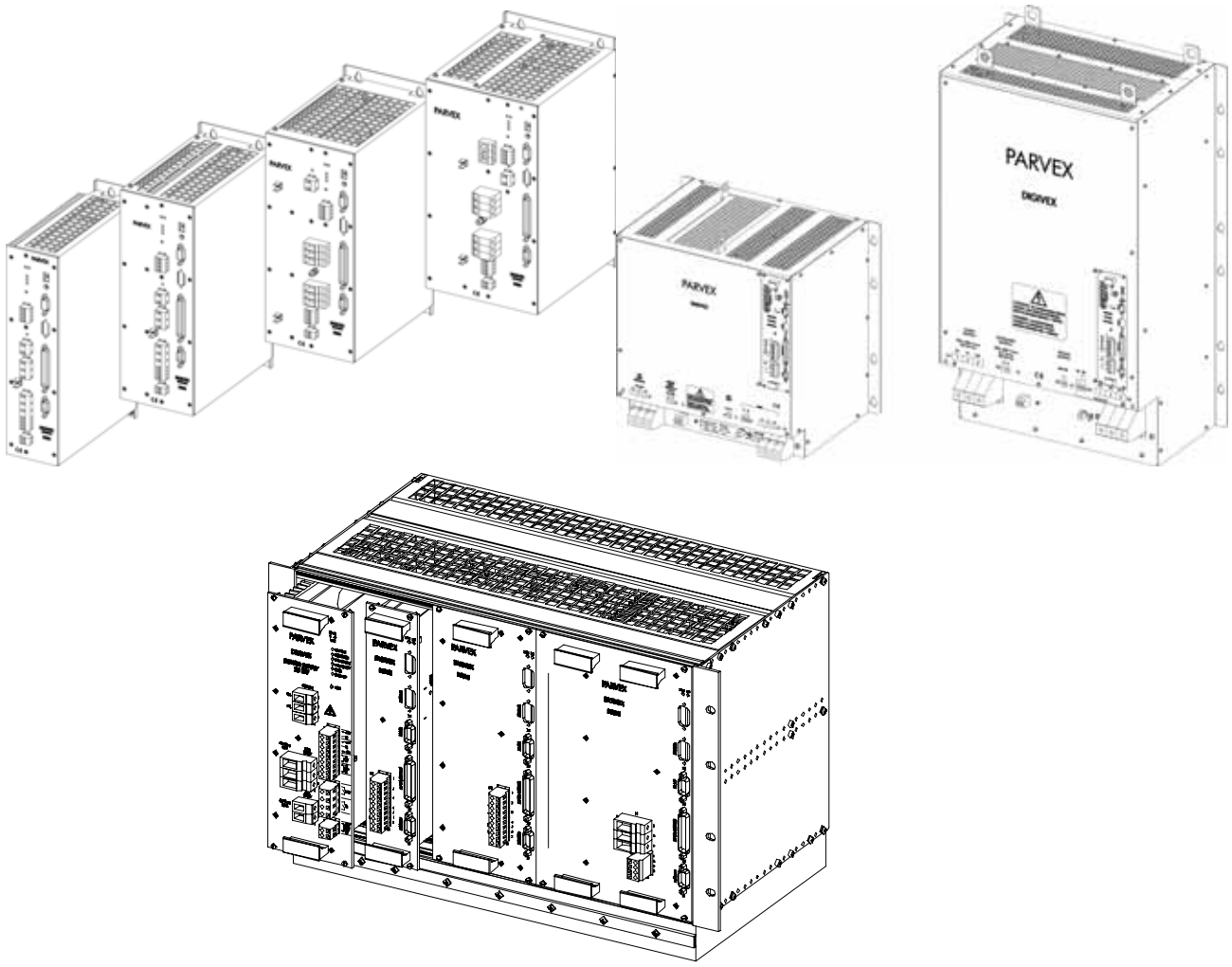
SSD Parvex SAS
8, avenue du Lac - B.P. 249
F-21007 Dijon Cedex
www.SSDdrives.com

DIGIVEX Motion

PROGRAMMATION

PVD 3517 F – 11/2003



PARVEX

GAMME DE PRODUITS

1 - SERVOENTRAÎNEMENTS « BRUSHLESS »

GAMME DE COUPLE OU DE PUISSANCE

- **SERVOMOTEURS BRUSHLESS**, **FAIBLE INERTIE**, **AVEC RESOLVER** :
  Très fort rapport Couple/Inertie (machines haute dynamique) :
  - ⇒ NX -HX - HXA — de 1 à 320 N.m
  - ⇒ NX - LX — de 0,45 à 64 N.m

  Inertie rotor élevée pour une meilleure adéquation de l'inertie de la charge :
  - ⇒ HS - LS — de 3,3 à 31 N.m

  Un choix géométrique varié :
  - ⇒ moteurs courts : HS - LS — de 3,3 à 31 N.m
  - ⇒ ou moteurs de faible diamètre : HD, LD — de 9 à 100 N.m

  Tension adaptée à différents réseaux :
  - ⇒ 230V triphasée pour la «série L - NX»
  - ⇒ 400V, 460V triphasée pour la «série H - NX»
- **SERVOAMPLIFICATEURS NUMERIQUES** « **DIGIVEX Drive** »
  - ⇒ MONOAXE DSD
  - ⇒ MONOAXE COMPACT DµD, DLD
  - ⇒ MONOAXE DE PUISSANCE DPD
  - ⇒ MULTIAXES (RACK) DMD
- **LOGICIEL** DE REGLAGE « PARVEX MOTION EXPLORER »

2 - ENTRAÎNEMENTS « DE BROCHE »

- **MOTEURS SYNCHRONES DE BROCHE**
  - ⇒ Série compacte « **HV** »
  - ⇒ ELECTROBROCHE « **HW** » livrée en kit, à intégrer, avec refroidissement à eau — de 5 à 110 kW, Jusqu'à 60 000 tr/min
- **SERVOAMPLIFICATEURS NUMERIQUES** « **DIGIVEX** » à zone étendue de puissance constante

3 - SERVOENTRAÎNEMENTS « COURANT CONTINU »

- **SERVOMOTEURS** Séries « **AXEM** », « **RS** » — 0.08 à 13 N.m
- **SERVOAMPLIFICATEURS** « **RTS** »
- **SERVOAMPLIFICATEURS** « **RTE** » pour moteurs courant continu + resolver donnant la mesure de position et de vitesse

4 - SERVOENTRAÎNEMENTS « ADAPTATIONS SPECIALES »

- **SERVOMOTEURS** « **EX** » Pour atmosphère explosible
- **SERVOREDUCTEURS** COMPACTS Série « **AXL** » — 5 à 700 N.m

5 - SYSTEMES DE POSITIONNEMENT

- **COMMANDE NUMERIQUE** « **CYBER 2000** » 1 à 2 axes
- **COMMANDE NUMERIQUE** « **CYBER 4000** » 1 à 4 axes
- **VARIATEUR POSITIONNEUR DIGIVEX Motion**
  - ⇒ MONOAXE DSM
  - ⇒ MONOAXE DE PUISSANCE DPM
  - ⇒ MULTIAXES (RACK) DMM
- **LOGICIEL** DE REGLAGE ET PROGRAMMATION PARVEX MOTION EXPLORER

TABLE DES MATIERES

Caractéristiques et dimensions peuvent être modifiées sans préavis

VOTRE CORRESPONDANT LOCAL

SSD Parvex SAS
8 Avenue du Lac / B.P 249 / F-21007 Dijon Cedex
Tél. : +33 (0)3 80 42 41 40 / Fax : +33 (0)3 80 42 41 23
www.SSDdrives.com

1. GENERALITES

1.1 Rappel des notices existantes du DIGIVEX Motion

♦ Notice d'utilisation DIGIVEX Single Motion	(DSM)	PVD3515
♦ Notice d'utilisation DIGIVEX Power Motion	(DPM)	PVD3522
♦ Notice d'utilisation DIGIVEX Multi Motion	(DMM)	PVD3523
♦ Notice DIGIVEX Motion - CANopen		PVD3518
♦ Notice DIGIVEX Motion - Profibus		PVD3554
♦ Notice de réglage PME-DIGIVEX Motion		PVD3516
♦ Répertoire des variables DIGIVEX Motion		PVD3527
♦ Notice de programmation DIGIVEX Motion		PVD3517
♦ DIGIVEX Motion - Fonction Came		PVD3538
♦ Notice d'utilisation PME Tool kit		PVD3528
♦ CANopen - Accès au bus CAN par CIM03		PVD3533
♦ CANopen - Contrôle à distance par messages PDO		PVD3543
♦ Logiciel d'application "Positionnement par blocs"		PVD3519
♦ Logiciel d'application "Coupes à longueur linéaires avec cisaille volante"		PVD3531
♦ Logiciel d'application "Coupes à longueur avec couteaux rotatifs"		PVD3532

1.2 Introduction

Ce document présente le langage de programmation ***Basic_DM***. Les produits numériques *PARVEX* de la famille *DIGIVEX Motion* proposent l'emploi de ce langage. Les informations développées dans ce manuel font référence aux documents :

- PVD 3516 « Logiciel *Parvex Motion Explorer*, module *DIGIVEX Motion* »,
- PVD 3527 « *DIGIVEX Motion, Répertoire des variables* ».

Organisation des chapitres :

- Le **chapitre 1** présente les généralités du langage *Basic_DM*.
- Le **chapitre 2** est le *Guide de l'utilisateur*. Il donne le mode d'emploi du langage et les aspects fonctionnels de la programmation.
- Le **chapitre 3** constitue le *Guide des instructions*.
  - ♦ Le lecteur trouvera en tête de chapitre, une présentation thématique regroupant les codes du langage *Basic_DM*, suivie du répertoire alphabétique des instructions.
  - ♦ Les exemples présentés dans ce chapitre sont donnés à titre didactique ; ils n'illustrent aucunement l'emploi du jeu d'instructions dans un contexte "*application machine"*, des fascicules complémentaires étant proposés à cette fin.
- Le **chapitre 4** indique comment effectuer la mise en œuvre d'un programme.
- Le **chapitre 5** décrit les modes de fonctionnement possibles d'un variateur positionneur DIGIVEX Motion et établit des liens entre les instructions de programmation Basic_DM et ces modes de fonctionnement.

1.3 Langage de programmation

Le langage de programmation est constitué par un ensemble de *codes* : *directives de compilation*, *déclarations* et *instructions*.

- Les principaux *codes* du langage de programmation sont identifiés par des termes *mnémoniques* empruntés au langage BASIC® et écrits de préférence en caractères *MAJUSCULES* (les caractères minuscules restent cependant autorisés). Des notions d'anglais suffisent généralement à en comprendre le sens.
  [ex : HOME, IF…THEN…ELSE…ENDIF…].

- Certains *codes* du langage font directement appel à des *variables*.
  Les *variables* sont identifiées de préférence par des caractères *minuscules*,
  [ex : timer3 = 100, hardlimit_en = 1, etc…].

- Certaines *variables* présentent une valeur dont l'usage est à lecture seule,
- d'autres permettent une lecture et une écriture (leur valeur est par conséquent modifiable).

  Les *variables* adoptent chacune un *format* de représentation :

format :		*caractéristiques :*	
double précision	***D***	réel 40 bits	($\pm$ 3.4∗10^{38} avec 10 chiffres significatifs)
flottant	***F***	réel 32 bits	($\pm$ 3.4∗10^{38} avec 7 chiffres significatifs)
entier	***E***	entier 32 bits	($\pm$ 2 147 483 647).
bit	***B***	valeur binaire	(0 ou 1).
chaîne	***C1***	chaîne de 16 caractères maxi.	

Attention :
Lorsque des grandeurs codées en double précision (*D*) ou en flottant (*F*) dépassent en valeur, le nombre de chiffres significatifs de leur format, il y a mise en place d'un exposant et risque de perte de précision…

Les valeurs entières seront généralement spécifiées au format décimal. On pourra toutefois les exprimer au format hexadécimal en faisant précéder leur valeur du caractère $.
Exemple : ui1 = $10 $\Rightarrow$ ui1 = 16

- L'opérateur utilise l'outil *Editeur de programmes* [*cf. PVD 3516 Chapitre 7*] pour constituer un texte appelé « *fichier_source**.bdm** »,* dans lequel figurent les différents *codes*. Ce fichier sera ensuite *assemblé et compilé* pour obtenir un « *fichier_objet**.odm** *», exécutable par le variateur positionneur.

1.4 Structure de programme

Le langage de programmation *Basic_DM* offre les moyens de structurer les programmes. L'opérateur se doit de diviser préalablement son application en plusieurs tâches distinctes :

- a) les tâches relevant de fonctions de positionnement, par exemple la gestion d'un mouvement programmé (programme de gestion de mouvement).
- b) les tâches à caractère prioritaire, comme la prise en compte d'une entrée logique sur un front d'interruption.
- c) les tâches relevant de fonctions automate, par exemple la gestion d'un automatisme séquentiel utilisant des entrées sorties logiques.
- d) les tâches relevant de fonctions automate et devant s'exécuter avec un temps de cycle fixe et imposé (régulation de certains process).

De cette *partition* naissent le *programme principal* et les divers *sous-programmes* afférents qui composent le *programme utilisateur*.

Il est conseillé de commenter les programmes pour une meilleure relecture.

1.5 Répertoire des variables

Toutes les variables utilisateur et système des DIGIVEX Motion sont accessibles par programmation. Ces variables sont décrites dans le document "PVD 3527 - DIGIVEX Motion - Répertoire des variables".

2. GUIDE DE L'UTILISATEUR

2.1 Introduction

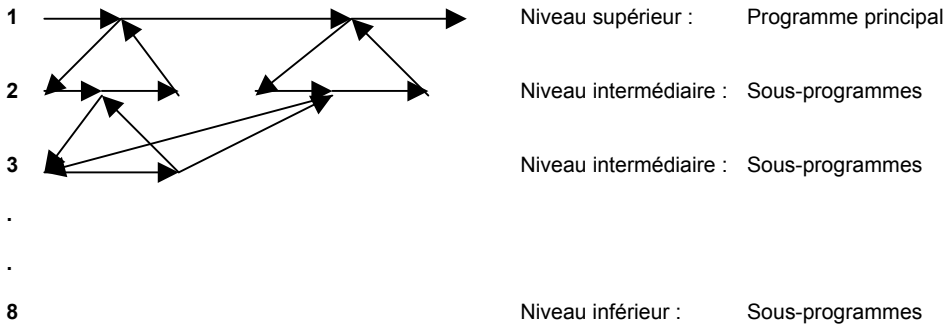
Programmer en *Basic_DM* consiste à écrire une suite ordonnée de codes.
Chaque code traduit une action logicielle élémentaire. Un *index* appelé « *pointeur opérationnel* » désigne le code en cours d'exécution. Le cheminement du *pointeur opérationnel* d'un code à l'autre définit une séquence de programme.

L'opérateur organise son application en plusieurs tâches distinctes pour obtenir, à l'exécution du code, un résultat « programmé ». Il dispose pour cela de codes particuliers, appelés « *structures de contrôle* », destinés à maîtriser l'itinéraire du *pointeur opérationnel* au sein de chaque tâche.

2.2 Hiérarchie des niveaux

Le *programme principal* constitue le niveau supérieur. A partir du *programme principal* sont appelés les *sous-programmes* ; ils constituent les niveaux intermédiaires. Un *sous-programme* peut appeler un autre *sous-programme* ; la hiérarchie peut atteindre 8 niveaux distincts. Le huitième et dernier niveau constitue le niveau inférieur.

L'appel à un *sous programme* fait référence à l'instruction : **GOSUB**
Le retour au programme appelant fait référence à l'instruction : **RETURN**



2.3 Structures de contrôle

Plusieurs structures contrôlent le déroulement et l'exécution d'un programme.

2.3.1 La séquence

La séquence enchaîne une suite d'actions élémentaires :

⇒ Action 1
Action 2
Action 3
…
Action n

Lorsque l'*Action 1* sera achevée, le *pointeur opérationnel*, représenté ici par une flèche [⇒], désignera l'*Action 2* en cours d'exécution.

2.3.2 La sélection

La *sélection* se présente comme une alternative :

Si *telle condition est vraie* Alors Action 1
Sinon Action 2
Fin Si

Le code de programmation correspondant à la *sélection* est la macro instruction :

IF *telle condition est vraie* **THEN** Action 1
ELSE Action 2
ENDIF

2.3.3 La répétition

Cette *structure de contrôle* répète une opération ou une séquence d'opérations, tant qu'une condition est vérifiée à l'entrée.

Tant que la *condition* est vérifiée au test d'entrée,
Effectuer *telle Action 1*
Effectuer telle Action 2…
Fin Tant que

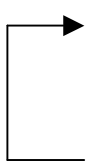
La répétition des opérations comprises entre Tant que et Fin Tant que se traduit par une « boucle ». Lorsque la *condition* est vérifiée au test d'entrée, le nombre de passages dans la boucle est indéterminé. Au contraire, si la *condition* n'est pas vérifiée au test d'entrée, il n'y a aucun passage dans la boucle.

Le code de programmation correspondant à la *répétition* est la macro-instruction :

WHILE condition **DO**
action
WEND

2.3.4 La répétition indexée

Cette *structure de contrôle* répète une opération ou une séquence d'opérations, un certain nombre de fois ; la répétition est dite « indexée ».



```
#début de boucle
 Pour index = 1 à 3   Effectuer  telle Action 1
                                 telle Action 2
                                 …
 Suivant
#fin de boucle
```

La *structure de contrôle* prend en compte :

- l'initialisation et la valeur finale du compteur d'itérations *index*
- la progression du compteur d'un incrément : *index* = *index*+1
- le test du compteur pour sortir de la boucle :

```
Si index > nombre de passages souhaité
      Alors  branchement à l'adresse   #fin de boucle
      Sinon  branchement à l'adresse   #début de boucle
Fin
```

Le nombre de passages dans la boucle est déterminé.

Le code de programmation correspondant à la *répétition indexée* est la macro instruction :

```
FOR variable = n TO m   action
NEXT
```

2.3.5 Le branchement

Le *branchement* est un saut inconditionnel à une adresse déterminée.
L'usage de cette instruction doit s'effectuer avec circonspection. En effet, nombreux sont les exemples où l'emploi inconsidéré de « GOTO » conduit à une programmation non structurée et au final, à une séquence d'opérations non maîtrisée.

Le code de programmation correspondant à un *branchement* est l'instruction :

GOTO #LABEL

2.4 Partition des programmes

Voici un exemple générique de *programme utilisateur*. Chaque tâche à réaliser fait l'objet d'un module spécifique qui sera décrit ci-après.
Un *programme utilisateur* comprend un *programme principal* et des *sous-programmes* appelés au cours de son exécution.

```
%PROG0
;Programme principal
#INIT
PLC1 = PROG100
PLC2 = PROG200
DEFPLC1 = 1000
DEFPLC2 = 100
PLC1 = START
PLC2 = START
INTERRUPT0 = PROG300
INTERRUPT1 = PROG301
INTERRUPT2 = PROG302
INTERRUPT3 = PROG303
ERROR = PROG999
#START
        GOSUB PROG1
        GOSUB PROG13
RESTART
%ENDPROG
```

```
%PROG1
;Sous-programme n°1

RETURN
%ENDPROG
```

```
%PROG13
;Sous-programme n°13

RETURN
%ENDPROG
```

```
%PROG100
;Programme automate
;type 1




END
%ENDPROG
```

```
%PROG200
;Programme automate
;type 2




END
%ENDPROG
```

```
%PROG999
;Programme de gestion
;d'erreurs
;(routine d'exception)




END
%ENDPROG
```

```
%PROG300
;Sous-programme
;prioritaire relatif à in0


END
%ENDPROG
```

```
%PROG301
;Sous-programme
;prioritaire relatif à in1


END
%ENDPROG
```

```
%PROG302
;Sous-programme
;prioritaire relatif à in2


END
%ENDPROG
```

```
%PROG303
;Sous-programme
;prioritaire relatif à in3


END
%ENDPROG
```

2.5 Programme principal

Le *programme principal* s'intitule *PROG**0***.

%PROG0	La directive *%PROG0* figure en « en-tête » du programme principal ; elle définit le début du code relatif au programme principal et n'est destinée qu'au compilateur.
#INIT	La déclaration *#INIT* indique quelle est l'adresse de démarrage du programme principal (et par voie de conséquence, celle du *programme utilisateur*). A l'apparition de l'alimentation auxiliaire, le pointeur opérationnel se place à cette adresse si la variable système *exec_en* = 1 [*cf. PVD 3516 Chapitre 5.2.2.2*]. L'espace de code figurant entre les adresses *#INIT* et *#START* est appelé *zone déclarative*. Cette zone est utilisée pour désigner quels sont les *programmes automate* et les *sous-programmes prioritaires*.
PROG_INIT	L'instruction *PROG_INIT* permet de repositionner le pointeur opérationnel à l'adresse *#INIT*. La structure de prise en compte des imbrications de sous-programmes est complètement effacée. L'instruction *PROG_INIT* peut être employée dans le programme principal ou dans un sous-programme.
#START	La déclaration *#START* est obligatoirement placée après *#INIT*. Elle mentionne l'adresse d'exécution effective du programme principal. Le corps du programme principal débute en fait à cette adresse. Remarque : Lorsque la variable système *userprog_option* = 1, le pointeur opérationnel peut marquer l'arrêt à l'adresse *#START*, jusqu'à l'apparition de la puissance sur le variateur positionneur [*cf. PVD 3516 Chapitre 5.2.2.2*].
RESTART	L'instruction *RESTART* entraîne un saut inconditionnel du pointeur opérationnel à l'adresse *#START*. La structure de prise en compte des imbrications de sous-programmes est complètement effacée. L'instruction *RESTART* peut être employée dans le programme principal ou dans un sous-programme.
%ENDPROG	La directive *%ENDPROG* figure en « pied de page » du programme principal ; elle définit la fin du code relatif au programme principal et n'est destinée qu'au compilateur.

Remarques d'ordre général :

- Chaque ligne de programme est repérée par un numéro. La numérotation est générée automatiquement par l'*éditeur de programmes* ; elle facilite le repérage d'une ligne particulière dans un programme de grande taille. Dans le cas d'une insertion ou d'une suppression de ligne, la numérotation est remise à jour automatiquement.
- Il est permis de sauter des lignes à des fins de clarté.
- L'emploi de tabulations permet d'aérer le texte du programme par mise en exergue des *structures de contrôle*.
- Un commentaire est identifié par un point virgule [**;**]. Il peut figurer à la suite d'une instruction. Un bloc-commentaire se compose de plusieurs lignes de commentaires. Il débute par une ouverture d'accolade **{** et se termine par une fermeture d'accolade **}**.
- Une adresse de branchement est identifiée par un label précédé de **#**.
- On peut utiliser la directive « %VDM= » pour spécifier un fichier contenant les noms « en clair » des variables utilisateur présentes dans le programme utilisateur (programme principal et sous-programmes). La programmation n'en sera alors que plus conviviale.

Remarques concernant le programme principal :

- Le *programme principal* se situe au niveau supérieur de la hiérarchie des programmes. Sa taille est très souvent réduite, car seules y figurent les *déclarations* générales et les *instructions* d'appel aux sous-programmes des niveaux intermédiaires.

Exemple :

```
%PROG0
; Ce texte est un commentaire : il commence par un point virgule.
#INIT
; Voici le label d'initialisation : le programme PROG0 démarre son exécution à cette
; adresse.
PLC1 = NONE      ; pas de programme automate de type 1
PLC2 = NONE      ; pas de programme automate de type 2
#START
HOME
; Cette macro instruction gouverne la prise d'origine de l'axe piloté par le variateur-
; positionneur.
WAIT_UNTIL home_made = 1
{home_made est une variable système. Elle passe à 1 quand la prise d'origine est
effectuée.
L'instruction WAIT_UNTIL permet d'attendre que la prise d'origine soit réalisée avant de
poursuivre le programme.}
#ENCORE
IF in7 = 1 AND in8 = 1      THEN  GOSUB PROG10
                            ELSE  GOSUB PROG20
ENDIF
GOTO #ENCORE
%ENDPROG
```

Commentaires :

- Le déroulement d'un programme s'effectue en séquence : dans notre exemple, la première instruction rencontrée est *PLC1=*, puis viennent *PLC2=, HOME, WAIT_UNTIL, IF…THEN…ELSE…ENDIF,* etc…

- La macro-instruction *IF …THEN …ELSE…ENDIF* permet ici de tester un combinatoire d'entrées logiques :

SI (in7 = 1 et in8 = 1),
Alors le programme principal appelle le sous-programme *PROG10* grâce à l'instruction *GOSUB PROG10*.
Sinon (in7 = 0 ou in8 = 0), le programme principal appelle le sous-programme *PROG20* grâce à l'instruction *GOSUB PROG20*.

Lorsque l'exécution du sous-programme *PROG10* ou *PROG20* est terminée, le pointeur opérationnel se positionne sur *ENDIF*. Cette directive clôt la macro-instruction *IF… THEN… ELSE…ENDIF*.

- L'instruction suivante est *GOTO* : un *saut inconditionnel* replace le pointeur opérationnel à l'adresse *#ENCORE*. L'exécution d'une première boucle de *programme principal* s'achève.

2.6 Sous-programmes

Définition

Un *sous-programme* est une séquence de codes particulière contrôlée par les instructions *GOSUB* et *RETURN*. Le programme contenant l'instruction *GOSUB* est le *programme appelant*. Le programme contenant l'instruction *RETURN* est le *programme appelé* ou *sous-programme*.
Un *sous-programme* s'intitule *PROG**n*** avec ***n*** le n° de sous-programme considéré. ***n*** est un nombre entier compris entre **1** et 999.

%PROG*n*	La directive *%PROG**n*** figure en « en-tête » du sous-programme considéré ; elle définit le début du code relatif à ce sous-programme et n'est destinée qu'au compilateur.
RETURN	L'instruction *RETURN* provoque le retour au programme appelant. Le pointeur opérationnel se place à la ligne qui suit l'instruction *GOSUB PROG**n*** dans le programme appelant. Plusieurs instructions *RETURN* peuvent figurer dans un même sous-programme.
%ENDPROG	La directive *%ENDPROG* figure en « pied de page » d'un sous-programme ; elle définit la fin du code relatif au sous-programme considéré et n'est destinée qu'au compilateur.

Exemple :

```
10      %PROG10
11      ; Voici un exemple de sous-programme. Il porte le numéro 10.
12      #CYCLE
13      ; Ce label est une adresse de branchement pour boucler le sous-programme.
14      WAIT_UNTIL in3 = 1 AND pos2 > 0
15      ; Cette instruction définit une attente conditionnelle.
16      WHILE in4 = 1
17              SPEED = 2000            ; Définit la vitesse de déplacement de l'axe.
18              MOVER = 10              ; Définit un mouvement relatif.
19              WAIT_UNTIL in_position = 1; Attend l'arrivée à destination.
20      WEND
21      IF in5 = 1 THEN GOTO #CYCLE
22      ; Si l'entrée logique n°5 vaut 1 Alors branchement à l'adresse #CYCLE.
23      ENDIF
24      ; La structure IF…THEN ne contient pas nécessairement ELSE.
25      RETURN
26      %ENDPROG
```

Remarques concernant les sous-programmes :

- Un *sous-programme* peut appeler un autre *sous-programme*. La « cascade » d'appels peut atteindre 8 niveaux consécutifs.
- Attention : un *sous-programme* peut s'appeler lui-même ; il est alors « réentrant ». L'utilisation de sous-programmes réentrants est cependant délicate et déconseillée.
- *Basic_DM* gère l'ensemble des *variables* de manière globale. Toute *variable* est par conséquent disponible dans n'importe quel *sous-programme* et à tout moment ; la notion de transfert de paramètres, d'un programme appelant à un *sous-programme* appelé, est ici sans objet.
- Un *sous-programme* est employé pour accomplir une tâche bien déterminée. Lorsque cette tâche est susceptible d'être répétée, un *sous-programme* évite la réécriture de lignes de codes.
- Les *sous-programmes* sont associés au *programme principal* pour composer la « partition ». Une *partition* structurée favorise la compréhension et la maintenance du *programme utilisateur*.

2.7 Sous-programmes prioritaires

Définition

Un *sous-programme prioritaire* est un sous-programme appelé par le changement d'état d'une entrée logique spécifiée. Un sous-programme prioritaire s'intitule *PROG**n***, avec ***n*** le numéro de sous-programme prioritaire considéré. ***n*** est un nombre entier compris entre 1 et 999.
Un sous-programme prioritaire ne suspend pas l'exécution du programme de gestion des mouvements (programme principal ou sous-programmes appelés par celui-ci), son exécution se réalisant en parallèle.

%PROG*n*	La directive *%PROG**n*** figure en « en-tête » du sous-programme prioritaire considéré ; elle définit le début du code relatif à ce sous-programme prioritaire et n'est destinée qu'au compilateur.
END	L'instruction *END* termine l'exécution d'un sous-programme prioritaire. Le pointeur opérationnel retrouve le contexte logiciel qui précédait l'appel au sous-programme prioritaire ; il se positionne en séquence au niveau de l'instruction suivante. Plusieurs instructions *END* peuvent figurer dans un même sous-programme prioritaire. Remarque : L'instruction *RETURN* est réservée aux sous-programmes normaux.
%ENDPROG	La directive *%ENDPROG* figure en « pied de page » d'un sous-programme prioritaire ; elle définit la fin du code relatif au sous-programme prioritaire considéré et n'est destinée qu'au compilateur.

Entrées logiques spécifiées :

Quatre entrées logiques sont susceptibles de déclencher chacune un sous-programme prioritaire. Il s'agit respectivement de : *in0*, *in1*, *in2* et *in3*.
[*cf. PVD 3516 Chapitre 5.4*]

Remarque Importante :

On veillera à ce que les programmes prioritaires soient les plus courts possibles. Ceux-ci ne doivent prendre en compte que la partie primordiale du traitement à réaliser. La partie secondaire du traitement sera effectuée par la suite, par exemple par l'un des programmes automate. Le respect de cette règle permettra de ne pas surcharger ponctuellement la gestion du process.

Gestion des sous-programmes prioritaires :

1] Déclarer le numéro de sous-programme prioritaire et le numéro de l'entrée logique déclenchant son appel.

INTERRUPT*x* = PROG*n*	La déclaration *INTERRUPT**x*** = *PROG**n*** figure le plus souvent dans la zone déclarative du programme principal. (Rappel : La zone déclarative est comprise entre les adresses *#INIT* et *#START*.) ***x*** désigne le numéro de l'entrée logique. C'est un nombre entier compris entre **0** et **3**, correspondant respectivement à *in**0**…in**3***. ***n*** désigne le numéro de sous-programme prioritaire. C'est un nombre entier compris entre 1 et 999. Cette instruction est inutile si aucun programme prioritaire n'est utilisé.

2] Déclarer la nature du front d'interruption (montant ↑ ou descendant ↓). Deux possibilités sont proposées :

2.1] Lors du réglage des paramètres variateur, dans l'environnement *Entrées / Sorties*, onglet *Entrées logiques*
[cf. PVD 3516 Chapitre 5.4] :

- valider **+** dans la rubrique *Front d'interruption* en regard de l'entrée choisie pour un front montant ↑.
- valider **–** pour un front descendant ↓.

2.2] Affecter directement la variable *level_in**x*** par programme :

level_in*x* =	***x*** désigne le numéro de l'entrée logique. C'est un nombre entier compris entre **0** et **3**, correspondant respectivement à *in**0**…in**3***. L'affectation *level_in**x*** = ***0*** spécifie un front montant ↑. L'affectation *level_in**x*** = ***1*** spécifie un front descendant ↓. Attention : seule la valeur ***0*** peut être attribuée à *level_in**0***. L'affectation *level_in**x*** = peut figurer dans le programme principal ou dans un sous-programme appelé préalablement.

3] Ordonner la validation de l'entrée logique :

IT_ON = IN*x*	La déclaration *IT_ON = IN**x*** valide l'entrée logique *in**x*** ***x*** désigne le numéro de l'entrée logique. C'est un nombre entier compris entre **0** et **3**, correspondant respectivement à *in**0**…in**3***. La déclaration *IT_ON = IN**x*** peut figurer dans le programme principal ou dans un sous-programme appelé préalablement. Désormais, à la suite d'un changement d'état, l'entrée logique *in**x*** déclenchera l'appel au sous-programme prioritaire intitulé *PROG**n***.

4] Ordonner, le cas échéant, l'invalidation de l'entrée logique :

IT_OFF = IN*x*	La déclaration *IT_OFF = IN**x*** invalide l'entrée logique *in**x***. ***x*** désigne le numéro de l'entrée logique. C'est un nombre entier compris entre **0** et **3**, correspondant respectivement à *in**0**…in**3***. La déclaration *IT_OFF = IN**x*** peut figurer dans le programme principal ou dans un sous-programme. Désormais, à la suite d'un changement d'état, l'entrée logique *in**x*** ne déclenchera pas l'appel au sous-programme prioritaire intitulé *PROG**n***.

5] **Récapitulatif :**

Entrée logique spécifiée :	Déclaration du sous-programme prioritaire :	Validation de l'entrée logique :	Invalidation de l'entrée logique :
*in**0***	INTERRUPT***0*** = PROG***n***	IT_ON = IN***0***	IT_OFF = IN***0***
*in**1***	INTERRUPT***1*** = PROG***n***	IT_ON = IN***1***	IT_OFF = IN***1***
*in**2***	INTERRUPT***2*** = PROG***n***	IT_ON = IN***2***	IT_OFF = IN***2***
*in**3***	INTERRUPT***3*** = PROG***n***	IT_ON = IN***3***	IT_OFF = IN***3***

Exemple :

Soit le sous-programme prioritaire n°**30**, déclenché par le front montant de l'entrée logique *in**2***.

```
1    %PROG0
2    ; programme principal
3    #INIT
4
5    INTERRUPT2 = PROG30 ; l'instruction figure dans la zone déclarative du programme.
6
7    #START
8    GOSUG PROG10 ; appel au sous-programme n°10
9    …
10   …
11   RESTART
12   %ENDPROG
```

```
20   %PROG10
21   ; sous-programme n°10
22   ; dans cet exemple, c'est le sous-programme 10 qui valide l'entrée logique in2.
23   IT_ON = IN2
24   …
25   …
26   …
27   …
28   RETURN
29   %ENDPROG
```

Sous-programme prioritaire :

```
30   %PROG30
31   ; ce sous-programme prioritaire sera déclenché par l'entrée logique in2 sur un front
32   ; montant.
33   ; le front montant est supposé avoir été choisi lors du réglage des paramètres
34   ; variateur, environnement Entrées / Sorties, onglet Entrées logiques.
35   …
36   …
37   …
38   STOP
39   ; cette instruction suspend les mouvements en cours.
40
41   END
42   %ENDPROG
```

Datation de l'événement interruptif, emploi de *in0*

L'entrée logique ***in0***, associée à un front **montant** ↑, permet la mémorisation de la position *Moteur* et de la position *Codeur extérieur*, à l'instant exact du front montant.
Comme le capteur présente un temps de retard, il est possible de compenser ce dernier dans le paramétrage de l'entrée *in0*. La position réelle mémorisée est alors égale à :

$$Position_{réelle} = Position\ exacte_{front\ montant\ \uparrow} - V*\Delta T$$

où *V* est la vitesse du mobile et ΔT le retard de propagation dû au capteur.

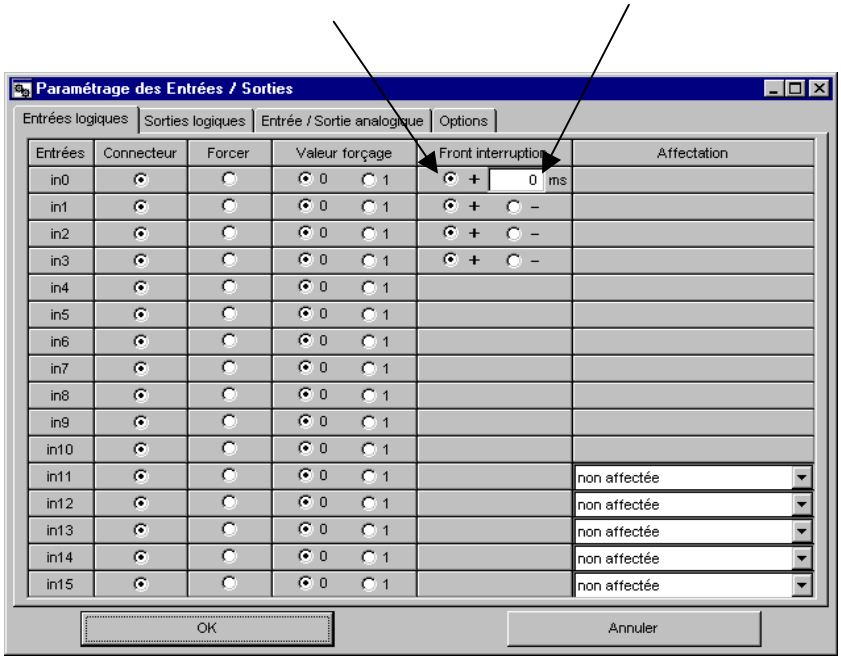
Remarques :

- Les positions réelles corrigées des axes asservi (axe 1) et mesuré (axe 2) sont attribuées respectivement aux variables système *pos1_c* et *pos2_c.* Ces variables sont automatiquement mises à jour lorsque l'entrée logique *in0* passe de 0 à 1 (si le sous-programme prioritaire déclaré par *INTERRUPT0* = *PROGn* est validé par *IT_ON* = *IN0*).
- Dans le sous-programme prioritaire déclenché par ***in0***, les instructions *INDEX* et *SYNCHRO_START* tiennent compte des positions réelles corrigées *pos1_c* et *pos2_c.* Le mode de fonctionnement de ces instructions est explicité au chapitre 3.

Cet environnement de saisie est décrit dans la notice *Logiciel Parvex Motion Explorer - Module DIGIVEX Motion*
[*cf. PVD 3516 Chapitre 5.4.1 Entrées logiques*].

Le temps de retard du capteur raccordé à ***in0*** est renseigné dans cette rubrique.

Seul le front montant est proposé pour ***in0***.



Remarques concernant les sous-programmes prioritaires :

- Un sous-programme prioritaire n'est pas un sous-programme traité sous interruption, dans le sens où il n'interrompt pas les autres programmes en cours d'exécution. Son caractère « prioritaire » provient du fait qu'il est ordonnancé dès que survient la transition d'état sur l'entrée logique.
- Le système opérationnel est gestionnaire du temps de calcul attribué à chaque tâche. Lorsqu'il ne dispose pas suffisamment de temps pour dérouler un sous-programme prioritaire en une seule fois, ce dernier sera segmenté dans son exécution. Plusieurs périodes d'échantillonnage peuvent s'avérer nécessaires pour exécuter complètement un sous-programme prioritaire de grande taille.

2.8 Programmes automate

2.8.1 Programme automate de type 1

Définition

Un programme automate de type 1 (automate générique) est un programme spécifique qui s'exécute de façon continue et "parallèlement" aux autres sous-programmes utilisateur ; il s'intitule *PROG**n***, avec ***n*** le numéro du programme automate considéré. ***n*** est un nombre entier compris entre 1 et 999.

%PROG*n*	La directive *%PROG**n*** figure en « en-tête » du programme automate de type 1 ; elle définit le début du code relatif au programme automate de type 1 et n'est destinée qu'au compilateur.
END	L'instruction *END* termine l'exécution d'une séquence de programme automate et relance ce dernier à son début pour enchaîner un nouveau cycle.
%ENDPROG	La directive *%ENDPROG* figure en « pied de page » d'un programme automate de type 1 ; elle définit la fin du code relatif au programme automate considéré et n'est destinée qu'au compilateur.

Gestion des programmes automate de type 1 :

1] Déclarer le numéro de programme automate de type 1.

PLC1 = PROGn	La déclaration *PLC1 = PROG**n*** figure le plus souvent dans la zone déclarative du programme principal. **n** désigne le numéro de programme automate de type 1. Selon les applications développées, *PLC1 = PROGn* peut figurer dans un sous-programme voire un programme automate de type 2.

2] Déclarer le temps de cycle du programme automate de type 1.

DEFPLC1 = t	Cette déclaration se situe de préférence dans la zone déclarative du programme principal. La déclaration *DEFPLC1 = **t*** fixe le temps de cycle maximal qui sera accordé à l'exécution d'une séquence complète de programme automate. ***t*** est un nombre entier, ou une variable entière [*format E*], dont la valeur est comprise entre 1 et 10^6 millisecondes. • Si le temps d'exécution devient supérieur à la durée déclarée par *DEFPLC1*, le « time out » survient et déclenche un défaut de programmation. • Si *DEFPLC1 = t* ne figure pas dans le programme, le temps de cycle maximal est alloué par défaut et vaut 10^6 ms.

3] Démarrer le programme automate de type 1.

PLC1 = START	L'instruction *PLC1 = START* figure le plus souvent dans la zone déclarative du programme principal. Elle a pour but de démarrer l'exécution du programme automate de type 1 ou éventuellement de le relancer, s'il avait été précédemment arrêté (Dans ce cas le programme automate redémarrera **à son début**). Selon les applications développées, *PLC1 = START* peut figurer dans un sous-programme voire un programme automate de type 2. N.B. Le pointeur opérationnel doit obligatoirement rencontrer *PLC1 = START* <u>après</u> *PLC1 = PROG**n***

4] Arrêter le programme automate de type 1.

PLC1 = STOP	L'instruction *PLC1 = STOP* arrête l'exécution du programme automate de type 1. Elle peut figurer dans le programme principal, dans un sous-programme ou un programme automate de type 1 ou 2. Cette instruction offre certaines possibilités de programmation ; son emploi doit être justifié et non systématisé.

5] Déclarer l'inexistence d'un programme automate de type 1.

PLC1 = NONE	La déclaration *PLC1 = NONE* mentionne qu'il n'y a pas de programme automate de type 1 dans la *partition* des programmes. Elle figure généralement dans la zone déclarative du programme principal. Les instructions *PLC1 = START* et *PLC1 = STOP* rendues caduques par *PLC1 = NONE*, sont ignorées par le système opérationnel.

<u>Exemple :</u>

```
1   %PROG0
2   ; programme principal
3   #INIT
4   PLC1 = PROG100          ; désignation du programme automate de type 1.
5   DEFPLC1 = 1000          ; le time out sera déclenché pour un temps de cycle > à 1s
6   PLC1 = START            ; ordre de démarrage du programme automate de type 1.
7   #START
8         GOSUG PROG10      ; appel au sous-programme n°10.
9         GOSUB PROG20      ; appel au sous-programme n°20.
10  …
11  …
12  RESTART
13  %ENDPROG
```

Programme automate de type 1 :

```
14  %PROG100
15  ; le programme n° 100 est un programme automate de type 1, désigné comme tel et
16  ; démarré dans le programme  principal PROG0.
17  ; description de l'automatisme séquentiel de la machine :
18    IF in7 = 1 AND in3 = 0    THEN   out1 = 1
19                              ELSE   out2 = 1
20    ENDIF
21  …
22  …
23  …
24  END
25  ;l'instruction END provoque la reprise d'exécution du programme automate à son début.
26  %ENDPROG
```

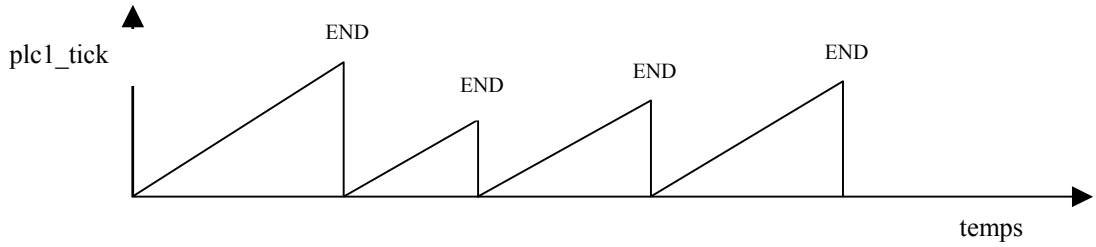
Remarques générales concernant les programmes automate :

- Un programme automate présente la même « priorité » que tout autre programme actif de la *partition*.
- Les labels *#INIT* et *#START* ne doivent pas figurer dans un programme automate.

Particularités d'un programme automate de type 1 :

- Un programme automate de type 1 fonctionne en « tâche de fond » : il tourne en permanence et se reboucle sur lui-même dès que le pointeur opérationnel rencontre l'instruction *END*.
- Un seul programme automate de type 1 est actif à la fois.
- Le temps de cycle d'un programme automate de type 1 n'est pas constant. La visualisation de la variable *plc1_tick* (compteur de passages dans le programme automate) à l'aide de l'outil *Oscilloscope*, illustre ce fonctionnement.

La variable système *plc1_tick* est incrémentée d'une unité à chaque fois que le pointeur opérationnel accède au programme automate de type 1. Lorsque ce dernier rencontre l'instruction *END*, une nouvelle séquence de programme automate débute.



- La description d'un programme automate de type 1 peut faire appel à de nombreuses lignes d'instructions. Il n'est pas limité en taille. Un temps de passage lui est alloué toutes les 250 μs.

Quand choisir une programmation automate de type 1 ?

La gestion de l'environnement machine (sécurités, marche - arrêt, automatismes séquentiels usant des entrées sorties logiques, supervision) est associée de préférence à un programme automate de type 1.

2.8.2 Programme automate de type 2

Définition

Un programme automate de type 2 (programme automate cyclique) est un programme spécifique dont l'exécution cyclique présente un « temps de cycle » contrôlé ; il se déroule "parallèlement" aux autres sous-programmes utilisateur. Il s'intitule *PROG**n***, avec ***n*** le numéro du programme automate considéré. ***n*** est un nombre entier compris entre 1 et 999.

%PROG*n*	La directive *%PROG**n*** figure en « en-tête » du programme automate de type 2 ; elle définit le début du code relatif au programme automate de type 2 et n'est destinée qu'au compilateur.
END	L'instruction *END* termine l'exécution d'une séquence de programme automate. Elle relance ce dernier à son début pour enchaîner un nouveau cycle à la fin de la période fixée par *DEFPLC2*.
%ENDPROG	La directive *%ENDPROG* figure en « pied de page » d'un programme automate de type 2 ; elle définit la fin du code relatif au programme automate considéré et n'est destinée qu'au compilateur.

Gestion des programmes automate de type 2 :

1] Déclarer le numéro de programme automate de type 2.

PLC2 = PROG*n*	La déclaration *PLC2 = PROG**n*** figure le plus souvent dans la zone déclarative du programme principal. ***n*** désigne le numéro de programme automate de type 2. Selon les applications développées, *PLC2 = PROG**n*** peut figurer dans un sous-programme voire un programme automate de type 1.

2] Déclarer la périodicité d'exécution du programme automate de type 2.

DEFPLC2 = t	Cette déclaration se situe de préférence dans la zone déclarative du programme principal La déclaration *DEFPLC2* = ***t*** fixe la périodicité de la séquence de traitement. **t** est un nombre entier, ou une variable entière [*format E*], dont la valeur est comprise entre 1 et 10^6 millisecondes. • Il appartient au programmeur de vérifier si la séquence de programme « passe » effectivement dans le temps de cycle prescrit par *DEFPLC2* ⇒ visualiser à cet effet la variable *plc2_tick* à l'aide de la fonction *Oscilloscope* (voir illustrations sur le graphique ci-après). • Si le temps d'exécution devient supérieur à la durée déclarée par *DEFPLC2*, le « time out » survient et déclenche un défaut de programmation. • Si *DEFPLC2* = ***t*** ne figure pas dans le programme, la périodicité d'exécution est fixée par défaut et vaut *100 ms*.

3] Démarrer le programme automate de type 2.

PLC2 = START	L'instruction *PLC2 = START* figure le plus souvent dans la zone déclarative du programme principal. Elle a pour but de démarrer l'exécution du programme automate de type 2 ou éventuellement de le relancer, s'il avait été précédemment arrêté (Dans ce cas le programme automate redémarrera **à son début**). Selon les applications développées, *PLC2 = START* peut figurer dans un sous-programme voire un programme automate de type 1. N.B. Le pointeur opérationnel doit obligatoirement rencontrer *PLC2 = START* <u>après</u> *PLC2 = PROG**n***

4] Arrêter le programme automate de type 2.

PLC2 = STOP	L'instruction *PLC2 = STOP* arrête l'exécution du programme automate de type 2. Elle peut figurer dans le programme principal, dans un sous-programme ou un programme automate de type 1 ou 2. Bien que cette instruction soit disponible, elle présente peu d'intérêt et sera rarement employée.

5] Déclarer l'inexistence d'un programme automate de type 2.

PLC2 = NONE	La déclaration *PLC2 = NONE* mentionne qu'il n'y a pas de programme automate de type 2 dans la *partition* des programmes. Elle figure généralement dans la zone déclarative du programme principal. Les instructions *PLC2 = START* et *PLC2 = STOP*, rendues caduques par *PLC2 = NONE*, sont ignorées par le système opérationnel.

<u>Exemple :</u>

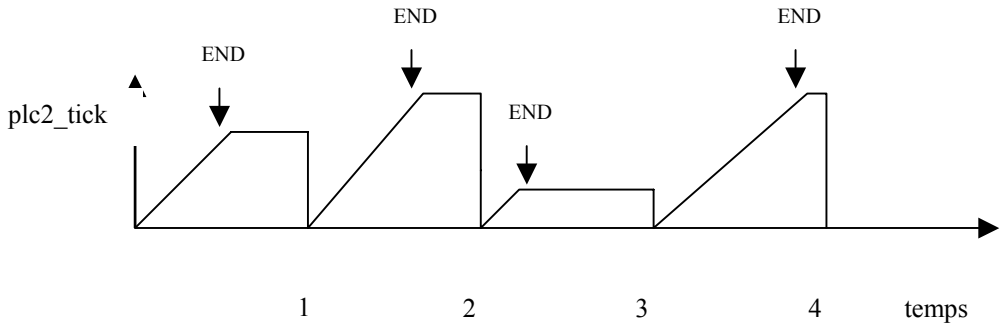
```
1    %PROG0
2    ; programme principal.
3    #INIT
4    PLC2 = PROG200      ; désignation du programme automate de type 2.
5    DEFPLC2 = 125       ; le temps de cycle est fixé à 125 ms.
6    PLC2 = START        ; ordre de démarrage du programme automate de type 2.
7    #START
8      GOSUB PROG15      ; appel au sous-programme n°15.
9      GOSUB PROG25      ; appel au sous-programme n°25.
10   …
11   …
12   RESTART
13   %ENDPROG
```

Programme automate de type 2 :

```
14   %PROG200
15   { le programme n° 200 est un programme automate de type 2, désigné comme tel et
16   démarré dans le programme principal PROG0.}
17    IF in7 = 1 THEN   ui0 =ui0+1   ; compteur incrémenté de 1 unité toutes les 125 ms
18   ENDIF
19   …
20   …
21   END   { l'instruction END provoque la reprise d'exécution au début du programme
22             automate.}
23   %ENDPROG
```

Particularités d'un programme automate de type 2 :

- Un programme automate de type 2 fonctionne en « tâche de fond » : il tourne en permanence et se reboucle sur lui-même de manière cyclique (périodicité programmée par *DEFPLC2*).
- Un seul programme automate de type 2 est actif à la fois.
- La description d'un programme automate de type 2 doit faire appel à un nombre d'instructions restreint. Un temps de passage lui est alloué toutes les 250 µs. Pour qu'il puisse être entièrement exécuté dans le « temps de cycle » fixé par *DEFPLC2*, il est obligatoirement limité en taille.
- En suivant les recommandations énoncées précédemment, le « temps de cycle » d'un programme automate de type 2 est constant.



La variable système *plc2_tick* est incrémentée d'une unité à chaque fois que le pointeur opérationnel accède au programme automate de type 2. Lorsque ce dernier rencontre l'instruction *END*, une nouvelle séquence de programme automate débute à la fin de la période fixée par *DEFPLC2*.

L'illustration montre que le « temps de cycle » est constant : il existe des laps de temps, où la variable *plc2_tick* présente des plateaux. Ceci signifie que la durée spécifiée avec *DEFPLC2* n'a pas été dépassée lors de l'exécution d'une séquence complète de programme automate.

Quand choisir un programme automate de type 2 ?

L'intérêt majeur de ce type de programme réside dans le fait que son « temps de cycle » est constant et fixé par l'utilisateur. Les exemples qui suivent sont associés de préférence à un programme automate cyclique du type 2 :

- Lecture périodique d'une information,
- Affichage « temps réel » d'une variable,
- Calcul récurrent à une cadence déterminée,
- Algorithme d'asservissement particulier qui modifie de manière cyclique le mouvement, suite à la mesure de grandeurs externes (par exemple : une pression, une température, une force, etc…).
- …

2.9 Flags

Définition :
Un *flag* est un mot logique adoptant la valeur booléenne *1* ou *0,* selon qu'une variable choisie dans une liste prédéfinie, appartient ou non, à un intervalle spécifié. (Le mot *flag*, en français « drapeau », rappelle l'emploi des sémaphores ferroviaires qui autorisent ou interdisent les voies aux circulations).

Le système gère deux *flags* dénommés respectivement *flag0* et *flag1*. La périodicité de leur mise à jour est de 500 µs [*flag0* et *flag1* sont rafraîchis alternativement toutes les 250 µs].

Gestion :

Les *flags* sont gérés par l'intermédiaire de *variables.* ⇒ [*cf. documentation : PVD 3527 DIGIVEX Motion, Répertoire des variables, chapitre 6.4, Flags*].

- Le numéro du *flag* concerné est renseigné par la valeur que prend ***n*** dans l'instruction *FLAG* (***n*** = 0 ou 1 respectivement pour *flag**0*** ou *flag**1***).
- La *variable* signalée par le *flag* est renseignée comme premier argument de l'instruction *FLAG* (voir description ci-après).
- Les bornes inférieure et supérieure de l'intervalle de test sont mentionnées respectivement comme 2ème et 3ème argument de l'instruction *FLAG* (voir description ci-après).
- La valeur booléenne du *flag* est attribuée à la variable système *flag**n***.

Variables signalées par *flag**n*** :		valeur prise par les variables d'affectation *a_flag0* et *a_flag1*
none	pas d'affectation (situation par défaut)	0
pos1	position réelle axe asservi	1
pos2	position réelle axe mesuré	2
pos_th	consigne de position théorique	3
tracking_error	erreur de poursuite	4
speed1	vitesse réelle axe asservi	5
speed2	vitesse réelle axe mesuré	6
speed_th	consigne de vitesse théorique	7
synchro_error	erreur de vitesse en synchro (*master_Vn-master_Vf)*	8
i_setpoint	consigne de courant	9
uf0	variable utilisateur	10
uf1	variable utilisateur	11
ui0	variable utilisateur	12
ui1	variable utilisateur	13
ina	entrée analogique, état après masque	14
outa	sortie analogique	15
torque_setpoint	consigne de couple	16
var0	variable physique	17
var1	variable physique	18
filter0	variable filtrée	19
filter1	variable filtrée	20

Syntaxe :

FLAG (***n,*** arg1,arg2,arg3) avec

n : numéro du *flag* concerné (***n*** = 0 ou 1)
arg1 : variable signalée par *flag**n***
arg2 : borne inférieure de l'intervalle de test
arg3 : borne supérieure de l'intervalle de test

Exemple :

- Le *flag0* est utilisé pour signaler si la position réelle de l'axe mesuré [variable système *pos2*] est comprise entre 100 et 200 *unit2*. L'instruction à programmer est la suivante :

  FLAG (***0,*** pos2,100,200)

  La variable système *flag0* sera égale :
  à *1* si *pos2* est comprise entre 100 et 200 *unit2* et
  à *0* dans le cas contraire.

  L'opérateur peut ensuite utiliser *flag0* dans les structures de contrôle du type « sélection » ou « répétition » :

```
IF flag0 = 1  THEN GOSUB PROG40
              ELSE GOSUB PROG41
ENDIF
```

Remarque :

- A l'initialisation, les variables *flag0* et *flag1* présentent des valeurs nulles (les variables d'affectation correspondantes *a_flag0* et *a_flag1* sont mises à *0* par défaut [*cf. PVD 3527 Répertoire des variables, chapitre 6.4, Flags*]).

2.10 Filtres

Définition :

Le système gère deux filtres dont la périodicité de mise à jour est de 250 µs. Ces filtres fournissent deux variables *filter0* et *filter1* qui représentent les valeurs filtrées des variables spécifiées par la fonction FILTER().

Gestion :

Les *filtres* sont gérés par l'intermédiaire de *variables.* ⇒ [*cf. documentation : PVD 3527 DIGIVEX Motion, Répertoire des variables, chapitre 6.5, Filtres*].

- Le numéro du *filtre* concerné est renseigné par la valeur que prend *n* dans l'instruction *FILTER* (*n* = 0 ou 1 respectivement pour *filter0* ou *filter1*).
- La *variable* prise en compte par le *filtre* est renseignée comme premier argument de l'instruction *FILTER* (voir description ci-après).
- Le coefficient de filtrage est spécifié par le 2ème argument de l'instruction *FILTER* (voir description ci-après).
- La valeur en sortie du *filtre* est attribuée à la variable système *filtern*.

Variables signalées par *filtern* :		valeur prise par les variables d'affectation *a_filter0* et *a_filter1*
none	pas d'affectation (situation par défaut)	0
pos1	position réelle axe asservi	1
pos2	position réelle axe mesuré	2
pos_th	consigne de position théorique	3
tracking_error	erreur de poursuite	4
speed1	vitesse réelle axe asservi	5
speed2	vitesse réelle axe mesuré	6
speed_th	consigne de vitesse théorique	7
synchro_error	erreur de vitesse en synchro (*master_Vn-master_Vf)*	8
i_setpoint	consigne de courant	9
uf0	variable utilisateur	10
uf1	variable utilisateur	11
ui0	variable utilisateur	12
ui1	variable utilisateur	13
ina	entrée analogique, état après masque	14
outa	sortie analogique	15
torque_setpoint	consigne de couple	16
var0	variable physique	17
var1	variable physique	18

- La formule de filtrage est la suivante (réactualisation toutes les 250 µs) :

filter*n* = [*arg2* ∗ *variable_spécifiée_par_arg1*] + [(1 - *arg2*) ∗ *valeur_précédente_de_filtern*]

constante de temps du filtre : τ = 250 e^{-6} / *arg2*

Syntaxe :

FILTER (***n****,* arg1,arg2) avec

n **:** numéro du *filtre* concerné (*n* = 0 ou 1)
arg1 : nom de la variable prise en compte par *filtern*
arg2 : coefficient de filtrage (0 < *arg2* < 1)

Exemple :

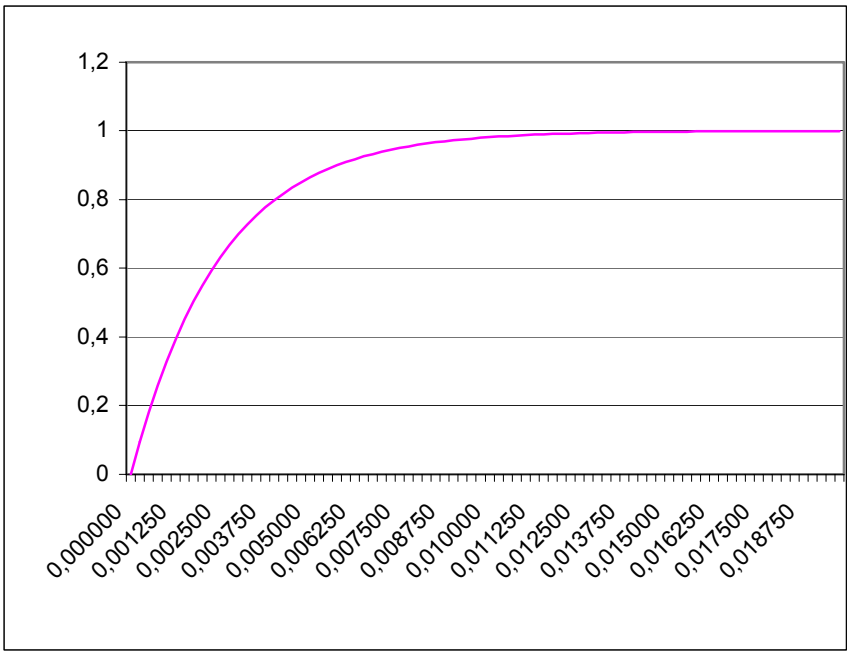
- *filter0* est utilisé pour filtrer la position réelle d'un axe mesuré [variable système *pos2*] avec une constante de temps de 2,5 ms (τ = 250 e^{-6} / 0.1 = 2.5 ms). L'instruction à programmer est la suivante :

  FILTER (*0,* pos2,0.1)

  La variable système *filter0* correspond à :
  filter0 = 0.1 ∗ *pos2* + 0.9 ∗ *filter0*

Remarque :

- A l'initialisation, les variables d'affectation correspondantes *a_filter0* et *a_filter1* sont mises à *0* par défaut [*cf. PVD 3527 Répertoire des variables, chapitre 6.5, Filtres*].
- La réponse à un échelon 0 → 1 d'un filtre de constante de temps τ = 2.5ms sera la suivante :



à un temps correspondant à τ, la valeur filtrée sera égale à 0.63

à un temps correspondant à 3τ, la valeur filtrée sera égale à 0.95

2.11 Programme de gestion d'erreurs

Définition :

Un programme de gestion d'erreurs est un sous-programme spécifique appelé par l'instruction *EXEC_ERR*. C'est une « routine d'exception » qui permet de suspendre momentanément l'exécution du programme de gestion des mouvements (programme principal et sous-programmes afférents). Les instructions du programme de gestion d'erreurs sont alors exécutées en séquence, en lieu et place.

Mise en œuvre :

Le programme de gestion d'erreurs est déclaré par l'instruction *ERROR = PROG**n***, avec ***n*** un nombre entier compris entre 1 et 999.

La fin du programme de gestion d'erreurs, indiquée par l'instruction *END*, engendre la reprise d'exécution du programme de gestion des mouvements à l'endroit où il fut précédemment interrompu.

L'instruction *RESTART* rencontrée dans le programme de gestion d'erreurs, entraîne le retour d'exécution à l'adresse *#START* du programme principal *PROG0*.

L'instruction *PROG_INIT* rencontrée dans le programme de gestion d'erreurs, provoque le retour d'exécution à l'adresse *#INIT* du programme principal *PROG0*.

Exemple :

Dans l'exemple qui suit, le programme de gestion d'erreurs est déclaré dans le programme principal *PROG0*. Il porte le nom *PROG999*. Le programme automate *PROG200* surveille la valeur de la variable utilisateur *uf0*. Si cette dernière devient négative ou nulle, le programme de gestion d'erreurs est appelé. Selon les valeurs prises par les variables utilisateur *ui1* et *ui2*, le programme de gestion d'erreurs provoque le retour d'exécution :

- au programme principal à la ligne 3 (adresse *#INIT*) si *ui1* = 1 ,
- au programme principal à la ligne 9 (adresse *#START*) si *ui1* ≠ 1 et *ui2* = 1 ,
- au programme de gestion des mouvements précédemment interrompu (*PROG0, PROG10 ou PROG20)*, si *ui1* ≠ 1 et *ui2* ≠ 1 .

```
1     %PROG0
2      ; programme principal
3     #INIT
4     PLC1 = PROG200
5     DEFPLC1 = 1000
6     PLC1 = START
7     ERROR = PROG999
8      ; cette instruction déclare le programme n° 999 comme étant un programme d'erreur.
9     #START
10            GOSUB PROG10
11            GOSUB PROG20
12       …
13       …
14    RESTART
15    %ENDPROG
```

```
16   %PROG200
17    ; programme automate
18   …
19   …
20   …
21   IF uf0<=0 THEN EXEC_ERR ; Appel au programme de gestion d'erreurs
22   uf0=uf0+10
23   …
24   END
25   %ENDPROG
```

```
26   %PROG999
27    ; voici un exemple de programme de gestion d'erreurs
28   ADR = 1
29   PAGE = 2
30   LINE = 3
31   COL = 4
32   CLEAR_PAGE ; effacement de la page 2 sur abonné CAN n°1
33   PRINT(" valeur uf0 incorrecte ! ")
34   ; affichage du message sur l'abonné au bus CAN n°1 à la page 2, ligne 3, colonne 4.
35   IF ui1 = 1    THEN PROG_INIT
36                 ; retour au programme principal au label #INIT si ui1 = 1.
37                 ELSE
38                 IF ui2 = 1THEN RESTART
39                 ; retour au programme principal au label #START si ui1 ≠ 1 et ui2 = 1 .
40                 ENDIF
41   ENDIF
42   END ; retour au programme de gestion des mouvements précédemment interrompu si ui1 ≠ 1 et
     ui2 ≠ 1
43   %ENDPROG
```

2.12 Durée d'exécution des instructions

Il n'est pas possible de définir avec précision quel sera le temps d'exécution d'une suite d'instructions. En contrepartie, quelques grandes règles générales peuvent être établies :

- Pour l'ensemble des programmes, environ une dizaine d'instructions élémentaires assembleur pourront être exécutées cycliquement (un cycle toutes les 250 µs).
- Lorsqu'un programme prioritaire (%INTERRUPT0, %INTERRUPT1, %INTERRUPT2 ou %INTERRUPT3) est activé, il lui est réservé la totalité des possibilités d'exécution. Le programme principal et les programmes automates ne passent plus (un programme prioritaire devra donc être le plus court possible).
- Lorsque deux programmes prioritaires sont activés en même temps, c'est le programme prioritaire portant le plus petit numéro qui sera servi en premier (%INTERRUPT0).
- Lorsqu'aucun programme prioritaire n'est activé, le programme principal et les programmes automate se partagent équitablement les possibilités d'exécution : première moitié des possibilités d'exécution pour le programme principal, l'autre moitié pour les programmes automate (PLC1 et PLC2).

La méthode la plus simple et la plus précise pour estimer le temps d'exécution d'un programme consiste à introduire un mouchard en début et en fin de programme (variable mise à 1 puis à 0) et à l'observer sur plusieurs cycles de programme.

Remarque :

Après compilation, les instructions Basic_DM se décomposent en une ou plusieurs instructions élémentaires assembleur. On visualise les instructions en langage assembleur en ouvrant le fichier .asm correspondant au programme Basic_DM. Ce fichier n'est visualisable que si l'option de compilation "Conserver les fichiers intermédiaires" a été validée.

Programme assembleur

```
39  %line 66
40  WTB1 ub0
41  %line 69
42  SET out4,0
43  %line 72
44  SET wi0,ui0
45  ADD wi0,1
46  SET ui0,wi0
47  %line 75
48  SET speed_move,10
49  %line 78
50  EQ in2,0
51  SET wb1,wb0
52  SET wb0,wb1
53  JZR #LABEL10002
54  %line 79
55  MVR 20
56  %line 80
57  JMP #LABEL10003
58  #LABEL10002
59  %line 81
60  MVA 0
61  %line 82
62  #LABEL10003
63  %line 85
64  WTB1 in_position
```

1 instruction élémentaire

3 instructions élémentaires

4 instructions élémentaires

1 instruction élémentaire

Programme Basic_DM

```
65  ; waiting for ub0=1
66  WAIT_UNTIL ub0=1
67
68  ; reset output out4
69  out4 = 0
70
71  ; counter
72  ui0 = ui0 + 1
73
74  ; speed
75  SPEED = 10
76
77  ; move choice
78  IF in2=0 THEN
79        MOVER = 20        ; relative position setpoint
80  ELSE
81        MOVEA = 0         ; absolute position setpoint
82  ENDIF
83
84  ; waiting for moving completed
85  WAIT_UNTIL in_position=1
```

3. GUIDE DES INSTRUCTIONS

3.1 Présentation thématique

3.1.1 Déclarations

%PROG***n***	Définit le nom du programme principal (***n***=0) ou d'un sous-programme (0<**n**≤999).
%ENDPROG	Définit la fin du code spécifiant un programme ou un sous-programme.
%VDM=	Définit le nom et le chemin d'accès au fichier contenant les noms « en clair » des variables utilisateur
INTERRUPT***x*** = PROG***n***	Définit le nom du sous-programme prioritaire activé par l'entrée logique interruptive in***x*** (si *IT_ON = IN**x***)
ERROR = PROG***n***	Définit le nom du sous-programme de gestion d'erreurs (0<**n**≤999).
PLC1 = PROG***n***	Déclare le programme automate de type 1 (0<**n**≤999).
PLC1 = NONE	Déclare qu'il n'y a pas de programme automate de type 1.
PLC2 = PROG***n***	Déclare le programme automate de type 2 (0<**n**≤999).
PLC2 = NONE	Déclare qu'il n'y a pas de programme automate de type 2.
DEFPLC1 = ***t***	Définit le temps de cycle maxi ***t*** alloué au programme automate de type 1.
DEFPLC2 = ***t***	Définit la périodicité d'exécution ***t*** du programme automate de type 2.
#INIT	Adresse de démarrage de l'exécution du programme principal *PROG0* .
#START	Adresse du début du corps de programme principal *PROG0*.
#LABEL	Adresse de branchement ordinaire.
{texte**}**	Définit un commentaire développé sur plusieurs lignes.
;texte	Définit un commentaire limité à une seule ligne.

3.1.2 Instructions de contrôle des programmes

PLC1 = START	Démarre l'exécution du programme automate de type 1.
PLC1 = STOP	Arrête l'exécution du programme automate de type 1.
PLC2 = START	Démarre l'exécution du programme automate de type 2.
PLC2 = STOP	Arrête l'exécution du programme automate de type 2.
RETURN	Définit la fin d'un sous-programme ; provoque le retour d'exécution à la ligne suivant le *GOSUB appelant*.
END	Fin de programme : de type automate, de gestion d'erreurs ou prioritaire.

3.1.3 Instructions de branchement

GOSUB PROG***n***	Appelle le sous-programme intitulé *PROG**n***.
GOSUB PROG(variable utilisateur)	Appelle le sous-programme spécifié par la *variable utilisateur* entre parenthèses.
EXEC_ERR	Appelle le programme de gestion d'erreurs déclaré par *ERROR = PROG**n***.
GOTO #LABEL	Définit un saut inconditionnel à l'adresse #*LABEL*.
RESTART	Définit un saut à l'adresse #*START* du programme principal *PROG0*.
PROG_INIT	Définit un saut à l'adresse #*INIT* du programme principal *PROG0*.

3.1.4 Gestion des temporisations et attentes

timer***n*** = valeur tempo	Charge le timer ***n*** à la *valeur tempo* ; le timer est décrémenté de 1 unité toutes les *ms*.
WAIT = valeur tempo	Spécifie une temporisation exprimée en *ms*.
WAIT_UNTIL condition	Définit une attente conditionnelle.

3.1.5 Gestion des conditions et répétitions

IF condition THEN
action1
ELSE
action2
ENDIF
	Définit une structure de contrôle de type *sélection*.

FOR variable = n TO m action NEXT	Définit une structure de contrôle de type *répétition indexée*.
WHILE condition DO action WEND	Définit une structure de contrôle de type *répétition « tant que »*.

3.1.6 Opérateurs et fonctions mathématiques

+	Addition.
-	Soustraction.
/	Division.
*	Multiplication.
**	Puissance.
%%	Modulo.
NOT	Opérateur logique NON.
AND	Opérateur logique ET.
NAND	Opérateur logique NON ET (ET complémenté).
OR	Opérateur logique OU inclusif.
NOR	Opérateur logique NON OU (OU complémenté).
XOR	Opérateur logique OU exclusif.
COS	Cosinus (argument en **radians**).
SIN	Sinus (argument en **radians**).
TAN	Tangente (argument en **radians**).
ACOS	Arc cosinus (résultat en **radians**).
ASIN	Arc sinus (résultat en **radians**).
ATAN	Arc tangente (résultat en **radians**).
DCOS	Cosinus (argument en **degrés**).
DSIN	Sinus (argument en **degrés**).
DTAN	Tangente (argument en **degrés**).
DACOS	Arc cosinus (résultat en **degrés**).
DASIN	Arc sinus (résultat en **degrés**).
DATAN	Arc tangente (résultat en **degrés**).
EXP	Exponentielle.
LN	Logarithme népérien.
LOG	logarithme base 10.
SQR	Racine carrée.
ABS	Valeur absolue.
FIX	Partie entière.
FRAC	Partie fractionnaire.
MIN(val1,val2)	Minimum de deux valeurs.
MAX(val1,val2)	Maximum de deux valeurs.
FLOAT	Conversion d'une variable de type entier en flottant ou double précision.
INT	Conversion d'une variable de type flottant ou double précision en entier.

3.1.7 Constantes prédéfinies

pi	π.
kpi_180	$2\pi/360$.
k180_pi	$360/2\pi$.

3.1.8 Instructions de programmation des filtres et des flags

FILTER (***n,*** arg1, arg2)	Programmation d'un *filtre*. ***n*** = **0** pour ***filter0*** et ***n*** = **1** pour ***filter1*** *arg1* : variable prise en compte, *arg2* : coefficient de filtrage.
FLAG (***n,*** arg1, arg2, arg3)	Programmation d'un *flag*. ***n*** = **0** pour ***flag0*** et ***n*** = **1** pour ***flag1*** *arg1* : variable signalée, *arg2* : borne inférieure, *arg3* : borne supérieure.

3.1.9 Sauvegarde de variables en mémoire EEPROM

SAVE	Sauvegarde la valeur d'une variable utilisateur mémorisable en EEPROM.

3.1.10 Modification provisoire de paramètres machine

kp = valeur gain	Permet de modifier le gain de la boucle de position.
trackerror_max = valeur	Redéfinit la valeur d'erreur de poursuite maxi.
target = valeur	Redéfinit la valeur de la fenêtre d'arrêt.
softlimit_p = valeur	Redéfinit la valeur du fin de course logiciel **+**.
softlimit_m = valeur	Redéfinit la valeur du fin de course logiciel **-**.
hardlimit_en = 0	Invalide la prise en compte des fins de course électriques.
hardlimit_en = 1	Valide la prise en compte des fins de course électriques.
softlimit_en = 0	Invalide la prise en compte des fins de course logiciels.
softlimit_en = 1	Valide la prise en compte des fins de course logiciels.
drive_mode = 0	Déclare un pilotage en position de l'axe
drive_mode = 1	Déclare un pilotage en vitesse de l'axe
drive_mode = 2	Déclare un pilotage en courant de l'axe

3.1.11 Gestion des synchronisations maître / esclave

KSYNC = valeur	Définit la valeur du rapport de recopie (coefficient de synchronisation entre axe maître et axe esclave).
SYNCHRO_START	Valide une recopie d'axe dans le sous-programme déclaré par *INTERRUPT**0** = PROGn*.
SYNCHRO	Valide une recopie d'axe.
ABORT	Permet d'arrêter tout déplacement en cours et la recopie d'axe.

3.1.12 Instructions relatives au déplacement

FSPEED = valeur vitesse finale	Définit la vitesse finale en fin de bloc (*FSPEED* = 0 par défaut).
FSPEED_IM = valeur vitesse finale	Définit la vitesse finale en fin de bloc immédiatement.
SPEED = valeur vitesse	Définit la vitesse de déplacement.
SPEED_IM = valeur vitesse	Définit la vitesse de déplacement immédiatement.
ACCEL = valeur accel	Définit l'accélération et la décélération lors d'un déplacement.
ACCEL_IM = valeur accélération	Définit l'accélération et la décélération immédiatement.
STOP	Suspend les mouvements en cours (variable *move_en = 0*).
START	Reprend l'exécution de mouvements suspendus (variable *move_en = 1*).
ABORT	Arrête tout déplacement en cours.
HOME	Appelle la séquence de prise d'origine.
MOVEA = cote à atteindre	Spécifie un mouvement absolu.
MOVER = valeur déplacement	Spécifie un mouvement relatif.
FORWARD	Définit un mouvement continu dans le sens **+**.
REVERSE	Définit un mouvement continu dans le sens **-**.
INDEX = valeur déplacement	Spécifie un mouvement relatif en mode *Stop cote* (s'emploie exclusivement dans le sous-programme déclaré par *INTERRUPT**0** = PROGn*).
master_speedmax = valeur	Définit la vitesse max. de ligne dans des applications de coupe à longueur

3.1.13 Commandes diverses-modification de variables

ki_red = valeur	Définit le coefficient de réduction appliqué au courant de limitation moteur (*ki_red* $\in$ [0,1]).
speed_att = valeur	Définit la valeur de l'atténuation de vitesse appliquée à la vitesse programmée (*speed_att* $\in$ [0,1]).
torque_cmd = **1**	Demande la mise sous couple du moteur.
torque_cmd = **0**	Demande la mise à couple nul du moteur.
brake_cmd = **1**	Demande la fermeture du frein.
brake_cmd = **0**	Demande l'ouverture du frein.
brake_emergency = 1	Commande la fermeture d'urgence du frein
emergency_cmd = 1	Commande l'arrêt d'urgence des mouvements
reset_cmd = 1	Commande le « reset » des défauts
speed_value = valeur	Définit la valeur de la consigne de vitesse en mode commande en vitesse
torque_value = valeur	Définit la valeur de la consigne de couple en mode commande en courant

3.1.14 Commande relative à la gestion de position

variable utilisateur = pos1	Range la position de l'axe 1 lue précédemment dans la variable utilisateur spécifiée.
variable utilisateur = pos2	Range la position de l'axe 2 lue précédemment dans la variable utilisateur spécifiée.
DEFPOS1 = valeur	Redéfinit la position de l'axe 1 (cette instruction n'engendre aucun mouvement).
DEFPOS2 = valeur	Redéfinit la position de l'axe 2 (cette instruction n'engendre aucun mouvement).

3.1.15 Gestion de l'entrée analogique

variable = in**a** scale_in**a** = valeur	Affecte la valeur de l'entrée analogique à la variable spécifiée. Permet de changer le facteur d'échelle de l'entrée analogique.

3.1.16 Gestion de la sortie analogique

out**a** = valeur numérique out**a** = variable scale_out**a** = valeur	Impose une tension sur la sortie analogique. Impose une tension sur la sortie analogique. Permet de changer le facteur d'échelle de la sortie analogique.

3.1.17 Gestion des entrées logiques

variable utilisateur = in***x*** variable système = in***x***	Lecture de l'entrée logique spécifiée par ***x*** , sa valeur est attribuée à la variable_destination (*variable utilisateur* ou *variable système*)

3.1.18 Entrées logiques relatives aux sous-programmes prioritaires

IT_**ON** = nom de l'entrée logique (IN**0** à IN**3**)	Valide la prise en compte de l'entrée logique interruptive spécifiée. ⇒ Un changement d'état sur l'entrée indiquée, respectant les conditions de prise en compte choisies en paramètres machine, déclenche le traitement du sous-programme prioritaire déclaré par *INTERRUPT**x** = PROG**n***.
IT_**OFF** = nom de l'entrée logique (IN**0** à IN**3**)	Invalide la prise en compte de l'entrée logique interruptive spécifiée.

3.1.19 Gestion des sorties logiques

out***x*** = 0 out***x*** = 1	Mise à 0 de la sortie logique spécifiée par ***x***. Mise à 1 de la sortie logique spécifiée par ***x***.

3.1.20 Gestion d'un clavier - afficheur µVision

ADR	Spécifie l'identificateur *CAN* de l'abonné adressé.
PAGE	Spécifie le numéro de page du clavier - afficheur.
LINE	Spécifie le numéro de ligne du clavier - afficheur.
COL	Spécifie le numéro de colonne du clavier - afficheur.
DISPLAY	Affiche la page sélectionnée par *ADR* et *PAGE*.
CLEAR_PAGE	Efface la page sélectionnée par *ADR* et *PAGE*.
CLEAR_LINE	Efface la ligne sélectionnée par *ADR*, *PAGE* et *LINE*.
PRINT ("texte") PRINT (variable, format)	Affiche un texte ou une variable aux coordonnées (adresse, page, ligne et colonne) spécifiées par les instructions *ADR*, *PAGE*, *LINE* et *COL*.
READ (variable, min, max)	Attend la réponse à une question posée à un clavier-afficheur.

3.1.21 Acquisition, lecture, écriture de variables, échange de données

ENQ (id, var1, var2) ENQ (id, index, sous_index, var)	Lit une variable appartenant à un autre variateur positionneur. Lit une *variable* d'un autre abonné *CAN*.
STC (id, var1, var2) STC (id, var1, valeur) STC (id, index, sous_index, var) STC (id, index, sous_index, valeur, type)	Modifie une variable d'un autre variateur positionneur. Modifie une variable d'un autre variateur positionneur. Modifie une variable d'un autre abonné *CAN*. Modifie une variable d'un autre abonné *CAN*.

3.1.22 Gestion de cames

Voir notice PVD3538 « DIGIVEX Motion – Fonction came »

3.2 Opérateurs mathématiques

3.2.1 Introduction

- Un ***opérateur*** mathématique est représenté par un symbole ou un terme spécialisé (par exemple : +, /, AND, …).
  Les opérateurs mathématiques interviennent sur des éléments appelés opérandes. Une expression mathématique peut réunir plusieurs opérateurs.

- Les familles d'opérateurs sont décrites ci-après. Chaque opérateur détient une priorité dans l'exécution des opérations mathématiques.

- Les opérations placées entre parenthèses internes sont effectuées en premier, puis viendront les opérations contenues entre parenthèses externes. Les règles arithmétiques sont respectées selon l'ordre de priorité détenu par les opérateurs.

- Dans une expression mathématique, il est possible d'accéder à 8 niveaux de parenthèses. Le nombre de parenthèses ouvertes doit toujours être égal au nombre de parenthèses fermées. Le non respect de cette règle entraîne une erreur à la compilation.

- Les éventuels espaces blancs situés à l'intérieur d'une expression mathématique (précédant ou suivant un opérateur) sont ignorés et ne provoquent aucune erreur lors de la compilation.

Remarque :

- Les ***fonctions*** mathématiques intervenant sur un ou plusieurs opérandes sont décrites dans le répertoire alphabétique des instructions au chapitre 3.3 (par exemple : SIN, ATAN, MIN, MAX,…).

3.2.2 Opérateurs arithmétiques

+

Fonction

Opérateur addition

Synopsis

variable_destination = opérande_1 **+** opérande_2

Propriétés

variable_destination, *opérande_1* et *opérande_2* doivent être de formats compatibles.

format *variable_destination* :	format *opérande*:
E	B
E	E
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
ui1 = 2 + ui2          ; somme d'une valeur algébrique et d'une variable utilisateur
uf1 = uf1 + pos1       ; somme de deux variables
```

–

Fonction

Opérateur soustraction

Synopsis

variable_destination = opérande_1 – opérande_2

Propriétés

variable_destination, *opérande_1* et *opérande_2* doivent être de formats compatibles.

format *variable_destination* :	format *opérande*:
E	B
E	E
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
ui1 = 2 - ui2          ; soustraction d'une valeur algébrique et d'une variable utilisateur
uf1 = uf1 - pos1       ; soustraction de deux variables
```

/

Fonction

Opérateur division

Synopsis

variable_destination = dividende **/** diviseur

Propriétés

variable_destination, *dividende* et *diviseur* doivent être de formats compatibles.

format *variable_destination* :	format *opérande*:
E	B
E	E
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
uf1 = uf2 / pos1        ; division de deux variables
```

*

Fonction

Opérateur multiplication

Synopsis

variable_destination = multiplicande ***** multiplicateur

Propriétés

variable_destination, *multiplicande* et *multiplicateur* doivent être de formats compatibles.

format *variable_destination* :	format *opérande*:
E	B
E	E
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
uf1 = uf2 * pos1        ; multiplication de deux variables
```

%%

Fonction

Opérateur *modulo*

Synopsis

variable_destination = opérande_1 **%%** opérande_2

Description

L'opérateur arithmétique **%%** détermine le reste de la division algébrique base 10 de l'*opérande_1* par l'*opérande_2*. Le résultat obtenu est affecté à la *variable_destination*.

Propriétés

variable_destination, *opérande_1* et *opérande_2* doivent être de formats compatibles.

Exemples

```
uf5 = 5
uf2 = 2
uf1 = uf5 %% uf2        ; uf1 vaut 1.
```

**

Fonction

Opérateur *élévation en puissance*

Synopsis

variable_destination = opérande_1 **** opérande_2

Propriétés

variable_destination, *opérande_1* et *opérande_2* doivent être de formats compatibles.

Exemples

```
uf1 = uf2 ** 3          ; élévation à la puissance 3 de la variable uf2.
```

3.2.3 Opérateurs relationnels

<, <=, =, >, >=, <>

Fonction

<	inférieur au sens strict	**<=**	inférieur ou égal
>	supérieur au sens strict	**>=**	supérieur ou égal
=	équivalent	**<>**	différent

Synopsis

variable_destination = (opérande_1 **<** opérande_2)
variable_destination = (opérande_1 **>=** opérande_2)
variable_destination = (opérande_1 **<>** opérande_2)

Description

- Les opérateurs relationnels établissent une assertion logique. Lorsque la proposition est vraie, la *variable_destination* reçoit la valeur booléenne 1. Si la proposition est fausse, la *variable_destination* présente la valeur booléenne 0.
- Les opérandes d'un opérateur relationnel sont des valeurs (algébriques ou logiques), des variables ou une combinaison des deux.
- Le format des opérandes peut être de nature différente :

format *opérande_1*	format *opérande_2*
E	E, B
F	F, D
D	D
B	B (fonctions = et <> uniquement)
Toute autre combinaison de formats provoque une erreur à la compilation.	

Propriétés

- La *variable_destination* est obligatoirement une variable booléenne.
- La *variable_destination* est implicite dans les structures de contrôle. Exemple :
  IF ui1 **<** *5 THEN GOTO #LABEL* se traduit par :
  Si *ui1* est *strictement inférieur* à 5 alors, la *variable_destination* implicite vaut 1 (la proposition est vraie) et l'exécution du programme se poursuit à l'adresse #LABEL (cf. description de *IF…THEN…ELSE…ENDIF* au chapitre 3.3).
- L'introduction d'opérandes au format *C1* engendre une erreur à la compilation ; par exemple, ub1 = (uc1 **=** "MESSAGE") est interdit.
- Les opérateurs gouvernant une relation d'ordre (<, <=, >, >=) ne s'appliquent pas aux opérandes de format *B*.
- Attention ! Ne pas confondre le signe = correspondant à une *affectation* de variable (par exemple ui1 = 5 , la valeur algébrique 5 est attribuée à la variable *ui1*) et le signe **=** , opérateur relationnel "*équivalent* " . L'emploi de parenthèses évite cette confusion.

Exemples

ub1 = (ui1 **<=** 5) ; la *variable_destination ub1* vaut 1, si *ui1* est inférieur ou égal à 5.
ub1 = (in1 **=** 0) ; *ub1* vaut 1 si l'entrée logique *in1* est à l'état 0.
IF ui1 **<>** 10 THEN GOTO #LABEL ; branchement à l'adresse *LABEL* si *ui1* différent de 10.

3.2.4 Opérateurs logiques

NOT, AND, NAND, OR, NOR, XOR

Fonction

NOT NON logique (complément logique)
AND ET logique (multiplication booléenne)
NAND ET logique complémenté (NON ET)
OR OU logique inclusif (somme booléenne)
NOR OU logique complémenté (NON OU)
XOR OU logique exclusif

Synopsis

variable_destination = **NOT**(opérande_1)
variable_destination = (opérande_1 **AND** opérande_2)
variable_destination = (opérande_1 **OR** opérande_2)

Description

- Les opérateurs logiques établissent la table de vérité issue de l'algèbre de Boole. La *variable_destination* reçoit le résultat de cette opération logique.

opérateur →		**AND**	**NAND**	**OR**	**NOR**	**XOR**
opérande_1	*opérande_2*	*variable_destination*				
0	0	0	1	0	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	0	1	0	0

- Lorsque les opérandes présentent un format entier *E*, le calcul de la table de vérité s'effectue bit à bit (mot codé sur 32 bits).
- Lorsque la *variable_destination* présente un format entier *E*, le résultat binaire de l'opération logique est converti en valeur entière sur 32 bits.
- Formats autorisés pour la *variable_destination* et les opérandes :

variable_destination	*opérande_1*	*opérande_2*
B	B	B
E	E	E
Toute autre combinaison de formats provoque une erreur à la compilation.		

Propriétés

- La *variable_destination* est implicite dans les structures de contrôle. Exemple :
  WAIT_UNTIL (in4 = 0) ***OR*** *(in1= 1* ***AND*** *in3 = 1)* se traduit par :
  Attendre jusqu'à ce que *in4* soit à 0 OU jusqu'à ce que *ui1* ET *in3* soient à 1 alors, la *variable_destination* implicite vaut 1 (la proposition est vraie) et la poursuite du déroulement de programme est autorisée. (cf. description de *WAIT_UNTIL* au chapitre 3)

Exemples

```
ub1 = (ub0 AND 0)    ; ET entre une variable booléenne et une valeur logique
ub1 = (ub0 OR in3)   ; Opérateur OU intervenant sur deux variables logiques
ui3 = (ui4 AND 12)   ; ET entre le masque 1100 (valeur binaire de 12) et ui4
```

3.3 Répertoire alphabétique des instructions

; commentaire ou { commentaire }

Fonction

définit un commentaire

Synopsis

; commentaire
{ commentaire }

Description

Un commentaire est identifié par un point virgule [**;**]. Il peut figurer à la suite d'une instruction.
Un bloc-commentaire se compose de plusieurs lignes de commentaires. Il débute par une ouverture d'accolade **{** et se termine par une fermeture d'accolade **}**.

Exemple

```
10      %PROG100
11      ; sous-programme de déplacement relatif
12      MOVER = 1   ; déplacement relatif sens positif
13      RETURN
14      END
15      %ENDPROG
```

ABORT

Fonction

Arrête tout déplacement en cours mais sans stopper l'exécution des programmes.

Synopsis

ABORT

Description

La commande ABORT s'utilise pour arrêter tout type de déplacement (FORWARD, REVERSE, SYNCHRO, SYNCHRO_START, HOME, MOVEA, MOVER).
Lorsque la commande ABORT est prise en compte, la variable système *move_abort* est mise à 1 (information "arrêt de l'axe en cours"). L'axe éventuellement en mouvement est arrêté selon les caractéristiques d'accélération / décélération actives [accel_prog].
L'arrêt de l'axe est prioritaire par rapport aux nouvelles demandes de mouvements qui pourraient être transmises (Lors de l'arrêt de l'axe, les demandes de mouvement sont rejetées. Celles-ci n'étant pas mémorisées, elles sont donc définitivement perdues).
La position obtenue en fin de décélération reste asservie et devient la nouvelle consigne de position à atteindre [*posa* = position d'arrêt]. La variable système *move_abort* est remise 0, indiquant l'arrêt de l'axe.
Par la suite, toute nouvelle instruction de mouvement programmée est prise en compte (l'instruction ABORT n'est pas équivalente à *move_en* = 0).
Attention !
L'exécution des programmes n'est pas suspendue !
A la fin de l'arrêt de l'axe, vérifier que le programme de gestion des mouvements ne reste pas bloqué sur une instruction de type WAIT_UNTIL dont la condition ne pourrait jamais être validée.

Propriétés

La commande ABORT correspond à mettre à 1 la variable système *abort_cmd* (*abort_cmd* est remis automatiquement à 0 dès que la commande est prise en compte).

Exemple

```
16      %PROG100
17      ; programme automate type 2
18      IF in7 = 1 THEN ABORT
19      ENDIF
20      …
21      …
22      END
23      %ENDPROG
```

Dans le programme automate, le passage à 1 de l'entrée logique n°7 provoque l'arrêt du mouvement en cours.

Voir aussi

WAIT_ UNTIL (Description des variables système permettant de savoir si un déplacement est en cours ou terminé)

ABS

Fonction

Valeur absolue.

Synopsis

variable_destination = ABS(variable_source)

Description

La fonction *ABS* calcule la valeur absolue d'une *variable_source*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
E	E
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = ABS(uf2)

La valeur absolue de la variable utilisateur *uf2* (*variable_source*) est affectée à la variable utilisateur *uf1* (*variable_destination*). Le résultat est présenté au format flottant *F*.

ACCEL

Fonction

Définit, ou réinitialise, l'accélération et la décélération des prochains déplacements.

Synopsis

ACCEL = valeur_de_l'accélération

ACCEL = variable_source

Description

- L'instruction *ACCEL* détermine l'accélération et la décélération du déplacement programmé par la prochaine instruction *MOVEA*, *MOVER*, *HOME*, *FORWARD* ou *REVERSE* rencontrée (l'action de cette instruction n'est donc pas immédiate).
- *valeur_de_l'accélération* est une donnée, *variable_source* une variable, exprimées en *unit1/s*^{2}, au format flottant *F* ou double précision *D*.
- Valeur_de_l'accélération et variable_source doivent être strictement positives.
- La variable système relative à l'instruction *ACCEL* est : *accel_move*.
  N.B. Au début de l'instruction *MOVEA*, *MOVER*, *HOME*, *FORWARD* ou *REVERSE* suivante, la valeur de *accel_move* est chargée dans *accel_prog*.
  Remarque : A la mise sous tension du variateur positionneur, accel_move = accel_max et accel_prog = accel_max.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*accel_move*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

ACCEL = 1000
{ La valeur de l'accélération vaut 1000 *unit1*/s^2.
N.B. La valeur *1000* peut à la fois être considérée comme présentant les formats *E*, *F* ou *D*. La valeur *1000.0* est considérée comme ayant le format flottant *F* ou le format double précision *D*.}

ACCEL_IM

Fonction

Modifie de manière immédiate mais provisoire l'accélération et la décélération du mouvement en cours.

Synopsis

ACCEL_IM = valeur_de_l'accélération

ACCEL_IM = variable_source

Description

- L'instruction ACCEL_IM modifie l'accélération et la décélération du mouvement en cours. Cette instruction est à prise d'effet immédiate.
  N.B. La modification est effective même si au moment de la prise en compte de l'instruction, on se trouve en phase d'accélération ou de décélération.

- *valeur_de_l'accélération* est une donnée, *variable_source* une variable , exprimées en *unit1/s*^{2}, au format flottant *F* ou double précision *D*.

- Valeur_de_l'accélération et variable_source doivent être strictement positives.

- La variable système relative à l'instruction *ACCEL_IM* est : *accel_prog*.
  Attention : Au début de l'instruction MOVEA, MOVER, HOME, FORWARD ou REVERSE suivante , la valeur de accel_prog sera écrasée par celle accel_move.
  Remarque : A la mise sous tension du variateur positionneur, accel_move = accel_max et accel_prog = accel_max.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*accel_prog*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

ACCEL_IM = 1000
{ La valeur de l'accélération vaut immédiatement 1000 *unit1*/s^2.
N.B. La valeur *1000* peut à la fois être considérée comme présentant les formats *E*, *F* ou *D*. La valeur *1000.0* est considérée comme ayant le format flottant *F* ou le format double précision *D*.}

ACOS

Fonction

Arc Cosinus

Synopsis

variable_destination = ACOS(variable_source)

variable_destination = ACOS(valeur)

Description

La valeur de l'angle résultant, exprimé en radians, est attribuée à la *variable_destination*.

Propriétés

Si *valeur* est <-1 ou >1, une erreur sera générée lors de la compilation.
Si *variable_source* est <-1 ou >1, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = ACOS(uf2)

ADR

Fonction

Spécifie l'identificateur *CAN* du terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

ADR = variable_source
ADR = valeur

Description

L'instruction *ADR* spécifie le numéro d'identification d'un abonné au bus *CAN.*

L'instruction *ADR* attribue la valeur programmée dans *variable_source* (ou *valeur*) à la *variable_destination* associée *can_id*.

variable_source (ou *valeur*) doit être comprise entre [1 et 63].

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*can_id*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

ADR = 4
PAGE = 2
LINE = 3
COL = 1
PRINT("texte") {affichage de *texte* débutant à la colonne n°1, ligne n°3, page n°2, sur le terminal abonné au bus *CAN* n°4}

Voir aussi

DISPLAY, PRINT, READ, CLEAR_LINE, CLEAR_PAGE

a_in*11*… à …a_in*15*

Fonction

Affecte l'entrée logique spécifiée à une variable système ou à un paramètre.

Synopsis

a_in***x*** = **n**

Description

- L'instruction *a_in**x*** = ***n*** est la traduction en langage de programmation, de l'affectation choisie pour *in**x***, à l'aide du logiciel *PME* [cf. notice *PVD 3516 : Entrées / Sorties*, *chapitre 5.4.1 Entrées logiques*].
- ***n*** est une variable ou une valeur binaire (égale à 0 ou 1)
- ***x*** est un nombre entier compris entre *11* et *15* identifiant l'entrée logique considérée.

Propriétés

Si ***n*** ou ***x*** est une *valeur* hors limites, une erreur sera générée lors de la compilation.
Si ***n*** est une *variable* hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*a_inx*]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

a_in***x*** = 0 ⇔ in***x*** *non affecté*
a_in***x*** = 1 ⇔ in***x*** *affecté à une fonctionnalité*

Pour les entrées logiques, les affectations sont imposées :

a_in*11* = 1 ⇔ *hardp_input* = in11	état de l'entrée fin de course électrique +.
a_in*12* = 1 ⇔ *hardm_input* = in12	état de l'entrée fin de course électrique -.
a_in*13* = 1 ⇔ *switch0_input* = in13	état de l'entrée came origine.
a_in*14* = 1 ⇔ *move_en* = in14	commande l'autorisation d'avance.
a_in*15* = 1 ⇔ *exec_en* = in15	commande l'autorisation d'exécution des programmes.

Voir aussi

Utilisation des variables *in**x***.

a_ina

Fonction

Affecte l'entrée analogique à une variable système.

Synopsis

a_in**a** = **n**

Description

- L'instruction *a_in**a*** = ***n*** est la traduction en langage de programmation, de l'affectation choisie pour *in**a***, à l'aide du logiciel *PME* [cf. notice *PVD 3516 : Entrées / Sorties*, *chapitre 5.4.3.1 Entrée analogique*].

- ***n*** est une variable ou une valeur entière comprise entre 0 et 4.

Propriétés

Si ***n*** est une *valeur* hors limites, une erreur sera générée lors de la compilation.
Si ***n*** est une *variable* hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*a_ina*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

a_in**a** = 0	⇔ *in**a*** non affecté	
a_in**a** = 1	⇔ *ki_red = in**a***	réduction de courant appliquée à *i_red*.
a_in**a** = 2	⇔ *speed_att = in**a***	atténuation de vitesse appliquée à *speed_prog*
a_in**a** = 3	⇔ speed_value=ina	consigne de vitesse en mode commande en vitesse
a_in**a** = 4	⇔ torque_value=ina	consigne de couple en mode commande en courant

Voir aussi

Utilisation des variables *in**a** et scale_in**a***.

a_out*0*… à …a_out*7*

Fonction

Affecte la sortie logique spécifiée à une variable système.

Synopsis

a_out***x*** = **n**

Description

- L'instruction ***a_outx = n,*** avec ***n*** une variable entière comprise entre 0 et 11, est la traduction en langage de programmation, de l'affectation choisie pour ***outx***, à l'aide du logiciel *PME* [cf. notice *PVD 3516 : Entrées / Sorties*, *chapitre 5.4.2 Sorties logiques*].

- ***n*** est une variable ou une valeur entière comprise entre 0 et 11.

- ***x*** est un entier compris entre 0 et 7 identifiant la sortie logique considérée.

Propriétés

Si ***n*** ou ***x*** est une *valeur* hors limites, une erreur sera générée lors de la compilation.
Si ***n*** est une *variable* hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*a_outx*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

a_out***x*** = 0 ⇔ *out**x*** *non affecté*	situation par défaut.
a_out***x*** = 1 ⇔ *out**x*** = *home_made*	information origine faite.
a_out***x*** = 2 ⇔ *out**x*** = *moving*	information mouvement en cours (théorique).
a_out***x*** = 3 ⇔ *out**x*** = *in_position*	information but atteint.
a_out***x*** = 4 ⇔ *out**x*** = *move_end*	information fin de mouvement (théorique).
a_out***x*** = 5 ⇔ *out**x*** = *exec_on*	information programme en exécution.
a_out***x*** = 6 ⇔ *out**x*** = *drive_ok*	information variateur positionneur ok (pas de défaut majeur et puissance présente).
a_out***x*** = 7 ⇔ *out**x*** = *brake_supplied*	information frein alimenté.
a_out***x*** = 8 ⇔ *out**x*** = *flag0*	flag0, état.
a_out***x*** = 9 ⇔ *out**x*** = *flag1*	flag1, état.
a_out***x*** = 10 ⇔ *out**x*** = *fatal_error*	information présence défaut majeur.
a_out***x*** = 11 ⇔ *out**x*** =*fault*	information présence défaut.
a_out***x*** = 12 ⇔ *out**x*** =*cam_out**x***	variable logique issue de la came.

Voir aussi

Utilisation des variables *out**x***.

a_outa

Fonction

Affecte la sortie analogique à une variable système.

Synopsis

a_out**a** = **n**

Description

- L'instruction *a_out**a*** = ***n*** est la traduction en langage de programmation, de l'affectation choisie pour *out**a***, à l'aide du logiciel *PME* [cf. notice *PVD 3516 : Entrées / Sorties*, *chapitre 5.4.3.2 Sortie analogique*].
- ***n*** est une variable ou une valeur entière comprise entre 0 et 20.

Propriétés

Si ***n*** est une *valeur* hors limites, une erreur sera générée lors de la compilation.
Si ***n*** est une *variable* hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*a_outa*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

a_out**a** = 0 ⇔ *out**a***	*non affecté*	situation par défaut.
a_out**a** = 1 ⇔ *out**a*** =	*pos1*	position réelle axe asservi.
a_out**a** = 2 ⇔ *out**a*** =	*pos2*	position réelle axe mesuré.
a_out**a** = 3 ⇔ *out**a*** =	*pos_th*	consigne de position.
a_out**a** = 4 ⇔ *out**a*** =	*tracking_error*	erreur de poursuite.
a_out**a** = 5 ⇔ *out**a*** =	*speed1*	vitesse réelle axe asservi.
a_out**a** = 6 ⇔ *out**a*** =	*speed2*	vitesse réelle axe mesuré.
a_out**a** = 7 ⇔ *out**a*** =	*speed_th*	consigne de vitesse.
a_out**a** = 8 ⇔ *out**a*** =	*synchro_error*	erreur de vitesse en synchro.
a_out**a** = 9 ⇔ *out**a*** =	*i_setpoint*	consigne de courant.
a_out**a** = 10 ⇔ *out**a*** =	*uf0*	variable utilisateur.
a_out**a** = 11 ⇔ *out**a*** =	*uf1*	variable utilisateur.
a_out**a** = 12 ⇔ *out**a*** =	*ui0*	variable utilisateur.
a_out**a** = 13 ⇔ *out**a*** =	*ui1*	variable utilisateur.
a_out**a** = 14 ⇔ *out**a*** =	*in**a***	entrée analogique, état après masque.
a_out**a** = 15 ⇔ *out**a*** =	*cam_out**a***	variable analogique issue de la came.
a_out**a** = 16 ⇔ *out**a*** =	*torque_setpoint*	consigne de couple.
a_out**a** = 17 ⇔ *out**a*** =	*var0*	variable physique.
a_out**a** = 18 ⇔ *out**a*** =	*var1*	variable physique.
a_out**a** = 19 ⇔ *out**a*** =	*filter0*	variable filtrée.
a_out**a** = 20 ⇔ *out**a*** =	*filter1*	variable filtrée.

Voir aussi

Utilisation des variables *out**a** et scale_out**a***.

ASIN

Fonction

Arc Sinus

Synopsis

variable_destination = ASIN(variable_source)

variable_destination = ASIN(valeur)

Description

La valeur de l'angle résultant, exprimé en radians, est attribuée à la *variable_destination*.

Propriétés

Si *valeur* est <-1 ou >1, une erreur sera générée lors de la compilation.
Si *variable_source* est <-1 ou >1, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = ASIN(uf2)

ATAN

Fonction

Arc Tangente

Synopsis

variable_destination = ATAN(variable_source)

variable_destination = ATAN(valeur)

Description

La valeur de l'angle résultant, exprimé en radians, est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = ATAN(uf2)

brake_cmd

Fonction

Commande de fermeture ou d'ouverture du frein. Cette commande n'est prise en compte que si un frein est déclaré et géré par le variateur positionneur (cf. notice PVD3516, chapitre 5.2)

Synopsis

brake_cmd = 0

brake_cmd = 1

brake_cmd = variable_source

Description

- *brake_cmd = 0* demande l'ouverture du frein (si le système ne s'y oppose pas).
- *brake_cmd = 1* demande la fermeture du frein.

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*brake_cmd*]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
IF in3 = 1THEN brake_cmd = 0
ENDIF
```

brake_ emergency

Fonction

Commande de fermeture d'urgence du frein. Cette commande n'est prise en compte que si un frein est déclaré et géré par le variateur positionneur (cf. notice PVD3516, chapitre 5.2)

Synopsis

brake_emergency = 0
brake_emergency = 1
brake_emergency = variable_source

Description

brake_emergency = 0 permet d'acquitter la demande de fermeture d'urgence du frein. Le frein reste dans l'état dans lequel il se trouve.

brake_emergency = 1 demande la fermeture d'urgence du frein. Par rapport à brake_cmd = 1, il n'y a pas attente que la vitesse devienne inférieur à 60 tr/mn pour fermer le frein (cf. notice PVD3516, chapitre 5.2).

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [brake_emergency]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
IF in3 = 1THEN brake_emergency = 1
ENDIF
```

cam

Voir notice PVD3538 « DIGIVEX Motion – Fonction came »

CLEAR_LINE

Fonction

Efface la ligne sélectionnée sur le terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

CLEAR_LINE

Description

L'instruction *CLEAR_LINE* efface la ligne de la page sélectionnée précédemment par les instructions *LINE* et *PAGE*, sur le terminal spécifié précédemment par l'instruction *ADR*.

Propriétés

- L'instruction *CLEAR_LINE* fait appel aux arguments implicites :

can_id, variable système spécifiant l'identificateur *CAN* de l'abonné adressé ; cette variable entière (*format E*) est comprise entre [1 et 63] et est programmée par l'instruction *ADR*.
can_page, variable système spécifiant le numéro de page du terminal ; cette variable entière (*format E*) est comprise entre [0 et 255] et est programmée par l'instruction *PAGE*.
can_line, variable système spécifiant le numéro de ligne du terminal ; cette variable entière (*format E*) est comprise entre [0 et 255] et est programmée par l'instruction *LINE*.

- Lorsque le terminal spécifié est un terminal µVision :

can_page doit être comprise entre 1 et 4
can_line doit être comprise entre 1 et 2
Toute autre valeur de *can_page* ou *can_line* appartenant à l'intervalle [0, 255] est sans effet, mais aucun message d'erreur n'est généré.

Exemple

```
ADR = 4
PAGE = 2
LINE = 3
CLEAR_LINE   ;effacement de la ligne n°3 de la page n°2 sur le terminal abonné CAN n°4
```

Voir aussi

ADR, PAGE, LINE

CLEAR_PAGE

Fonction

Efface la page sélectionnée sur le terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

CLEAR_PAGE

Description

L'instruction *CLEAR_PAGE* efface la page sélectionnée précédemment par l'instruction *PAGE* sur le terminal spécifié précédemment par l'instruction *ADR*.

Propriétés

- L'instruction *CLEAR_PAGE* fait appel aux arguments implicites :

can_id, variable système spécifiant l'identificateur *CAN* de l'abonné adressé ; cette variable entière (*format E*) est comprise entre [1 et 63] et est programmée par l'instruction *ADR*.

can_page, variable système spécifiant le numéro de page du terminal ; cette variable entière (*format E*) est comprise entre [0 et 255] et est programmée par l'instruction *PAGE*.

- Lorsque le terminal spécifié est un terminal µVision :

can_page doit être comprise entre 1 et 4
Toute autre valeur de *can_page* appartenant à l'intervalle [0, 255] est sans effet, mais aucun message d'erreur n'est généré.

Exemple

```
ADR = 4
PAGE = 2
CLEAR_PAGE  ;effacement de la page n°2 sur le terminal abonné CAN n°4
```

Voir aussi

ADR, PAGE

COL

Fonction

Spécifie le n° de la colonne du terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

COL = variable_source
COL = valeur

Description

L'instruction *COL* positionne le curseur dans la ligne et la page spécifiées précédemment par *LINE* et *PAGE*.

L'instruction *COL* attribue la valeur programmée dans *variable_source* (ou *valeur*) à la *variable_destination* associée *can_col*.

variable_source (ou *valeur*) doit être comprise entre [1 et 255].

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

- Lorsque le terminal spécifié est un terminal µVision :

variable_source (ou *valeur*) doit être comprise entre 1 et 16
Toute autre valeur attribuée à *can_col* appartenant à l'intervalle [1, 255] est sans effet, mais aucun message d'erreur n'est généré.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*can_col*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
ADR = 4
PAGE = 3
LINE = 2
COL = 1
PRINT("texte")      {affichage de texte débutant à la colonne n°1, ligne n°2, page n°3, sur le
                     terminal abonné CAN n°4}
```

Voir aussi

ADR, PAGE, LINE, PRINT, READ

COS

Fonction

Cosinus.

Synopsis

variable_destination = COS(variable_source)

variable_destination = COS(valeur)

Description

variable_source ou *valeur sont* exprimées en radians.
La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = COS(uf2)

DACOS

Fonction

Arc Cosinus.

Synopsis

variable_destination = DACOS(variable_source)

variable_destination = DACOS(valeur)

Description

La valeur de l'angle résultant, exprimé en degrés, est attribuée à la *variable_destination*.

Propriétés

Si *valeur* est <-1 ou >1, une erreur sera générée lors de la compilation.
Si *variable_source* est <-1 ou >1, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = DACOS(uf2)

DASIN

Fonction

Arc Sinus.

Synopsis

variable_destination = DASIN(variable_source)

variable_destination = DASIN(valeur)

Description

La valeur de l'angle résultant, exprimé en degrés, est attribuée à la *variable_destination*.

Propriétés

Si *valeur* est <-1 ou >1, une erreur sera générée lors de la compilation.
Si *variable_source* est <-1 ou >1, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = DASIN (uf2)

DATAN

Fonction

Arc Tangente.

Synopsis

variable_destination = DATAN(variable_source)

variable_destination = DATAN(valeur)

Description

La valeur de l'angle résultant, exprimé en degrés, est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = DATAN(uf2)

DCOS

Fonction

Cosinus.

Synopsis

variable_destination = DCOS(variable_source)

variable_destination = DCOS(valeur)

Description

variable_source ou *valeur* sont exprimées en degrés.
La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = DCOS(uf2)

DEFPLC1

Fonction

Attribue au programme automate de type 1 (programme automate générique), une durée limite pour effectuer son cycle. Voir à ce sujet le chapitre 2.8.1

Synopsis

DEFPLC1 = t

Description

Cette déclaration se situe de préférence dans la zone déclarative du programme principal.
La déclaration *DEFPLC1* = ***t*** fixe le temps de cycle maximal qui sera accordé à l'exécution d'une séquence complète de programme automate.

t est un nombre entier, ou une variable entière [*format E*], exprimés en ms, dont la valeur est comprise entre 1 et 10^6.

- Si le temps d'exécution devient supérieur à la durée déclarée par *DEFPLC1*, le « time out » survient et déclenche un défaut de programmation

- Si *DEFPLC1 = t* ne figure pas dans le programme, le temps de cycle maximal est alloué par défaut et vaut 10^6 ms.

La variable système relative à l'instruction *DEFPLC1* est : *defplc1*.

Propriétés

Si ***t*** est une valeur hors limites, une erreur sera générée lors de la compilation.
Si ***t*** est une variable hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*defplc1*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

DEFPLC2

Fonction

Définit la périodicité d'exécution du programme automate de type 2 (programme automate cyclique). Voir à ce sujet le chapitre 2.8.2

Synopsis

DEFPLC2 = t

Description

Cette déclaration se situe de préférence dans la zone déclarative du programme principal
La déclaration *DEFPLC2* = ***t*** fixe le temps de cycle (la périodicité d'une séquence de traitement).

t est un nombre entier, ou une variable entière [*format E*], exprimés en ms, dont la valeur est comprise entre 1 et 10^6.

- Il appartient au programmeur de vérifier si la séquence de programme « passe » effectivement dans le temps de cycle prescrit par *DEFPLC2* ⇒ visualiser à cet effet la variable *plc2_tick* à l'aide de la fonction *Oscilloscope* (cf. illustration au chapitre 2.8.2).

- Si le temps d'exécution devient supérieur à la durée déclarée par *DEFPLC2*, le « time out » survient et déclenche un défaut de programmation.

- Si *DEFPLC2* = ***t*** ne figure pas dans le programme, le temps de cycle est fixé par défaut et vaut *100 ms*.

La variable système relative à l'instruction *DEFPLC2* est : *defplc2*.

Propriétés

Si ***t*** est une valeur hors limites, une erreur sera générée lors de la compilation.
Si ***t*** est une variable hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*defplc2*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

DEFPOS1

Fonction

Redéfinit la position de l'axe asservi (axe1).

Synopsis

DEFPOS1 = variable_source
DEFPOS1 = valeur

Description

- L'instruction *DEFPOS1* effectue un changement de référentiel sans provoquer de déplacement. Le compteur de position de l'axe 1 (variable système *pos1*) est réinitialisé avec la valeur programmée dans *variable_source* (ou *valeur*).

- L'instruction *DEFPOS1* attribue la valeur programmée dans *variable_source* (ou *valeur*) à la *variable_destination* associée *val_pos1*. L'ordre est transmis par la mise à 1 de la variable booléenne *def_pos1*.

- *variable_source* et *valeur* sont exprimées en *unit1*.

Propriétés

- Cette fonction modifie l'*Origine Mesure*. Son utilisation se justifie pour les machines ne possédant pas de limites finies. La valeur programmée correspond à un décalage : l'*Origine Mesure* ayant été déplacée, les fins de course logiciels sont repoussés d'autant.

- Les principales variables système modifiées par *DEFPOS1* sont : *pos1*, *pos_th*, *posa* et *poscod1.*

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*val_pos1*]	format *variable_source* :
D	F
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

DEFPOS1 = 10

DEFPOS2

Fonction

Redéfinit la position de l'axe mesuré (axe 2).

Synopsis

DEFPOS2 = variable_source
DEFPOS2 = valeur

Description

- L'instruction *DEFPOS2* effectue un changement de référentiel. Le compteur de position de l'axe 2 (variable système *pos2*) est réinitialisé avec la valeur programmée dans *variable_source* (ou *valeur*).

- L'instruction *DEFPOS2* attribue la valeur programmée dans *variable_source* (ou *valeur*) à la *variable_destination* associée *val_pos2*. L'ordre est transmis par la mise à 1 de la variable booléenne *def_pos2*.

- *variable_source* et *valeur* sont exprimées en *unit2*.

Propriétés

- Les principales variables système modifiées par *DEFPOS2* sont : *pos2*, *pos_th*, *master_Pn* et *poscod2.*

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*val_pos2*]	format *variable_source* :
D	F
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

DEFPOS2 = 10

drive_mode

Fonction

Définit le mode de pilotage de l'axe.

Synopsis

drive_mode = 0
drive_mode = 1
drive_mode = 2
drive_mode = variable_source

Description

variable_source doit être égale à 0, 1 ou 2.

drive_mode = 0 ⇔ commande en position
drive_mode = 1 ⇔ commande en vitesse
drive_mode = 2 ⇔ commande en courant

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*drive_mode*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
IF in1 = 1 AND in2 = 0  THEN drive_mode = 1
                        ELSE drive_mode = 0
ENDIF
```

DISPLAY

Fonction

Affiche la page sélectionnée sur le terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

DISPLAY

Description

L'instruction *DISPLAY* affiche la page sélectionnée précédemment par l'instruction *PAGE* sur le terminal spécifié précédemment par l'instruction *ADR*.

Propriétés

L'instruction *DISPLAY* fait appel aux arguments implicites :

can_id, variable système spécifiant l'identificateur *CAN* de l'abonné adressé ; cette variable entière (*format E*) est comprise entre [1 et 63] et est programmée par l'instruction *ADR*.

can_page, variable système spécifiant le numéro de page du terminal ; cette variable entière (*format E*) est comprise entre [0 et 255] et est programmée par l'instruction *PAGE*.

Exemple

```
ADR = 4
PAGE = 2
DISPLAY ; affichage de la page n°2 sur le terminal abonné CAN n°4
```

Voir aussi

ADR, PAGE

Remarque Importante

Il est fortement conseillé de ne gérer un terminal donné qu'à partir d'une tâche unique (soit programme de gestion de mouvements, soit programme automate de type 1, soit programme automate de type 2). En effet, la gestion d'un même terminal à partir de différentes tâches s'exécutant en parallèle pourrait conduire à des comportements incohérents (risque de télescopage d'instructions).

DO

Fonction *WHILE / DO / WEND*
Se reporter à la description de *WHILE*.

DSIN

Fonction

Sinus.

Synopsis

variable_destination = DSIN(variable_source)

variable_destination = DSIN(valeur)

Description

variable_source ou *valeur* sont exprimées en degrés.
La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = DSIN(uf2)

DTAN

Fonction

Tangente.

Synopsis

variable_destination = DTAN(variable_source)

variable_destination = DTAN(valeur)

Description

variable_source ou *valeur* sont exprimées en degrés.
La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = DTAN(uf2)

emergency_cmd

Fonction

Provoque l'arrêt d'urgence de l'axe.

Synopsis

emergency_cmd = 1
emergency_cmd = 0
emergency_cmd = variable_source

Description

emergency_cmd = 1 provoque l'arrêt d'urgence de l'axe et l'affichage du message d'erreur E sur l'afficheur 7 segments. S'il y avait un déplacement en cours, l'axe décélère avec freinage maximum [accel_max] puis s'arrête. La position obtenue en fin de décélération reste asservie et devient la nouvelle consigne de position à atteindre [posa = position d'arrêt]. Les mouvements sont ensuite interdits tant qu'un acquittement de défaut n'a pas été réalisé [reset_cmd = 1]. emergency_cmd est remis automatiquement à 0 lors de l'acquittement du défaut.

Attention ! L'exécution des programmes n'est pas suspendue !
Vérifier que le programme de gestion des mouvements ne reste pas bloqué sur une instruction de type WAIT_UNTIL dont la condition ne pourrait jamais être validée.

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format variable_destination : [emergency_cmd]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
IF in1 = 1 AND in2 = 0     THEN  emergency_cmd = 1
ENDIF
```

ELSE

Fonction *IF / THEN / ELSE / ENDIF*
Se reporter à la description de *IF*.

END

Fonction

Définit la fin de cycle d'une séquence de programme automate.
Définit la fin d'exécution d'un sous-programme prioritaire.
Définit la fin d'exécution d'un sous-programme de gestion d'erreurs.

Synopsis

END

Description

END conclut une séquence de programme automate, et provoque une reprise d'exécution de ce dernier à son début.

END termine l'exécution d'un sous-programme prioritaire déclenché par une entrée logique à caractère interruptif.

END termine l'exécution d'un sous-programme de gestion d'erreurs, déclaré par *ERROR*, et provoque la reprise d'exécution normale du programme de gestion des mouvements. Le pointeur opérationnel retrouve le contexte logiciel qui précédait l'appel au sous-programme de gestion d'erreurs.

Exemple

```
10      %PROG50
11      ; programme automate type 1
12      uf1=uf1+1
13      …
14      …
15      …
16      …
17      END          ; Fin de cycle d'une séquence de programme automate.
18      %ENDPROG     ; Fin de code du programme n° 50.
```

Propriétés

END est une instruction opérationnelle à ne pas confondre avec *%ENDPROG* directive uniquement destinée au compilateur.

ENDIF

Fonction *IF / THEN / ELSE / ENDIF*
Se reporter à la description de *IF*.

%ENDPROG

Fonction

Définit la fin d'un programme ou d'un sous-programme.

Synopsis

%ENDPROG

Description

%ENDPROG correspond à la fin du code de description d'un programme ou d'un sous-programme.

Exemple

```
10      %PROG30
11      WAIT_UNTIL in_position = 1
12      …
13      …
14      …
15      RETURN
16      %ENDPROG        ; Fin du sous-programme n° 30.
```

Propriétés

%ENDPROG est une directive exclusivement destinée au compilateur.

ENQ

Fonction

Lit une variable d'un autre variateur positionneur [*fonction A*].
Lit une variable d'un autre abonné *CAN* [*fonction B*].
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

ENQ(id, var1, var2) ; [*fonction A*]
ENQ(id, index, sous_index, var) ; [*fonction B*]

Description

- [*fonction A*]

id est un entier compris entre [1 et 127] correspondant au numéro d'abonné *CAN*. Sa valeur est affectée à la variable *can_id.*
var1 est le nom de la *variable_source* (à lire).
var2 est le nom de la *variable_destination* (à écrire).

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
B	B
E	B
E	E
F	F
F	D (avec perte de la précision)
D	D
C1	C1
C1	C2 (troncature de la chaîne)
C2	C2
Toute autre combinaison de formats entraîne une erreur à la compilation.	

- [*fonction B*]

La variable lue est référencée par les identificateurs *CAN* nommés *index* et *sous-index* :

index est l'index *CANopen* de la *variable_source*. Sa valeur est affectée à la variable *can_index.*
sous_index est le sous-index *CANopen* de la *variable_source*. Sa valeur est affectée à la variable *can_subindex.*
var est le nom de la *variable_destination*.

Pour connaître les numéro d'*index* et de *sous_index CAN*, se reporter au document intitulé :
[PVD 3527 *DIGIVEX Motion, Répertoire des variables*]

Propriétés

Attention !
L'instruction *ENQ* utilise la variable *can_id*. Elle peut donc modifier une programmation précédente effectuée à l'aide de *ADR.*

Remarque

Utiliser de préférence avec cette fonction les adresses des canaux SDO 1, 2 ou 3 (voir notice PVD 3516 chapitre 5.2.5.1).

L'exécution des instructions programmées après ENQ ne se fera que lorsque l'instruction ENQ aura reçu une réponse (timeout de 75 ms avec génération d'un "défaut exécution programme" en cas de non-réponse). On comprend facilement l'intérêt de cette propriété dans l'exemple suivant :

```
uf3 = 10
ENQ(1,uf3,uf3)
MOVEA =  uf3
```

Il n'y aura pas déplacement à la cote 10, mais à la cote correspondant à la valeur uf3 récupérée par l'instruction ENQ

Exemples

ENQ(3,pos1,ud0)	{ la valeur *pos1* du variateur positionneur abonné CAN n°3 est attribuée à la variable *ud0*.}
ENQ(3,$2100,1,ui8)	{ la *variable_source* d'*index* hexadécimal $2100 et de *sous-index* 1 de l'abonné au bus CAN n°3 est attribuée à la variable *ui8*.}

Voir aussi

STC

ERROR

Fonction

Définit un programme de gestion d'erreurs. Voir à ce sujet le chapitre 2.11.

Synopsis

ERROR = PROG***n***

Description

Un programme de gestion d'erreurs est déclaré par l'instruction *ERROR* = *PROG**n*** avec ***n*** un entier compris entre **1** et 999.
Le programme ainsi déclaré est appelé par l'instruction *EXEC_ERR*.

Propriétés

Si *PROG**n*** n'existe pas ou si ***n*** est hors limites, une erreur sera générée lors de la compilation.

Exemple

```
1   %PROG0
2    ; programme principal
3   #INIT
4   ERROR = PROG999
5   {cette déclaration indique que le programme n°999 est un sous-
6   programme de gestion d'erreurs.}
7    #START
8       GOSUB PROG10
9       GOSUB PROG20
10  ...
11  …
12  …
13  RESTART
14  %ENDPROG
```

EXEC_ERR

Fonction

Provoque l'appel au programme de gestion des erreurs. Voir à ce sujet le chapitre 2.11.

Synopsis

EXEC_ERR

Description

- L'instruction *EXEC_ERR* appelle le programme de gestion d'erreurs préalablement déclaré par *ERROR* = *PROG**n*** avec ***n*** un entier compris entre **1** et 999.
- *EXEC_ERR* est ignorée quand un programme de gestion d'erreurs est déjà en cours d'exécution.

Propriétés

L'exécution de *EXEC_ERR* équivaut à mettre à 1 la variable système *call_err*.

EXP

Fonction

Exponentielle.

Synopsis

variable_destination = EXP(variable_source)

variable_destination = EXP(valeur)

Description

La valeur résultante est attribuée à la *variable_destination*.

variable_source et *valeur* ne doivent pas excéder 88.7

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = EXP(uf2)

FILTER

Fonction

Programmation d'un *filtre*.

Synopsis

FILTER (n, arg1, arg2)

Description

n vaut 0 ou 1 et correspond au numéro du filtre choisi (*filter0* ou *filter1*).
arg1 est la variable système prise en compte par le *filtre*.
arg2 est le coefficient de filtrage.

Propriétés

- Le système gère deux filtres dont la périodicité de mise à jour est de 250 µs. Ces filtres fournissent deux variables *filter0* et *filter1* qui représentent les valeurs filtrées des variables spécifiées par *arg1*.

  arg1 est choisi dans la liste ci-dessous :

Liste des variables pouvant être assignées à *arg1* :	
none	pas d'affectation (situation par défaut)
pos1	position réelle axe asservi
pos2	position réelle axe mesuré
pos_th	position théorique
tracking_error	erreur de poursuite
speed1	vitesse réelle axe asservi
speed2	vitesse réelle axe mesuré
speed_th	vitesse théorique
synchro_error	erreur de vitesse en synchro (*master_Vn-master-Vf)*
i_setpoint	consigne de courant
uf0	variable utilisateur
uf1	variable utilisateur
ui0	variable utilisateur
ui1	variable utilisateur
ina	variable utilisateur
outa	sortie analogique
torque_setpoint	consigne de couple
var0	variable physique
var1	variable physique

- *arg2* est une *valeur* comprise entre 0 et 1 qui indique le coefficient de filtrage. Plus arg2 sera proche de 0 et plus le filtrage sera élevé.

  La formule de filtrage est la suivante :

filter*n* = [*arg2* ∗ *variable_spécifiée_par_arg1*] + [(1 - *arg2*) ∗ *valeur_précédente_de_filte*m]

La constante de temps du filtre est : $\tau = 250\ e^{-6} / arg2$

Exemples

FILTER (0, pos2,0.1)
FILTER (1, speed1,0.05)

Voir aussi

Description de la programmation des *filtres* au chapitre 2.10 de ce document.

FIX

Fonction

Partie entière d'un nombre.

Synopsis

variable_destination = FIX(variable_source)

Description

La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
uf2 = 3.4
uf1 = FIX(uf2)    ; uf1 est égal à 3

uf2 = -3.4
uf1 = FIX(uf2)    ; uf1 est égal à –3
```

FLAG

Fonction

Programmation d'un *flag*.

Synopsis

FLAG (n, arg1, arg2, arg3)

Description

n vaut 0 ou 1 et correspond au numéro du flag choisi (*flag0* ou *flag1*).
arg1 est la variable système signalée par le *flag*.
arg2 est la borne inférieure de l'intervalle de test.
arg3 est la borne supérieure de l'intervalle de test.

Propriétés

- *arg1* est choisi dans la liste ci-dessous :

Liste des variables pouvant être assignées à *arg1* :	
none	pas d'affectation (situation par défaut)
pos1	position réelle axe asservi
pos2	position réelle axe mesuré
pos_th	position théorique
tracking_error	erreur de poursuite
speed1	vitesse réelle axe asservi
speed2	vitesse réelle axe mesuré
speed_th	vitesse théorique
synchro_error	erreur de vitesse en synchro (*master_Vn-master-Vf)*
i_setpoint	consigne de courant
uf0	variable utilisateur
uf1	variable utilisateur
ui0	variable utilisateur
ui1	variable utilisateur
ina	variable utilisateur
outa	sortie analogique
torque_setpoint	consigne de couple
var0	variable physique
var1	variable physique
filter0	variable filtrée
filter1	variable filtrée

- *arg2* et *arg3* sont des *variables_source* ou des *valeurs.*

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*flag0_low, flag1_low, flag0_up, flag1_up*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

FLAG (0, pos2,100, 200)
FLAG (1, tracking_error, -0.01, 0.01)

Voir aussi

Description de la programmation des *flags* au chapitre 2.9 de ce document.

FLOAT

Fonction

Convertit une variable entière (format *E*) ou binaire (format *B*) au format flottant *F* ou au format double précision *D*.

Synopsis

variable_destination = FLOAT(variable_source)

Description

La nature de la conversion au format *F* ou *D* dépend du format de la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	B
F	E
D	B
D	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

uf1 = FLOAT(ui1)
{conversion de la variable entière *ui1* au format flottant dans la variable *uf1*.}

ud1 = FLOAT(ui1)
{conversion de la variable entière *ui1* au format double précision dans la variable *ud1*.}

FOR…NEXT

Fonction

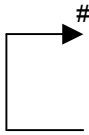
Définit une structure de contrôle de type *répétition indexée*.

Synopsis

FOR *index* = ***n*** TO ***m***
action(s)
NEXT

Description

- *FOR…NEXT* s'utilisent pour répéter une opération ou une séquence d'opérations, un certain nombre de fois ; la répétition est dite « indexée ». La structure de contrôle peut se traduire de la façon suivante :



#début de boucle
Pour *index* évoluant de ***n*** à ***m*** Effectuer *telle Action 1*
telle Action 2
…
Suivant
#fin de boucle

La *structure de contrôle* prend en compte :
- ❖ l'initialisation et la valeur finale du compteur d'itérations *index*
- ❖ la progression du compteur d'un incrément : *index* = *index* + 1
- ❖ le test du compteur pour sortir de la boucle :

Si *index* est supérieur au nombre de passages souhaité
Alors branchement à l'adresse #fin de boucle
Sinon branchement à l'adresse #début de boucle
Fin

- Le nombre de passages dans la boucle *FOR…NEXT* est déterminé.
- *index* est une variable utilisateur entière (format *E*).
- ***n*** et ***m*** sont des nombres entiers (ou des variables entières) signés.

Propriétés

Lorsque ***m*** est supérieur ou égal à ***n*** , le nombre de répétitions *FOR…NEXT* effectué vaut (***m*** – ***n*** +1), sinon il n'y a qu'un seul passage dans la boucle *FOR…NEXT*.

Exemples

```
10   %PROG10
11    ;
12   FOR ui0 = 1 TO ui1
13       MOVER = 1000
14       WAIT_UNTIL move_end=1
15   NEXT
16   {une succession de déplacements relatifs valant chacun
17   1000 unit1 s'accomplit un nombre de fois égal à ui1.}
18   RETURN
19   %ENDPROG
```

FORWARD

Fonction

Définit un déplacement continu dans le sens +.

Synopsis

FORWARD

Description

- L'instruction *FORWARD* se traduit par *MOVEA* = +∞ (déplacement absolu infini dans le sens positif, variable système *posa* = +∞).

- Les phases d'accélération, de décélération et à vitesse constante sont gouvernées par les paramètres machines actifs au moment de l'exécution de *FORWARD* (accel_move et speed_move). Remarque : accélération/décélération et vitesse peuvent être mofidifées lors du déplacement en utilisant les instructions *ACCEL_IM* et *SPEED_IM*.

Exemple

```
10  %PROG10
11  ACCEL = 1000
12  SPEED = 500
13  ;
14  IF in1 = 1 THEN FORWARD
15  {l'entrée logique in1 à l'état 1 provoque un déplacement continu dans le
16   sens positif, à la vitesse de 500 unit1/s.}
17  ENDIF
18  …
19  …
20  …
21  RETURN
22  %ENDPROG
```

Voir aussi

REVERSE

WAIT_UNTIL (Description des variables système permettant de savoir si un déplacement est en cours ou terminé).

FRAC

Fonction

Partie fractionnaire d'un nombre.

Synopsis

variable_destination = FRAC(variable_source)

Description

La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
uf2 = 3.4
uf1 = FRAC(uf2) ;    uf1 est égal à 0.4

uf2 = -3.4
uf1 = FRAC(uf2) ;    uf1 est égal à -0.4
```

FSPEED

Fonction

Définit la vitesse en fin de mouvement des prochains déplacements.
N.B. Cette instruction n'est en général pas utilisée dans les applications courantes.

Synopsis

FSPEED = variable_source

FSPEED = valeur_vitesse_finale

Description

- L'instruction *FSPEED* détermine la vitesse finale du déplacement programmé par la prochaine instruction *MOVEA*, *MOVER*, *HOME*, *FORWARD* ou *REVERSE* rencontrée (l'action de cette instruction n'est donc pas immédiate).
- *valeur_vitesse_finale* est une donnée, *variable_source* une variable , exprimées en *unit1/s*, au format flottant *F* ou double précision *D*.
- Par défaut, la vitesse en fin de mouvement est nulle (*FSPEED = 0*).
- La variable système relative à l'instruction *FSPEED* est : *fspeed_move.*
  N.B. Au début de l'instruction *MOVEA*, *MOVER*, *HOME*, *FORWARD* ou *REVERSE* suivante, la valeur de *fspeed_move* est chargée dans *fspeed_prog*.

  Remarque : A la mise sous tension du variateur positionneur, fspeed_move = 0 et fspeed_prog = 0.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* [*fspeed_move*] :	format *variable_source* ou *valeur_vitesse_finale* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Voir aussi

FSPEED_IM

FSPEED_IM

Fonction

Définit de manière immédiate mais provisoire la vitesse en fin de mouvement.
N.B. Cette instruction n'est en général pas utilisée dans les applications courantes.

Synopsis

FSPEED_IM = variable_source

FSPEED_IM = valeur_vitesse_finale

Description

- L'instruction FSPEED_IM modifie la vitesse finale du mouvement en cours. Cette instruction est à prise d'effet immédiate
- *valeur_vitesse_finale* est une donnée, *variable_source* une variable , exprimées en *unit1/s*, au format flottant *F* ou double précision *D*.
- Par défaut, la vitesse en fin de mouvement est nulle (*FSPEED_IM = 0*).
- La variable système relative à l'instruction *FSPEED_IM* est : *fspeed_prog.*

Attention ! Au début de l'instruction MOVEA, MOVER, HOME, FORWARD ou REVERSE suivante, la valeur de fspeed_prog sera écrasée par celle de fspeed_move.

Remarque : A la mise sous tension du variateur positionneur, fspeed_move = 0 et fspeed_prog = 0.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* [*fspeed_prog*] :	format *variable_source* ou *valeur_vitesse_finale* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Voir aussi

FSPEED

GOSUB

Fonction

Appelle un sous-programme.

Synopsis

GOSUB PROG***n***

GOSUB PROG(variable_utilisateur)

Description

- *GOSUB* est l'instruction qui interrompt le programme en cours pour appeler le sous-programme spécifié par :
  *PROG**n*** ⇒ ***n*** est un nombre entier compris entre 1 et 999 ou
  PROG(variable_utilisateur) ⇒ *variable_utilisateur* est une variable entière (*format E*) dont la valeur est comprise entre 1 et 999.

- Le pointeur opérationnel suspend l'exécution du programme en cours et se place à la première ligne du sous-programme appelé par *GOSUB*.

Propriétés

Si *PROG**n*** n'existe pas ou si ***n*** est hors limites, une erreur sera générée lors de la compilation.
Si *variable_utilisateur* est hors limites ou si *PROG(variable_utilisateur)* n'existe pas, une erreur sera générée à l'exécution.

Exemple

```
1       %PROG0
2       #INIT
3       ui0 = 30
4        ; la valeur 30 est affectée à la variable_utilisateur entière ui0.
5       #START
6          IF in1 = 1  THEN GOSUB PROG10
7        ; Appel au sous-programme n°10.
8                          ELSE GOSUB PROG(ui0)
9        ; Appel au sous-programme n°30.
10         ENDIF
11         …
12         …
13       RESTART
14      %ENDPROG
```

GOTO

Fonction

Définit un saut inconditionnel.

Synopsis

GOTO #LABEL

Description

GOTO est l'instruction de branchement inconditionnel.
Le pointeur opérationnel se déroute à l'adresse spécifiée par *#LABEL*.

Exemple

```
20  %PROG10
21   …
22  #DEBUT
23    uf0 = uf0+1
24    IF in1 = 1  THEN
25                GOTO #FIN ; branchement à l'adresse #FIN.
26                ELSE
27                GOTO #DEBUT ; branchement à l'adresse #DEBUT.
28    ENDIF
29    …
30  #FIN
31  RETURN
32  %ENDPROG
```

hardlimit_en

Fonction

Valide, ou non, la prise en compte effective des fins de course électriques.

Synopsis

hardlimit_en = 1

hardlimit_en = 0

hardlimit_en = variable_source

Description

hardlimit_en = ***1*** **valide** la prise en compte des fins de course électriques.
hardlimit_en = ***0*** **invalide** cette prise en compte.

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*hardlimit_en*]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
IF in1 = 1 AND in2 = 0  THEN hardlimit_en = 1
                        ELSE hardlimit_en = 0
ENDIF
```

Remarque

Lorsqu'un fin de course électrique est atteint (*hardp_input* = 0 ou *hardm_input* = 0), il y a décélération de l'axe et arrêt. La position destination devient la position d'arrêt. L'exécution du programme n'est pas interrompue. Un mouvement programmé pour se dégager du fin de course activé est ensuite possible.

HOME

Fonction

Lance l'exécution de la séquence de prise d'origine.

Synopsis

HOME

Description

L'exécution de la séquence de prise d'origine est totalement transparente pour l'utilisateur.
La procédure de déplacement de l'axe et les caractéristiques du mouvement sont développées dans la notice *PVD 3516 au chapitre 5.2*.

Au début de la séquence de prise d'origine, la variable système homing est mise à 1 et la variable home_made à 0.
homing repasse à 0 lorsque la prise d'origine est terminée ou interrompue (axe à l'arrêt). home_made ne passe à 1 que si la séquence de prise d'origine s'est déroulée avec succès

Propriétés

La commande *HOME* correspond à mettre à 1 la variable système *home_cmd*. (home_cmd est remis automatiquement à 0 dès que la commande a été prise en compte).

Attention ! La vitesse en prise d'origine est définie par home_speed (cette valeur est chargée dans speed_prog au moment de la prise en compte de l'instruction HOME)

Exemple

```
1    %PROG0
2     ; programme principal
3    #INIT
4    PLC1 = PROG100
5    PLC1 = START
6    #START
7    HOME                          ; lancement de la séquence de prise d'origine.
8    WAIT_UNTIL home_made = 1      ; attente de prise d'origine effectuée.
9    …
10      GOSUB PROG10
11      GOSUB PROG20
12   …
13   …
14   RESTART
15   %ENDPROG
```

Voir aussi

WAIT_UNTIL (Description des variables système permettant de savoir si un déplacement est en cours ou terminé).

IF…THEN…ELSE…ENDIF

Fonction

Définit une structure de décision.

Synopsis

```
IF condition  THEN action1
              ELSE action2
ENDIF
```

Description

- *IF…THEN…ELSE…ENDIF* s'utilisent pour programmer une sélection :

Si *telle condition est vérifiée* Alors *exécuter l'action1*
Sinon *exécuter l'action2*
Fin d'alternative.

- Forme réduite :

Si *telle condition est vérifiée* Alors *exécuter telle action*
Fin d'alternative.

- Forme imbriquée :

Si *telle condition1* est vérifiée Alors
  Si *telle condition2* est vérifiée Alors
    Si *telle condition3* est vérifiée Alors *exécuter l'action1*
    Sinon *exécuter l'action2*
    Fin d'alternative n°3.
  Sinon *exécuter l'action3*
  Fin d'alternative n°2.
Sinon *exécuter l'action4*
Fin d'alternative n°1.

- « *telle condition* » est une assertion logique.

Lorsque la proposition logique est vraie, la condition est vérifiée.

Exemples de « *conditions* » pouvant figurer à la suite de « IF » :

```
in3 = 1          ; proposition vraie pour l'entrée logique in3 à l'état haut.
pos2 < 2000      ; proposition vraie pour une position réelle de l'axe 2
                 ; inférieure strictement à 2000 unit2.
timer3 = 0       ; proposition vraie pour un timer n°3 expiré, ou initialisé à 0.
```

- « *exécuter telle action* » est une résultante de l'alternative.

Exemples d'« *actions* » pouvant figurer à la suite de « *THEN* » ou « *ELSE* » :

```
out1 = 1              ; affectation de la valeur 1 à la sortie logique out1.
GOSUB PROG10          ; appel au sous-programme n° 10.
RETURN                ; fin d'exécution du sous-programme incluant la
                      ; structure IF…THEN…ELSE…ENDIF.
```

Exemple

```
1     %PROG0
2      ; Voici un exemple de programme principal.
3     #INIT
4     PLC1 = PROG100
5       #START
6           IF in1 = 1 THEN PLC1 = START
7      ;    Si l'entrée in1 est à 1 Alors  démarrage du programme
8      ;                                   automate type 1.
9                       ELSE  GOSUB PROG10
10     ;                Sinon appel au sous-programme n°10.
11          ENDIF
12
13      GOSUB PROG20
14
15      IF in2 = 1 THEN RESTART
16    {cette forme de sélection ne comprend pas l'alternative « ELSE ».}
17      ENDIF
18    …
19    PROG_INIT
20    %ENDPROG
```

Propriétés

La structure de contrôle est complète avec *IF…THEN…ELSE…ENDIF* ou *IF… THEN…ENDIF*. L'absence de *ENDIF* (ou un nombre incorrect de *ENDIF* lors d'imbrications) conduit à une erreur de syntaxe lors de la compilation.

in*0*… à …in*15*

Fonction

Donne l'état de l'entrée logique spécifiée.

Synopsis

variable_destination = in***x***

Description

- ***x*** est un entier compris entre 0 et 15 désignant l'entrée logique considérée.
- L'instruction *variable_destination = in**x*** attribue à la *variable_destination* l'état de l'entrée logique mentionnée par ***x***.
- *variable_destination* est une variable binaire [*format B*].

Propriétés

Si ***x*** est une valeur hors limites, une erreur sera générée lors de la compilation.

Exemple

IF in1 = 1 THEN RESTART { test de l'entrée logique *in1* : si l'état est vrai alors, branchement à l'adresse *#START* }
ENDIF

Voir aussi

Utilisation des variables système *a_in11* à *a_in15*

ina

Fonction

Donne la valeur de l'entrée analogique.

Synopsis

variable_destination = in**a**

Description

- L'instruction *variable_destination* = *in**a*** attribue à la *variable_destination* la valeur de l'entrée analogique *in**a***.
- *variable_destination* est une variable flottante [*format F*] exprimée en *unit4*.

Exemple

uf1 = in**a** ; la variable flottante *uf1* reçoit la valeur *in**a***

ksync = in**a** {la variable système *ksync* est à l'image de l'entrée analogique *in**a***.}

Soit le paramètre *unit4* égal à "*m/s*".
Une tension de 5 Volts à l'entrée correspond à l'acquisition d'une vitesse linéaire de 40 m/s.
scale_ina = (40/5) = 8 m/s ⇒ *scale_ina* = 8

La pleine échelle de ± 10 Volts est convertie en ± 80 mètres/seconde, accessible par la variable *in**a***.

```
10  %PROG10
11   ; sous-programme d'affichage de la vitesse linéaire, acquise par l'entrée
     analogique.
12  unit4 = "m/s"
13  scale_ina = 8
14  ADR = 42
15  PAGE = 1
16  LINE = 1
17  COL = 3
18  PRINT(ina, f8_3)  ; affichage de la vitesse acquise, en mètres/seconde.
19  …
20  uf1 = ina*3.6
21  LINE = 2
22  PRINT(uf1, f8_3)  ; affichage de la vitesse acquise, en kilomètres/heure.
23  …
24  …
25  …
26  RETURN
27  %ENDPROG
```

Voir aussi

Utilisation des variables *a_in**a*** *et scale_in**a***.

INDEX

Fonction

Spécifie un déplacement relatif en mode « stop cote ».

Synopsis

INDEX = valeur déplacement

INDEX = variable_source

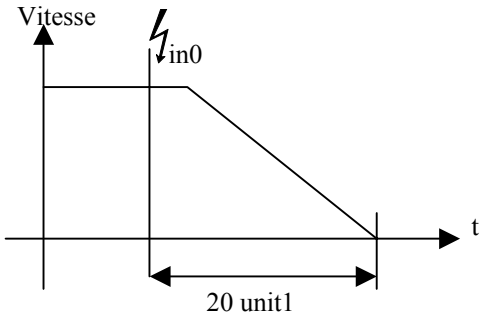
Description

- L'instruction *INDEX* ne s'utilise que dans le sous-programme prioritaire associé à l'entrée interruptive *in0*.
- Dès lors que cette instruction est rencontrée, le système considère la position de l'axe et la corrige des éventuels décalages temporels dus au temps de propagation du capteur et de l'entrée logique *in0*
  (*cf. chapitre 2.7, datation de l'événement interruptif, emploi de in0*).
- La position absolue à atteindre est ainsi déterminée à l'aide de ces données et de la distance d'arrêt programmée (déplacement relatif à accomplir, spécifié par *INDEX*).
- Le mouvement est alors commandé pour atteindre cette destination : c'est le mode « stop cote ».
- *variable_source* est une variable au format *F* ou *D*.

Exemple

```
10   %PROG10
11    ; sous programme utilisateur
12    ;
13   INTERRUPT0 = PROG11
14   IT_ON = IN0
15   { activation de l'entrée
16     interruptive in0.}
17    ;
18   ACCEL = 1000
19   SPEED = 500
20   FORWARD
21    ; ordre de déplacement
22    ; continu
23   WAIT_UNTIL moving = 0
24   {attente de la fin du
25    mouvement géré dans
26    PROG11.}
27   …
28   …
29   RETURN
30   %ENDPROG
```

```
31   %PROG11
32    ; sous-programme prioritaire déclenché
33    ; par in0.
34    ;
35   IT_OFF = IN0
36    ; désactivation de l'entrée interruptive in0.
37    ; prise d'effet de la position réelle de l'axe
38    ; corrigée des décalages temporels.
39    ;
40   INDEX = 20
41   {ordre de stopper le mouvement, intégrant
42    le déplacement relatif souhaité comme
43    distance d'arrêt (stop cote).}
44    ;
45   END
46   %ENDPROG
```



Voir aussi

WAIT_UNTIL (Description des variables système permettant de savoir si un déplacement est en cours ou terminé).

#INIT

Fonction

Le programme utilisateur démarre son exécution à l'adresse *#INIT*.

Synopsis

#INIT

Description

- *#INIT* s'utilise comme en-tête de la zone déclarative dans le programme principal *PROG0*. Cette partie de programme définit et initialise les éventuels programmes automate et sous-programmes prioritaires.

- Cette adresse représente l'adresse de départ de l'exécution des programmes.

Propriétés

#INIT s'utilise exclusivement dans le programme principal intitulé *PROG0*.
L'instruction *PROG_INIT* permet d'accéder directement à l'adresse *#INIT* depuis le programme principal, ou un quelconque autre sous-programme.

Exemple

```
1       %PROG0
2        ; exemple de programme principal.
3       #INIT
4        ;
5       PLC1 = PROG100
6       DEFPLC1 = 1000
7       PLC1 = START
8        ;
9       #START
10      …
11                  GOSUB PROG10
12                  GOSUB PROG20
13      …
14         IF in2 = 1     THEN    RESTART
15         ENDIF
16      PROG_INIT ; sinon, saut à l'adresse #INIT
17      %ENDPROG
```

Voir aussi

PROG_INIT

INT

Fonction

Convertit une variable flottante [format *F*] ou double précision [format *D*], au format entier *E*.

Synopsis

variable_destination = INT(variable_source)

Description

L'instruction *INT* convertit la valeur d'une *variable_source* exprimée au format *F* ou *D*, dans un format *E* et attribue le résultat de cette conversion à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
E	F
E	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

- Si un dépassement de capacité survient lors de l'exécution de *INT,* selon le signe affecté à la *variable_source*, le résultat de la conversion se trouve saturé au plus grand ou au plus petit entier possible.

Exemples

ui1 = INT(uf1)
{conversion de la variable flottante *uf1* au format entier dans la variable *ui1*.}

ui1 = INT(ud1)
{conversion de la variable double précision *ud1* au format entier dans la variable *ui1*.}

INTERRUPT

Fonction

Déclare quel est le sous-programme prioritaire associé à une entrée logique interruptive. Voir à ce sujet le chapitre 2.7

Synopsis

INTERRUPT***x*** = PROG***n***

Description

INTERRUPT s'utilise dans la zone déclarative du programme principal ou d'un sous-programme.
x est un entier compris entre **0** et **3** correspondant à la sélection de l'entrée logique in**0**, in1, in2 ou in**3**.
*PROG**n*** désigne le nom du sous-programme prioritaire associé à l'entrée logique spécifiée ; ***n*** est un entier compris entre 1 et 999.

Propriétés

Si *PROG**n*** n'existe pas ou si ***n*** est hors limites, une erreur sera générée lors de la compilation. De même si ***x*** est hors limites.

Exemple

```
1       %PROG0
2        ; programme principal.
3       #INIT
4       …
5       INTERRUPT2 = PROG30
6       {le programme  n°30 est déclaré sous-programme prioritaire,
7       il est associé à l'entrée logique interruptive in2.}
8       …
9       #START
10      …
11      …
12          GOSUB PROG10
13          GOSUB PROG20
14      …
15      …
16          IF in2 = 1THEN RESTART
17          ENDIF
18      PROG_INIT
19      %ENDPROG
```

Voir aussi

IT_ON, IT_OFF

IT_OFF

Fonction

Invalide la prise en compte de l'entrée interruptive spécifiée. Voir à ce sujet le chapitre 2.7

Synopsis

IT_OFF = IN***x***

Description

*IT_OFF = IN**x*** interdit le traitement du sous-programme prioritaire déclaré par *INTERRUPT*, lorsque survient un changement d'état sur l'entrée interruptive spécifiée *IN**x*.**
x est un entier compris entre **0** et **3** correspondant à la sélection de l'entrée logique *in**0***, *in**1***, *in**2*** ou *in**3***.

Propriétés

Si ***x*** est une valeur hors limites, une erreur sera générée lors de la compilation.

L'instruction *IT_OFF = IN**x*** correspond à la mise à 0 de la variable système *int_in**x*.**

Exemple

```
10      %PROG10
11      ; Voici un exemple de sous-programme.
12      INTERRUPT2 = PROG30
13      IT_ON = IN2       ; l'entrée interruptive in2 est validée.
14      …
15      #DEBUT
16          HOME
17          WAIT_UNTIL home_made = 1
18          …
19          …
20          IT_OFF = IN2
21          { l'entrée interruptive in2 est invalidée. Dorénavant, un changement d'état sur cette entrée
22           ne provoquera plus le traitement du sous-programme prioritaire associé n°30.}
23          …
24          …
25      #FIN
26      RETURN
27      %ENDPROG
```

Voir aussi

INTERRUPT, IT_ON

IT_ON

Fonction

Valide la prise en compte de l'entrée interruptive spécifiée. Voir à ce sujet le chapitre 2.7

Synopsis

IT_ON = IN***x***

Description

- L'instruction *IT_ON = IN**x*** autorise le traitement du sous-programme prioritaire déclaré par *INTERRUPT* lorsque survient un changement d'état sur l'entrée interruptive spécifiée *in**x*.**

- La sélection du front montant ou descendant est effectuée lors du réglage des paramètres variateur ou par l'intermédiaire des variables système *level_in0* à *level_in3*.

- ***x*** est un entier compris entre **0** et **3** correspondant à la sélection de l'entrée logique *in**0***, *in**1***, *in**2*** ou *in**3***.

Propriétés

Si ***x*** est une valeur hors limites, une erreur sera générée lors de la compilation.

L'instruction *IT_ON = IN**x*** correspond à la mise à 1 de la variable système *int_in**x*.**

Exemple

```
10      %PROG10
11      ; Voici un exemple de sous-programme.
12      INTERRUPT2 = PROG30
13       …
14      #DEBUT
15          HOME
16          WAIT_UNTIL home_made = 1
17       …
18       …
19          IT_ON = IN2
20          { un changement d'état sur l'entrée interruptive in2 provoquera désormais le traitement du
21            sous-programme prioritaire associé n°30.}
22          …
23          …
24      #FIN
25      RETURN
26      %ENDPROG
```

Voir aussi

INTERRUPT, IT_OFF

ki_red

Fonction

Coefficient de réduction de courant.

Synopsis

ki_red = valeur

ki_red = variable_source

Description

- *ki_red* s'applique à *i_red* , la variable système de limitation du courant.
- *valeur* est une donnée, *variable_source* une variable, exprimées au format flottant *F* ou double précision *D*. Leur valeur est comprise entre :
  0 ⇒ la réduction de courant est maximale ⇔ le moteur est à couple nul.
  1 ⇒ il n'y a pas de réduction de courant appliquée à *i_red*.
- *ki_red* vaut 1 par défaut à la mise sous tension de l'appareil.

Exemples

La valeur de limitation courant est supposée avoir été programmée à 50 A dans l'environnement *PME Réglages asservissements* [variable système correspondante *: i_red*].

- Exemple 1. Dans le sous_programme n°10, l'entrée analogique *in**a*** sera affectée à la fonction réduction de courant. La limitation de courant est portée à 40 Ampères quand l'entrée analogique vaut 8 volts. 8 Volts ⇔ 80% de réduction courant ⇔ ki_red = 0.8 ⇔ limitation courant = 0.8*50 = 40 A.

```
10      %PROG10
11      …
12      a_ina = 1
13      { le numéro d'ordre 1 correspond à l'affectation de l'entrée analogique à ki_red.}
14      …
15      …
16      …
17      RETURN
18      %ENDPROG
```

- Exemple 2. Dans le sous-programme n°11, la limitation courant est portée à 15 Ampères quand l'entrée logique *in0* passe à 1. ki_red = 0.3 ⇔ limitation courant = 0.3*50 = 15 A.

```
30      %PROG11
31      …
32      IF in0 = 1 THEN ki_red = 0.3
33      { la réduction de courant sera portée à 0.3 * i_red soit 15 A si in0 vaut 1.
34      i_red est la valeur du courant de limitation ayant été programmée en Ampères dans
35      l'environnement PME Réglages asservissements.}
36      ENDIF
37      …
38      …
39      RETURN
40      %ENDPROG
```

kp

Fonction

Permet de modifier le gain de la boucle d'asservissement en position.

Synopsis

kp = valeur_de_gain

kp = variable_source

Description

- La variable *kp* gouverne le gain proportionnel de l'asservissement de position.
- *valeur_de gain* ou *variable_source* présentent un format flottant *F* ou double précision *D* et s'expriment en s^{-1}.

Exemple

kp = 100 ; le gain de la boucle de position est porté à 100 s^{-1}.

kpi_180

Fonction

Constante prédéfinie valant $\pi/180$

Description

- *kpi_180* est une constante au format flottant *F*.
- *kpi_180* s'utilise dans les calculs de conversion entre les angles exprimés en degrés et ceux exprimés en radians.

Exemple

Dans l'exemple suivant, *uf2* désigne une position angulaire exprimée en degrés, et *uf1* sa correspondance en radians.

uf1 = kpi_180 $*$ uf2

si uf2 = 90° $\rightarrow$ uf1 = $^{\pi}/2$

k180_pi

Fonction

Constante prédéfinie valant 180/π

Description

- *k180_pi* est une constante au format flottant *F*.
- Elle s'utilise dans les calculs de conversion entre les angles exprimés en radians et ceux exprimés en degrés.

Exemple

Dans l'exemple suivant, *uf2* désigne une position angulaire exprimée en radians, et *uf1* sa correspondance en degrés.

uf1 = k180_pi $*$ uf2

si uf2 = π/2 $\rightarrow$ uf1 = 90°

KSYNC / ksync_d

Fonction

Définissent, l'un ou l'autre, la valeur du rapport de recopie dans les applications de synchronisme de type « maître-esclave ».

Synopsis

KSYNC = valeur_du_rapport_de_recopie

KSYNC = variable_source

ksync_d =valeur_du_rapport_de_recopie

ksync_d = variable_source

Description

- *valeur_du_rapport_de_recopie* ou *variable_source* sont au format flottant *F* ou double précision *D*.
- L'unité du rapport de recopie s'exprime en *unit1/unit2*.
- La variable associée à l'instruction *KSYNC* est *ksync*.

Remarque :
- A la mise sous tension du variateur positionneur, ksync = 0 et ksync_d = 0.
- Si une grande précision de synchronisme est requise, on utilisera l'instruction « ksync_d = » en lieu et place de l'instruction KSYNC (ksync_d est une variable de type réel 40 bits alors que ksync n'est qu'une variable de type réel 32 bits).

Exemple

Dans l'exemple qui suit, le rapport de recopie est fixé à –1.
L'axe asservi (*axe1*) copiera l'axe maître (*axe2*) dans le même rapport, mais en sens contraire. (voir à ce sujet la description des instructions *SYNCHRO* et *SYNCHRO_START*)

KSYNC = -1 (ou ksync_d = -1)

Voir aussi

SYNCHRO, SYNCHRO_START

#LABEL

Fonction

Définit une adresse de branchement.

Synopsis

#LABEL

Description

Une adresse de branchement est définie par le symbole **#** suivi d'une chaîne de caractères (sans limitation du nombre de caractères).

Exemple

```
10      %PROG10
11
12      #DEBUT
13        uf0 = uf0+1
14        IF in1 = 1  THEN GOTO #FIN
15                                ; branchement à l'adresse #FIN.
16                    ELSE  GOTO #DEBUT
17                                ; branchement à l'adresse #DEBUT.
18        ENDIF
19        …
20        …
21      #FIN
22      RETURN
23      %ENDPROG
```

Propriétés

Les caractères autorisés sont alphanumériques ; l'usage préconise de préférence l'emploi des majuscules.
L'adresse de branchement définie doit être unique.

LINE

Fonction

Spécifie le numéro de la ligne du terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

LINE = variable_source
LINE = valeur

Description

L'instruction *LINE* sélectionne la ligne d'une page du terminal spécifié précédemment par *ADR* et *PAGE*.

L'instruction *LINE* attribue la valeur de *variable_source* (ou *valeur*) à la *variable_destination* associée *can_line*.

variable_source (ou *valeur*) doit être comprise entre [1 et 255].

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

- Lorsque le terminal spécifié est un terminal µVision :
  variable_source (ou *valeur*) doit être comprise entre 1 et 2.
  Toute autre valeur attribuée à *can_line* appartenant à l'intervalle [1, 255] est sans effet, mais aucun message d'erreur n'est généré.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*can_line*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
ADR = 4
PAGE = 3
LINE = 2
COL = 1
PRINT("texte")      {affichage de texte débutant à la page n° 3, ligne n° 2, colonne n°1, sur le
                     terminal abonné au bus CAN n°4}
```

Voir aussi

ADR, PAGE, CLEAR_LINE, PRINT, READ

LN

Fonction

Logarithme népérien.

Synopsis

variable_destination = LN(variable_source)

variable_destination = LN(valeur)

Description

La valeur résultante est attribuée à la *variable_destination*.

variable_source et *valeur* doivent être strictement positives.

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

uf1 = LN(uf2)
uf2 = LN(2.71828182846) ; *uf2* sera sensiblement égal à 1.

LOG

Fonction

Logarithme base 10.

Synopsis

variable_destination = LOG(variable_source)

variable_destination = LOG(variable_destination

Description

La valeur résultante est attribuée à la *variable_destination*.

variable_source et *valeur* doivent être strictement positives.

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

uf1 = LOG(uf2)
uf2 = LOG(10) ; *uf2* sera égal à 1.

master_speedmax

Fonction

Définit la vitesse max de ligne dans des applications de coupe à longueur (applications comportant des synchronisations maître/esclave avec déplacements relatifs supplémentaires de l'axe esclave). Pour plus de détails, voir notices : PVD 3531 "Logiciel d'application Coupes à longueur linéaires avec cisaille volante" et PVD 3532 "Logiciel d'application Coupes à longueur avec couteaux rotatifs".

Synopsis

master_speedmax = valeur_vitesse

master_speedmax = variable_source

Description

- l'instruction master_speedmax = valeur permet d'enclencher un processus qui a pour objet de rendre la vitesse de déplacement de tout mouvement relatif programmé en cours de synchro, proportionnel à la vitesse d'avance de la ligne. Ce processus est comparable à un atténuateur de vitesse :

  synchro_att = speed2 / master_speedmax
  si synchro_att > 1 alors synchro_att = 1
  si synchro_att < master_speedthreshold alors synchro_att = 0

- La programmation de master_speedmax = 0 arrête le processus décrit au paragraphe précédent.

- valeur_vitesse est une donnée, variable_source une variable, exprimées en unit1/s, au format flottant ou double précision D.

- valeur_vitesse et variable_source doivent être strictement positives.

- à la mise sous tension du variateur positionneur, master_speedmax = 0, master_speedthreshold = 0.001 et synchro_att = 1.

Propriétés

Les formats attribués aux variables sont les suivants :

format variable_destination [master_speedmax]	format variable_source
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

MAX

Fonction

Maximum

Synopsis

variable_destination = MAX(var1, var2)

variable_destination = MAX(var1, valeur)

Description

- L'instruction *MAX* détermine quelle est la valeur maximale entre deux variables ou entre une variable et une valeur.
- La valeur maximale issue de la comparaison est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
E	E
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
uf2 = 100
uf3 = 1000

uf1 = MAX(uf2,uf3)          ; uf1 sera égal à 1000
uf1 = MAX(uf2,5000)         ; uf1 sera égal à 5000
```

MIN

Fonction

Minimum

Synopsis

variable_destination = MIN(var1, var2)

variable_destination = MIN(var1, valeur)

Description

- L'instruction *MIN* détermine quelle est la valeur minimale entre deux variables ou entre une variable et une valeur.
- La valeur minimale issue de la comparaison est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
E	E
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

```
uf2 = 100
uf3 = 1000

uf1 = MIN(uf2,uf3)          ; uf1 sera égal à 100
uf1 = MIN(uf2,5000)         ; uf1 sera égal à 100
```

MOVEA

Fonction

Spécifie un mouvement dans le repère absolu.

Synopsis

MOVEA = variable_source

MOVEA = cote_à_atteindre

Description

cote_à_atteindre est une donnée, exprimée au format entier E, flottant *F* ou double précision *D*, attribuée à la variable système ***posa***.

variable_source est une variable au format entier E, flottant *F* ou double précision *D*, dont la valeur est attribuée à la variable système ***posa***.

variable_source et *cote_à_atteindre* sont exprimées en *unit1*.

Les phases d'accélération, de décélération et à vitesse constante sont gouvernées par les paramètres machines actifs au moment de l'exécution de *MOVEA* (*accel_move* et *speed_move*). Remarque : accélération/décélération et vitesse peuvent être modifiés lors du déplacement en utilisant les instructions *ACCEL_IM* et *SPEED_IM*.

Exemples

```
10  %PROG10
11  ACCEL = 1000
12  SPEED = 500
13   ;
14  MOVEA = 70
15  {déplacement de 70 unit1 à la
16  vitesse de 500 unit1/s et phase
17  d'accélération et décélération à
18  1000 unit1/s².}
19   ;
20   ;
21  WAIT_UNTIL moving = 0
22   ;
23  {attente de fin de mouvement
24  théorique et poursuite du
25  sous-programme.}
26  …
27  …
28  …
29  …
30  RETURN
31  %ENDPROG
```

```
32  %PROG20
33  ACCEL = 1000
34  SPEED = 500
35   ;
36  MOVEA = 70
37   ;
38  WAIT_UNTIL in_position = 1
39   ;
40  {attente de l'information système
41  « but atteint » et poursuite du
42   sous-programme.}
43  …
44  …
45  …
46  …
47
48  RETURN
49  %ENDPROG
```

Voir aussi

WAIT_UNTIL (Description des variables système permettant de savoir si un déplacement est en cours ou terminé).

move_en

Fonction

Autorise ou interdit l'exécution des mouvements sans action directe sur l'exécution des programmes.

Synopsis

move_en = 0

move_en = 1

move_en = variable_source

Description

- move_en = **0** permet de de suspendre un mouvement en cours ou d'interdire tout nouveau déplacement. S'il y avait un déplacement en cours, l'axe décélère [accel_prog] puis s'arrête. La position obtenue en fin de décélération reste asservie et la consigne de position à atteindre [posa] n'est pas modifiée. Il n'y a aucune action directe sur l'exécution des programmes.

- *move_en* = ***1*** autorise l'exécution des mouvements.

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

- La transition d'un état haut vers un état bas (*move_en* = 1 → 0) suspend un mouvement en cours ; la position obtenue en fin de décélération reste asservie. La consigne de position à atteindre [*posa*] n'est pas modifiée.

- La transition d'un état bas vers un état haut (*move_en* = 0 → 1) redémarre un mouvement suspendu avec sa consigne de position initiale (s'il y en avait un en cours). Il n'y a aucun à-coup à la reprise du mouvement.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*move_en*]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Voir aussi

PVD 3516, chapitre 5.2., autorisation mouvement à la mise sous tension.
START
STOP

MOVER

Fonction

Spécifie un mouvement relatif.

Synopsis

MOVER = variable_source
MOVER = valeur_déplacement

Description

valeur_déplacement est une donnée, exprimée au format entier E, flottant *F* ou double précision *D*, attribuée à la variable système ***posr***.

variable_source est une variable au format entier E, flottant *F* ou double précision *D*, dont la valeur est attribuée à la variable système ***posr***.

variable_source et *valeur_déplacement* sont exprimées en *unit1*.

Les phases d'accélération, de décélération et à vitesse constante sont gouvernées par les paramètres machines actifs au moment de l'exécution de *MOVER* (*accel_move* et *speed_move*). Remarque : accélération/décélération et vitesse peuvent être modifiés lors du déplacement en utilisant les instructions *ACCEL_IM* et *SPEED_IM*.

Exemple

Dans l'exemple qui suit, un déplacement relatif de 30 *unit1* est ordonné.
La position initiale (absolue) est de 100 *unit1*.
La position résultante, à l'issue du déplacement relatif *MOVER*, sera de 130 *unit1* dans le repère absolu.

```
10  %PROG10
11  ACCEL = 1000
12  SPEED = 500
13  MOVER = 30
14  WAIT_UNTIL moving = 0
15  {attente de fin de mouvement théorique et poursuite
16  du sous-programme}
17  …
18  …
19  RETURN
20  %ENDPROG
```

Voir aussi

WAIT_UNTIL (Description des variables système permettant de savoir si un déplacement est en cours ou terminé).

NEXT

Fonction *FOR / NEXT*
Se reporter à la description de *FOR*.

out*0*… à …out*7*

Fonction

Contrôle une sortie logique.

Synopsis

out***x*** = 1
out***x*** = 0
out***x*** = variable_source

Description

- *out**x*** = *1* avec ***x***, un entier compris entre **0** et **7**, met à 1 la sortie logique [*out**0**,…out**7***] concernée.

- *out**x*** = *0* met à zéro la sortie logique spécifiée.

- *variable_source* est une variable binaire [format *B*].

Attention : La sortie logique mentionnée ne doit pas être affectée à une variable système (*home_made*, *moving*, *in_position,* etc…)
[*cf. PVD 3516 chapitre 3.4.2, Affectation des sorties logiques*].
Dans le cas contraire, la valeur spécifiée au cours du programme serait « écrasée » immédiatement par la variable d'affectation.

Propriétés

Si ***x*** est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Exemples

out1 = 0 ; mise à 0 de la sortie logique n°1.

out2 = ub4 ; attribution de la valeur binaire de *ub4* à la sortie *out2*.

Voir aussi

Utilisation des variables système *a_out0* à *a_out7*

outa

Fonction

Impose une tension sur la sortie analogique.

Synopsis

out**a** = valeur
out**a** = variable_source

Description

valeur ou *variable_source* présentent un format flottant *F* ou double précision *D*.

valeur et *variable_source* sont exprimées en *unit3.*

Exemples

```
outa = 7.5        ; la sortie analogique est fixée à 7.5 unit3.

uf1 = 10
outa = uf1        ; la sortie analogique vaut 10 unit3.

outa = pos1       { la sortie analogique reproduit la position réelle de l'axe
                    asservi pos1. Le facteur d'échelle, exprimé en Volt/Unités,
                    est donné par la variable système scale_outa.
                    [cf. PVD 3516 chapitre 5.4.3, Sortie analogique] }
```

Voir aussi

Utilisation des variables *a_out**a** et scale_out**a***.

PAGE

Fonction

Spécifie le numéro de page du terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

PAGE = variable_source
PAGE = valeur

Description

L'instruction *PAGE* sélectionne la page du terminal spécifiée précédemment par *ADR*.

L'instruction *PAGE* attribue la valeur de *variable_source* (ou *valeur*) à la *variable_destination* associée *can_page*.

variable_source (ou *valeur*) doit être comprise entre [1 et 255].

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

- Lorsque le terminal spécifié est un terminal µVision :
  variable_source (ou *valeur*) doit être comprise entre 1 et 4.
  Toute autre valeur attribuée à *can_page* appartenant à l'intervalle [1, 255] est sans effet, mais aucun message d'erreur n'est généré.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*can_page*]	format *variable_source* :
E	E
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
ADR = 4
PAGE = 2
LINE = 3
COL = 1
PRINT("texte")      {affichage de texte débutant à la page n°2, ligne n°3, colonne n°1, sur le
                     terminal abonné au bus CAN n°4}
```

Voir aussi

ADR, CLEAR_PAGE, DISPLAY, PRINT, READ

pi

Fonction

Constante prédéfinie valant π

Description

- *pi* est une constante au format flottant *F.*

Exemple

Dans l'exemple suivant, *uf2* désigne un diamètre, et *uf1* la circonférence du cercle correspondant.

uf1 = pi ∗ uf2

PLC1

Fonction

Opérateur de gestion du programme automate de type 1 (programme automate générique). Voir à ce sujet le chapitre 2.8.1

Synopsis

PLC1 = PROG***n***
PLC1 = START
PLC1 = STOP
PLC1 = NONE

Description

PLC1 = PROGn déclare le numéro de programme automate de type 1.

La déclaration *PLC1 = PROG**n*** figure le plus souvent dans la zone déclarative du programme principal. ***n*** désigne le numéro de programme automate de type 1. Selon les applications développées, *PLC1 = PROGn* peut figurer dans un sous-programme voire un programme automate de type 2.

PLC1 = START démarre le programme automate de type 1.

L'instruction *PLC1 = START* figure le plus souvent dans la zone déclarative du programme principal. Elle a pour but de démarrer l'exécution du programme automate de type 1 ou éventuellement de le relancer, s'il avait été précédemment arrêté (Dans ce cas le programme automate redémarrera **à son début**). Selon les applications développées, *PLC1 = START* peut figurer dans un sous-programme voire un programme automate de type 2.

PLC1 = STOP arrête le programme automate de type 1.

L'instruction *PLC1 = STOP* arrête l'exécution du programme automate de type 1. Elle peut figurer dans le programme principal, dans un sous programme ou un programme automate de type 1 ou 2. Cette instruction offre certaines possibilités de programmation ; son emploi doit être justifié et non systématisé.

PLC1 = NONE déclare l'inexistence d'un programme automate de type 1.

La déclaration *PLC1 = NONE* mentionne qu'il n'y a pas de programme automate de type 1 dans la *partition* des programmes. Elle figure généralement dans la zone déclarative du programme principal. Les instructions *PLC1 = START* et *PLC1 = STOP* rendues caduques par *PLC1 = NONE*, sont ignorées par le système opérationnel.

Exemple

Voir la présentation *Programme automate de type 1* au chapitre 2.8.1 de ce document.

PLC2

Fonction

Opérateur de gestion du programme automate de type 2 (programme automate cyclique). Voir à ce sujet le chapitre 2.8.2

Synopsis

PLC2 = PROG***n***
PLC2 = START
PLC2 = STOP
PLC2 = NONE

Description

PLC2 = PROGn déclare le numéro de programme automate de type 2.

La déclaration *PLC2 = PROG**n*** figure le plus souvent dans la zone déclarative du programme principal. ***n*** désigne le numéro de programme automate de type 2. Selon les applications développées, *PLC2 = PROG**n*** peut figurer dans un sous-programme voire un programme automate de type 1.

PLC2 = START démarre le programme automate de type 2.

L'instruction *PLC2 = START* figure le plus souvent dans la zone déclarative du programme principal. Elle a pour but de démarrer l'exécution du programme automate de type 2 ou éventuellement de le relancer, s'il avait été précédemment arrêté (Dans ce cas le programme automate redémarrera **à son début**). Selon les applications développées, *PLC2 = START* peut figurer dans un sous-programme voire un programme automate de type 1.

PLC2 = STOP arrête le programme automate de type 2.

L'instruction *PLC2 = STOP* arrête l'exécution du programme automate de type 2. Elle peut figurer dans le programme principal, dans un sous-programme ou un programme automate de type 1 ou 2. Bien que cette instruction soit disponible, elle présente peu d'intérêt et sera rarement employée.

PLC2 = NONE déclare l'inexistence d'un programme automate de type 2.

La déclaration *PLC2 = NONE* mentionne qu'il n'y a pas de programme automate de type 2 dans la *partition* des programmes. Elle figure généralement dans la zone déclarative du programme principal. Les instructions *PLC2 = START* et *PLC2 = STOP*, rendues caduques par *PLC2 = NONE*, sont ignorées par le système opérationnel.

Exemple

Voir la présentation *Programme automate de type 2* au chapitre 2.8.2 de ce document.

pos1 / pos1_f

Fonction

Variable système qui permet l'accès à la position réelle de l'axe asservi (axe 1).

La position est issue de l'information resolver (ou de l'information codeur si *option_card* = 1 ou 3 et *encoder_use* = 0)

Synopsis

variable_destination = pos1

Description

- *variable_destination* présente un format double précision *D* et est exprimée en *unit1*.
- Le contenu de la variable_source *pos1* est attribué à la *variable_destination*.
- *pos1* est une variable système : elle reflète la position réelle de l'axe asservi ; cette donnée est rafraîchie à chaque période d'échantillonnage de 250 μsecondes.
- *pos1_f* représente la variable système *pos1* exprimée au format flottant F.

Exemple

```
10    %PROG10
11     ; sous programme utilisateur
12     ;
13    ACCEL = 1000
14    SPEED = 500
15    FORWARD                    ; ordre de déplacement continu
16     ;
17    WAIT_UNTIL pos1 >= 100     ; attendre que l'axe 1 ait dépassé 100 unit1
18     ;
19    STOP
20    WAIT_UNTIL moving = 0
21    ud0 = pos1                 ; mémorisation de la position d'arrêt de l'axe
22    …
23    …
24    …
25    RETURN
26    %ENDPROG
```

Voir aussi

pos2, pos2_f

pos2 / pos2_f

Fonction

Variable système qui permet l'accès à la position réelle de l'axe mesuré (axe 2).

La position est issue de l'information codeur extérieur (*option_card* = 1 ou 3).

Synopsis

variable_destination = pos2

Description

- *variable_destination* présente un format double précision *D* et est exprimée en *unit2*.
- Le contenu de la variable_source *pos2* est attribué à la *variable_destination*.
- *pos2* est une variable système : elle reflète la position réelle de l'axe mesuré ; cette donnée est rafraîchie à chaque période d'échantillonnage de 250 μsecondes.
- *pos2_f* représente la variable système *pos2* exprimée au format flottant F.

Exemple

```
10    %PROG10
11     ; sous programme utilisateur
12
13    ACCEL = 1000
14    SPEED = 500
15    FORWARD
16     ; ordre de déplacement continu de l'axe 1
17     ;
18    WAIT_UNTIL pos2 >= 100
19    {attendre que l'axe 2 ait dépassé 100 unit2 avant de commander l'arrêt}
20     ;
21    STOP
22    …
23    …
24    …
25    RETURN
26    %ENDPROG
```

Voir aussi

pos1, pos1_f

PRINT

Fonction

Affiche un texte ou une variable sur le terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

PRINT (“texte”)
PRINT (variable, format)

Description

Nom du format	Format des variables	Description
def	toutes	format par défaut de la variable : *B* bit : b1 *E* entier : i11 *F* flottant : e *D* double précision : e *C1* : pas de format *C2* : pas de format
b1	B	1 digit : variable binaire = 0 ⇒ “0” variable binaire = 1 ⇒ “1”
b3	B	3 digits : variable binaire = 0 ⇒ “ NO” variable binaire = 1 ⇒ “YES”
i6	E	6 digits : ⇒ “±12345”
i11	E	11 digits : ⇒ “±1234567890”
e	F, D	16 digits : ⇒ “±1.123456789E±12”
f8_0	F, D	8 digits, 0 après le point ⇒ “±123456.”
f8_1	F, D	8 digits, 1 après le point ⇒ “±12345.1”
f8_2	F, D	8 digits, 2 après le point ⇒ “±1234.12”
f8_3	F, D	8 digits, 3 après le point ⇒ “±123.123”
f8_4	F, D	8 digits, 4 après le point ⇒ “±12.1234”
f8_5	F, D	8 digits, 5 après le point ⇒ “±1.12345”
f8_6	F, D	8 digits, 6 après le point ⇒ “±.123456”
f12_0	F, D	12 digits, 0 après le point ⇒ “±1234567890.”
f12_1	F, D	12 digits, 1 après le point ⇒ “±123456789.1”
f12_2	F, D	12 digits, 2 après le point ⇒ “±12345678.12”
f12_3	F, D	12 digits, 3 après le point ⇒ “±1234567.123”
f12_4	F, D	12 digits, 4 après le point ⇒ “±123456.1234”
f12_5	F, D	12 digits, 5 après le point ⇒ “±12345.12345”
f12_6	F, D	12 digits, 6 après le point ⇒ “±1234.123456”
f12_7	F, D	12 digits, 7 après le point ⇒ “±123.1234567”
f12_8	F, D	12 digits, 8 après le point ⇒ “±12.12345678”
f12_9	F, D	12 digits, 9 après le point ⇒ “±1.123456789”
f12_10	F, D	12 digits, 10 après le point ⇒ “±.1234567890”

Description (suite)

L'instruction *PRINT* affiche aux coordonnées indiquées par les variables *can_id*, *can_page*, *can_line*, *can_col* :
- Soit une chaîne de caractères alphanumériques "texte", pouvant atteindre 255 caractères (il n'est pas recommandé d'utiliser les caractères accentués).
- Soit la valeur d'une variable, présentée au format spécifié.

can_id est une variable entière de valeur comprise entre [1 et 63] correspondant au numéro d'abonné *CAN* du terminal. Cette variable est gérée par l'instruction *ADR*.

can_page est une variable entière de valeur comprise entre [0 et 255] spécifiant le numéro de la page. Cette variable est gérée par l'instruction *PAGE*.

can_line est une variable entière de valeur comprise entre [0 et 255] correspondant au numéro de la ligne. Cette variable est gérée par l'instruction *LINE*.

can_col est une variable entière de valeur comprise entre [0 et 255] correspondant au numéro de la colonne. Cette variable est gérée par l'instruction *COL*.

variable est le nom de la variable à afficher.

format est le nom du format d'affichage.

Exemple

```
10      %PROG10
11       ; affichage d'un texte et d'une variable sur la même ligne
12      COL = 1
13      PRINT("piece n°")
14      COL = 12
15      PRINT(ui15,def)
16      …
17      …
18      RETURN
19      %ENDPROG
```

Remarque Importante

Il est fortement conseillé de ne gérer un terminal donné qu'à partir d'une tâche unique (soit programme de gestion de mouvements, soit programme automate de type 1, soit programme automate de type 2). En effet, la gestion d'un même terminal à partir de différentes tâches s'exécutant en parallèle pourrait conduire à des comportements incohérents (risque de télescopage d'instructions).

%PROG

Fonction

Définit le numéro d'un programme ou d'un sous-programme.

Synopsis

%PROG***n***

Description

*%PROG**n*** correspond au début du code de description d'un programme ou d'un sous-programme.

n est un nombre entier compris entre 0 et 999.

Le programme principal se distingue par ***n*** = 0.
Les sous-programmes se caractérisent par ***n*** > 0.

Propriétés

Si ***n*** est hors limites, une erreur sera générée lors de la compilation.

*%PROG**n*** est une directive exclusivement destinée au compilateur.

Exemples

%PROG***0*** Début de code du programme principal.

%PROG***30*** Début de code du sous-programme n° 30.

Voir aussi

%ENDPROG

PROG

Fonction *ERROR* = *PROG**n***
Se reporter à la description de *ERROR*.

Fonction *INTERRUPT**x*** = *PROG**n***
Se reporter à la description de *INTERRUPT*.

Fonction *GOSUB PROG**n***
Se reporter à la description de *GOSUB*.

PROG_INIT

Fonction

Définit un saut inconditionnel à l'adresse *#INIT* du programme *PROG0*.

Synopsis

PROG_INIT

Description

PROG_INIT s'utilise pour se brancher à l'adresse *#INIT* de *PROG0* afin de redémarrer l'exécution du programme principal.

Exemple

```
1   %PROG0
2    ; programme principal.
3   #INIT  ; démarrage du programme principal.
4   PLC1 = PROG100
5   DEFPLC1 = 1000
6    ; fin de zone déclarative.
7   #START ; début du corps de programme principal.
8   IF in1 = 1 THEN PLC1 = START
9    ; démarrage du programme automate de type 1.
10  ENDIF
11  …
12     GOSUB PROG10
13     GOSUB PROG20
14  …
15  …
16  RESTART
17  %ENDPROG

18  %PROG10
19   ; sous-programme n°10
20  …
21  …
22  MOVEA = 500
23  …
24   IF in2 = 1 THEN PROG_INIT
25  {saut à l'adresse #INIT du programme PROG0.}
26   ENDIF
27  …
28  …
29  RETURN
30  %ENDPROG
```

Propriétés

La structure de prise en compte des imbrications de sous-programmes est complètement effacée à l'exécution de l'instruction *PROG_INIT*.

L'instruction PROG_INIT permet d'accéder directement à l'adresse #INIT depuis le programme principal, ou un quelconque autre sous-programme.

Voir aussi

#INIT

READ

Fonction

Acquiert une donnée en provenance du terminal µVision adressé.
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

READ(variable_destination, valeur_min, valeur_max)

Description

- L'instruction *READ* attend une réponse de la part d'un terminal de type *µVision*.

variable_destination est le nom de la variable où sera stockée la donnée acquise.
valeur_min est la valeur minimale d'acceptation de la réponse (donnée introduite au format flottant *F*).
valeur_max est la valeur maximale d'acceptation de la réponse (donnée introduite au format flottant *F*).

- La donnée introduite au clavier est vérifiée par l'instruction *READ* : elle doit être comprise dans l'intervalle d'acceptation [*valeur_min*, *valeur_max*], sinon elle est refusée. La situation est bloquée (pointeur opérationnel non évolutif), tant que l'acquisition au clavier n'est pas correcte.
- L'instruction *READ* utilise les coordonnées indiquées par les variables (*can_id*, *can_page*, *can_line*, *can_col*) :

can_id est une variable entière de valeur comprise entre [1 et 63] correspondant au numéro d'abonné *CAN* du terminal de saisie. Cette variable est gérée par l'instruction *ADR.*
can_page est une variable entière de valeur comprise entre [0 et 255] spécifiant le numéro de la page. Cette variable est gérée par l'instruction *PAGE*.
can_line est une variable entière de valeur comprise entre [0 et 255] correspondant au numéro de la ligne. Cette variable est gérée par l'instruction *LINE*.
can_col est une variable entière de valeur comprise entre [0 et 255] correspondant au numéro de la colonne. Cette variable est gérée par l'instruction *COL*.

Exemple

```
%PROG10
; sous-programme d'acquisition des données de fabrication.
ADR = 10
PAGE = 1
LINE = 1
COL = 1
PRINT("NB DE PIECES ?")
LINE = 2
COL = 1
READ(ui1,1,100)   ; la réponse est acceptée si la donnée saisie est comprise entre 1 et 100.
…
RETURN
%ENDPROG
```

Remarques Importantes

Il est fortement conseillé de ne gérer un terminal donné qu'à partir d'une tâche unique (soit programme de gestion de mouvements, soit programme automate de type 1, soit programme automate de type 2). En effet, la gestion d'un même terminal à partir de différentes tâches s'exécutant en parallèle pourrait conduire à des comportements incohérents (risque de télescopage d'instructions).
READ bloque l'exécution de la tâche dans laquelle elle a été programmée tant qu'une réponse correcte n'a pas été fournie. Il est donc fortement déconseillé d'utiliser cette instruction dans un programme automate.

reset_cmd

Fonction

Assure l'acquittement des défauts

Synopsis

reset_cmd = 1

reset_cmd = 0

reset_cmd = variable_source

Description

reset_cmd = 0 ne provoque aucun effet
reset_cmd = 1 efface tout défaut signalé par l'afficheur 7 segments en face avant du variateur positionneur. Il y a suppression de la mémorisation du défaut, mais pas de sa cause.
reset_cmd = 1 n'a aucun effet si aucun défaut n'était signalé.
reset_cmd est remis automatiqement à 0 dès que la commande a été prise en compte.

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*reset_cmd*]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

IF in1 = 1 AND in2 = 0 THEN *reset_cmd* = 1

ENDIF

RESTART

Fonction

Définit un saut inconditionnel à l'adresse *#START* du programme *PROG0*.

Synopsis

RESTART

Description

RESTART s'utilise pour se brancher à l'adresse *#START* afin de répéter une séquence d'instructions appartenant au corps de programme principal.

Exemple

```
1    %PROG0
2     ; programme principal.
3    #INIT     ; démarrage du programme principal.
4    PLC1 = PROG100
5    DEFPLC1 = 1000
6     ; fin de zone déclarative.
7    #START ; début du corps de programme principal.
8    IF in1 = 1 THEN PLC1 = START
9     ; démarrage du programme automate de type 1.
10   ENDIF
11   …
12        GOSUB PROG10
13        GOSUB PROG20
14   …
15   …
16   RESTART
17   %ENDPROG

18   %PROG10
19    ; sous-programme n°10
20   …
21   …
22   MOVEA = 500
23   …
24    IF in2 = 1 THEN RESTART
25   {saut à l'adresse #START du programme PROG0.}
26    ENDIF
27   …
28   …
29   RETURN
30   %ENDPROG
```

Propriétés

L'instruction RESTART permet d'accéder directement à l'adresse #START depuis le programme principal, ou un quelconque autre sous-programme.

La structure de prise en compte des imbrications de sous-programmes est complètement effacée à l'exécution de l'instruction RESTART.

Voir aussi

#START

RETURN

Fonction

Définit la fin d'exécution d'un sous-programme.

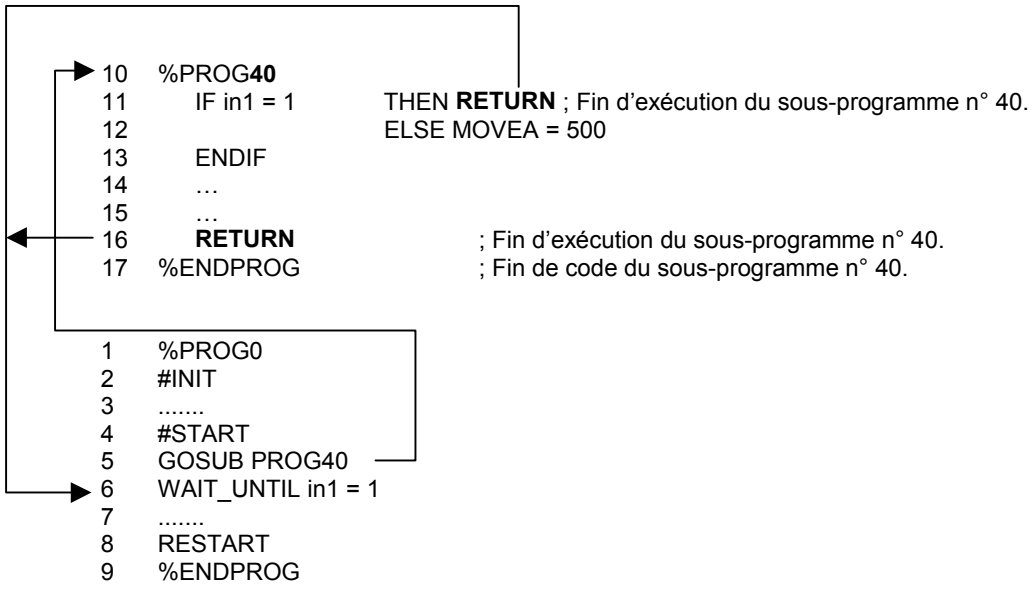
Synopsis

RETURN

Description

RETURN termine le déroulement d'un sous-programme et provoque un retour au programme appelant.
Le pointeur opérationnel se place à la ligne de programme qui suit l'instruction *GOSUB* qui avait appelé le sous-programme.

Exemple



Propriétés

- *RETURN* s'utilise exclusivement dans les sous-programmes de gestion des mouvements.
- *RETURN* est une instruction opérationnelle à ne pas confondre avec *%ENDPROG* directive uniquement destinée au compilateur.

Voir aussi

%ENDPROG, GOSUB

REVERSE

Fonction

Définit un déplacement continu dans le sens **-**.

Synopsis

REVERSE

Description

- L'instruction *REVERSE* se traduit par *MOVEA* = **-**∞ (déplacement absolu infini dans le sens négatif, variable système *posa* = **-**∞).
- Les phases d'accélération, de décélération et à vitesse constante sont gouvernées par les paramètres machines actifs au moment de l'exécution de *REVERSE* (accel_move et speed_move). Remarque : accélération/décélération et vitesse peuvent être modifiées lors du déplacement en utilisant les instructions *ACCEL_IM* et *SPEED_IM*.

Exemple

```
20  %PROG10
21  ACCEL = 1000
22  SPEED = 500
23   ;
24  IF in1 = 1 THEN REVERSE
25  {l'entrée logique in1 à l'état 1 provoque un déplacement continu dans le
26   sens négatif, à la vitesse de 500 unit1/s.}
27  ENDIF
28  …
29  …
30  …
31  RETURN
32  %ENDPROG
```

Voir aussi

FORWARD
WAIT_UNTIL (Description des variables système permettant de savoir si un déplacement est en cours ou terminé).

SAVE

Fonction

Sauvegarde la valeur d'un paramètre (R/W : accès en lecture et écriture) en EEPROM_DM.

Synopsis

SAVE(paramètre_R/W)

Description

- L'instruction *SAVE* sauvegarde en mémoire non volatile *EEPROM_DM*, la valeur d'un *paramètre_R/W* présent en mémoire *RAM_DM*.
- Consulter la liste des *paramètres_R/W* dans le document [*PVD 3527 DIGIVEX Motion, Répertoire des variables*].
- L'instruction SAVE permet aussi de sauvegarder en mémoire non volatile EEPROM_DM, les variables utilisateur :
  - ub0 à ub15
  - ui0 à ui15
  - uf0 à uf15
  - ud0 à ud15

  Attention ! Ces variables utilisateur sont initialisées à la mise sous tension du variateur positionneur par les dernières valeurs sauvegardées en EEPROM_DM au moyen de la commande SAVE.

Exemple

SAVE(uf1) {sauvegarde la valeur de la *variable_utilisateur uf1* en mémoire *EEPROM*.}

Attention !

Le nombre maximum d'écritures possibles en EEPROM est limité à environ 1 Million pour chaque variable. Au-delà de ces limites, le fonctionnement de la fonction SAVE n'est plus assuré (nécessité de changer la mémoire EEPROM).

scale_ina

Fonction

Permet de changer le facteur d'échelle de l'entrée analogique.

Synopsis

scale_in**a** = valeur_gradient
scale_in**a** = variable_source

Description

- *valeur_gradient* est une donnée, *variable_source* une variable, présentant un format flottant *F* ou double précision *D*.
- *scale_in**a*** est un paramètre exprimé en *unit4/Volt.* C'est le coefficient multiplicateur de la conversion analogique – numérique.
- *unit4* est une variable correspondant à l'unité programmée pour l'entrée analogique. [*voir PVD 3516 chapitre 3.4.3.1 Entrée analogique*].

Exemple

Soit le paramètre *unit4* égal à "*m/s"*.
Une tension de 5 Volt à l'entrée correspond à l'acquisition d'une vitesse linéaire de 40 m/s.
Détermination de *scale_ina* : (40/5) = 8 m/s ⇒ *scale_ina* = 8

La pleine échelle de ± 10 Volts est convertie en ± 80 mètres/seconde, accessible par la variable *in**a***.

```
10  %PROG10
11   ; sous-programme d'affichage de la vitesse linéaire, acquise par l'entrée analogique.
12  unit4 = "m/s"
13  scale_ina = 8
14  ADR = 42
15  PAGE = 2
16  LINE = 1
17  COL = 3
18  PRINT(ina, f8_3)     ; affichage de la vitesse acquise, en mètres/seconde.
19  …
20  uf1 = ina*3.6
21  LINE = 2
22  PRINT(uf1, f8_3)     ; affichage de la vitesse acquise, en kilomètres/heure.
23  …
24  …
25  …
26  RETURN
27  %ENDPROG
```

Voir aussi

*a_in**a**, in**a***

scale_outa

Fonction

Permet de changer le facteur d'échelle de la sortie analogique.

Synopsis

scale_out**a** = valeur_gradient
scale_out**a** = variable_source

Description

- *valeur_gradient* est une donnée, variable_source une variable, présentant un format flottant *F* ou double précision *D*.
- *scale_out**a*** est un paramètre exprimé en *Volt/unit3*. C'est le coefficient multiplicateur de la conversion numérique – analogique.
- *unit3* est une variable correspondant à l'unité programmée pour la sortie analogique. [*voir PVD 3516 chapitre 3.4.3.2 Sortie analogique*].

Exemple

Soit le paramètre *unit3* égal à "*m/s*".
Une valeur programmée de vitesse linéaire égale à 40 m/s doit correspondre à une tension de 5 Volts en sortie analogique.

Détermination de *scale_out**a*** : (5/40) = 0.125 *Volt/unit3* $\Rightarrow$ *scale_outa* = 0.125
Une valeur programmée *out**a*** égale à ± 80 mètres/seconde, délivrera en sortie analogique ±10 Volts.

```
10  %PROG10
11  {sous-programme de génération d'une tension analogique à l'image d'une vitesse
12  linéaire, donnée par la variable outa.}
13  …
14  unit3 = "m/s"
15  scale_outa = 0.125
16  outa = -40
17   ; la tension présente en sortie analogique sera de -5 Volts.
18  …
19  …
20  …
21  RETURN
22  %ENDPROG
```

Voir aussi

*a_out**a**, out**a***

SIN

Fonction

Sinus

Synopsis

variable_destination = SIN(variable_source)

variable_destination = SIN(valeur)

Description

variable_source ou *valeur sont* exprimées en radians.
La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = SIN(uf2)

softlimit_en

Fonction

Valide, ou non, la prise en compte effective des fins de course logiciels.

Synopsis

softlimit_en = 1
softlimit_en = 0
softlimit_en = variable_source

Description

softlimit_en = ***1*** **valide** la prise en compte des fins de course logiciels.
softlimit_en = ***0*** **invalide** cette prise en compte.

Propriétés

Les valeurs *softlimit_m* et *softlimit_p* ne sont prises en compte que si une prise d'origine a été réalisée au préalable (*home_made = 1*).

Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*softlimit_en*]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
IF in1 = 1 AND in2 = 0    THEN  softlimit_en = 1
                          ELSE  softlimit_en = 0
ENDIF
```

Remarque

Lorsqu'un fin de course logiciel est sur le point d'être atteint, il y a décélération de l'axe et arrêt sur le fin de course logiciel. La position destination devient la position d'arrêt. L'exécution du programme n'est pas interrompue. Un mouvement programmé pour se dégager du fin de course activé est ensuite possible.

Voir aussi

softlimit_m, softlimit_p

softlimit_m

Fonction

Redéfinit, ou réinitialise, la valeur de fin de course inférieure gérée par le logiciel (fin de course logiciel “moins”).

Synopsis

softlimit_m = valeur

softlimit_m = variable_source

Description

Les valeurs softlimit_m et soflimit_p ne sont prises en compte que si les fins de course logiciel sont valides (soflimit_en = 1) et seulement si une prise d’origine a été réalisée au préalable (home_made = 1).

- L’instruction *softlimit_m* redéfinit la valeur inférieure d’activation du fin de course logiciel *moins*.
- *valeur* est une donnée, *variable_source* une variable, présentant un format flottant *F* ou double précision *D*.
- *softlimit_m* est une variable exprimée en *unit1.* Seules les valeurs négatives sont acceptées par cette variable.

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l’exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*softlimit_m*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

softlimit_m = -50

Voir aussi

softlimit_en, *softlimit_p*

softlimit_p

Fonction

Redéfinit, ou réinitialise, la valeur de fin de course supérieure gérée par le logiciel (fin de course logiciel “plus”).

Synopsis

softlimit_p = valeur

softlimit_p = variable_source

Description

Les valeurs softlimit_m et soflimit_p ne sont prises en compte que si les fins de course logiciel sont valides (soflimit_en = 1) et seulement si une prise d’origine a été réalisée au préalable (home_made = 1).

- L’instruction *softlimit_p* redéfinit la valeur supérieure d’activation du fin de course logiciel *plus*.
- *valeur* est une donnée, *variable_source* une variable, présentant un format flottant *F* ou double précision *D*.
- *softlimit_p* est une variable exprimée en *unit1*. Seules les valeurs positives sont acceptées par cette variable.

Propriétés

Si *variable_source* est hors limites, une erreur sera générée à l’exécution.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*softlimit_p*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

softlimit_p = +50

Voir aussi

softlimit_en, *softlimit_m*

speed1

Fonction

Variable système qui permet l'accès à la vitesse réelle de l'axe asservi (axe 1). Valeur exprimée en unit1/s, au format flottant F.

Description

La vitesse est issue de l'information resolver (*speed1* = *resol_speed*) :
- si *option_card* = 0 ou 2
- si *option_card* = 1 ou 3 et *encoder_use* = 1 ou 2

La vitesse est issue de l'information codeur extérieur (*speed1* = *speed2*) :
- si *option_card* = 1 ou 3 et *encoder_use* = 0

Voir aussi

speed2

speed2

Fonction

Variable système qui permet l'accès à la vitesse réelle de l'axe mesuré (axe 2). Valeur exprimée en unit2/s, au format flottant F.

Description

La vitesse est issue de l'information codeur extérieur :
- si *option_card* = 1 ou 3

La valeur speed2 est nulle :
- si *option_card* = 0 ou 2

Voir aussi

speed1

SPEED

Fonction

Définit, ou réinitialise, la vitesse des prochains déplacements.

Synopsis

SPEED = valeur_de_la_vitesse

SPEED = variable_source

Description

- L'instruction *SPEED* détermine la vitesse du déplacement programmé par la prochaine instruction *MOVEA*, *MOVER*, *FORWARD* ou *REVERSE* rencontrée (l'action de cette instruction n'est donc pas immédiate).
- *valeur_de_la_vitesse* est une donnée, *variable_source* une variable, exprimées en *unit1/s*, au format flottant *F* ou double précision *D*.
- valeur_de_la_vitesse et variable_source doivent être strictement positives.
- La variable système relative à l'instruction *SPEED* est : *speed_move*.
  N.B. Au début de l'instruction *MOVEA*, *MOVER*, *FORWARD* ou *REVERSE* suivante, la valeur de *speed_move* est chargée dans *speed_prog*.

  Remarque : A la mise sous tension du variateur positionneur, speed_move = home_speed et speed_prog = home_speed.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*speed_move*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
SPEED = 1000
{ La valeur de la vitesse vaut 1000 unit1/s.
  N.B. La valeur 1000 peut à la fois être considérée comme présentant les
  formats E, F ou D. La valeur 1000.0 est considérée comme ayant le format
  flottant F ou le format double précision D.}
```

Voir aussi

SPEED_IM

speed_att

Fonction

Coefficient d'atténuation de la vitesse programmée.

Synopsis

speed_att = valeur

speed_att = variable_source

Description

- L'instruction *speed_att* s'applique aux variables système :
  speed_prog correspondant à la vitesse programmée et
  fspeed_prog correspondant à la vitesse à atteindre en fin de mouvement.

- *valeur* est une donnée, *variable_source* une variable, exprimées au format flottant *F* ou double précision *D*. Leur valeur est comprise entre :
  0 ⇒ l'atténuation de vitesse est maximale ⇔ le moteur est à l'arrêt.
  1 ⇒ il n'y a pas d'atténuation de vitesse.

- *speed_att* vaut 1 par défaut à la mise sous tension de l'appareil.

Exemples

La vitesse du déplacement est programmée à 1000 t/min dans les sous-programmes *PROG10* et *PROG11* à l'aide de l'instruction *SPEED_IM* [variable correspondante *speed_prog*].

- Exemple 1. Dans le sous_programme n°10, l'entrée analogique *in**a*** sera affectée à la fonction *atténuation de vitesse*. Le coefficient de l'atténuation est par exemple de 0.8, quand l'entrée analogique vaut ± 8 volts ⇔ coefficient = 80% ⇔ *speed_att* = 0.8 ⇔ vitesse mouvement = 0.8*1000 = 800 t/min.

```
10      %PROG10
11      unit1 = "t/min"
12      SPEED_IM = 1000
13       ; la variable speed_prog  vaut 1000
14      scale_ina = 0.1
15      a_ina = 2
16       { le numéro d'ordre 2 correspond à
17        l'affectation de l'entrée analogique
18        à speed_att.}
19      RETURN
20      %ENDPROG
```

- Exemple 2. Dans le sous-programme n°11, la vitesse du mouvement est portée à 300 t/min, si l'entrée logique *in0* est à 1 : *speed_att* = 0.3 ⇔ vitesse mouvement = 0.3*1000 = 300 t/min.

```
21      %PROG11
22      unit1 = "t/min"
23      SPEED_IM = 1000
24       ; la variable speed_prog  vaut 1000
25      IF in0 = 1 THEN speed_att = 0.3
26      { l'atténuation de vitesse sera portée à :
27      0.3 * speed_prog = 300 t/min si in0 vaut 1.}
28      ENDIF
29      RETURN
30      %ENDPROG
```

SPEED_IM

Fonction

Modifie de manière immédiate mais provisoire la vitesse du mouvement en cours.

Synopsis

SPEED_IM = valeur_de_la_vitesse

SPEED_IM = variable_source

Description

- L'instruction SPEED_IM modifie la vitesse de déplacement du mouvement en cours. Cette instruction est à prise d'effet immédiate.
- *valeur_de_la_vitesse* est une donnée, *variable_source* une variable, exprimées en *unit1/s*, au format flottant *F* ou double précision *D*.
- Valeur_de_la_vitesse et variable_source doivent être strictement positives.
- La variable système relative à l'instruction *SPEED_IM* est : *speed_prog.*

Attention ! Au début de l'instruction MOVEA, MOVER, HOME, FORWARD ou REVERSE suivante, la valeur de speed_prog sera écrasée par celle de speed_move ou de home_speed.

Remarque : A la mise sous tension du variateur positionneur, speed_move = home_speed et speed_prog = home_speed.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*speed_prog*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

```
SPEED_IM = 1000
{ La valeur de la vitesse vaut immédiatement 1000 unit1/s.
  N.B. La valeur 1000 peut à la fois être considérée comme présentant les
  formats E, F ou D. La valeur 1000.0 est considérée comme ayant le format
  flottant F ou le format double précision D.}
```

Voir aussi

SPEED

speed_value

Fonction

Redéfinit ou réinitialise la valeur de la consigne de vitesse en mode commande en vitesse.

Synopsis

speed_value = valeur

speed_value = variable_source

Description

- L'instruction speed_value redéfinit la valeur de la consigne de vitesse en mode commande en vitesse [drive_mode = 1]
- valeur est une donnée, variable_source une variable, exprimées en unit1/s au format flottant F ou double précision D.
- valeur et variable_source peuvent être positives ou négatives. Elles ne doivent pas dépasser la valeur speed_max en valeur absolue.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* : *[speed_value]*	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

speed_value est forcée à 0 lorsque l'on se trouve en mode commande en courant

Exemples

speed_value = -0.01
speed_value = uf3
speed_value = ina

SQR

Fonction

Racine carrée.

Synopsis

variable_destination = SQR(variable_source)

variable_destination = SQR(valeur)

Description

La valeur résultante est attribuée à la *variable_destination*.

variable_source et *valeur* ne doivent pas être inférieures à 0.

Propriétés

Si *valeur* est hors limites, une erreur sera générée lors de la compilation.
Si *variable_source* est hors limites, une erreur sera générée à l'exécution.

- Seule une valeur positive ou nulle est acceptée comme argument de l'instruction *SQR*.

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

uf1 = SQR(uf2)
uf2 = SQR(16) ; uf2 sera égal à 4.

#START

Fonction

Le corps du programme principal débute à l'adresse *#START*.

Synopsis

#START

Description

- *#START* s'utilise comme en-tête du corps de programme principal et se trouve donc en fin de zone déclarative.
- La variable système *userprog_option* = 1 permet de gérer un éventuel arrêt à l'adresse *#START* [*cf. PVD 3516 Chapitre 5.2*].

Exemple

```
1       %PROG0
2        ; Voici un exemple de programme principal.
3       #INIT
4       INTERRUPT2 = PROG40
5       PLC1 = PROG100
6       PLC2 = PROG200
7       DEFPLC1 = 1000
8       DEFPLC2 = 100
9       ; fin de zone déclarative.
10      #START ; début du corps de programme principal.
11          IF in1 = 1 THEN    PLC1 = START ; démarrage des programmes
12                             PLC2 = START ; automate de type 1 et 2.
13                     ELSE    GOSUB PROG10
14          ENDIF
15      …
16          GOSUB PROG20
17          GOSUB PROG30
18      …
19          IF in2 = 1 THEN    RESTART ; saut à l'adresse #START.
20          ENDIF
21      PROG_INIT
22      %ENDPROG
```

Propriétés

#START s'utilise exclusivement dans le programme principal *PROG0*.
L'instruction *RESTART* permet d'accéder directement à l'adresse *#START* depuis le programme principal, ou un quelconque autre sous-programme.

Voir aussi

RESTART

START

Fonction

Reprend l'exécution de mouvements suspendus.

Synopsis

START

Description

- L'instruction *START* permet la reprise d'exécution des mouvements précédemment suspendus par l'instruction *STOP*. Les mouvements reprennent leur cours, à l'endroit précis où l'axe s'était arrêté.
- La variable système associée à l'instruction *START* est : *move_en = 1*.
- Cette instruction n'a aucun effet direct sur l'exécution des programmes.

Exemple

Le sous-programme *PROG10* lance un mouvement continu dans le sens positif. Ce mouvement est suspendu si l'entrée logique *in0* passe à 1. La reprise du mouvement interrompu est rendue possible si l'entrée logique *in1* passe à 1.

```
10   %PROG10
11    ; sous-programme
12    ;
13   ACCEL = 1000
14   SPEED = 500
15   FORWARD
16   …
17   …
18   …
19   RETURN
20   %ENDPROG
```

```
21   %PROG100
22    ; programme automate
23   IF in1 = 1 THEN START
24   ENDIF
25   IF in0 = 1 THEN STOP
26   ENDIF
27   …
28   …
29   END
30   %ENDPROG
```

Voir aussi

STOP
move_en

STC

Fonction

Modifie une variable d'un autre variateur positionneur [*fonction A*].
Modifie une variable d'un autre abonné *CAN* [*fonction B*].
Cette fonction n'est pas disponible dans le cas de l'utilisation d'un DIGIVEX Motion Profibus.

Synopsis

STC(id, var1, var2) ; [*fonction A*]
STC(id, var1, valeur1) ; [*fonction A*]
STC(id, index, sous_index, var) ; [*fonction B*]
STC(id, index, sous_index, valeur, type) ; [*fonction B*]

Description

L'instruction *STC* permet d'accéder à l'espace mémoire d'un variateur positionneur, ou d'un autre dispositif numérique connecté au bus *CAN*, à partir d'un variateur positionneur donné, afin d'en modifier une variable.

- [*fonction A*]

id est un entier compris entre [1 et 127] désignant le numéro d'abonné *CAN*. Sa valeur est affectée à la variable *can_id*.
var1 est le nom de la *variable_destination* (à modifier).
var2 est le nom de la *variable_source* (à lire).
valeur1 est la valeur attribuée à la *variable_destination var1*.

- [*fonction B*]

index est l'index *CANopen* de la *variable_destination*. Sa valeur est affectée à la variable *can_index*.
sous_index est le sous-index *CANopen* de la *variable_destination*. Sa valeur est affectée à la variable *can_subindex*.
var est le nom de la *variable_source*.
valeur est la valeur attribuée à la *variable_destination* indexée.
type est un entier compris entre 1 et 6 désignant le format assigné à *valeur*.

format *valeur* :	*type*
B	1
E	2
F	3
D	4
C1	5
C2	6

Pour connaître les numéro d'*index* et de *sous_index CAN*, se reporter au document intitulé :
[PVD 3527 *DIGIVEX Motion*, *Répertoire des variables*]

Propriétés

Attention ! L'instruction *STC* utilise la variable *can_id*. Elle peut donc modifier une programmation précédente effectuée à l'aide de *ADR.*

Remarque

Utiliser de préférence avec cette fonction les adresses des canaux SDO 1,2 ou 3 (voir notice PVD 3516 chapitre 5.2.5.1).

Exemples

- [*fonction A*]

```
STC(3,trackerror_max,uf1)
STC(4,trackerror_max,0.01)
```

- [*fonction B*]

```
STC(4, $2840, 1, ub1)
STC(4, $3000, 10, 1.56E+5,3)
```

Voir aussi

ENQ

STOP

Fonction

Suspend l'exécution des mouvements en cours sans action directe sur l'exécution des programmes.

Synopsis

STOP

Description

- L'instruction STOP permet de suspendre un mouvement en cours ou d'interdire tout nouveau déplacement. S'il y avait un déplacement en cours, l'axe décélère [accel_prog] puis s'arrête. La position obtenue en fin de décélération reste asservie et la consigne de position à atteindre [posa] n'est pas modifiée. Il n'y a aucune action directe sur l'exécution des programmes.
- La variable système associée à l'instruction *STOP* est : *move_en = 0*.

Exemple

Le sous-programme *PROG10* lance un mouvement continu dans le sens positif. Ce mouvement est suspendu si l'entrée logique *in0* passe à 1. La reprise du mouvement interrompu est rendue possible si l'entrée logique *in1* passe à 1.

```
%PROG10
 ; sous-programme
 ;
ACCEL = 1000
SPEED = 500
FORWARD
…
…
…
RETURN
%ENDPROG
```

```
%PROG100
 ; programme automate
IF in0 = 1 THEN STOP
ENDIF
IF in1=1 AND in0 = 0 THEN START
ENDIF
…
…
END
%ENDPROG
```

Voir aussi

START
move_en

SYNCHRO

Fonction

Démarre le mode de recopie d'axe.

Synopsis

SYNCHRO

Description

- L'instruction *SYNCHRO* active le mode de recopie d'axe. L'axe asservi (*axe 1*) devient l'esclave de l'axe maître (*axe 2*) en se synchronisant dans un certain rapport *KSYNC*.
- Le rapport de synchronisation *KSYNC* doit être défini préalablement à l'emploi de l'instruction *SYNCHRO*.
- La position réelle de l'axe 2 n'est aucunement corrigée d'éventuels décalages temporels, lorsque *SYNCHRO* est désignée par le pointeur opérationnel ; $\Rightarrow$ si cette fonction est désirée, utiliser l'instruction *SYNCHRO_START.*
- La validation du mode de recopie d'axe correspond à une mise à 1 de la variable système : *synchro_en* = 1.
- L'instruction *ABORT* permet d'arrêter le mode de recopie d'axe.

Exemple

```
10  %PROG17
11  ; sous-programme d'activation du mode de recopie d'axe.
12  ACCEL = 1000
13  KSYNC = 0,5
14  ; l'axe 2 recopiera l'axe 1 avec un rapport de synchronisation égal à 0,5.
15  ; lorsque l'axe 2 se déplace de 2 unit2, l'axe 1 effectue un déplacement de 1 unit1.
16  SYNCHRO
17  ; le mode de fonctionnement en  recopie d'axe est validé.
18  …
19  …
20  …
21  RETURN
22  %ENDPROG
```

Voir aussi

SYNCHRO_START
KSYNC

SYNCHRO_START

Fonction

Démarre le mode de recopie d'axe.

Synopsis

SYNCHRO_START

Description

- L'instruction *SYNCHRO_START* s'utilise exclusivement dans le sous-programme prioritaire associé à l'entrée interruptive *in0*.

- Dès que cette instruction est rencontrée, le système considère la position de l'axe 2 et la corrige des éventuels décalages temporels dus au temps de propagation du capteur et de l'entrée logique *in0* (*cf. chapitre 2.7, datation de l'événement interruptif, emploi de in0*).

- L'instruction *SYNCHRO_START* active le mode de recopie d'axe. L'axe asservi (*axe 1*) devient l'esclave de l'axe maître (*axe 2*) en se synchronisant dans un certain rapport *KSYNC*.

- Le rapport de synchronisation *KSYNC* doit être défini préalablement à l'emploi de l'instruction *SYNCHRO_START*.

- Valider le mode de recopie d'axe correspond à mettre à 1 la variable système : *synchro_en* = 1.

- L'instruction *ABORT* permet d'arrêter le mode de recopie d'axe.

Exemple

```
%PROG10
 ; sous programme utilisateur
INTERRUPT0 = PROG11
ACCEL = 1000
SPEED = 500
 ;
 ;
KSYNC = 1 ; choix du rapport de recopie
IT_ON = IN0
{ activation de l'entrée  interruptive in0.}
WAIT_UNTIL in2 = 0
 ; attente demande fin de synchronisation
ABORT
; arrêt freiné
WAIT_UNTIL moving = 0
 ; attente fin de mouvement théorique
…
RETURN
%ENDPROG
```

```
%PROG11
 ; sous-programme prioritaire déclenché
 ; par in0.
 ;
IT_OFF = IN0
 ; désactivation de l'entrée interruptive in0.
 ; prise d'effet de la position réelle de l'axe
 ; corrigée des décalages temporels.
 ;
SYNCHRO_START
{validation du mode de synchronisation
 perpétuelle}
 ;
END
%ENDPROG
```

Voir aussi

SYNCHRO
KSYNC

TAN

Fonction

Tangente.

Synopsis

variable_destination = TAN(variable_source)

variable_destination = TAN(valeur)

Description

variable_source ou *valeur sont* exprimées en radians.
La valeur résultante est attribuée à la *variable_destination*.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* :	format *variable_source* :
F	F
F	D (avec perte de la précision)
D	D
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemple

uf1 = TAN(uf2)

target

Fonction

Redéfinit, ou réinitialise, la valeur de la fenêtre d'arrêt.

Synopsis

target = valeur

target = variable_source

Description

- L'instruction *target* redéfinit la valeur de la fenêtre d'arrêt.
- *valeur* est une donnée, *variable_source* une variable, exprimées en *unit1* au format flottant *F* ou double précision *D*.
- Valeur et variable_source doivent être strictement positives.

Propriétés

- *posa* étant la consigne de position absolue (cote à atteindre), l'axe asservi (axe 1) est déclaré à l'arrêt à l'issue d'un déplacement [variable *in_position = 1*], quand son mouvement théorique n'a plus cours [variable *moving = 0*] et que sa position réelle a convergé dans l'intervalle [*posa - target*, *posa + target*].

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*target*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

target = 0.01
target = uf3

THEN

Fonction *IF / THEN / ELSE / ENDIF*
Se reporter à la description de *IF*.

; texte {texte}

Fonction

Définit un commentaire.

Synopsis

; commentaire écrit sur une ligne
{ commentaire et explications
nécessitant plusieurs lignes.**}**

Description

Un commentaire est détecté par le compilateur, dès que le caractère point virgule « **;** », ou ouverture de parenthèse « **{** », figure sur une ligne de programme.

Exemple

```
1       %PROG0
2        ; ceci est un commentaire pour désigner l'intitulé du programme principal.
3       #INIT
4        PLC1 = PROG100 ; ce commentaire fait suite à une déclaration.
5
6       #START
7           { les commentaires sont utiles à la compréhension :
8           - de l'aspect fonctionnel ( traduction du cahier des charges )
9           - de la méthode de programmation employée.}
10
11       IF in7 = 1 AND in2 = 1 THEN
12                  GOSUB PROG10        ; déroutement normal
13                                      ; lancement de production
14
15                  ELSE
16                  GOSUB PROG20        ; déroutement d'urgence
17                  { une condition de sécurité n'est pas satisfaite
18                    in7 = 0 correspond à un capot ouvert
19                    in2 = 0 correspond à manque d'air comprimé.}
20
21      ENDIF
22      RESTART
23      %ENDPROG
```

Propriétés

Les commentaires n'influencent pas le temps de cycle des programmes.

timer*n*

Fonction

Initialise un *timer* ou compte à rebours exprimé en ms.

Synopsis

timer***n*** = valeur_tempo

timer***n*** = variable_source

Description

- L'instruction *timer**n*** démarre un compte à rebours qui est décrémenté automatiquement toutes les millisecondes jusqu'à la valeur zéro. La valeur de départ est fixée par *valeur_tempo*, un entier positif ou *variable_source*, une variable entière contenant le nombre de millisecondes souhaitées avant l'expiration du *timer.*

- Quatre *timer* sont disponibles ; ils sont référencés par ***n***, un entier compris entre 0 et 3.

Propriétés

- Le test d'un *timer* fait appel à la variable système qui porte son nom : par exemple, le timer n°3 est associé à la variable système *timer3* (variable entière).

- Les variables système *timer**n*** sont toujours à l'image d'un nombre positif ou nul. Si *valeur_tempo* ou *variable_source* présente une valeur négative, le *timer* sera fixé à zéro.

- Un *timer* peut à tout moment être réinitialisé à une nouvelle *valeur_tempo* ou *variable_source* (son expiration première ayant ou non, eu lieu).

Exemple

```
10      %PROG10
11      ; exemple d'utilisation d'un timer
12      ADR = 1
13      PAGE = 3
14      LINE = 2
15      COL = 4
16      timer3 = 500
17      ; le timer n° 3 est initialisé à 500 millisecondes.
18      MOVER = 1000
19      WAIT_UNTIL move_end = 1
20      IF timer3 = 0 THEN PRINT(″Temps de déplacement > à 0.5 s !″)
21      ENDIF
22      …
23      RETURN
24      %ENDPROG
```

torque_cmd

Fonction

Commande la mise sous couple, ou la mise à couple nul, du moteur.

Synopsis

torque_cmd = 0

torque_cmd = 1

torque_cmd = variable_source

Description

- *torque_cmd* = ***0*** demande la mise à couple nul du moteur.
- *torque_cmd* = ***1*** demande la mise sous couple du moteur (si le système variateur positionneur ne s'y oppose pas).

Propriétés

- Si variable_source est hors limites, une erreur sera générée à l'exécution.
- Lorsque la variable *torque_cmd = 0*, la consigne de position absolue *posa* est égale à la position réelle de l'axe asservi (axe 1). L'erreur de poursuite *tracking_error* est donc nulle.
- La transition d'un état bas vers un état haut (*torque_cmd* = 0 → 1) s'effectue sans à-coup sur l'axe qui passe alors en mode asservi.

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*torque_cmd*]	format *variable_source* :
B	B
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Voir aussi

PVD 3516, chapitre 5.2, autorisation couple à la mise sous tension.

torque_value

Fonction

Redéfinit, ou réinitialise la valeur de la consigne de couple en mode commande en courant.

Synopsis

torque_value = valeur

torque_value = variable_source

Description

- L'instruction torque_value redéfinit la valeur de la consigne de couple en mode commande en courant [drive_mode = 2].
- *valeur* est une donnée, *variable_source* une variable, exprimées en *N.m* au format flottant *F* ou double précision *D*.
- Valeur et variable_source peuvent être positives ou négatives. Elle ne doivent pas dépasser la valeur torque_max en valeur absolue.

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [torque_value]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

torque_value = -0.01
torque_value = uf3
torque_value = ina

trackerror_max

Fonction

Redéfinit, ou réinitialise, la valeur maximale de l'erreur de poursuite autorisée.

Synopsis

trackerror_max = valeur

trackerror_max = variable_source

Description

- L'instruction *trackerror_max* redéfinit la valeur maximale de l'erreur de poursuite autorisée.
- *valeur* est une donnée, *variable_source* une variable, exprimées en *unit1* au format flottant *F* ou double précision *D*.
- valeur et variable_source doivent être strictement positives.
- Si au cours d'un déplacement, l'erreur de poursuite *tracking_error* sort de l'intervalle [-*trackerror_max*, +*trackerror_max*], l'axe passe en *Défaut erreur de poursuite* et s'arrête [variable *status_number* = 28].

Propriétés

Les formats attribués aux variables sont les suivants :

format *variable_destination* : [*trackerror_max*]	format *variable_source* :
F	F
F	D (avec perte de la précision)
Toute autre combinaison de formats entraîne une erreur à la compilation.	

Exemples

trackerror_max = 1000

trackerror_max = uf1

ub*n*, uc*n*, ud*n*, uf*n*, ui*n*, ub[ui*n*], uc[ui*n*], ud[ui*n*], …

Fonction

Variables utilisateur pouvant servir à :

- structurer la programmation
- paramétrer le programme
- effectuer des calculs
- mémoriser des états, des points de passage

Description

nom	description	format	ui*n*
ub0 à ub127 ou ub[ui*n*]	variables utilisateur de type binaire (valeur 0 ou 1)	B	ui*n* = ui0 à ui127 $0 \leq$ ui*n* ≤ 127
uc0 à uc7 ou uc[ui*n*]	variables utilisateur de type chaîne de caractères (16 caractères maxi)	C1	ui*n* = ui0 à ui127 $0 \leq$ ui*n* ≤ 7
ud0 à ud63 ou ud[ui*n*]	variables utilisateur de type flottant double précision (réel 40 bits)	D	ui*n* = ui0 à ui127 $0 \leq$ ui*n* ≤ 63
uf0 à uf191 ou uf[ui*n*]	variables utilisateur de type flottant simple précision (réel 32 bits)	F	ui*n* = ui0 à ui127 $0 \leq$ ui*n* ≤ 191
ui0 à ui127 ou ui[ui*n*]	variables utilisateur de type entier (entier 32 bits)	E	ui*n* = ui0 à ui127 $0 \leq$ ui*n* ≤ 127

Les formes ub[ ], uc[ ], ud[ ], uf[ ] et ui[ ] ne représentent qu'une facilité d'écriture pour accéder aux variables ub*n*, uc*n*, ud*n*, uf*n* et ub*n* de manière indexée (tableaux à une dimension) :

si ui*m* = n, ub*n* et ub[ui*m*] représentent la même variable
si ui*m* = n, uc*n* et uc[ui*m*] représentent la même variable
si ui*m* = n, ud*n* et ud[ui*m*] représentent la même variable
si ui*m* = n, uf*n* et uf[ui*m*] représentent la même variable
si ui*m* = n, ui*n* et ui[ui*m*] représentent la même variable

Exemple

```
15      %PROG10
16      ; sous-programme de réinitialisation
17      ; (remise à 0 de uf0 à uf10)
18      FOR ui0 = 0 TO 10
19          uf[ui0] = 0
20      NEXT
21      RETURN
22      %ENDPROG
```

%VDM

Fonction

Définit le nom et le chemin d'accès au fichier contenant les noms "en clair" des variables utilisateur et système [cf PVD 3516 chapitre 7, éditeur de programmes].

Synopsis

%VDM = C:\chemin\nom_de_fichier.vdm

%VDM = ..\nom_de_fichier.vdm

%VDM = nom_de_fichier.vdm

Remarque : Il n'est pas nécessaire de spécifier le chemin complet d'accès au fichier .vdm, si celui-ci se trouve dans le même répertoire que le fichier .bdm contenant le programme utilisateur BASIC_DM.

Description

Afin de rendre la programmation plus conviviale, il est possible de remplacer, dans les programmes BASIC_DM, les noms de variables utilisateur et système par des noms plus "parlant". Par exemple, pour incrémenter un nombre de pièces, on pourra programmer : nb_pieces = nb_pieces + 1 au lieu de ui1 = ui1 + 1.

La directive %VDM permet d'indiquer au compilateur les nouveaux noms "en clair" des variables utilisateur et système. Ces déclarations sont enregistrées dans un fichier portant l'extension .vdm qui a été créé par l'éditeur de programmes BASIC_DM [cf PVD 3516 chapitre 5, éditeur de programmes].

Propriétés

%VDM est une directive facultative, exclusivement destinée au compilateur. Elle figurera le plus souvent dans la zone déclarative du programme principal (entre les adresses #INIT et #START).

Rien n'interdit l'utilisation d'un même fichier .vdm avec plusieurs programmes utilisateur BASIC_DM différents.

Exemple

```
1   %PROG0
2   #INIT
3   %VDM = definition.vdm      ; avec dans ce fichier : nb_pieces = ui1
4   PLC1 = NONE                ;                        arret_axe = move_end
5   PLC2 = NONE
6   nb_pieces = 0
7   #START
8   ...
9   WHILE nb_pieces < 10 DO
10  ...
11  nb_pieces = nb_pieces + 1
12  MOVER = 10
13  WAIT_UNTIL arret_axe = 1
14  WEND
15  RESTART
16  %ENDPROG
```

WAIT

Fonction

Définit une temporisation exprimée en ms

Synopsis

WAIT = valeur_tempo

WAIT = variable_source

Description

- L'instruction *WAIT* lance une temporisation. La valeur de cette temporisation est fixée par *valeur_tempo*, un entier positif ou *variable_source*, une variable entière contenant le nombre de millisecondes souhaitées.

Propriétés

- Si *valeur_tempo* ou *variable_source* présente une valeur négative, la temporisation sera fixée à zéro.

- Le pointeur opérationnel stationne sur l'instruction *WAIT* jusqu'à ce que la temporisation soit écoulée (ceci constitue la différence de comportement majeure comparée à l'instruction *timer**n*** qui lance un compte à rebours, le programme poursuivant son exécution en séquence).

Exemple

```
10      %PROG10
11      ; exemple d'utilisation d'une temporisation.
12      IF in1 = 1   THEN
13      ; démarrage retardé de 5 secondes.
14                      WAIT = 5000
15      ; le pointeur opérationnel patiente 5 secondes avant de poursuivre.
16                      MOVER = 1000
17      ENDIF
18      …
19      …
20      RETURN
21      %ENDPROG
```

Voir aussi

*timer**n***, WAIT_UNTIL

WAIT_UNTIL

Fonction

Définit une attente conditionnelle.

Synopsis

WAIT_UNTIL *condition*

Description

- L'instruction *WAIT_UNTIL* autorise la poursuite du déroulement d'un programme, lorsque la *condition* spécifiée est satisfaite. Dans le cas contraire, le pointeur opérationnel stationne à cette instruction.
- « *condition* » est une assertion logique. Lorsque la proposition logique est vraie, la condition est vérifiée.
- Exemples de « *condition* » pouvant figurer à la suite de *WAIT_UNTIL :*

WAIT_UNTIL in3 = 1	Attendre que l'entrée logique in3 soit à l'état haut.
WAIT_UNTIL pos2 < 2000	Attendre que la position réelle de l'axe mesuré soit inférieure à 2000 unit2. Remarque : Il aurait aussi été possible d'utiliser un flag : FLAG(0,pos2,2000,1E20) ... WAIT_UNTIL flag0 = 0
WAIT_UNTIL timer3 = 0	Attendre que le timer n°3 soit arrivé à expiration.
WAIT_UNTIL drive_ok = 1	Attendre que la puissance soit présente (drive_ok = 1 si la puissance est présente et si l'axe ne présente pas de défaut majeur, drive_ok est à 0 dans le cas contraire. Lorsque drive_ok = 0, l'axe est forcé à couple nul, la consigne de position absolue *posa* est égale à la position réelle de l'axe asservi *pos1*, l'erreur de poursuite *tracking_error* est donc nulle).
WAIT_UNTIL home_made = 1	Attendre que la prise d'origine ait été réalisée avec succès (home_made = 1 indique que la prise d'origine a été faite et est valide. home_made est initialisé à 0 à la mise sous tension de l'axe et est réinitialisé à 0 en début de prise d'origine). Remarque : Pour ne pas rester bloqué sur cette condition si la séquence de prise d'origine a été interrompue, il est préférable de programmer : HOME WAIT_UNTIL homing = 0 IF home_made = 0 THEN GOSUB PROG10 ; traitement d'erreur ENDIF
WAIT_UNTIL homing = 0	Attendre que l'axe ne soit plus en prise d'origine (prise d'origine réalisée avec succès ou interrompue). Remarque : homing est mis à 1 en début de prise d'origine et repasse à l'état 0, en fin de prise d'origine, après arrêt complet de l'axe.

WAIT_UNTIL moving = 0	Attendre que le mouvement théorique en cours (consigne au niveau du générateur de trajectoire) soit terminé (moving est mis à 1 lorsqu'une instruction de type MOVEA, MOVER, FORWARD, REVERSE, INDEX, SYNCHRO ou SYNCHRO_START est en cours d'exécution, moving = 0 lorsqu'il n'y a pas de synchro maître esclave en cours et que le générateur de trajectoire ne traite plus aucune demande de déplacement). Remarque : remplir cette condition ne signifie pas que l'axe est à l'arrêt (celui-ci peut être en train de résorber son erreur de poursuite).
WAIT_UNTIL move_end = 1	Attendre que le mouvement théorique élémentaire en cours (consigne au niveau du générateur de trajectoire) soit terminé (move_end = 1 lorsque le générateur de trajectoire ne traite plus aucune demande de déplacement, move_end est mis à 0 lorsqu'une instruction de type MOVEA, MOVER, FORWARD, REVERSE, INDEX, SYNCHRO ou SYNCHRO_START est en cours d'exécution) Remarque : remplir cette condition ne signifie pas que l'axe est à l'arrêt (celui-ci peut être en train de résorber son erreur de poursuite). On utilisera l'instruction WAIT_UNTIL move_end = 1 plutôt que WAIT_UNTIL moving = 0, lorsque l'on voudra s'assurer qu'un mouvement relatif programmé en plus d'une synchro maître/esclave est terminé.
WAIT_UNTIL in_position = 1	Attendre que le mouvement réel soit terminé → axe à l'arrêt : mouvement en cours terminé (moving = 0) et erreur de poursuite [tracking_error] inférieure à la fenêtre d'arrêt [target] Remarque : remplir cette condition ne signifie pas que le but est atteint (axe à l'arrêt et arrivé à destination), le mouvement ayant pu être interrompu (par exemple suite à la programmation de ABORT). Pour être sûr que l'axe est arrivé à destination, il faut aussi vérifier que (position à atteindre actuelle) posa = position destination programmée initialement
WAIT_UNTIL move_abort = 0	Attendre l'arrêt de l'axe, suite à programmation de l'instruction ABORT (la prise en compte de l'instruction ABORT provoque la mise à 1 de move_abort qui ne repasse à 0 qu'après arrêt complet de l'axe).
WAIT_UNTIL synchro_ok = 1	Attendre que les axes en synchro maître/esclave soient en synchronisme parfait (position et vitesse). Il est possible d'agir sur synchro_ok en ajustant les fenêtres synchro_postarget et synchro_speedtarget. synchro_postarget permet de modifier le seuil de mise à 1 d'une variable système indiquant qu'il y a synchronisme en position. synchro_speedtarget permet de modifier le seuil de mise à 1 d'une variable système indiquant qu'il y a synchronisme en vitesse synchro_ok ne passe à 1 que si l'on se trouve en synchro (synchro_on =1) et que si les 2 variables système citées précédemment sont toutes les deux à 1.

Exemple

```
10    %PROG10
11    ; exemple d'utilisation de la structure WAIT_UNTIL.
12    ; appui sur bouton poussoir « marche » pour démarrer.
13    WAIT_UNTIL in1 = 1
14    HOME
15    WAIT_UNTIL home_made = 1 ; attente de fin de prise d'origine.
16    MOVEA = 1000
17    WAIT_UNTIL in_position = 1 ; attente d'arrivée à destination.
18    …
19    …
20    …
21    …
22    RETURN
23    %ENDPROG
```

Voir aussi

*timer**n***, WAIT

WEND

Fonction *WHILE / DO / WEND*
Se reporter à la description de *WHILE*.

WHILE…DO…WEND

Fonction

Définit une structure de répétition tant que la condition spécifiée reste vraie.

Synopsis

WHILE condition spécifiée DO
action(s) à exécuter
WEND

Description

- L'instruction *WHILE…DO…WEND* s'utilise pour programmer une répétition du type :

Tant que la *condition spécifiée* au test d'entrée est vérifiée,
Exécuter *telle action*
Fin.

- Si la *condition spécifiée* reste vérifiée, le pointeur opérationnel se place directement sur Exécuter [*DO*]. Le nombre de passages dans la boucle Tant que…Fin est indéterminé.

- Si la *condition spécifiée* n'est pas vérifiée, l'action n'est pas exécutée ; le pointeur opérationnel se place directement sur Fin [*WEND*], le programme poursuivant son exécution en séquence.

Exemple

```
10      %PROG10
11       ; sous_programme d'illustration de WHILE…DO…WEND
12       ;
13      SPEED = 100
14      WHILE in4 = 1 DO
15            { une succession de déplacements relatifs de 50 unit1 s'accomplit à la
16              vitesse de 100 unit1/s  tant que l'entrée logique in4 reste à l'état 1.}
17            MOVER = 50
18            WAIT_UNTIL move_end = 1
19      WEND
20      …
21      …
22      …
23      RETURN
24      %ENDPROG
```

4. MISE EN ŒUVRE

4.1 Ecriture d'un programme

Un éditeur de programme est fourni avec le logiciel Parvex Motion Explorer (voir notice PVD 3516).

4.2 Exemples

Divers exemples sont fournis dans les répertoires :

C:\Program Files\Parvex\Pmex.xx\App_Parvex\Exemples\ (commentaires en français)
C:\ Program Files\Parvex\Pmex.xx\App_Parvex \Samples\ (commentaires en anglais)

Ces programmes sont présentés à titre purement didactique. PARVEX ne pourra en aucun cas être tenu responsable d'éventuels problèmes liés à l'utilisation qui en sera faite.

4.3 Compilation

Un compilateur est intégré à l'éditeur de programme.

Il permet :

- d'effectuer le contrôle syntaxique des programmes écrits en Basic_DM.
- de générer un fichier objet (.odm)

4.4 Chargement

Un outil de chargement de programme est intégré à l'éditeur de programmes. Il permet d'effectuer le chargement d'un fichier objet (.odm) de la mémoire de masse du PC vers la mémoire FLASH_DM du variateur positionneur (mémoire de type flash_eprom).

Si on utilise PME Tool Kit (voir notice PVD 3528), il est possible d'intégrer cette opération à un applicatif client sur PC.

Seuls les fichiers objet (.odm) peuvent être chargés dans un DIGIVEX Motion.

Ces fichiers intégrent :

- le code exécutable
- le code source Basic_DM (si l'option "*Inclure les fichiers sources*" a été sélectionnée). Remarque : L'intérêt principal d'inclure le code source réside dans le fait, qu'une fois sauvegardé dans le variateur, le programme écrit en langage Basic_DM ne risque pas d'être perdu : on pourra donc modifier facilement l'application, si besoin est, sans avoir à faire appel à une sauvegarde extérieure. La maintenance sera donc facilitée.

4.5 Exécution

A la mise sous tension du variateur, le code exécutable contenu dans la mémoire FLASH_DM est transféré dans une mémoire vive (PROG_DM). C'est là qu'il sera exécuté.

Une commande spéciale permet d'effectuer à la demande le transfert du code de FLASH_DM vers PROG_DM. Cette opération devra être réalisée à la fin du chargement ("Appliquer les programmes"), si l'on désire exécuter tout de suite le programme que l'on vient de charger (dans le cas contraire c'est le programme présent à la mise sous tension qui continuera de s'exécuter).

Si on utilise PME Tool Kit (voir notice PVD 3528), il est possible d'intégrer cette opération à un applicatif client sur PC.

L'exécution des programmes s'effectue dès la mise sous tension du variateur, à condition que la variable *exec_en* soit à 1 (départ de l'exécution à l'adresse #INIT de PROG0).

exec_en = 0 provoque l'arrêt de l'exécution des programmes utilisateur (programme principal et programmes automates) et l'arrêt des mouvements (ABORT). Lorsque *exec_en* repasse à 1, l'exécution redémarre à son début (adresse *#INIT* du programme principal *PROG0*).

4.6 Contrôle d'exécution (Debug)

Un outil de contrôle d'exécution est fourni avec le logiciel Parvex Motion Explorer (voir notice PVD 3516). Il sera utilisé principalement en phase de conception et de test afin de vérifier le bon déroulement des programmes.

5. MODES DE FONCTIONNEMENT

Ce chapitre décrit les modes de fonctionnement possibles d'un variateur positionneur DIGIVEX Motion et établit des liens entre les instructions de programmation Basic_DM et ces modes de fonctionnement.

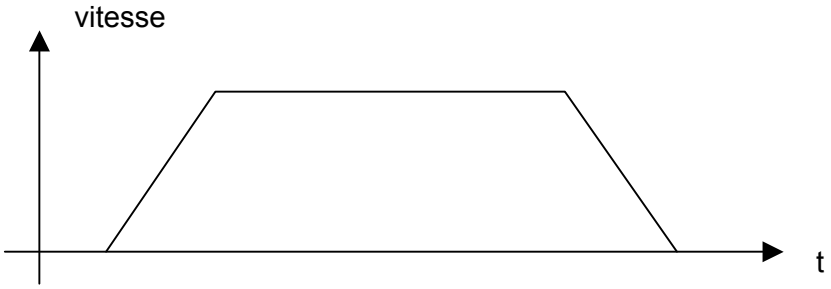
5.1 Commande en position

5.1.1 Description

La fonction commande en position est réalisée en mettant à 0 le paramètre *drive_mode*.
C'est le mode de fonctionnement le plus utilisé.

5.1.2 Déplacement élémentaires

Le générateur de trajectoire du variateur positionneur gère des rampes d'accélération / décélération de forme linéaire (graphe de vitesse en forme de trapèze) : instruction *SPEED* pour programmer la vitesse, instruction *ACCEL* pour programmer l'accélération et la décélération, ou utilisation du bus CAN ou Profibus (variables *speed_move*, *speed_prog*, *accel_move* et *accel_prog*). Il est possible d'enchaîner des mouvements et de changer vitesse et/ou accélération en cours de déplacement.



5.1.3 Déplacement absolu

Cette fonction est réalisée en utilisant l'instruction *MOVEA* ou par l'intermédiaire du bus CAN ou Profibus en agissant directement sur la variable *posa* (position absolue à atteindre).

5.1.4 Déplacement relatif

Cette fonction est réalisée en utilisant l'instruction *MOVER* ou par l'intermédiaire du bus CAN ou Profibus en agissant directement sur la variable *posr* (déplacement relatif à réaliser).

5.1.5 Déplacement continu

Cette fonction est réalisée en utilisant les instructions *FORWARD* (déplacement sens +) ou *REVERSE* (déplacement sens -).
L'instruction *FORWARD* est équivalente à la programmation de *MOVEA* = +∞, l'instruction *REVERSE* à la programmation de *MOVEA* = -∞.

5.1.6 Déplacement en mode manuel

Ce cas de figure correspond tout simplement à une combinaison de déplacements continus. La gestion de ce mode de fonctionnement pourra être réalisée par le programme automate.

5.1.7 Déplacement en mode JOG

Ce cas de figure correspond tout simplement à une combinaison de déplacement relatifs. La gestion de ce mode de fonctionnement pourra être réalisée par le programme automate.

5.1.8 Déplacement en stop cote

Un déplacement en stop cote consiste en un déplacement avec arrêt programmé après prise en compte d'un capteur.
Il est nécessaire d'utiliser une entrée avec datation de l'événement si l'on veut que la position d'arrêt ne dépende pas de la vitesse de déplacement lors de la prise en compte du capteur (Le capteur doit donc être raccordé sur l'entrée interruptive in0).
Le déplacement débute suite à la programmation des instructions *FORWARD* ou *REVERSE*. Un déplacement relatif par rapport à la position correspondant à la prise en compte du capteur est défini en programmant l'instruction *INDEX* dans le sous-programme prioritaire gérant l'entrée logique *in0* (programme *INTERRUPT0*).

5.1.9 Prise d'origine avec came d'origine

Cette fonction est réalisée en utilisant l'instruction *HOME* ou par l'intermédiaire du bus CAN ou Profibus en agissant directement sur la variable *home_cmd*.
Le paramètre *switch_en* doit être mis à 1. La coïncidence zéro resolver ou top0 codeur et came d'origine est utilisée pour référencer l'origine des mesures.

5.1.10 Prise d'origine sans came d'origine

Cette fonction est réalisée en utilisant l'instruction *HOME* ou par l'intermédiaire du bus CAN ou Profibus en agissant directement sur la variable *home_cmd*.
Le paramètre *switch_en* doit être mis à 0. Seul le zéro resolver ou top0 codeur est utilisé pour référencer l'origine des mesures.

5.1.11 Prise d'origine simplifiée

On ne désire prendre en compte seulement qu'un capteur pour référencer l'origine des mesures. Ce cas de figure correspond tout simplement à un stop cote (instructions *FORWARD* ou *REVERSE* associées à l'instruction *INDEX*) suivi d'une remise à 0 ou d'une réinitialisation du compteur de position, en fin de déplacement (instruction *DEFPOS1*).

5.1.12 Remise à zéro ou réinitialisation du compteur de position

Il est possible de remettre à 0 ou de réinitialiser le compteur de position sans qu'il y ait de déplacement physique. Cette fonction est réalisée en utilisant l'instruction *DEFPOS1* (associée aux variables *val_pos1* et *def_pos1*) pour l'axe asservi ou *DEFPOS2* (associée aux variables *val_pos2* et *def_pos2*) pour l'axe mesuré.

5.2 Commande en vitesse

5.2.1 Description

La fonction commande en vitesse est réalisée en mettant à 1 le paramètre *drive_mode*.

La consigne de vitesse est donnée par la variable signée *speed_value* qui peut être modifiée :
- par programme
- par bus CAN ou Profibus
- par affectation de l'entrée analogique à la fonction commande en vitesse (*a_ina* = 3)

5.2.2 Caractéristiques de la commande en vitesse

- L'axe n'est plus asservi en position
- L'erreur de poursuite (*tracking_error*) est forcée à 0
- La consigne de position à atteindre (*posa*) est forcée à $\pm\infty$ suivant le signe de *speed_value*
- La gestion des rampes d'accélération et de décélération reste active (prise en compte de la variable *accel_prog* ou de l'instruction *ACCEL_IM*).

5.2.3 Passage du mode commande en position au mode commande en vitesse

- La variable *speed_value* est initialisée avec la vitesse théorique de l'axe au moment du changement (*slave_Vn*).
- Si un mouvement était en cours et si *speed_value* n'est pas modifié, l'axe continue de se déplacer dans le même sens et à la même vitesse.

5.2.4 Passage du mode commande en vitesse au mode commande en position

Deux options sont possibles :
- *speed_option* = 1 :
  Il y a arrêt du mouvement comme si l'on avait programmé l'instruction *ABORT*. L'axe décélère (*accel_prog*) puis s'arrête. La position obtenue en fin de décélération reste asservie et la consigne de position à atteindre (*posa*) devient la position d'arrêt.
- *speed_option* = 0 :
  Il y a continuation du mouvement avec la vitesse courante (*speed_prog* = *slave_Vn*). L'axe part à l'infini et il n'y a pas de discontinuité sur la vitesse lors du changement de mode. Le mouvement final est équivalent à la commande *FORWARD* ou *REVERSE*.

5.3 Commande avec limitation de courant

Utilisable dans les modes "Commande en vitesse" et "Commande en position", la limitation de courant s'effectue à l'aide de la variable *ki_red* (précision effective : 2% à 5% sur le courant, 5% à 10% sur le couple).
Utilisations : travail sur butée mécanique, vissage avec contrôle de serrage, ...

5.4 Commande en couple

5.4.1 Description

La fonction commande en couple est réalisée en mettant à 2 le paramètre *drive_mode*.

La consigne de couple est donnée par la variable signée *torque_value* qui peut être modifiée :

- par programme
- par bus CAN ou Profibus
- par affectation de l'entrée analogique à la fonction commande en couple (*a_ina* = 4)

5.4.2 Caractéristiques de la commande en couple

- L'axe n'est plus asservi en position, accélération et décélération ne sont pas prises en compte
- L'erreur de poursuite (*tracking_error*) est forcée à 0 (consigne position = position mesurée)
- Attention au risque d'emballement du moteur si la charge disparaît !

5.5 Synchronisation Maître / Esclave

La fonction synchronisation maître / esclave est réalisée par le mode commande en position (*drive_mode* = 0). Cette fonction nécessite la présence d'un codeur incrémental sur l'axe maître et d'une carte option entrée codeur sur le variateur positionneur.
Il est possible de programmer sur l'axe esclave des mouvements relatifs en plus des mouvements de recopie de l'axe maître (instruction *MOVER*). Il est aussi possible de changer le rapport de recopie (instruction *KSYNC* ou variable *ksync_d*) en cours de synchro.

5.6 Cames

Les variateurs DIGIVEX Motion permettent de gérer des "cames électroniques" : cames en fonction du temps ou cames fonction de la position d'un axe maître. Ce mode de fonctionnement est détaillé dans le document " PVD 3538 - DIGIVEX Motion - Fonction came ".

5.7 Manivelle électronique

Ce cas de figure correspond tout simplement à une séquence de fonctionnement en mode maître / esclave. La manivelle électronique doit se présenter sous la forme d'un codeur incrémental que l'on raccorde sur la carte option entrée codeur du variateur positionneur.

5.8 Manipulateur analogique / Joystick

Ce cas de figure correspond tout simplement à une séquence de fonctionnement en mode commande en vitesse (*drive_mode* = 1). Le joystick doit être de type ±10V et être raccordé sur l'entrée analogique ina → affectation de l'entrée analogique à la fonction commande en vitesse (*a_ina* = 3).
Afin d'éviter une dérive de l'axe en position repos, il est possible de valider une bande morte de ±0,1V autour du zéro, sur l'entrée analogique : ina_option = 1 → pas de déplacement pour une consigne joystick inférieure à 0,1V (S'assurer que le manipulateur analogique possède lui aussi une plage de 0V élargie).

5.9 Contrôle à distance

5.9.1 Généralités

Toutes les variables et paramètres du DIGIVEX Motion sont accessibles par l'intermédiaire du bus CAN ou Profibus.

5.9.2 Contrôle à distance d'un DIGIVEX Motion CANopen à partir d'un PC possédant une liaison série RS232

Il n'est pas nécessaire que l'utilisateur connaisse et sache gérer le protocole CANopen. Par contre, une bonne connaissance de la programmation sous environnement Windows® est indispensable (C, Delphi®, Visual Basic®). Une bibliothèque de fonctions de type "DLL" est fournie avec le logiciel PME. Celle-ci permet d'accéder aux variables et paramètres du variateur positionneur au travers du module d'interface CRS232. Cette "DLL" contient toutes les fonctions nécessaires à l'initialisation et au transfert des messages de type SDO. Le document " PVD 3527 - PME Tool Kit - Notice d'utilisation" décrit ces fonctions.

5.9.3 Contrôle à distance d'un DIGIVEX Motion CANopen à partir d'un automate (ou un superviseur) possédant une liaison série RS232

Il n'est pas nécessaire que l'utilisateur connaisse et sache gérer le protocole CANopen. Par contre, une bonne connaissance de programmation est indispensable (gestion de liaison série de l'automate dans le langage de l'automate). Un document explicitant la mise en œuvre à réaliser pour permettre l'accès aux variables et paramètres du variateur positionneur au travers du module d'interface CIM03 est disponible : " PVD 3533 - Accès au bus CAN par CIM03 ".

5.9.4 Contrôle à distance d'un DIGIVEX Motion CANopen par un autre DIGIVEX Motion CANopen ou par un terminal µVision

Il n'est pas nécessaire que l'utilisateur connaisse et sache gérer le protocole CANopen. Des commandes Basic_DM de lecture et d'écriture permettent d'accéder aux variables et paramètres des variateurs positionneurs. Ces commandes sont décrites dans le document : " PVD 3517 - Programmation ".

5.9.5 Contrôle à distance d'un DIGIVEX Motion CANopen à partir d'un superviseur muni d'une interface CANopen

5.9.5.1 Par consignes ponctuelles (messages SDO)

Il est nécessaire que l'utilisateur connaisse et sache gérer le protocole CANopen. Les données (index et sous-index) permettant d'adresser les variables et paramètres du variateur positionneur au travers des messages de type SDO sont récapitulées dans le document " PVD 3527 - DIGIVEX Motion - Répertoire des variables".

5.9.5.2 Par consignes cycliques (messages PDO)

Il est nécessaire que l'utilisateur connaisse et sache gérer le protocole CANopen. Ce mode de fonctionnement est détaillé dans le document " PVD 3543 - DIGIVEX Motion – Contrôle à distance par messages PDO ".

5.9.6 Contrôle à distance d'un DIGIVEX Motion Profibus à partir d'un superviseur muni d'une interface PROFIBUS

5.9.6.1 Par consignes cycliques

Il est nécessaire que le superviseur supporte la norme PROFIBUS DP-V0 et le profil PROFIdrive V2.0. Ce mode de fonctionnement est détaillé dans le document " PVD 3554 - DIGIVEX Motion - PROFIBUS - Notice d'utilisation ".

5.9.6.2 Par consignes acycliques

Il est nécessaire que le superviseur supporte la norme PROFIBUS DP-V1 et le profil PROFIdrive V3.0 - Appli class 3. Ce mode de fonctionnement est détaillé dans le document " PVD 3554 - DIGIVEX Motion - PROFIBUS - Notice d'utilisation ".