à condition de l'initialiser à 9600 bauds au lieu des 19200 comme ci-dessus. Il faut se référer à la documentation de chaque amplificateur pour le protocole de communication. Le message reçu se trouve dans la variable système USTRREC. Il dépend de la valeur du Pn100 contenu dans l'amplificateur.

Note importante :

Chez Yaskawa (SIGMA II, XTRADRIVE), les niveaux de sortie (TX Yaskawa) sont de type RS422 (de 0 à 4 volts environ). Cela signifie que la communication RS232 (de –5 à +5 volts selon la norme) est sensible. La longueur des câbles ne doit donc pas excéder 2m. Ces câbles ne doivent pas être exposés aux perturbations. En cas de mauvais fonctionnement, il faut envisager de mettre un convertisseur sériel RS232 / RS422.



**WAIT tempo

L'instruction **WAIT** effectue une attente de *tempo* secondes avant de passer à l'instruction suivante.

- *tempo* : définit la durée de l'attente en secondes.

Notes :

ATTENTION : les instructions ***PECK***, ***RX232***, ***SPEC 3***, ***SPEC 4***, ***SPEC 5***, et ***WAIT*** utilisent le timer de la tâche simultanée courante. Si donc le programme utilise ce timer à travers les variables système TMR et TAF, l'utilisation de l'une de ces instructions entre l'initialisation de TMR et le test de fin de timer avec TAF perturberait le timer, et les résultats finaux seraient aléatoires.

Exemple de problème d'utilisation du timer avec l'instruction **WAIT** :

```
            MOV     #TMR[R8] 2000          ; Compter 2000 ms = 2 s
TMRACTIF:   <instructions>
            WAIT    0.5                    ; Pause de 0.5 secondes
                                           ; ICI LE TIMER EST INCOHERENT!
            <instructions>
            BRIN1   #TAF[R8] TMRACTIF      ; Boucler si timer en cours
            <instructions suivantes>       ; Timer termine, suite
```

Exemple :

Fichier « MSG.INI » :

```
[Msg]
m0  = "Bonjour"
m1  = "Bienvenue"
m2  = "Exemple du wait"
m3  = "Succession de messages"
m4  = "Toutes les 1.5 secondes"
m5  = "Bonne journee."
```

En Uniprog :

```
            MOV     R0 -1                  ; Initialisation index
            REP     6                      ; Repeter pour R0 de 0 a 5
              INC     R0                   ; Increment index
              GTXY    5 3                  ; Curseur en (3,5)
              DISPST  R0                   ; Afficher le message m<R0>
              DISPC   3                    ; Effacer fin de ligne
              WAIT    1.5                  ; Pause 1.5 secondes
            ENDRP                          ; Fin de boucle
;
            WAIT    3                      ; Pause 3 secondes
            END                            ; Fin du programme
```

Cet exemple en Uniprog effectue séquentiellement les opérations suivantes:

- Répéter pour un index i allant de 0 à 5 (utilisation de R0 comme index i) :
  - Afficher le message m<i> et effacer la fin de ligne.
  - Faire une pause de 1.5 secondes.
- Faire une pause de 3 secondes puis terminer le programme.

WAIT0 src [nb]

L'instruction **WAIT0** surveille *src*, et attend tant que *src* = 0 avant de passer à l'instruction suivante.

- *src* : variable à évaluer et à surveiller. **WAIT0** passe à l'instruction suivante dès que *src* ≠ 0.
- *nb* : nombre de lectures, argument optionnel. Cet argument est positif ou nul et signifie que WAIT0 doit être exécuté nb fois de suite avec *src* à 1 pour que WAIT0 se termine.

Notes :

Il est bien entendu que si *src* représente une valeur immédiate ou une variable dont la valeur n'est pas modifiée par ailleurs, l'instruction **WAIT0** effectue toujours la même opération (attente indéfinie si *src* ≠ 0, ou continuer immédiatement si *src* = 0), ce qui n'a pas de sens.

L'argument *nb* optionnel est utile lorsque l'on travaille avec des entrées non fiables qui ont tendance à passer à 1 brièvement et aléatoirement. Ce phénomène peut être provoqué par des mauvais contacts ou des perturbations électromagnétiques. Ainsi, on peut s'assurer que le changement d'état est bien réel et ce n'est pas un simple glitch. Pour plus de précisions à ce propos, lire le document intitulé *Traitement des entrées non fiables.doc*.

https://docs.google.com/open?id=0B7-TXUHxqqRyWUI4TWc2RUlRXzZaVVd1Z1hVbS0wUQ

Voir aussi Instructions ***WAIT1***.

Exemple :

```
;                          CORPS DU PROGRAMME
;
              WAIT0    #IN[3]                  ; Attendre si IN[3] = 0
;
              WAIT     3                       ; Pause 3 secondes
              END                              ; Fin du programme
```

Cet exemple en Uniprog attend tant que l'entrée IN[3] est nulle, et continue pour terminer le programme si cette entrée passe à 1.

WAIT1 src [nb]

L'instruction **WAIT1** surveille *src*, et attend tant que *src* ≠ 0 avant de passer à l'instruction suivante.

- *src* : variable à évaluer et à surveiller. **WAIT1** passe à l'instruction suivante dès que *src* = 0.
- *nb* : nombre de lectures, argument optionnel. Cet argument est positif ou nul et signifie que WAIT1 doit être exécuté nb fois de suite avec *src* à 0 pour que WAIT1 se termine.

Notes :

Il est bien entendu que si *src* représente une valeur immédiate ou une variable dont la valeur n'est pas modifiée par ailleurs, l'instruction **WAIT1** effectue toujours la même opération (attente indéfinie si *src* = 0, ou continuer immédiatement si *src* ≠ 0), ce qui n'a pas de sens.

L'argument *nb* optionnel est utile lorsque l'on travaille avec des entrées non fiables qui ont tendance à passer à 0 brièvement et aléatoirement. Ce phénomène peut être provoqué par des mauvais contacts ou des perturbations électromagnétiques. Ainsi, on peut s'assurer que le changement d'état est bien réel et ce n'est pas un simple glitch. Pour plus de précisions à ce propos, lire le document intitulé *Traitement des entrées non fiables.doc.*

https://docs.google.com/open?id=0B7-TXUHxqqRyWUI4TWc2RUlRXzZaVVd1Z1hVbS0wUQ

Voir aussi Instructions ***WAIT0***.

Exemple :

Cet exemple reprend l'exemple de l'instruction ***WAIT0***, mais attend ici tant que IN[3] est non nulle.

En Uniprog :

```
;                          CORPS DU PROGRAMME
;
            WAIT1   #IN[3]                ; Attendre si IN[3] non nulle
;
             WAIT    3                    ; Pause 3 secondes
             END                          ; Fin du programme
```

Cet exemple en Uniprog attend tant que l'entrée IN[3] est non nulle, et continue pour terminer le programme si cette entrée passe à 0.

WAITK dst

L'instruction **WAITK** attend que l'utilisateur ait pressé une des touches actives F1 à F6, et renvoie le numéro de la touche pressée dans *dst* (**WAIT K**ey).

- *dst* : De 1 - 6], à l'issue de l'instruction **WAITK**, *dst* contient le numéro de la touche de fonction pressée F<*dst*>, de *dst* = 1 pour F1 à *dst* = 6 pour F6.

  Les touches de fonctions sont actives si elles ont un libellé associé :

  - Le libellé a été défini dans le Display actif sous la forme (fichier « DISPLAY.INI », *i* de 1 à 6, *nom* jusqu'à 6 caractères):

```
f<i> = "nom"
```

  - Et si la touche de fonction définie dans le Display actif n'a pas été désactivée par l'instruction ***MENU***.
  - Ou si la touche de fonction a été activée avec l'instruction ***MENU***.

  L'action sur une touche de fonction non active est sans effet.

Notes :

ATTENTION : L'instruction **WAITK** arrête l'exécution de l'ensemble des tâches (touche AUTO éteinte), les différentes tâches reprennent leur activité lorsqu'une des touches de fonctions active est appuyée. Pour continuer l'exécution des tâches pendant l'attente d'une touche, voir l'instruction ***GETKF*** ou l'utilisation de la variable système CURKEY (cf. exemple instruction ***DISPS*** en page ***63***).

Exemple :

Fichier « DISPLAY.INI » :

```
[Display0]
l0 = "EIP SA – Exemple instruction WAITK:      "
f1 = "DO 1"
f2 = "DO 2"
f6 = "FIN"
```

Fichier « MSG.INI » :

```
[Msg]
m1  = "F1 appuyee, action 1 ..."
m2  = "F2 appuyee, action 2 ..."
```

En Uniprog :

```
;                CORPS DU PROGRAMME
DEBUT:      WAITK   R0                      ; Attendre touche pressee
            SWITCH  R0                      ; Suivant touche pressee
              CASE    1 ACTION1 FINSWI      ; F1: procedure ACTION1
              CASE    2 ACTION2 FINSWI      ; F2: procedure ACTION2
                                            ; Sinon F6: continuer
FINSWI:     ENDS                            ; Fin SWITCH
;
            CMP     R0 6                    ; F6 pressee ?
            JNE     DEBUT                   ; Non: Boucle attendre touche
            END                             ; Fin du programme
;
;                            PROCEDURES
;
; Procedure ACTION1, affiche "Action 1":
```

```
ACTION1:    GTXY   5 3                    ; Curseur en (3,5)
            DISPST 1                      ; Afficher m1
            DISPC  3                      ; Effacer fin de ligne
            END                           ; Fin procedure ACTION1
;
; Procedure ACTION2, affiche "Action 2":
ACTION2:    GTXY   5 3                    ; Curseur en (3,5)
            DISPST 2                      ; Afficher m2
            DISPC  3                      ; Effacer fin de ligne
            END                           ; Fin procedure ACTION2
```

Cet exemple en Uniprog effectue les opérations suivantes :

- Attendre une touche de fonction F1, F2, ou F6:
  - Si F1 est appuyée, effectuer la procédure ACTION1.
  - Si F2 est appuyée, effectuer la procédure ACTION2.
  - Si F6 est appuyée, ne rien faire.
- Recommencer l'attente d'une touche tant que touche appuyée est différente de F6.
- Terminer le programme.

Cet exemple utilise deux procédures ACTION1 et ACTION2 :

- ACTION1 affiche la chaîne m1.
- ACTION2 affiche la chaîne m2.

XOR dst src

L'instruction **XOR** effectue le OU exclusif logique de ses deux arguments (**XOR**, noté aussi « ^ »), et retourne le résultat dans le premier de ces arguments.

Le xor logique est tel qu'un bit a pour résultat 1 si l'un des deux bits est à 1 mais pas les deux :

- *dst* : valeur modifiée, **dst ← dst XOR src**.
- *src* : autre argument du **XOR**.

Voir aussi Instruction ***OR.***

Exemple en logique :

```
3.14159 | -589.545      =     3 | -590 (valeurs entières arrondies)

                        =     0000 0000  0000 0000  0000 0000  0000 0011
                        xor   1111 1111  1111 1111  1111 1101  1011 0010

                        =     1111 1111  1111 1111  1111 1101  1011 0001

                        =     -591
```

Donc 3.14159 xor -590.545 = -591

Exemple en Uniprog :

```
;               DECLARATION CONSTANTES ET VARIABLES
              CNS = 11                        ; Constante CNS vaut 11
              VAL =                           ; Variable VAL
;
;                           INITIALISATIONS
              MOV    R0   3.14159             ; R0 vaut 3.14159
              MOV    #VAL -589.545            ; VAL vaut -589.545
;
;                          CORPS DU PROGRAMME
              XOR    R0   #VAL                ; R0 <- 3 xor -590 = -591
              XOR    R0   CNS                 ; R0 <- -589 xor 11 = -582
              XOR    #VAL #IN[3               ; VAL <- -590 xor IN[3]:
                                              ; = -589 si entree IN[3]=1
                                              ; = -590 si entree IN[3]=0
              XOR    #VAL 2.71828             ; VAL <- VAL xor 3:
                                              ; = -592 si entree IN[3]=1
                                              ; = -591 si entree IN[3]=0
              END                             ; Fin du programme
```

Cet exemple montre les différentes opérations logiques avec valeurs immédiates, constantes, registres, variables, élément de tableau correspondant à une entrée numérique.