

MANUEL DE RÉFÉRENCE VAL3

Version 7

Documentation addenda and errata can be found in the "readme.pdf" document delivered with the controller's CdRom.

TABLE DES MATIÈRES

1 - INTRODUCTION.....	13
2 - ÉLÉMENTS DU LANGAGE VAL 3	17
2.1 APPLICATIONS	19
2.1.1 Définition	19
2.1.2 Contenu par défaut.....	19
2.1.3 Démarrage et arrêt	19
2.1.4 Paramètres d'application	19
2.1.4.1 Unité de longueur.....	20
2.1.4.2 Quantité de mémoire d'exécution.....	20
2.2 PROGRAMMES	21
2.2.1 Définition	21
2.2.2 Réentrance.....	21
2.2.3 Programme Start().....	21
2.2.4 Programme Stop().....	21
2.2.5 Instructions de contrôle du programme.....	22
Commentaire //	22
Appel de sous-programme call	22
return	22
Instruction de contrôle if	23
Instruction de contrôle while	24
Instruction de contrôle do ... until	24
Instruction de contrôle for	25
Instruction de contrôle switch	26
2.3 VARIABLES	28
2.3.1 Définition	28
2.3.2 Types simples.....	28
2.3.3 Types structurés.....	28
2.3.4 Conteneurs.....	29
2.4 INITIALISATION DES DONNÉES	29
2.4.1 Données de type simple	29
2.4.2 Données de type structuré	29
2.5 VARIABLES	30
2.5.1 Définition	30
2.5.2 Portée d'une variable.....	30
2.5.3 Accès à la valeur d'une variable.....	30
2.5.4 Instructions valables pour toutes les variables	31
num size (*)	31
bool isDefined (*)	31
bool insert (*)	32
bool delete (*)	33
num getData (string sNomDonnées, *)	33

2.5.5	Instructions spécifiques des variables de tableau	35
	void append (*)	35
	num size (*, num nDimension)	35
	void resize (*, num nDimension, num nSize)	35
2.5.6	Instructions spécifiques des variables de collection	36
	string first (*)	36
	string next (*)	36
	string last (*)	36
	string prev (*)	36
2.6	PARAMÈTRES DU PROGRAMME	37
2.6.1	Paramètre par valeur d'élément	38
2.6.2	Paramètre par référence d'élément	38
2.6.3	Paramètre par référence de tableau ou de collection	39
3	- TYPE SIMPLES	41
3.1	TYPE BOOL	43
3.1.1	Définition	43
3.1.2	Opérateurs	43
3.2	TYPE NUM	44
3.2.1	Définition	44
3.2.2	Opérateurs	45
3.2.3	Instructions	46
	num sin (num nAngle)	46
	num asin (num nValeur)	46
	num cos (num nAngle)	46
	num acos (num nValeur)	46
	num tan (num nAngle)	46
	num atan (num nValeur)	47
	num abs (num nValeur)	47
	num sqrt (num nValeur)	47
	num exp (num nValeur)	47
	num power (num nX, num nY)	47
	num ln (num nValeur)	48
	num log (num nValeur)	48
	num roundUp (num nValeur)	48
	num roundDown (num nValeur)	48
	num round (num nValeur)	48
	num min (num nX, num nY)	49
	num max (num nX, num nY)	49
	num limit (num nValeur, num nMini, num nMaxi)	49
	num sel (bool bCondition, num nValeur1, num nValeur2)	49
3.3	TYPE DE CHAMP DE BITS	50
3.3.1	Définition	50
3.3.2	Opérateurs	50
3.3.3	Instructions	50
	num bNot (num nChampBit)	50
	num bAnd (num nChampBit1, num nChampBit2)	50
	num bOr (num nChampBit1, num nChampBit2)	51
	num bXor (num nChampBit1, num nChampBit2)	51
	num toBinary (num nValeur[], num nTailleValeur, string sFormatDonnées, num& nOctetDonnées[])	52
	num fromBinary (num nOctetDonnées[], num nTailleDonnées, string sFormatDonnées, num& nValeur[])	52

3.4	TYPE STRING.....	54
3.4.1	Définition	54
3.4.2	Opérateurs	54
3.4.3	Instructions	54
	string toString (string sFormat, num nValeur)	54
	string toNum (string sChaîne, num& nValeur, bool& bRapport)	55
	string chr (num nCodePoint)	56
	num asc (string sTexte, num nPosition)	57
	string left (string sTexte, num nTaille)	57
	string right (string sTexte, num nTaille)	57
	string mid (string sTexte, num nTaille, num nPosition)	57
	string insert (string sTexte, string sInsertion, num nPosition)	58
	string delete (string sTexte, num nTaille, num nPosition)	58
	string replace (string sTexte, string sRemplacement, num nTaille, num nPosition)	58
	num find (string sTexte1, string sTexte2)	58
	num len (string sTexte)	58
3.5	TYPE DIO	59
3.5.1	Définition	59
3.5.2	Opérateurs	59
3.5.3	Instructions	60
	void dioLink (dio& diVariable, dio diSource)	60
	num dioGet (dio diTableau[])	60
	num dioSet (dio diTableau[], num nValeur)	61
	num ioStatus (dio diEntréeSortie)	61
	num ioStatus (dio diEntréeSortie, string& sDescription, string& sChemin physique)	62
3.6	TYPE AIO	63
3.6.1	Définition	63
3.6.2	Instructions	63
	void aioLink (aio& aiVariable, aio aiSource)	63
	num aioGet (aio aiEntrée)	63
	num aioSet (aio aiSortie, num nValeur)	63
	num ioStatus (aio aiEntréeSortie)	64
	num ioStatus (aio diEntréeSortie, string& sDescription, string& sChemin physique)	64
3.7	TYPE SIO	65
3.7.1	Définition	65
3.7.2	Opérateurs	65
3.7.3	Instructions	66
	void sioLink (sio& siVariable, sio siSource)	66
	num clearBuffer (sio siVariable)	66
	num sioGet (sio siEntrée, num& nDonnées[])	66
	num sioSet (sio siSortie, num& nDonnées[])	66
	num sioCtrl (sio siCanal, string nParamètre, *valeur)	67

4 - INTERFACE UTILISATEUR..... 69

4.1	PAGE UTILISATEUR.....	71
4.2	TYPE D'ÉCRAN	71
4.2.1	Sélection de l'écran utilisateur	71
4.2.2	Ecrire dans un écran utilisateur	71
4.2.3	Lecture dans un écran utilisateur	71

4.3 INSTRUCTIONS	72
void userPage (), void userPage (screen scPage), void userPage (bool bFixe)	72
void gotoxy (num nX, num nY), void gotoxy (screen scPage, num nX, num nY)	72
void cls (), void cls (screen scPage)	72
void setTextMode (num nMode), void setTextMode (screen scPage, num nMode)	72
num getDisplayLen (string sTexte)	73
void put (string sTexte), void put (screen scPage, string sTexte)	
void put (num nValeur), void put (screen scPage, num nValeur),	
void putln (string sTexte), void putln (screen scPage, string sTexte),	
void putln (num nValeur), void putln (screen scPage, num nValeur),	74
void title (string sTexte), void title (screen scPage, string sTexte)	74
num get (string& sTexte), num get (screen scPage, string& sTexte),	
num get (num& nValeur), num get (screen scPage, num& nValeur),	
num get (), num get (screen scPage)	74
num getKey (), num getKey (screen scPage)	76
bool isKeyPressed (num nCode), bool isKeyPressed (screen scPage, num nCode)	76
void popUpMsg (string sTexte)	76
bool logMsg (string sTexte)	77
string getProfile ()	77
num setProfile (string sLoginUtilisateur, string sMotDePasseUtilisateur)	77
string getLanguage ()	78
bool setLanguage (string sLangue)	79
string getDate (string sFormat)	79

5 - TÂCHES 81

5.1 DÉFINITION	83
5.2 REPRISE APRÈS UNE ERREUR D'EXÉCUTION	83
5.3 VISIBILITÉ	83
5.4 SÉQUENCEMENT	84
5.5 TÂCHES SYNCHRONES	85
5.6 ERREUR DE CADENCE	85
5.7 RAFRAÎCHISSEMENT DES ENTRÉES/SORTIES	85
5.8 SYNCHRONISATION	86
5.9 PARTAGE DE RESSOURCE	87
5.10 INSTRUCTIONS	88
void taskSuspend (string sNom)	88
void taskResume (string sNom, num nSaut)	88
void taskKill (string sNom)	89
void setMutex (bool& bMutex)	89
string help (num nCodeErreur)	89
num taskStatus (string sNom)	90
void taskCreate string sNom, num nPriorité, programme(...)	91
void taskCreateSync string sNom, num nPériode, bool& bOverrun, programme(...)	92
void wait (bool bCondition)	93
void delay (num nSecondes)	93
num clock ()	94
bool watch (bool bCondition, num nSecondes)	94

6 - LIBRAIRIES	95
6.1 DÉFINITION	97
6.2 INTERFACE	97
6.3 IDENTIFIANT D'INTERFACE	97
6.4 CONTENU	97
6.5 CRYPTAGE	98
6.6 CHARGEMENT ET DÉCHARGEMENT	99
6.7 INSTRUCTIONS	101
num identifiant:libLoad(string sChemin)	101
num identifiant:libLoad(string sChemin, string sMot de passe)	101
num identifiant:libSave(), num libSave()	101
num libDelete(string sChemin)	101
string identifiant:libPath(), string libPath()	102
bool libList(string sChemin, string& sContenu[])	102
bool identifier:libExist(string sNomSymbole)	102
7 - TYPE D'UTILISATEUR	103
7.1 DÉFINITION	105
7.2 CRÉATION	105
7.3 DOMAINE D'UTILISATION	105
8 - CONTRÔLE DU ROBOT	107
8.1 INSTRUCTIONS	109
void disablePower()	109
void enablePower()	109
bool isPowered()	109
bool isCalibrated()	110
num workingMode(), num workingMode(num& nEtat)	110
num esStatus()	111
bool safetyFault(string& sNom du signal)	111
num ioBusStatus(string& sDescription de l'erreur[])	111
num getMonitorSpeed()	112
num setMonitorSpeed(num nVitesse)	112
string getVersion(string sComposant)	113
9 - POSITIONS DU BRAS	115
9.1 INTRODUCTION	117
9.2 TYPE JOINT	117
9.2.1 Définition	117
9.2.2 Opérateurs	118
9.2.3 Instructions	118
joint abs(joint jPosition)	118
joint herej()	119
bool isInRange(joint jPosition)	119
void setLatch(dio diEntrée) (CS8C only)	120
bool getLatch(joint& jPosition) (CS8C only)	120

9.3	TYPE TRSF	121
9.3.1	Définition	121
9.3.2	Orientation.....	122
9.3.3	Opérateurs	124
9.3.4	Instructions	124
	num distance (trsf trPosition1, trsf trPosition2)	124
	trsf interpolateL (trsf trDémarrage, trsf trFin, num nPosition)	125
	trsf interpolateC (trsf trDémarrage, trsf trIntermédiaire, trsf trFin, num nPosition)	126
	trsf align (trsf trPosition, trsf Reference)	126
9.4	TYPE FRAME	127
9.4.1	Définition	127
9.4.2	Utilisation.....	127
9.4.3	Opérateurs	128
9.4.4	Instructions	128
	num setFrame (point pOrigine, point pAxeOx, point pPlanOxy, frame& fResultat)	128
	trsf position (frame fRepere, frame fReference)	128
	void link (frame fRepere, frame fReference)	128
9.5	TYPE TOOL	129
9.5.1	Définition	129
9.5.2	Utilisation.....	129
9.5.3	Opérateurs	130
9.5.4	Instructions	130
	void open (tool tOutil)	130
	void close (tool tOutil)	131
	trsf position (tool tOutil, tool tReference)	131
	void link (tool tOutil, tool tReference)	131
9.6	TYPE POINT	132
9.6.1	Définition	132
9.6.2	Opérateurs	132
9.6.3	Instructions	133
	num distance (point pPosition1, point pPosition2)	133
	point compose (point pPosition, frame fReference, trsf trTransformation)	133
	point apro (point pPosition, trsf trTransformation)	134
	point here (tool tOutil, frame fReference)	134
	point jointToPoint (tool tOutil, frame fReference, joint jPosition)	134
	bool pointToJoint (tool tOutil, joint jInitial, point pPosition, joint& jResultat)	135
	trsf position (point pPosition, frame fReference)	135
	void link (point pPoint, frame fReference)	135
9.7	TYPE CONFIG	136
9.7.1	Introduction	136
9.7.2	Définition	136
9.7.3	Opérateurs	137
9.7.4	Configuration (bras RX/TX).....	138
	9.7.4.1 Configuration de l'épaule	138
	9.7.4.2 Configuration du coude.....	139
	9.7.4.3 Configuration du poignet.....	139
9.7.5	Configuration (bras RS/TS).....	140
9.7.6	Instructions	140
	config config (joint jPosition)	140

10 - CONTRÔLE DES MOUVEMENTS	141
10.1 CONTRÔLE DE TRAJECTOIRE	143
10.1.1 Types de mouvement : point-à-point, ligne droite, cercle.....	143
10.1.2 Enchaînement de mouvements : Lissage	145
10.1.2.1 Lissage.....	145
10.1.2.2 Annulation du lissage	146
10.1.2.3 Lissage de l'articulation, lissage cartésien	146
10.1.3 Reprise de mouvement	147
10.1.4 Particularités des mouvements cartésiens (ligne droite, cercle)	148
10.1.4.1 Interpolation de l'orientation	148
10.1.4.2 Changement de configuration (Bras RX/TX).....	150
10.1.4.3 Singularités (Bras RX/TX)	152
10.2 ANTICIPATION DES MOUVEMENTS	152
10.2.1 Principe	152
10.2.2 Anticipation et lissage.....	153
10.2.3 Synchronisation	153
10.3 CONTRÔLE DE VITESSE	154
10.3.1 Principe	154
10.3.2 Réglage simple.....	154
10.3.3 Réglage avancé	155
10.3.4 Erreur de traînée	155
10.4 CONTRÔLE DU MOUVEMENT EN TEMPS RÉEL	155
10.5 TYPE MDESC	156
10.5.1 Définition	156
10.5.2 Opérateurs	156
10.6 INSTRUCTIONS DE MOUVEMENT	157
num movej (joint jPosition, tool tOutil, mdesc mDesc)	157
num movej (point pPosition, tool tOutil, mdesc mDesc)	157
num movei (point pPosition, tool tOutil, mdesc mDesc)	157
num movec (point pIntermédiaire, point pCible, tool tOutil, mdesc mDesc)	158
void stopMove ()	159
void resetMotion (), void resetMotion (joint jdépart)	159
void restartMove ()	160
void waitEndMove ()	160
bool isEmpty ()	161
bool isSettled ()	161
void autoConnectMove (bool bActif), void autoConnectMove ()	161
num getSpeed (tool tOutil)	162
joint getPositionErr ()	162
void getJointForce (num& nForce)	162
num getMoveId ()	162
num setMoveId (num nIdMvt)	163

11 - OPTIONS..... 165

11.1 MOUVEMENTS COMPLIANTS AVEC CONTRÔLE EN EFFORT 167

11.1.1 Principe	167
11.1.2 Programmation.....	167
11.1.3 Contrôle de l'effort.....	167
11.1.4 Limitations	168
11.1.5 Instructions	168
num movejf (joint jPosition, tool tOutil, mdesc mDesc, num nForce)	168
num movelf (point pPosition, tool tOutil, mdesc mDesc, num nForce)	169
bool isCompliant ()	169

11.2 ALTER : CONTROLE EN TEMPS REEL D'UNE TRAJECTOIRE..... 170

11.2.1 Principe	170
11.2.2 Programmation.....	170
11.2.3 Contraintes.....	170
11.2.4 Sécurité	171
11.2.5 Limitations	171
11.2.6 Instructions	171
num alterMovej (joint jPosition, tool tOutil, mdesc mDesc)	171
num alterMovej (point pPosition, tool tOutil, mdesc mDesc)	171
num alterMoveI (point pPosition, tool tOutil, mdesc mDesc)	172
num alterMoveC (point pIntermédiaire, point pCible, tool tOutil, mdesc mDesc)	172
num alterBegin (frame fAlterReference, mdesc mVitesseMax)	173
num alterBegin (tool tAlterReference, mdesc mVitesseMax)	173
num alterEnd ()	174
num alter (trsf trAltération)	174
num alterStopTime ()	175

11.3 CONTRÔLE DE LA LICENCE OEM..... 176

11.3.1 Principes	176
11.3.2 Instructions	176
string getLicence (string sOemNomLicence, string sOemMot de passe)	176

11.4 ROBOT ABSOLU 177

11.4.1 Principe	177
11.4.2 Exploitation.....	177
11.4.3 Limitations	177
11.4.4 Instructions	178
void getDH (num& theta[], num& d[], num& a[], num& alpha[], num& beta[])	178
void getDefaultDH (num& theta[], num& d[], num& a[], num& alpha[], num& beta[])	178
bool setDH (num& theta[], num& d[], num& a[], num& b[], num& alpha[], num& beta[])	178

11.5 AXE CONTINU..... 179

11.5.1 Principe	179
11.5.2 Instructions.....	179
joint resetTurn (joint jReference)	179

12 - ANNEXES	181
12.1 CODES D'ERREUR D'EXÉCUTION	183
12.2 CODES DES TOUCHES DU CLAVIER DU PUPITRE	184
13 - ILLUSTRATIONS	185
14 - INDEX.....	187

CHAPITRE 1

INTRODUCTION

VAL 3 est un langage de programmation de haut niveau, destiné à la commande des robots **Stäubli** dans toutes sortes d'applications.

Le langage **VAL 3** combine les caractéristiques de base d'un langage informatique temps réel standard de haut niveau et les fonctionnalités spécifiques du contrôle d'une cellule de robot industriel :

- outils de contrôle du robot
- outils de modélisation géométrique
- outils de contrôle d'entrées sorties

Ce manuel de référence explique les notions indispensables à la programmation d'un robot, et détaille les instructions du langage **VAL 3**, classées suivant les catégories suivantes :

- Eléments du langage
- Types simples
- Interface utilisateur
- Tâches
- Librairies
- Types d'utilisateurs
- Contrôle du robot
- Position du bras
- Contrôle des mouvements
- Type d'écran : pour l'affichage sur l'écran MCP

Chaque instruction, avec sa syntaxe, apparaît dans la table des matières pour une consultation rapide.

CHAPITRE 2

ÉLÉMENTS DU LANGAGE VAL 3

Le langage de programmation **VAL 3** se compose d'applications. Une application **VAL 3** contient à la fois des programmes et des données. Une application **VAL 3** peut aussi faire référence à d'autres applications utilisées soit comme librairies, soit comme définitions de types d'utilisateurs.

2.1. APPLICATIONS

2.1.1. DÉFINITION

Une application **VAL 3** est un logiciel autonome destiné à commander les robots et les entrées-sorties associées à un contrôleur.

Une application **VAL 3** est constituée des éléments suivants :

- un ensemble de **programmes** : les instructions **VAL 3** à exécuter
- un ensemble de **données globales** : les données partagées par tous les programmes de l'application
- un ensemble de **librairies** : applications externes utilisées pour partager des programmes et/ou des données
- un ensemble **types d'utilisateurs** : applications externes utilisées comme modèles pour définir des données structurées dans l'application

Lorsqu'une application est en cours d'exécution, elle contient également :

- un ensemble de tâches : les programmes exécutés simultanément

2.1.2. CONTENU PAR DÉFAUT

Pour créer une nouvelle application **VAL 3**, il faut copier le contenu d'une application prédéfinie servant de modèle. Il est possible de créer de nouveaux modèles personnalisés. Ceux-ci se composent simplement d'une application **VAL 3** standard, placée dans un répertoire dédié dans le contrôleur.

Une application **VAL 3** ne peut être lancée que si elle contient un programme **start()** et un programme **stop()**. Sans les programmes **start()** et **stop()**, une application **VAL 3** ne peut être utilisée que comme librairie ou comme définition de type d'utilisateur. Il est possible de définir des applications contenant seulement des données ou seulement des programmes.

2.1.3. DÉMARRAGE ET ARRÊT

Le lancement d'une application **VAL 3** est géré par le contrôleur. Il peut soit être demandé par l'utilisateur sur l'interface utilisateur du **MCP**, soit être automatique dans le cadre du processus de démarrage.

Une seule application **VAL 3** peut être lancée à la fois. Elle peut cependant utiliser plusieurs autres applications en même temps (par exemple des librairies) et lancer plusieurs tâches d'exécution différentes.

lorsqu'une application **VAL 3** est lancée, son programme **start()** s'exécute.

Une application **VAL 3** s'arrête d'elle-même lorsque la dernière de ses tâches se termine : le programme **stop()** est alors exécuté. Toutes les tâches créées par des librairies, s'il en reste, sont détruites dans l'ordre inverse de leur création.

Si l'arrêt d'une application **VAL 3** est provoqué depuis l'interface utilisateur du contrôleur **MCP**, la tâche de démarrage, si elle existe encore, est immédiatement détruite. Le programme **stop()** est ensuite exécuté, puis toutes les tâches de l'application, s'il en reste, sont détruites dans l'ordre inverse de l'ordre de leur création.

2.1.4. PARAMÈTRES D'APPLICATION

Une application **VAL 3** peut être configurée par les paramètres suivants :

- l'unité de longueur
- quantité de mémoire d'exécution

Ces paramètres sont accessibles à l'aide d'une instruction **VAL 3** et ne peuvent être modifiés qu'à l'aide de l'interface utilisateur du **MCP** ou de **VAL 3 Studio** dans la **Stäubli Robotics Suite**.

2.1.4.1. UNITÉ DE LONGUEUR

Dans les applications **VAL 3**, l'unité de longueur est le millimètre ou le pouce. Elle est utilisée par les types de donnée géométriques du **VAL 3** : repère, point, articulation (pour les axes linéaires), transformation, outil et lissage de trajectoire.

L'unité de longueur d'une application est définie lors de sa création par l'unité de longueur courante du système, et ne peut plus être modifiée ensuite.

2.1.4.2. QUANTITÉ DE MÉMOIRE D'EXÉCUTION

Chaque tâche de **VAL 3** a besoin de mémoire pour enregistrer :

- La pile d'exécution (liste des appels du programme exécutés pendant cette tâche)
- Les paramètres de chaque programme de la pile d'exécution
- Les variables locales pour chaque programme de la pile d'exécution

Par défaut, chaque tâche dispose de **5000** octets pour la mémoire d'exécution. Il n'est habituellement pas nécessaire de modifier ce paramètre.

Cette mémoire peut toutefois ne pas être suffisante pour les applications contenant de grands tableaux de variables locales ou des algorithmes récursifs :

Elle doit alors être augmentée à l'aide de l'interface utilisateur de **MCP** ou de **VAL 3 Studio** dans la **Stäubli Robotics Suite** ou, si l'on veut optimiser l'application, par la réduction du nombre de programmes dans la pile d'appels ou l'utilisation de variables globales au lieu de variables locales.

2.2. PROGRAMMES

2.2.1. DÉFINITION

Un programme est une séquence d'instructions **VAL 3** à exécuter.

Un programme est constitué des éléments suivants :

- La séquence d'**instructions** : les instructions **VAL 3** à exécuter
- Un ensemble de **variables locales** : les données internes au programme
- Un ensemble de **paramètres** : les données fournies au programme lors de son appel

Les programmes permettent de regrouper des séquences d'instructions susceptibles d'être utilisées à plusieurs endroits dans une application. Outre qu'ils font gagner du temps de programmation, ils simplifient aussi la structure des applications, facilitent la programmation et l'entretien et améliorent la lisibilité.

Le nombre d'instructions d'un programme est limité seulement par la place mémoire disponible dans le système.

Le nombre de variables locales et de paramètres n'est limité que par la taille de la mémoire d'exécution pour l'application (voir chapitre 2.1.4.2).

2.2.2. RÉENTRANCE

Les programmes sont réentrants, c'est-à-dire qu'ils peuvent s'appeler de façon récursive (instruction **call**) ou être appelés simultanément par plusieurs tâches. Chaque appel d'un programme utilise ses propres variables et paramètres locaux spécifiques. Aucune interaction n'est possible entre deux appels différents du même programme.

2.2.3. PROGRAMME START()

Le programme **start()** est appelé lorsque l'application **VAL 3** s'exécute. Il ne peut avoir de paramètres.

Ce programme contient habituellement toutes les opérations requises pour exécuter l'application : initialisation des variables globales et des sorties, création des tâches d'application, etc.

L'application ne se ferme pas à la fin du programme **start()** tant que d'autres tâches de l'application sont encore en cours d'exécution.

Il est possible d'appeler le programme **start()** dans un programme (instruction **call**) comme n'importe quel autre programme.

2.2.4. PROGRAMME STOP()

Le programme **stop()** est le programme appelé lors de l'arrêt de l'application **VAL 3**. Il ne peut avoir de paramètres.

On trouvera typiquement dans ce programme toutes les opérations nécessaires pour terminer proprement l'application : réinitialisation des sorties et arrêt des tâches de l'application dans un ordre adéquat, etc.

Le programme **stop()** peut être appelé depuis un programme (instruction **call**) de la même manière que n'importe quel autre programme. L'appel du programme **stop()** n'arrête cependant pas l'application.

2.2.5. INSTRUCTIONS DE CONTRÔLE DU PROGRAMME

Commentaire //

Syntaxe

// <Chaîne>

Fonction

Une ligne commençant par « // » n'est pas exécutée ; l'exécution reprend à la ligne suivante. Il ne faut pas utiliser « // » au milieu d'une ligne, il doit former les premiers caractères de la ligne.

Exemple

```
// Ceci est un exemple de commentaire
```

Appel de sous-programme **call**

Syntaxe

call programme([Paramètre1][,Paramètre2])

Fonction

L'instruction d'appel exécute un programme personnalisé. Le nombre et le type d'expressions après le nom du programme doivent concorder avec l'interface du programme. Les expressions définies comme des paramètres sont d'abord exécutées dans leur ordre de spécification. Les variables locales sont ensuite initialisées et l'exécution du programme commence par son instruction de début.

L'exécution d'un appel est terminée quand le programme exécute une instruction de retour ou de fin.

Exemple

```
// Appel les programmes pick() et place() pour i,j entre 1 et 10
for i = 1 to 10
  for j = 1 to 10
    call pick(pPallet1[i,j])
    call place(pPallet2[i,j])
  endFor
endFor
```

return

Syntaxe

return

Fonction

L'instruction de retour interrompt immédiatement l'exécution du programme en cours. Si ce programme a été appelé par un **call**, son exécution reprend après le **call** dans le programme d'appel. Sinon (si le programme est le programme **start()** ou le point de lancement d'une tâche), la tâche courante se termine. L'instruction de retour a exactement le même effet que l'instruction de fin à la fin du programme.

Un programme est souvent plus facile à comprendre et à entretenir quand son exécution se termine toujours par l'instruction de fin. L'utilisation d'une instruction de retour au milieu d'un programme n'est donc pas souhaitable.

Instruction de contrôle if

Syntaxe

```

if <bool bCondition>
  <instructions>
[elseif <bool bConditionAlternée1>
  <instructions>]
../..
[elseif <bool bConditionAlternéeN>
  <instructions>]
[else
  <instructions>]
endif

```

Fonction

La séquence **if...elseif...else...endif** évalue successivement les expressions booléennes marquées par les mots-clés **if** ou **elseif** jusqu'à ce qu'une expression soit vraie. Les instructions suivant l'expression booléenne sont ensuite exécutées jusqu'au mot-clé **elseif**, **else** ou **endif** suivant. Le programme reprend finalement après le mot-clé **endif**.

Si toutes les expressions booléennes marquées par **if** ou **elseif** sont fausses, les instructions comprises entre les mots-clés **else** et **endif** sont exécutées (si le mot-clé **els** est présent). Le programme reprend ensuite après le mot-clé **endif**.

Il n'y a pas de limites au nombre d'expressions **elseif** dans une séquence **if...endif**.

La séquence **if...elseif...else...endif** peut être remplacée par la séquence **switch...case...default...endSwitch** pour tester les différentes valeurs possibles d'une même expression.

Exemple

Ce programme convertit un **jour** écrit dans une **string** (**sJour**) en un **num** (**nJour**).

```

put ( "Entrer un jour : ")
get ( sDay )
if sDay=="Lundi"
  nDay=1
elseif sDay=="Mardi"
  nDay=2
elseif sDay=="Mercredi"
  nDay=3
elseif sDay=="Jeudi"
  nDay=4
elseif sDay=="Vendredi"
  nDay=5
else
  // Week-end !
  nDay=0
endif

```

Voir aussi

Instruction de contrôle switch

Instruction de contrôle **for**

Syntaxe

```
for <num nCompteur> = <num nDébut> to <num nFin> [step <num nPas>]
  <instructions>
endFor
```

Fonction

Les instructions comprises entre **for** et **endFor** sont exécutées jusqu'à ce que le **nCompteur** dépasse la valeur spécifiée pour **nFin**.

Le **nCompteur** est initialisé par la valeur **nDébut**. Si le **nDébut** dépasse **nFin**, les instructions comprises entre **for** et **endFor** ne sont pas exécutées. A chaque itération, le **nCompteur** est incrémenté de la valeur de **nPas**, et les instructions entre **for** et **endFor** sont de nouveau exécutées si le **nCompteur** ne dépasse pas **nFin**.

Si le **nPas** est positif, la boucle **for** s'arrête quand le **nCompteur** est supérieur à **nFin**. Si le **nPas** est négatif, la boucle **for** s'arrête quand le **nCompteur** est inférieur à **nFin**.

Paramètres

num nCompteur	variable de type num utilisée comme compteur
num nDébut	expression numérique d'initialisation du compteur
num nFin	expression numérique de test de fin de boucle
[num nPas]	expression numérique d'incrément du compteur

Exemple

```
// Ce programme fait tourner l'axe 1 de -90° à +90° en -10° étapes
for nPos = 90 to -90 step -10
  jDest.j1 = nPos
  movej(jDest, flange, mNomSpeed)
  waitEndMove()
endFor
```

Instruction de contrôle **switch**

Syntaxe

```
switch <expression>
case <valeur1> [, <valeur2>]
    <instructions1-2>
    break
[case <valeur3> [, <valeur4>]
    <instructions3-4>
    break ]
[default
    <InstructionsParDéfaut>
    break ]
endSwitch
```

Fonction

La séquence **switch...case...default...endSwitch** évalue successivement les expressions signalées par le mot-clé **case** jusqu'à ce qu'une expression soit égale à l'expression initiale après le mot-clé **switch**.

Les instructions suivant l'expression sont ensuite exécutées, jusqu'au mot-clé **break**. Le programme reprend finalement après le mot-clé **endSwitch**.

Si aucune expression **case** n'est égale à l'expression **switch** initiale, les instructions comprises entre les mots-clés **default** et **endSwitch** sont exécutées (si le mot-clé **default** est présent).

Il n'y a pas de limites au nombre d'expressions **case** dans une séquence **switch...endSwitch**. Les expressions suivant le mot-clé **case** doivent être du même type que celles suivant le mot-clé **switch**.

La séquence **switch...case...default...endSwitch** est très semblable à la séquence **if...elseif...else...endif**.

Elle accepte non seulement les expressions booléennes, mais aussi tout type d'expressions acceptant l'opérateur standard "is equal to"=="

Exemple

Ce programme lit un **num** (**nMenu**) correspondant à une séquence de touches et modifie un **string** **s** en conséquence.

```
nMenu = get()
switch nMenu
case 271
    s = "Menu 1"
    break
case 272
    s = "Menu 2"
    break
case 273, 274, 275, 276, 277, 278
    s = "Menu 3 à 8"
    break
default
    s = "cette touche n'est pas une touche de menu"
    break
endSwitch
```

Ce programme convertit un **jour** écrit dans une **string** (**sJour**) en un **num** (**nJour**).

```
put("Entrer un jour : ")
get(sJour)
switch sJour
case "Lundi"
    nJour=1
    break
case "Mardi"
    nJour=2
```

```
break
case "Mercredi"
  nJour=3
break
case "Jeudi"
  nJour=4
break
case "Vendredi"
  nJour=5
break
default
  // N'est pas un jour de la semaine !
  nJour=0
break
endSwitch
```

2.3. VARIABLES

2.3.1. DÉFINITION

Une variable est un ensemble de valeurs à utiliser comme paramètre ou résultat d'instructions de **VAL 3**.

Les variables se composent des éléments suivants :

- un ensemble de valeurs
- un type définissant les valeurs possibles et les opérations autorisées sur les données. Les types de données les plus simples sont Booléen, Numérique et String
- un conteneur définissant la manière dont les valeurs sont stockées dans les données. Les conteneurs de données possibles dans **VAL 3** sont Élément, Tableau et Collection

2.3.2. TYPES SIMPLES

Le langage **VAL 3** supporte les types simples suivants :

- le type **bool** : pour les valeurs booléennes (vrai / faux)
- le type **num** : pour les valeurs numériques (nombres entiers ou à virgule flottante)
- le type **string** : pour les chaînes de caractères (caractères Unicode)
- le type **dio** : pour les entrées-sorties numériques
- le type **aio** : pour les entrées-sorties numériques (analogiques ou digitales)
- le type **sio** : pour les entrées-sorties sur liaison série, et socket ethernet
- le type **screen** : pour l'affichage sur l'écran du MCP et l'accès par le clavier

Le type de variable est indiqué dans la documentation par les premières lettres de son nom en minuscule :

- **bVariable** est une variable de type **bool**
- **nVariable** est une variable de type **num**
- **sVariable** est une variable de type **string**
- **diVariable** est une variable de type **dio**
- **aiVariable** est une variable de type **aio**
- **siVariable** est une variable de type **sio**
- **scVariable** est une variable de type **screen**

2.3.3. TYPES STRUCTURÉS

Un type structuré combine plusieurs types plus simples dans un nouveau type de niveau plus élevé. Chaque sous-type reçoit un nom et devient accessible individuellement en tant que champ de la structure. Les types adéquats dans une application organisent les données de façon à faciliter les calculs et les évolutions du programme.

Le langage **VAL 3** prend en charge les types structurés suivants, composés de types simples :

- le type **trsf** : pour les transformations géométriques cartésiennes
- le type **frame** : pour les repères géométriques cartésiens
- le type **tool** : pour les outils montés sur un robot
- le type **point** : pour les positions cartésiennes d'un outil
- le type **joint** : pour les positions des axes du robot
- le type **config** : pour les configurations du robot
- le type **mdesc** : pour les paramètres de déplacement du robot

Le langage **VAL 3** prend aussi en charge des types d'utilisateurs combinant des types d'utilisateurs **VAL 3** simples, structurés ou même d'autres dans un nouveau type. Un type d'utilisateur peut être utilisé dans une application de la même manière qu'un type standard.

Le type de variable est indiqué dans la documentation par les premières lettres de son nom en minuscule :

- **trVariable** est une variable de type **trsf**
- **fVariable** est une variable de type **frame**
- **tVariable** est une variable de type **tool**
- **pVariable** est une variable de type **point**

- **jVariable** est une variable de type **joint**
- **cVariable** est une variable de type **config**
- **mVariable** est une variable de type **mdesc**

2.3.4. CONTENEURS

Le conteneur de données définit la manière dont les valeurs sont stockées dans les données :

- Un conteneur "élément" se compose d'une valeur unique. True (booléen), 0 (numérique) ou 'text' (string) ont un conteneur de ce type.
- Un conteneur "tableau" se compose d'un ensemble de valeurs identifiées par 1, 2 ou 3 indices entiers. L'indice initial dans les tableaux est toujours 0.
- Un conteneur "collection" se compose d'un ensemble de valeurs identifié par une clé de chaîne. Une chaîne non vide peut être utilisée comme identifiant de valeur.

Un conteneur tableau à une dimension avec une seule valeur (index 0) est considéré comme un conteneur élément.

Lorsque cela est nécessaire dans la documentation, le conteneur de la variable est identifié avec le nom de celle-ci :

- **s1dArray** est un tableau à une dimension du type **string**
- **s2dArray** est un tableau à deux dimensions du type **string**
- **s3dArray** est un tableau à trois dimensions du type **string**
- **sColl** est une collection du type **string**

Certaines instructions (pour la gestion des tableaux ou des collections) sont indifférentes au type des données. Le type est alors remplacé par un astérisque dans la documentation : '*'.

2.4. INITIALISATION DES DONNÉES

2.4.1. DONNÉES DE TYPE SIMPLE

La syntaxe pour l'initialisation d'une donnée de type simple est indiquée dans le chapitre décrivant chaque type simple. Un tableau ou une collection doit être initialisé élément par élément. La valeur d'initialisation est **false** pour un bool, **0** pour un num et **""** (chaîne vide) pour une chaîne.

Exemple

Dans cet exemple, bBool est une variable booléenne, nPi une variable numérique et sChaîne une variable de chaîne.

```
bBool = true
nPi = 3.141592653
sString = "ceci est une chaîne"
```

2.4.2. DONNÉES DE TYPE STRUCTURÉ

La valeur d'une donnée de type structuré est définie par la séquence de ses valeurs de champ entre les crochets , séparées par des virgules. Les valeurs des champs vides sont remplacées par 0. L'ordre de la séquence est spécifié dans le chapitre détaillant chaque type structuré. La valeur d'une structure peut inclure des valeurs ou d'autres sous-structures imbriquées. Un tableau ou une collection d'un type structuré doit être initialisé élément par élément.

Exemple

Le type point est fait d'un trsf et d'un type config. Une variable de point peut être initialisée de la manière représentée :

```
pPosition = {{100, -50, 200, 0, 0, 0}, {sfree, efree, wfree}}
```

La transformation du point pourrait aussi être initialisée avec :

```
pPosition.trsf = {100, -50, 200,,,}
```

2.5. VARIABLES

2.5.1. DÉFINITION

Une variable est une donnée référencée par son nom dans une application ou un programme.

Une variable est caractérisée par :

- un nom : une chaîne de caractères
- une portée : là où la variable est accessible (dans un seul programme, partagée par des programmes dans une application ou partagée entre des applications)
- un ensemble de valeurs
- un type de données (type simple ou structuré)
- un conteneur de variables (élément, tableau ou collection)

Le nom d'une variable est un chaîne de **1 à 15** caractères parmi "**a..zA..Z0..9_**", commençant par une lettre.

2.5.2. PORTÉE D'UNE VARIABLE

La portée d'une variable peut être :

- globale : tous les programmes de l'application peuvent utiliser la variable, ou
- locale : la variable n'est accessible qu'au programme dans lequel elle est déclarée

Lorsqu'une variable globale et une variable locale ont le même nom, le programme où la variable locale est déclarée utilisera la variable locale, et n'aura pas accès à la variable globale.

Quand une application est utilisée comme librairie, chaque variable globale peut être déclarée comme publique ou privée. Une variable publique est accessible aux applications utilisant la librairie ; une variable privée n'est accessible qu'à l'intérieur de la librairie.

2.5.3. ACCÈS À LA VALEUR D'UNE VARIABLE

L'accès à la valeur d'une variable dépend de son conteneur :

- la valeur d'un conteneur élément est accessible par le nom de la variable (sans crochets) : `nVariable`.
- une valeur dans un tableau est accessible par ses indices numériques entre crochets carrés après le nom de la variable : `n1dArray[nIndex]`, `n2dArray[nIndex1, nIndex2]`, `n3dArray[nIndex1, nIndex2, nIndex3]`.
- une valeur d'une collection est accessible par sa clé de chaîne entre crochets carrés après le nom de la variable : `nCollection[sKey]`

Un conteneur tableau à une dimension avec une seule valeur (index 0) est considéré comme un conteneur élément. Il est possible d'accéder à sa valeur sans crochets : `n1dArray` est équivalent à `n1dArray[0]`.

Les indices numériques utilisés pour accéder à une valeur dans un tableau sont arrondis au nombre entier le plus proche : `n2dArray[5.01, 6.99]` est équivalent à `n2dArray[5, 7]`

L'indice utilisé pour accéder à une valeur dans un tableau est compris entre 0 et la taille de la dimension moins 1.

Les champs d'une variable de type structuré sont accessibles à l'aide d'un '.' suivi du nom du champ : `pPoint.trsf.x` renvoie à la valeur du champ 'x' ou du champ 'trsf' de la donnée de point `pPoint`.

Exemple

Initialisation de variables de type simple avec différents conteneurs :

```
nPi = 3.141592653
sMonth[0] = "Janvier"
sProductName["D 243 064 40 A"] = "VAL 3 CdRom"
```

2.5.4. INSTRUCTIONS VALABLES POUR TOUTES LES VARIABLES

num size(*)

Fonction

Cette instruction renvoie le nombre de valeurs accessibles avec la variable :

- la taille d'une variable de conteneur élément est 1.
- la taille d'un tableau à une dimension est le nombre d'éléments dans le tableau.
- la taille d'une collection est le nombre d'éléments qu'elle contient.

Pour les tableaux à deux et trois dimensions, l'instruction de taille nécessite un deuxième paramètre pour spécifier la dimension de taille. Pour un tableau à une dimension, `size(s1dArray)` est équivalent à `size(s1dArray, 1)`.

La taille d'un paramètre de tableau à une dimension donnée par une référence de tableau dépend de l'index spécifié lors de l'appel du programme. Seule la partie du tableau qui commence à partir de l'index spécifié est accessible dans le sous-programme : `size(s1dArray[nIndex]) = size(s1dArray) - nIndex`.

Paramètre

variable

variable de type quelconque

Exemple

La variable de taille doit être spécifiée sans crochets carrés :

```
// OK
nNbElements=size(sCollection)
// erreur de compilation : clé inattendue
nNbElements=size(sCollection[sKey])
```

Pour un tableau à une dimension, un index indique le début d'un sous-tableau : `size(s1dArray[nIndex])` est la taille du sous-tableau qui commence à l'index `nIndex`.

```
bool isDefined(*)
```

Fonction

Cette instruction renvoie **true** si l'élément spécifié est défini dans un tableau ou une collection, ou **false** si l'élément n'est pas défini.

Elle peut être utilisée pour tester si un élément est défini dans une collection ; elle peut aussi être utilisée pour tester si une librairie utilise une variable de son interface ou pas. Cette fonction est utile pour gérer l'évolution de l'interface d'une librairie et adapter son utilisation selon s'il s'agit d'une version récente ou ancienne de l'interface.

Example

Cet exemple ajoute une nouvelle clé d'article dans une collection.

```
// Demander un nouveau nom de référence
put ("Nouvelle référence •")
get (sReference)
if isDefined(sReferenceColl[sReference])==true
  putln("Erreur : référence déjà définie")
else
// Ajouter nouvel article à la collection
  insert(sReferenceColl[sReference])
endIf
```

L'exemple suivant teste l'interface d'une librairie.

```
// Charger librairie de pièces
nLoadCode = part:libLoad(sPartPath)
// part:sVersion n'a pas été défini dans la première version de la librairie
```

```
// Tester si cette librairie le définit
if (nLoadCode==0) or (nLoadCode==11)
  if (isDefined(part:sVersion)==false)
    // version initiale
    sLibVersion = "v1.0"
  else
    sLibVersion = part:sVersion
  endIf
endIf
```

bool insert(*)

Fonction

Cette instruction crée une nouvelle valeur du type de la variable et l'enregistre dans le conteneur de variables. La nouvelle valeur est initialisée avec la valeur par défaut du type. La taille de la variable est augmentée d'un.

Pour un tableau à une dimension, la nouvelle valeur est insérée dans la position d'index spécifiée. La position d'index peut être égale à la taille du tableau : l'insertion est alors effectuée en fin de tableau. "insert(s1dArray[size(s1dArray)])" est équivalent à "append(s1dArray)".

Pour les collections, l'insertion n'est acceptée que si la clé n'est pas déjà utilisée dans la collection. La nouvelle valeur est associée à la clé spécifiée et la fonction renvoie **true**. L'instruction est sans effet et renvoie **false** si la clé était déjà utilisée. L'instruction **isDefined()** peut être utilisée pour vérifier si une clé est ou non utilisée dans une collection.

Cette instruction n'est pas utilisable pour les tableaux à deux et trois dimensions (utiliser à la place l'instruction **resize()**) ni pour les variables de tableau locales. La taille d'une variable est limitée à 9999 valeurs. Une erreur de temps d'exécution est générée quand la taille de la variable dépasse cette limite.

L'instruction d'insertion attribue de la mémoire système. Les performances de l'attribution de mémoire ne sont pas garanties. Il vaut donc mieux éviter l'usage fréquent d'**insert()** dans les applications VAL 3 si les performances doivent rester bonnes.

Exemple

Cet exemple ajoute un nouvel article à une liste.

```
// Demander un nouveau nom d'article
put("Nouvel article •")
get(sNom de l'article)
println("")
// Demander la position dans la liste
put("Position ?")
get(nIndex)
if (nIndex<0) or (nIndex>size(sListe d'articles))
  println("Erreur: position incorrecte")
else
  // Ajouter le nouvel article à la liste
  insert(sListe d'articles[nIndex])
  sListe d'articles[nIndex] = sNom de l'article
endIf
```

Cet exemple ajoute une nouvelle clé d'article dans une collection.

```
// Demander un nouveau nom d'article
put("Nouvel article •")
get(sNom de l'article)
if isDefined(sCollection d'articles[sNom de l'article])==true
  println("Erreur: référence déjà définie")
else
  // Ajouter nouvel article à la collection
  insert(sCollection d'articles[sNom de l'article])
endIf
```


bool delete(*)

Fonction

Cette instruction supprime la valeur spécifiée du conteneur de la variable. La taille de la variable est réduite d'un.

Une erreur de temps d'exécution est générée si l'index ou la clé spécifiés dépassent les limites. L'instruction `isDefined()` peut être utilisée pour vérifier si une clé est ou non utilisée dans une collection.

La taille d'une collection peut être nulle, mais une variable de tableau doit toujours avoir au moins un élément. Une erreur de temps d'exécution est générée si l'on essaie de supprimer le dernier élément d'un tableau.

Cette instruction n'est pas utilisable pour les tableaux à deux et trois dimensions (utiliser à la place l'instruction `resize()`) ni pour les variables de tableau locales.

L'instruction `delete()` libère de la mémoire système. Les performances du recueil des données effacées ne sont pas garanties. Il vaut donc mieux éviter l'usage fréquent d'`delete()` dans les applications VAL 3 si les performances doivent rester bonnes.

Exemple

Cet exemple supprime un article dans une collection.

```
// Demander l'article à supprimer
put ("Article à supprimer •")
get (sNom de l'article)
if isDefined(sCollection d'articles[sNom de l'article]) == true
  // supprime l'article dans la collection
  delete(sCollection d'articles[sNom de l'article])
else
  println ("Erreur : article non défini")
endif
```

num getData(string sNomDonnées, *)

Fonction

Cette instruction copie la valeur des données, spécifiée par son nom **sNomDonnées**, dans la variable spécifiée. Si la donnée et la variable sont toutes deux des tableaux à une dimension, l'instruction `getData()` copie toutes les entrées du tableau jusqu'à la fin de celui-ci. L'instruction renvoie le nombre d'entrées copiées dans la variable.

Le nom des données doit avoir le format suivant : "library:name[index]", où "library:" et "[index]" sont facultatifs :

- "name" est le nom de la donnée
- "library" est le nom de l'identifiant de librairie où la donnée est définie
- "index" est la valeur numérique de l'index auquel il faut accéder quand la donnée est un tableau à une dimension

L'instruction renvoie un code d'erreur lorsque les données n'ont pas pu être copiées :

Valeur renvoyée	Description
n > 0	La variable a bien été actualisée avec n entrées copiées
-1	La donnée n'existe pas
-2	L'identifiant de librairie n'existe pas
-3	L'index est hors limites
-4	Le type de donnée ne correspond pas au type de variable

Exemple

Ce programme fusionne 2 tableaux de points pApproche[] et pTrajectoire[] provenant d'une librairie en un seul tableau local pChemin[].

```
// Copier les points d'approche dans le chemin
i = getData("Partie:pApproche", pChemin)
if(i > 0)
    nPoints = i
    // Joindre les points de trajectoire dans le chemin
    i = getData("Partie:pTrajectoire", pChemin[nPoints])
    if(i >0)
        nPoints=nPoints+i
    endIf
endIf
```

2.5.5. INSTRUCTIONS SPÉCIFIQUES DES VARIABLES DE TABLEAU

`void append(*)`

Fonction

Cette instruction crée une nouvelle valeur du type de la variable et l'enregistre à la fin de la variable de tableau à une dimension. La nouvelle valeur est initialisée avec la valeur par défaut du type. La taille de la variable est augmentée d'un.

Cette instruction n'est pas utilisable pour les tableaux à deux et trois dimensions ni pour les variables de tableau local. La taille d'une variable est limitée à 9999 valeurs. Une erreur de temps d'exécution est générée quand la taille de la variable dépasse cette limite.

L'instruction d'ajout affecte de la mémoire système. Les performances de l'attribution de mémoire ne sont pas garanties. Il vaut donc mieux éviter l'usage fréquent d'`append()` dans les applications VAL 3 si les performances doivent rester bonnes.

Exemple

Cet exemple joint un nouvel article dans une liste.

```
// Demander un nouveau nom d'article
put ("Nouvel article •")
get (sArticle)
println("")
append(sListe d'articles)
sListe d'articles[size(sListe d'articles)-1] = sArticle
```

`num size(*, num nDimension)`

Fonction

Cette instruction renvoie la taille de la dimension spécifiée dans le tableau. Si `nDimension` dépasse les dimensions du tableau, la fonction renvoie 0. Pour un tableau à une dimension, `size(s1dArray, 1)` est équivalent à `size(s1dArray)`.

Exemple

La variable de tableau doit être indiquée sans crochets carrés :

```
//OK
nNbElements=size(s3dArray, 3)
// Erreur de compilation : index inattendus
nNbElements=size(s3dArray[1, 2, 3], 3)
```

`void resize(*, num nDimension, num nSize)`

Fonction

Cette instruction crée ou supprime des valeurs dans un tableau de telle façon que la taille de la dimension spécifiée corresponde à la valeur `nSize`. La création ou la suppression de la valeur s'effectue à la fin du tableau. Le cas échéant, les nouvelles valeurs sont initialisées avec la valeur par défaut du type.

Cette instruction n'est pas utilisable pour les variables de tableau local. La taille d'une variable est limitée à 9999 valeurs. Une erreur de temps d'exécution est générée quand la taille de la variable dépasse cette limite.

L'instruction `resize()` affecte ou libère de la mémoire système. Les performances de gestion de la mémoire ne sont pas garanties. Il vaut donc mieux éviter l'usage fréquent d'`resize()` dans les applications VAL 3 si les performances doivent rester bonnes.

Exemple

La variable doit être définie sans index. L'instruction suivante modifie `s2dArray` de telle façon que sa seconde dimension soit 5.

```
resize(s2dArray, 2, 5)
```

2.5.6. INSTRUCTIONS SPÉCIFIQUES DES VARIABLES DE COLLECTION

string first(*)

Fonction

Cette instruction renvoie la première clé d'une collection dans l'ordre alphabétique. Si la collection est vide, l'instruction renvoie une chaîne vide "".

string next(*)

Fonction

Cette instruction renvoie la clé suivante d'une collection dans l'ordre alphabétique. Si la clé spécifiée est la dernière de la collection, l'instruction renvoie la chaîne vide "".

Exemple

Cet exemple imprime tous les éléments d'une collection sur la page utilisateur dans l'ordre alphabétique :

```
sKey = first(sCollection)
while sKey != ""
    sKey = next(sCollection[sKey])
    println(sKey)
endWhile
```

string last(*)

Fonction

Cette instruction renvoie la dernière clé d'une collection dans l'ordre alphabétique. Si la collection est vide, l'instruction renvoie une chaîne vide "".

string prev(*)

Fonction

Cette instruction renvoie la clé précédente d'une collection dans l'ordre alphabétique. Si la clé spécifiée est la première de la collection, l'instruction renvoie une chaîne vide "".

Exemple

Cet exemple imprime tous les éléments d'une collection sur la page utilisateur dans l'ordre alphabétique inverse des clés :

```
sKey = last(sCollection)
while sKey != ""
    sKey = prev(sCollection[sKey])
    println(sKey)
endWhile
```

2.6. PARAMÈTRES DU PROGRAMME

Les paramètres de sous-programme sont des données transmises d'un programme d'appel à un sous-programme avec l'instruction d'appel. Dans le sous-programme, les paramètres sont comme des variables locales initialisées automatiquement quand le sous-programme est lancé.

Il existe différentes manières de faire passer une variable à un sous-programme :

- on peut ne faire passer qu'une valeur (un élément) de la variable ou le conteneur (tableau ou collection) de la variable dans son ensemble.
- on peut laisser le sous-programme modifier la valeur de la variable (en transmettant la variable "par référence") ou faire passer simplement une copie de la valeur au sous-programme, en laissant la variable inchangée (transmission d'une variable "par valeur").

Une variable peut être transmise comme paramètre :

- par valeur d'élément.
- par référence d'élément.
- par référence de tableau ou de collection.

Il n'est pas permis de faire passer un conteneur (tableau ou collection) par sa valeur.

Une référence est notée d'un symbole '&' après le type de données dans la définition de l'interface du programme :

`num& nData` est un paramètre `num` transmis par référence à l'élément.

`num& n1dArray[]` est un `num` transmis par référence au tableau (tableau à une dimension).

`num& n2dArray[,]` est un paramètre `num` transmis par référence au tableau (tableau à deux dimensions).

`num& n3dArray[,,,]` est un paramètre `num` transmis par référence au tableau (tableau à trois dimensions).

`num& nCollection[""]` est un paramètre `num` transmis par référence à la collection.

La même notation est utilisée dans la documentation pour la description des instructions :

`bool pointToJoint(tool tOutil, joint jInitial, point pPosition, joint& jResultat)` est une instruction renvoyant une valeur booléenne, prenant un outil, une articulation et une donnée de point transmis par la valeur de l'élément et une articulation transmise par référence à l'élément.

`num fromBinary(num& nOctetDonnées[], num nTailleDonnées, string sFormatDonnées, num& nValeur[])` est une instruction renvoyant une valeur numérique, prenant un tableau à une dimension comme premier paramètre (transmis par référence), une donnée numérique et une donnée de chaîne transmises par valeur d'élément et un tableau à une dimension comme dernier paramètre (transmis par référence).

2.6.1. PARAMÈTRE PAR VALEUR D'ÉLÉMENT

Quand un paramètre est défini par la valeur de l'élément, le système crée une variable locale et l'initialise avec la valeur de l'instruction **VAL 3** fournie par le programme d'appel. Si l'instruction fournie est une variable, le paramètre est initialisé avec une copie de la valeur de la variable. Aucune modification apportée à la valeur du paramètre dans le sous-programme n'a d'effet sur celle de la variable dans le programme d'appel.

Exemple

Supposons que `sendMessage(string sMessage)` est un programme avec un seul paramètre transmis par la valeur de l'élément.

`sendMessage()` peut être utilisé avec une donnée constante ou le résultat du calcul :

```
call sendMessage ("Attente du signal StartCycle")
call sendMessage ("Attente du signal"+sSignalName)
```

`sendMessage()` peut être utilisé avec des valeurs d'éléments, de tableaux ou de collections :

```
call sendMessage(sMessage)
call sendMessage(sMessageArray[23])
call sendMessage(s2dArray[12,3])
call sendMessage(s3dArray[5,7,9])
call sendMessage(sMessageColl[sMessageName])
```

Après ces appels, la valeur de `sMessage`, `sMessageArray[23]`, `s2dArray[12,3]`, `s3dArray[5,7,9]`, `sMessageColl[sMessageName]` n'a pas été modifiée par les instructions dans `sendMessage()`.

2.6.2. PARAMÈTRE PAR RÉFÉRENCE D'ÉLÉMENT

Quand un paramètre est défini par une référence d'élément, le système crée une variable locale et l'initialise avec un lien vers la donnée fournie par le programme d'appel. La variable transmise par référence peut avoir un conteneur élément, tableau ou collection mais seule la valeur indiquée dans l'appel est transmise au sous-programme. Le conteneur du paramètre est toujours un élément. Toutes les modifications apportées à la valeur du paramètre dans le sous-programme affectent directement la valeur correspondante dans les données du programme d'appel. Il n'est pas possible de transmettre une donnée constante **VAL 3** ou le résultat d'une expression **VAL 3** par référence d'élément.

Exemple

Supposons que `sendMessage(string& sMessage)` est un programme avec un seul paramètre transmis par référence d'élément.

`sendMessage()` ne peut pas être utilisé avec une donnée constante ou le résultat d'un calcul :

```
// erreurs de compilation : variable attendue comme paramètre
call sendMessage ("Attente du signal StartCycle")
call sendMessage ("Attente du signal"+sSignalName)
```

`sendMessage()` peut être utilisé avec des valeurs d'éléments, de tableaux ou de collections :

```
call sendMessage(sMessage)
call sendMessage(sMessageArray[23])
call sendMessage(s2dArray[12,3])
call sendMessage(s3dArray[5,7,9])
call sendMessage(sMessageColl[sMessageName])
```

Après ces appels, la valeur de `sMessage`, `sMessageArray[23]`, `s2dArray[12,3]`, `s3dArray[5,7,9]`, `sMessageColl[sMessageName]` peut avoir été modifiée par les instructions de `sendMessage()`.

2.6.3. PARAMÈTRE PAR RÉFÉRENCE DE TABLEAU OU DE COLLECTION

Quand un paramètre est défini par une référence de tableau ou de collection, le système crée une variable locale et l'initialise avec un lien vers la donnée fournie par le programme d'appel. Le conteneur du paramètre dans le sous-programme est le même que celui de la variable fournie : un tableau à une, deux ou trois dimensions ou une collection. Tous les changements apportés à une valeur du paramètre dans le sous-programme affectent directement la valeur correspondante de la donnée du programme d'appel.

Avec les tableaux à une dimension, il est possible de transmettre seulement une partie du tableau au sous-programme, en indiquant le premier élément accessible. Pour les tableaux à deux et trois dimensions et les collections, il n'est pas possible de transmettre une partie seulement du tableau ou de la collection au sous-programme. La variable doit alors être transmise sans crochets [] indiquant un index ou une clé. Il n'est pas possible de transmettre une donnée constante **VAL 3** ou le résultat d'une expression **VAL 3** par référence de tableau ou de collection.

Exemple

Supposons que `send1dMessage(string& s1dArray[])` est un programme à un seul paramètre transmis par référence de tableau (une dimension).

Supposons que `send2dMessage(string& s2dArray[])` est un programme à un seul paramètre transmis par référence de tableau (deux dimensions).

Supposons que `send3dMessage(string& s3dArray[])` est un tableau à un paramètre transmis par référence de tableau (trois dimensions).

Supposons que `sendCollMessage(string& sMessageColl[""])` est un programme à un seul paramètre transmis par référence de collection.

Aucun de ces programmes ne peut être utilisé avec une donnée constante ou le résultat d'un calcul :

```
// erreurs de compilation : variable de tableau attendue comme paramètre
call send1dMessage ("Attente du signal StartCycle")
call send1dMessage ("Attente du signal"+1_sSignalName)
```

Le conteneur de la variable transmise doit correspondre au conteneur déclaré du paramètre :

```
// erreurs de compilation : variable de tableau 1d attendue
call send1dMessage (sMessageColl)
call send1dMessage (s2dArray[12,3])
// erreur de compilation : variable de collection attendue
call sendCollMessage (sMessage)
```

Il n'est pas possible de transmettre une partie d'un tableau ou d'une collection, sauf pour les tableaux 1d. Les tableaux et les collections doivent être indiqués sans indice ni clé.

```
// paramètre correct
call send2dMessage (s2dArray)
call send3dMessage (s3dArray)
call sendCollMessage (sMessageColl)
call send1dMessage (sMessageArray)
call send1dMessage (sMessageArray[23])
// erreurs de compilation : index inattendus pour le tableau
call send2dMessage (s2dArray[12,3])
call send3dMessage (s3dArray[5,7,9])
// erreur de compilation : index inattendus pour la collection
call sendCollMessage (sMessageColl[1_sMessageName])
```

Dans ce dernier cas, seule la partie du tableau `sMessageArray` commençant à l'index 23 est transmise au programme `send1dMessage()`. Les valeurs de `sMessageArray` ayant un index inférieur à 23 sont inaccessibles à `send1dMessage()`.

CHAPITRE 3

TYPE SIMPLES

3.1. TYPE BOOL

3.1.1. DÉFINITION

Les valeurs possibles des variables ou constantes de type bool sont :

- **true** : valeur vraie
- **false** : valeur fausse

Par défaut, une variable de type **bool** est initialisée à la valeur **false**.

3.1.2. OPÉRATEURS

Par ordre de priorité croissant :

bool < bool & bVariable > = < bool bCondition >	Affecte la valeur de bCondition à la variable bVariable et renvoie la valeur de bCondition
bool < bool bCondition1 > or < bool bCondition2 >	Renvoie la valeur du OU logique entre bCondition1 et bCondition2 . bCondition2 n'est évaluée que si bCondition1 est false .
bool < bool bCondition1 > and < bool bCondition2 >	Renvoie la valeur du ET logique entre bCondition1 et bCondition2 . bCondition2 n'est évaluée que si bCondition1 est true .
bool < bool bCondition1 > xor bool < bCondition2 >	Renvoie la valeur du OU exclusif entre bCondition1 et bCondition2
bool < bool bCondition1 > != < bool bCondition2 >	Teste l'égalité des valeurs de bCondition1 et bCondition2 . Renvoie true si les valeurs sont différentes, false sinon.
bool < bool bCondition1 > == < bool bCondition2 >	Teste l'égalité des valeurs de bCondition1 et bCondition2 . Renvoie true si les valeurs sont identiques, false sinon.
bool ! < bool bCondition >	Renvoie la négation de la valeur de bCondition

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de VAL 3 servant de paramètre d'instructions. **if**(**bCondition1**=**bCondition2**) serait interprété comme **bCondition1**=**bCondition2**; **if**(**bCondition1**==**true**). Cependant, l'intention était souvent d'écrire : **if**(**bCondition1**==**bCondition2**), ce qui est très différent !

3.2. TYPE NUM

3.2.1. DÉFINITION

Le type **num** représente une valeur numérique comprenant environ **14** chiffres significatifs.

Chaque calcul numérique se fait donc avec une précision limitée par ces **14** chiffres significatifs.

Il faut en tenir compte lorsque l'on veut tester l'égalité de deux numériques : le plus souvent, il est nécessaire de tester dans un intervalle.

Les constantes de type numérique ont le format suivant :

```
[ - ] <digits> [ . <digits> ] [ e [ - ] <digits> ]
```

Le 'e' est un marqueur de notation scientifique numérique, qui remplace '10^' : 1e3 est égal à 1×10^3 (ou 1000), 1e-2 est égal à 1×10^{-2} (ou 0.01).

Les variables de type **num** sont initialisées par défaut à la valeur **0**.

Exemple

Le test du résultat du calcul numérique doit tenir compte de l'inexactitude numérique des calculs.

```
if cos(nAngle)==0, if abs(cos(nAngle))<1e-10 sera de préférence remplacé par if
abs(cos(nAngle))<1e-10.
```

Voici quelques nombres constants :

```
1
0.2
-3.141592653
6.02214179e23
1.054571628e-34
```

3.2.2. OPÉRATEURS

Par ordre de priorité croissant :

num <num& nVariable> = <num nValeur>	Affecte nValeur à la variable nVariable et renvoie nValeur .
bool <num nValeur1> != <num nValeur2>	Renvoie true si nValeur1 n'est pas égal à nValeur2 , false sinon.
bool <num nValeur1> == <num nValeur2>	Renvoie true si nValeur1 est égal à nValeur2 , false sinon.
bool <num nValeur1> >= <num nValeur2>	Renvoie true si nValeur1 est supérieur ou égal à nValeur2 , false sinon.
bool <num nValeur1> > <num nValeur2>	Renvoie true si nValeur1 est strictement supérieur à nValeur2 , false sinon.
bool <num nValeur1> <= <num nValeur2>	Renvoie true si nValeur1 est inférieur ou égal à nValeur2 , false sinon.
bool <num nValeur1> < <num nValeur2>	Renvoie true si nValeur1 est strictement inférieur à nValeur2 , false sinon.
num <num nValeur1> - <num nValeur2>	Renvoie la différence entre nValeur1 et nValeur2 .
num <num nValeur1> + <num nValeur2>	Renvoie la somme de nValeur1 et nValeur2 .
num <num nValeur1> % <num nValeur2>	Opération modulo : Renvoie le reste de la division d'entier de nValeur1 par nValeur2 . Une erreur d'exécution est générée si nValeur2 est 0. Le signe du reste est le signe de nValeur1 .
num <num nValeur1> / <num nValeur2>	Renvoie le quotient de nValeur1 par nValeur2 . Une erreur d'exécution est générée si nValeur2 est 0.
num <num nValeur1> * <num nValeur2>	Renvoie le produit de nValeur1 et nValeur2 .
num - <num nValeur>	Renvoie l'opposé de nValeur .

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de VAL 3 servant de paramètre d'instructions. nCos=**cos**(nAngle=**30**) doit être remplacé par nAngle=**30** ; nCos=**cos**(nAngle).

3.2.3. INSTRUCTIONS

num sin(num nAngle)

Fonction

Retourne le sinus de **nAngle**.

Paramètre

num nAngle angle en degrés

Exemple

`sin(30)` renvoie 0.5

num asin(num nValeur)

Fonction

Retourne le sinus inverse de **nValeur**, en degrés. Le résultat est compris entre **-90** et **+90** degrés.

Une erreur d'exécution est générée si **nValeur** est supérieur à **1** ou inférieur à **-1**.

Exemple

`asin(0.5)` renvoie 30

num cos(num nAngle)

Fonction

Retourne le cosinus de **Angle**.

Paramètre

num nAngle angle en degrés

Exemple

`cos(60)` renvoie 0.5

num acos(num nValeur)

Fonction

Retourne le cosinus inverse de **nValeur**, en degrés. Le résultat est compris entre **0** et **180** degrés.

Une erreur d'exécution est générée si **nValeur** est supérieur à **1** ou inférieur à **-1**.

Exemple

`acos(0.5)` renvoie 60

num tan(num nAngle)

Fonction

Retourne la tangente de **Angle**.

Paramètre

num nAngle angle en degrés

Exemple

`tan(45)` renvoie 1.0

num atan(num nValeur)

Fonction

Retourne la tangente inverse de **nValeur**, en degrés. Le résultat est compris entre **-90** et **+90** degrés.

Exemple

atan(1) renvoie 45

num abs(num nValeur)

Fonction

Retourne la valeur absolue de **nValeur**.

Exemple

Le test du résultat du calcul numérique doit tenir compte de l'inexactitude numérique des calculs :

Il vaut mieux remplacer **if cos(nAngle)==0** par **if abs(cos(nAngle))<1e-10**.

abs(3.1415) renvoie 3.1415

abs(-3.1415) renvoie 3.1415

num sqrt(num nValeur)

Fonction

Retourne la racine carrée de **nValeur**.

Une erreur d'exécution est générée si **nValeur** est négatif.

Exemple

sqrt(9) renvoie 3

num exp(num nValeur)

Fonction

Retourne l'exponentielle de **nValeur**.

Une erreur d'exécution est générée si **nValeur** est trop grand.

Exemple

exp(1) renvoie 2.718281828459

num power(num nX, num nY)

Fonction

Renvoie **nX** à la puissance **nY** : nX^{nY}

Une erreur d'exécution est générée si **nX** est négatif ou nul ou si le résultat est trop grand.

Exemple

Ce programme calcule de 2 manières différentes 5 à la puissance 7.

```
// Première manière : instruction power
```

```
nResultat = power(5,7)
```

```
// Deuxième manière : power(x,y)=exp(y*ln(x)) (avec imprécision numérique)
```

```
nResultat = exp(7*ln(5))
```

num ln(num nValeur)

Fonction

Retourne le logarithme népérien de **nValeur**.

Une erreur d'exécution est générée si **nValeur** est négatif ou nul.

Exemple

ln(2.718281828) renvoie 0.99999999983113

num log(num nValeur)

Fonction

Retourne le logarithme décimal de **nValeur**.

Une erreur d'exécution est générée si **nValeur** est négatif ou nul.

Exemple

log(1000) renvoie 3

num roundUp(num nValeur)

Fonction

Retourne **nValeur** arrondi à l'entier immédiatement supérieur.

Exemple

roundUp(7.8) renvoie 8

roundUp(-7.8) renvoie -7

num roundDown(num nValeur)

Fonction

Retourne **nValeur** arrondi à l'entier immédiatement inférieur.

Exemple

roundDown(7.8) renvoie 7

roundDown(-7.8) renvoie -8

num round(num nValeur)

Fonction

Retourne **nValeur** arrondi à l'entier le plus proche.

Exemple

round(7.8) renvoie 8

round(-7.8) renvoie -8

round(0.5) renvoie 1

round(-0.5) renvoie 0

`num min(num nX, num nY)`

Fonction

Retourne le minimum de **nX** et **nY**.

Exemple

`min(-1, 10)` renvoie -1

`num max(num nX, num nY)`

Fonction

Retourne le maximum de **nX** et **nY**.

Exemple

`max(-1, 10)` renvoie 10

`num limit(num nValeur, num nMini, num nMaxi)`

Fonction

Renvoie **nValeur** borné par **nMini** et **nMaxi**.

Exemple

`limit(30, -90, 90)` renvoie 30

`limit(100, -90, 90)` renvoie 90

`limit(-100, -90, 90)` renvoie -90

`num sel(bool bCondition, num nValeur1, num nValeur2)`

Fonction

Renvoie **nValeur1** si **bCondition** est **true**, sinon **nValeur2**.

Exemple

`sel(true, -90, 90)` renvoie -90

`sel(false, -90, 90)` renvoie 90

3.3. TYPE DE CHAMP DE BITS

3.3.1. DÉFINITION

Un champ de bits est un moyen de stocker et d'échanger sous forme compacte une série de bits (valeurs booléennes ou entrées/sorties numériques). **VAL 3** ne donne pas un type de données spécifique pour gérer les champs de bits mais réutilise le type num pour stocker un champ de bits de 32 bits sous la forme d'un nombre entier positif dans la plage $[0, 2^{32}]$.

Toute valeur numérique de **VAL 3** peut être considérée comme un champ de bits de 32 bits ; les instructions de traitement du champ de bits arrondissent automatiquement une valeur numérique à un nombre entier positif de 32 bits, qui est alors traité comme un champ de bits de 32 bits.

3.3.2. OPÉRATEURS

Les opérateurs standard du type num s'appliquent dans un champ de bits : '=', '==', '!=', '<', '>', '<=', '>='.

3.3.3. INSTRUCTIONS

num bNot(num nChampBit)

Fonction

Cette instruction renvoie l'opération logique binaire "NOT" (négation) dans un champ de bits de 32 bits. (Le i^{ème} bit du résultat est initialisé à 1 si le i^{ème} bit de l'entrée est à 0). On obtient ainsi un nombre entier positif dans la plage $[0, 2^{32}]$.

L'entrée numérique est d'abord arrondie à un nombre entier positif dans la plage $[0, 2^{32}]$ avant que l'opération binaire soit appliquée.

Exemple

Ce programme réinitialise les bits i à j d'un champ de bits nChampBit à l'aide d'un masque nMask.

```
// Calculer un masque de bits avec les bits i à j initialisés à 1 (voir les explications sous bOr)
nMasque=(power(2, j-i+1)-1)*power(2, i)
// Inverser le masque pour que tous les bits soient à 1 sauf les bits i à j
nMasque=bNot(nMasque)
// Réinitialiser les bits i à j à l'aide de l'opération binaire 'and'
nChampBit=bAnd(nChampBit, nMasque)
```

num bAnd(num nChampBit1, num nChampBit2)

Fonction

Cette instruction renvoie l'opération logique binaire "ET" sur deux champs de bits de 32 bits. (Le i^{ème} bit du résultat est initialisé à 1 si les i^{èmes} bits des deux entrées sont initialisés à 1). On obtient ainsi un nombre entier positif dans la plage $[0, 2^{32}]$.

Les entrées numériques sont d'abord arrondies à un nombre entier positif dans la plage $[0, 2^{32}]$ avant que l'opération binaire soit appliquée.

Exemple

Ce programme affiche un champ de bits nChampBit de 32 bits à l'écran en testant successivement chaque bit :

```
for i=31 to 0 step -1
// Calculer le masque pour le ième bit
nMasque=power(2, i)
if bAnd(nChampBit, nMasque)==nMasque
put("1")
else
put("0")
endif
endFor
putln("")
```

num bOr(num nChampBit1, num nChampBit2)

Fonction

Cette instruction renvoie l'opération logique binaire "OU" sur deux champs de bits de 32 bits. (Le i^{ème} bit du résultat est initialisé à 1 si le i^{ème} bit d'au moins une entrée est initialisé à 1). On obtient ainsi un nombre entier positif dans la plage $[0, 2^{32}]$.

Les entrées numériques sont d'abord arrondies à un nombre entier positif dans la plage $[0, 2^{32}]$ avant que l'opération binaire soit appliquée.

Exemple

Ce programme calcule de deux manières différentes un masque de champ de bits dans lequel les i^{ème} à j^{ème} bits sont activés.

```
// Première manière : logique 'or' sur les bits i à j
nChampBit=0
for k=i to j
  nChampBit=bOr(nChampBit, power(2,k))
endFor
// Deuxième manière : calculer un masque de bits de (j-i) bits
nChampBit=(power(2,j-i+1)-1)
// Déplacer ensuite le masque de bits de i bits
nChampBit=nChampBit*power(2,i)
```

num bXor(num nChampBit1, num nChampBit2)

Fonction

Cette instruction renvoie l'opération logique binaire "XOR" (ou exclusif) sur deux champs de bits de 32 bits. (Le i^{ème} bit du résultat est initialisé à 1 si les i^{èmes} bits des deux entrées sont différents). On obtient ainsi un nombre entier positif dans la plage $[0, 2^{32}]$.

Les entrées numériques sont d'abord arrondies à un nombre entier positif dans la plage $[0, 2^{32}]$ avant que l'opération binaire soit appliquée.

Exemple

Ce programme inverse les bits i à j du champ de bits nChampBit :

```
// Calculer le masque pour les bits i à j (voir l'exemple bOr)
nMasque=(power(2,j-i+1)-1)*power(2,i)
// Inverser les bits i à j à l'aide du masque
nChampBit=bXor(nChampBit,nMasque)
```

num toBinary(num nValeur[], num nTailleValeur, string sFormatDonnées, num& nOctetDonnées[])

num fromBinary(num nOctetDonnées[], num nTailleDonnées, string sFormatDonnées, num& nValeur[])

Fonction

L'instruction **toBinary/fromBinary** a pour rôle de permettre l'échange de valeurs numériques entre deux appareils à l'aide d'une connexion série ou réseau. Les valeurs numériques sont tout d'abord codées sous la forme d'un flux d'octets. Les octets sont ensuite envoyés à l'appareil correspondant. Enfin, l'appareil correspondant décode les octets pour récupérer les valeurs numériques initiales. Différents codages binaires de valeurs numériques sont possibles.

L'instruction **toBinary** code les valeurs numériques sous forme de tableau d'octets (champ de bits de 8 bits, nombre entier positif dans la plage [0, 255]) selon les spécifications du format de données **sFormatDonnées**. Le nombre de valeurs numériques **nValeur** à coder est donné par le paramètre **nTailleValeur**. Le résultat est enregistré dans le tableau **nOctetDonnées** et le tableau renvoie le nombre d'octets codés dans ce tableau.

Une erreur d'exécution est générée si le nombre de valeurs à coder **nTailleValeur** dépasse la taille de **nValeur**, si le format spécifié n'est pas pris en charge ou si le tableau de résultats **nOctetDonnées** n'est pas assez grand pour coder toutes les données d'entrée.

L'instruction **fromBinary** décode un tableau d'octets en valeurs numériques **nValeur**, selon les spécifications du format de données **sFormatDonnées**. Le nombre d'octets à décoder est donné par le paramètre **nTailleDonnées**. Le résultat est enregistré dans le tableau **nValeur** et l'instruction renvoie le nombre de valeurs dans ce tableau. Si certaines données binaires sont altérées (octets en dehors de la plage [0, 255] ou codage de la virgule flottante invalide), l'instruction renvoie l'opposé du nombre de valeurs correctement décodées (valeur négative).

Une erreur d'exécution se produit si le nombre d'octets à décoder **nTailleDonnées** dépasse la taille de **nOctetDonnées**, si le format spécifié n'est pas pris en charge ou si le tableau de résultat **nValeur** n'est pas assez grand pour décoder toutes les données d'entrée.

Les codages binaires pris en charge sont indiqués dans le tableau ci-dessous :

- Le signe "-" indique le codage d'un nombre entier signé (le dernier bit du champ de bits code le signe de la valeur).
- Le chiffre indique le nombre d'octets pour le codage de chaque valeur numérique.
- L'extension ".0" marque le codage des valeurs de virgule flottante (les codages à simple et double précision selon IEEE 754 sont pris en charge).
- La dernière lettre indique l'ordre des octets : "l" pour "little endian" (le codage commence par le bit de plus faible poids), "b" pour "big endian" (le codage commence par le bit de plus fort poids). Le codage "big endian" est standard pour les applications en réseau (TCP/IP).

"-1"	Octet signé
"1"	Octet non signé
"-2l"	Mot signé, little endian
"-2b"	Mot signé, big endian
"2l"	Mot non signé, little endian
"2b"	Mot non signé, big endian
"-4l"	Double mot signé, little endian
"-4b"	Double mot signé, big endian
"4l"	Double mot non signé, little endian
"4b"	Double mot non signé, big endian
"4.0l"	Valeur à virgule flottante simple précision, little endian
"4.0b"	Valeur à virgule flottante simple précision, big endian
"8.0l"	Valeur à virgule flottante double précision, little endian
"8.0b"	Valeur à virgule flottante double précision, big endian

Le format natif de **VAL 3** pour les données numériques est le codage à double précision. Ce format doit être utilisé pour échanger des valeurs numériques sans perte de précision.

Exemple

Le premier programme code une donnée **trsf trShiftOut** sous la forme d'un tableau d'octets **nByteOut** et l'envoi par la liaison **siTcpClient**. Le deuxième programme lit les octets sur la connexion **siTcpServer** et les reconvertit en **trsf trShiftIn**.

```
// ---- Programme pour envoyer une trsf ----
// Copie les coordonnées de la trsf dans une variable numérique tampon
nTrsfOut[0]=trShiftOut.x
nTrsfOut[1]=trShiftOut.y
nTrsfOut[2]=trShiftOut.z
nTrsfOut[3]=trShiftOut.rx
nTrsfOut[4]=trShiftOut.ry
nTrsfOut[5]=trShiftOut.rz
// Code 6 valeurs numériques (virgule flottante double précision, donc 8 octets) dans 6*8=48 octets dans le tableau
nByteOut[48]
toBinary(nTrsfOut, 6, "8.0b", nByteOut)
// Envoyer le tableau nByte (48 octets) sur tcpClient
sioSet(siTcpClient, nByteOut)

// ---- Programme pour lire une trsf ----
nb=0
i=0
while (nb<48)
  nb=sioGet(siTcpServer, nByteIn[i])
  if (nb>0)
    i=i+nb
  else
    // Erreur de communication
    return
  endif
endwhile
if (fromBinary(nByteIn, 48, "8.0b", nTrsfIn) != 6)
  // Données altérées
  return
else
  trShiftIn.x=nTrsfIn[0]
  trShiftIn.y=nTrsfIn[1]
  trShiftIn.z=nTrsfIn[2]
  trShiftIn.rx=nTrsfIn[3]
  trShiftIn.ry=nTrsfIn[4]
  trShiftIn.rz=nTrsfIn[5]
endif
```

3.4. TYPE STRING

3.4.1. DÉFINITION

Les variables de type chaîne de caractères permettent de stocker des textes. Le type chaîne permet d'utiliser l'ensemble de caractères standard Unicode. On notera que l'affichage correct d'un caractère Unicode dépend des polices de caractères installées sur l'appareil d'affichage.

Une chaîne est mémorisée dans 128 octets ; le nombre maximum de caractères dans une chaîne dépend des caractères utilisés, le codage interne (Unicode UTF8) pouvant utiliser entre 1 octet (pour les caractères ASCII) et 4 octets pour les caractères (3 pour les caractères chinois).

La longueur maximale d'une chaîne ASCII est donc de 128 caractères, celle d'une chaîne en chinois de 42 caractères.

La valeur d'initialisation par défaut des variables de type chaîne est "" (chaîne vide).

3.4.2. OPÉRATEURS

Par ordre de priorité croissant :

string <string& sVariable> = <string sChaîne>	Affecte sChaîne à la variable sVariable et renvoie sChaîne .
bool <string sChaîne1> != <string sChaîne2>	Renvoie true si sChaîne1 et sChaîne2 ne sont pas identiques, sinon false .
bool <string sChaîne1> == <string sChaîne2>	Renvoie true si sChaîne1 et sChaîne2 sont identiques, sinon false .
string <string sChaîne1> + <string sChaîne2>	Renvoie les premiers caractères (limité à 128 octets) de sChaîne1 concaténés avec sChaîne2 .

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions. `nLen=len(sChaîne="hello wild world")` doit être remplacé par `sChaîne="hello wild world"; nLen=len(sChaîne)`.

3.4.3. INSTRUCTIONS

string toString(string sFormat, num nValeur)

Fonction

Cette instruction renvoie une chaîne de caractères représentant **nValeur** selon le format d'affichage **sFormat**.

Le format est "**size.précision**", où **size** est la taille minimale du résultat (des espaces sont ajoutés en début de chaîne au besoin), et **précision** est le nombre de chiffres significatifs après la virgule (les **0** en fin de chaîne sont remplacés par des espaces). Par défaut, **size** et **précision** valent **0**. La partie entière de la valeur n'est jamais tronquée, même si sa longueur d'affichage est plus grande que **size**.

Exemple

renvoie

```
nPi = 3.141592654
toString(".4", nPi) renvoie "3.1416"
toString("8", nPi) renvoie "      3" (7 espaces avant le '3')
toString("8.4", nPi) renvoie "  3.1416" (2 espaces avant le '3')
toString("8.4", 2.70001) renvoie "  2.7   " (2 espaces avant le '2', 3 espaces après le '7')
toString("", nPi) renvoie "3"
toString("1.2", 1234.1234) renvoie "1234.12"
```

Voir aussi

string chr(num nCodePoint)

string toNum(string sChaîne, num& nValeur, bool& bRapport)

`string toNum(string sChaîne, num& nValeur, bool& bRapport)`

Fonction

Cette instruction trouve le **nValeur** numérique représenté au début du **sChaîne** spécifié et renvoie **sChaîne** dans lequel tous les caractères ont été supprimés jusqu'à la représentation suivante d'une valeur numérique.

Si le début de **sChaîne** ne représente pas une valeur numérique, **bRapport** vaut **false** et **nValeur** n'est pas modifié ; sinon, **bRapport** vaut **true**.

Exemple

```
toNum("10 20 30", nVal, bOk) renvoie "20 30", nVal est égal à 10, bOk est égal à true
toNum("a10 20 30", nVal, bOk) renvoie "a10 20 30", nVal est inchangé, bOk est égal à false
toNum("10 end", nVal, bOk) renvoie "", nVal est égal à 10, bOk est égal à true
```

Ce programme affiche successivement 90, 0, -7.6, 17.3

```
sBuffer = "+90 0.0 -7.6 17.3"
do
    sBuffer = toNum(sBuffer, nVal, bOk)
    putln(nVal)
until (sBuffer=="") or (bOk != true)
```

Voir aussi

`string toString(string sFormat, num nValeur)`

string chr(num nCodePoint)

Fonction

Cette instruction renvoie la chaîne faite du caractère de code de point Unicode, s'il s'agit d'un code de point Unicode valide. Dans le cas contraire, il renvoie une chaîne vide.

Le tableau ci-dessous indique les points Unicode inférieurs à **128** (il correspond au tableau de caractères **ASCII**). Les caractères dans les cases grises sont des codes de contrôle qui peuvent être remplacés par un point d'interrogation quand la chaîne est affichée.

Tous les points de code Unicode valides sont pris en charge par le type de chaîne **VAL 3**. L'affichage du caractère dépend toutefois des polices de caractères installées sur l'appareil d'affichage. La liste complète des caractères Unicode est accessible à l'adresse <http://www.unicode.org> (chercher dans les "Code Charts").

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NUL	SOH	STX	ETX	EOT	ENQ	ACQ	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
" "	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	()	~	DEL

Exemple

chr(65) renvoie "A"

Voir aussi

num asc(string sTexte, num nPosition)

num asc(string sTexte, num nPosition)

Fonction

Cette instruction renvoie le code de point Unicode du caractère d'index **nPosition**.

Elle renvoie -1 si **nPosition** est négatif ou supérieur à la longueur de texte spécifiée.

Exemple

```
asc("A", 0) renvoie 65
```

Voir aussi

string chr(num nCodePoint)

string left(string sTexte, num nTaille)

Fonction

Cette instruction renvoie les **nTaille** premiers caractères de **sTexte**. Si **nTaille** est plus grand que la longueur de **sTexte**, l'instruction renvoie **sTexte**.

Une erreur d'exécution est générée si **nTaille** est négatif.

Exemple

```
left("salut tout le monde", 5) renvoie "salut"
```

```
left("salut tout le monde", 20) renvoie "salut tout le monde"
```

string right(string sTexte, num nTaille)

Fonction

Cette instruction renvoie les **nTaille** derniers caractères de **sTexte**. Si le nombre spécifié est plus grand que la longueur de **sTexte**, l'instruction renvoie **sTexte**.

Une erreur d'exécution est générée si **nTaille** est négatif.

Exemple

```
right("salut tout le monde", 5) renvoie "monde"
```

```
right("salut tout le monde", 20) renvoie "salut tout le monde"
```

string mid(string sTexte, num nTaille, num nPosition)

Fonction

Renvoie **nTaille** caractères de **sTexte** à partir du caractère d'index **nPosition**, en s'arrêtant à la fin de **sTexte**.

Une erreur d'exécution est générée si **nTaille** ou **nPosition** est négatif.

Exemple

```
mid("salut joli monde", 4, 6) renvoie "joli"
```

```
mid("salut joli monde", 20, 6) renvoie "joli monde"
```

string insert(string sTexte, string sInsertion, num nPosition)

Fonction

Cette instruction renvoie **sTexte** où **sInsertion** est inséré après le caractère d'index **nPosition**. Si **nPosition** est plus grand que la taille de **sTexte**, **sInsertion** est ajoutée en fin de **sTexte**. Le résultat est tronqué s'il dépasse 128 octets.

Une erreur d'exécution est générée si **nPosition** est négatif.

Exemple

`insert("salut tout le monde", "joli", 6)` renvoie "salut joli monde"

string delete(string sTexte, num nTaille, num nPosition)

Fonction

Cette instruction renvoie **sTexte** où **nTaille** a été supprimé du caractère d'index **nPosition**. Si **nPosition** est supérieur à la longueur de **sTexte**, l'instruction renvoie **sTexte**.

Une erreur d'exécution est générée si **nTaille** ou **nPosition** est négatif.

Exemple

`delete("salut joli monde", 5, 6)` renvoie "salut tout le monde"

string replace(string sTexte, string sRemplacement, num nTaille, num nPosition)

Fonction

Cette instruction renvoie **sTexte** où **nTaille** caractères ont été remplacés dans le caractère d'index **nPosition** par **sRemplacement**. Si **nPosition** est supérieur à la longueur de **sTexte**, l'instruction renvoie **sTexte**.

Une erreur d'exécution est générée si **nTaille** ou **nPosition** est négatif.

Exemple

`replace("salut • monde", "joli", 1, 6)` renvoie "salut joli monde"

num find(string sTexte1, string sTexte2)

Fonction

Cette instruction renvoie l'index (entre 0 et 127) du premier caractère dans la première occurrence de **sTexte2** dans **sTexte1**. Si **sTexte2** n'apparaît pas dans **sTexte1**, l'instruction renvoie -1.

Exemple

`find("salut joli monde", "joli")` renvoie 6

num len(string sTexte)

Fonction

Cette instruction renvoie le nombre de caractères dans **sTexte**.

Exemple

`len("salut joli monde")` renvoie 16

Voir aussi

`num getDisplayLen(string sTexte)`

3.5. TYPE DIO

3.5.1. DÉFINITION

Les variables de type **dio** servent à interfacer une application **VAL 3** avec les entrées et sorties numériques du système. Une variable **dio** enregistre un lien vers une entrée ou sortie numérique du système, ou "adresse physique".

Toutes les instructions utilisant une variable de type **dio** non liée à une entrée/sortie déclarée dans le système génèrent une erreur d'exécution. La valeur d'initialisation par défaut des variables de type **dio** est un lien indéfini. Le lien d'une variable **dio** peut être initialisé à partir d'une autre variable **dio**, du **MCP** du robot, ou à l'aide de **VAL 3 Studio** dans la **Stäubli Robotics Suite**.

3.5.2. OPÉRATEURS

Par ordre de priorité croissant :

bool < dio diSortie> = < bool bCondition>	Affecte bCondition à l'état de diSortie , et renvoie bCondition . Une erreur d'exécution est générée si diSortie n'est pas liée à une sortie système.
bool < dio diEntrée1> != < bool bEntrée2>	Renvoie true si diEntrée1 et bEntrée2 ne sont pas dans le même état, sinon renvoie false .
bool < dio diEntrée> != < bool bCondition>	Renvoie true si l'état de diEntrée n'est pas égal à bCondition , sinon renvoie false .
bool < dio diEntrée> == < bool bCondition>	Renvoie true si l'état de diEntrée est égal à bCondition , sinon renvoie false .
bool < dio diEntrée1> == < dio diEntrée2>	Renvoie true si diEntrée1 et diEntrée2 sont dans le même état, sinon renvoie false .

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions. **if**(diOutput=dilnInput) serait interprété comme : diOutput=dilnInput; **if**(diOutput==true). Cependant, l'intention était souvent d'écrire : **if**(diOutput==dilnInput), ce qui est très différent !

ATTENTION :

L'opérateur '=' entre les deux variables **dio** n'existe plus avec **VAL 3 s7** pour des raisons de cohérence avec les autres opérateurs '=' (voir la définition de l'opérateur '=' pour les types d'utilisateur). Il peut facilement être remplacé par l'opérateur '=' entre un **dio** et un **bool** : le diOut = diln des versions antérieures de **VAL 3** peut être remplacé par diOut = (diln==true).

3.5.3. INSTRUCTIONS

`void dioLink(dio& diVariable, dio diSource)`

Fonction

Cette instruction lie **diVariable** à l'entrée/sortie à laquelle **diSource** est lié.

Exemple

Cette application utilise un signal qui peut être configuré à l'aide de différents appareils matériels. Le programme ci-dessous détermine quel appareil est installé afin d'initialiser la variable diSignal qui est utilisée ensuite dans le reste de l'application.

```
if(ioStatus(diDevice1Signal)>=0)
// appareil 1 installé : utiliser cet appareil
dioLink(diSignal, diDevice1Signal)
elseIf (ioStatus(diDevice2Signal)>=0)
// appareil 2 installé : utiliser cet appareil
dioLink(diSignal, diDevice2Signal)
else
putln("Erreur : pas d'appareil e/s installé")
endif
```

`num dioGet(dio diTableau[])`

Fonction

Cette instruction renvoie la valeur numérique de **diTableau** lue sous la forme d'un entier écrit en code binaire, à savoir : $\text{diTableau}[0] + 2 * \text{diTableau}[1] + 4 * \text{diTableau}[2] + \dots + 2^k * \text{diTableau}[k]$, où $\text{diTableau}[i] = 1$ si $\text{diTableau}[i]$ est **true**, 0 sinon.

Une erreur d'exécution est générée si un élément de **diTableau** n'est pas lié à une entrée/sortie système.

Exemple

```
diArray[0] = false
diArray[1] = true
diArray[2] = false
diArray[3] = true
dioGet(diArray) renvoie 10 = 0+2*1+4*0+8*1
```

Voir aussi

`num dioSet(dio diTableau[], num nValeur)`

num dioSet(dio diTableau[], num nValeur)

Fonction

Cette instruction convertit la partie entière de **nValeur** en code binaire, l'affecte aux sorties de **diTableau** et renvoie la valeur correspondante, à savoir :

diTableau[0] + 2 * diTableau[1] + 4 * diTableau[2] + ... + 2^k * diTableau[k], où **diTableau[i] = 1** si **diTableau[i]** est **true**, **0** sinon.

Une erreur d'exécution est générée si un élément de **diTableau** n'est pas lié à une sortie système.

Exemple

En utilisant **di4bitsArray**, tableau de taille 4 :

dioSet(di4bitsArray, 10) renvoie 10

dioSet(di4bitsArray, 26) renvoie 10, parce que 26 demande 5 bits en codage binaire : $10 = 26 - 2^4$

Voir aussi

num dioGet(dio diTableau[])

num ioStatus(dio diEntréeSortie)

Fonction

Cette instruction renvoie un nombre positif si la variable d'entrée-sortie spécifiée fonctionne et un nombre négatif si elle est en erreur. La valeur renvoyée détaille le statut de l'entrée-sortie :

0	L'entrée-sortie fonctionne.
1	L'entrée-sortie fonctionne mais elle est verrouillée par l'opérateur. Les entrées ont alors une valeur fixe (contrôlée par l'opérateur) qui peut être différente de la valeur matérielle. Les sorties ont alors une valeur fixe contrôlée par l'opérateur : l'écriture sur la sortie n'a aucun effet. Le mode de verrouillage est un moyen de débogage.
2	L'entrée-sortie est simulée (entrée-sortie logicielle, sans effet sur le matériel).
-1	L'entrée-sortie ne fonctionne pas parce que le lien (adresse physique) n'est pas défini.
-2	L'entrée-sortie ne fonctionne pas parce que le lien (adresse physique) ne correspond à aucune entrée-sortie du système. Le matériel correspondant à l'adresse physique n'est pas installé ou n'a pas pu être initialisé.
-3	L'entrée-sortie ne fonctionne pas parce que l'appareil d'entrée-sortie est en erreur.

Exemple

Cette application utilise un signal qui peut être configuré à l'aide de différents appareils matériels. Le programme ci-dessous détermine quel appareil est installé afin d'initialiser la variable **diSignal** qui est utilisée ensuite dans le reste de l'application.

```

if(ioStatus(diDevice1Signal)>=0)
// appareil 1 installé : utiliser cet appareil
  dioLink(diSignal, diDevice1Signal)
elseif (ioStatus(diDevice2Signal)>=0)
// appareil 2 installé : utiliser cet appareil
  dioLink(diSignal, diDevice2Signal)
else
  putln("Erreur : pas d'appareil e/s installé")
endif

```

Voir aussi

```
num ioStatus(dio diEntréeSortie, string& sDescription, string& sChemin physique)
num ioStatus(aio aiEntréeSortie)
num ioBusStatus(string& sDescription de l'erreur[])
```

num ioStatus(dio diEntréeSortie, string& sDescription,
string& sChemin physique)

Fonction

Cette instruction se comporte exactement comme l'instruction ioStatus décrite plus haut mais renvoie en outre le texte descriptif et le lien (adresse physique) de l'entrée-sortie spécifiée.

La description est un texte libre, défini à l'aide des outils de commande de l'entrée-sortie. Le format du lien physique dépend de l'appareil d'entrée-sortie. Il prend habituellement la forme : 'deviceName\moduleName\ioAddress'.

Exemple

Ce programme teste un signal et affiche des informations d'erreur si celui-ci ne fonctionne pas.

```
if ioStatus(diSignal, sDescription, sPath)<0
    putln("Signal "+sPath+ "en erreur")
    putln("Description :"+sDescription)
endif
```

Voir aussi

```
num ioStatus(aio aiEntréeSortie)
num ioStatus(dio diEntréeSortie)
num ioBusStatus(string& sDescription de l'erreur[])
```

3.6. TYPE AIO

3.6.1. DÉFINITION

Les variables de type **aio** servent à interfacer une application **VAL 3** avec les entrées et sorties analogiques du système. Une variable **aio** enregistre un lien vers une entrée ou sortie analogique du système, ou "adresse physique".

Toutes les instructions utilisant une variable de type **aio** non liée à une entrée/sortie déclarée dans le système génèrent une erreur d'exécution. La valeur d'initialisation par défaut des variables de type **aio** est un lien indéfini. Le lien d'une variable **aio** peut être initialisé à partir d'une autre variable **aio**, du **MCP** du robot, ou à l'aide de **VAL 3 Studio** dans la **Stäubli Robotics Suite**.

3.6.2. INSTRUCTIONS

void aioLink(aio& aiVariable, aio aiSource)

Fonction

Cette instruction lie **aiVariable** à l'entrée/sortie à laquelle **aiSource** est lié.

Exemple

Cette application utilise un signal qui peut être configuré à l'aide de différents appareils matériels. Le programme ci-dessous détermine quel appareil est installé afin d'initialiser la variable **aiSignal** qui est utilisée ensuite dans le reste de l'application.

```
if(ioStatus(aiDevice1Signal)>=0)
// appareil 1 installé : utiliser cet appareil
aioLink(aiSignal, aiDevice1Signal)
elseif (ioStatus(aiDevice2Signal)>=0)
// appareil 2 installé : utiliser cet appareil
aioLink(aiSignal, aiDevice2Signal)
else
println("Erreur : pas d'appareil e/s installé")
endif
```

num aioGet(aio aiEntrée)

Fonction

Cette instruction renvoie la valeur numérique d'**aiEntrée**.

Une erreur d'exécution est générée si **aiEntrée** n'est pas liée à une entrée/sortie système.

Exemple

aioGet(aiSensor) renvoie la valeur actuelle du capteur

Voir aussi

num aioSet(aio aiSortie, num nValeur)

num aioSet(aio aiSortie, num nValeur)

Fonction

Cette instruction affecte **nValeur** à **aiSortie** et renvoie **nValeur**. Si la valeur réglée ne correspond pas à la plage de valeurs de l'**aio**, le nombre donné est la valeur réelle de la sortie **aio**.

Une erreur d'exécution est générée si **aiSortie** n'est pas liée à une sortie système.

Exemple

aioSet(aiCommand, -12.3) écrit -12.3 dans la commande de sortie et renvoie -12.3 si **aiCommand** est une sortie à virgule flottante.

aioSet(aiCommand, 12.3) écrit 12 sur la commande de sortie et renvoie 12 si **aiCommand** est une sortie entière.

Voir aussi

num aioGet(aio aiEntrée)

num ioStatus(aio aiEntréeSortie)

Fonction

Cette instruction renvoie un nombre positif si la variable d'entrée-sortie spécifiée fonctionne et un nombre négatif si elle est en erreur. La valeur renvoyée détaille le statut de l'entrée-sortie :

0 : L'entrée-sortie fonctionne.

1 : L'entrée-sortie fonctionne mais elle est verrouillée par l'opérateur. Les entrées ont alors une valeur fixe (contrôlée par l'opérateur) qui peut être différente de la valeur matérielle. Les sorties ont alors une valeur fixe contrôlée par l'opérateur : l'écriture sur la sortie n'a aucun effet. Le mode de verrouillage est un moyen de débogage.

2 : L'entrée-sortie est simulée (entrée-sortie logicielle, sans effet sur le matériel).

-1 : L'entrée-sortie ne fonctionne pas parce que le lien (adresse physique) n'est pas défini.

-2 : L'entrée-sortie ne fonctionne pas parce que le lien (adresse physique) ne correspond à aucune entrée-sortie du système. Le matériel correspondant à l'adresse physique n'est pas installé ou n'a pas pu être initialisé.

-3 : L'entrée-sortie ne fonctionne pas parce que l'appareil d'entrée-sortie est en erreur.

Exemple

Cette application utilise un signal qui peut être configuré à l'aide de différents appareils matériels. Le programme ci-dessous détermine quel appareil est installé afin d'initialiser la variable aiSignal qui est utilisée ensuite dans le reste de l'application.

```
if(ioStatus(aiDevice1Signal)>=0)
// appareil 1 installé : utiliser cet appareil
aioLink(aiSignal, aiDevice1Signal)
elseIf (ioStatus(aiDevice2Signal)>=0)
// appareil 2 installé : utiliser cet appareil
aioLink(aiSignal, aiDevice2Signal)
else
putln("Erreur : pas d'appareil e/s installé")
endif
```

Voir aussi

num ioStatus(dio diEntréeSortie)

**num ioStatus(aio diEntréeSortie, string& sDescription,
string& sChemin physique)**

Fonction

Cette instruction se comporte exactement comme l'instruction ioStatus décrite plus haut mais renvoie en outre le texte descriptif et le lien (adresse physique) de l'entrée-sortie spécifiée.

La description est un texte libre, défini à l'aide des outils de commande de l'entrée-sortie. Le format du lien physique dépend de l'appareil d'entrée-sortie. Il prend habituellement la forme : 'deviceName\moduleName\ioAddress'.

Exemple

Ce programme teste un signal et affiche des informations d'erreur si celui-ci ne fonctionne pas.

```
if ioStatus(aiSignal, sDescription, sPath)<0
putln("Signal "+sPath+ "en erreur")
putln("Description :"+sDescription)
endif
```

Voir aussi

num ioStatus(aio aiEntréeSortie)

num ioStatus(dio diEntréeSortie)

3.7. TYPE SIO

3.7.1. DÉFINITION

Le type **sio** permet de lier une variable **VAL 3** à une entrée-sortie série du système ou une connexion par socket Ethernet. Une entrée-sortie **sio** est caractérisée par :

- Des paramètres propres au type de communication, définis dans le système
- Un caractère de fin de chaîne, pour permettre l'utilisation du type **string**
- Un délai d'attente de communication

Les entrées-sorties série du système sont toujours actives. Les connexions sur embase Ethernet sont ouvertes au moment du premier accès d'un programme **VAL 3** en lecture ou en écriture. Elles sont fermées automatiquement quand l'application **VAL 3** est fermée.

Toutes les instructions utilisant une variable de type **sio** non liée à une entrée/sortie déclarée dans le système génèrent une erreur d'exécution. La valeur d'initialisation par défaut des variables de type **sio** est un lien indéfini. Le lien d'une variable **sio** peut être initialisé à partir d'une autre variable **sio**, du **MCP** du robot, ou à l'aide de **VAL 3 Studio** dans la **Stäubli Robotics Suite**.

3.7.2. OPÉRATEURS

Une erreur d'exécution est générée quand le délai de communication est dépassé lors de la lecture ou de l'écriture de l'entrée/sortie.

string < sio siSortie> = < string sTexte>	Ecrit successivement dans siSortie les caractères sTexte des codes Unicode UTF8 , suivis du caractère de fin de chaîne, et renvoie sTexte .
num < sio siSortie> = < num nDonnée>	Ecrit sur siSortie l'entier le plus proche de nDonnée , modulo 256 , et renvoie la valeur effectivement envoyée.
num < num nDonnées> = < sio siEntrée>	Lit un octet sur siEntrée et affecte la valeur de l'octet à nDonnées .
string < string sTexte> = < sio siEntrée>	Lit dans siEntrée une chaîne de caractères Unicode UTF8 et affecte cette chaîne à sTexte . Les caractères non supportés par le type string sont ignorés. La chaîne est terminée lorsque le caractère de fin de chaîne est lu ou lorsque sTexte atteint la taille maximale d'un string (128 octets). Le caractère de fin de chaîne n'est pas recopié dans sTexte .

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions. nLen=len(sString=silnput) doit être remplacé par sString=silnput; nLen=len(sString).

3.7.3. INSTRUCTIONS

void sioLink(sio& siVariable, sio siSource)

Fonction

Cette instruction lie **siVariable** à l'entrée/sortie série du système auquel **siSource** est lié.

Voir aussi

void dioLink(dio& diVariable, dio diSource)

void aioLink(aio& aiVariable, aio aiSource)

num clearBuffer(sio siVariable)

Fonction

Cette instruction vide la mémoire de lecture **siVariable** et renvoie le nombre de caractères effacés

Pour une connexion sur embase Ethernet, **clearBuffer** désactive (ferme) l'embase. **clearBuffer** renvoie **-1** si l'embase a déjà été désactivée.

Une erreur d'exécution est générée si **siVariable** n'est pas reliée à une liaison série système ou Ethernet.

num sioGet(sio siEntrée, num& nDonnées[])

Fonction

Cette instruction lit un caractère unique ou un tableau de caractères sur **siEntrée** et renvoie le nombre de caractères lus.

La séquence de lecture s'arrête lorsque le tableau **nDonnées** est plein ou lorsque le tampon de lecture des entrées est vide.

Pour une connexion sur embase Ethernet, **sioGet** essaie d'abord d'ouvrir une connexion s'il n'en existe pas une ouverte. Lorsque le délai d'attente de communication d'entrée est atteint, **sioGet** retourne **-1**. Si la connexion est ouverte mais que le tampon de lecture d'entrée ne contient pas de données, **sioGet** attend la réception de données ou la fin du délai d'exécution.

Une erreur d'exécution est générée si **siEntrée** n'est pas liée à un port série du système ou Ethernet, ou si **nDonnées** n'est pas une variable **VAL 3**.

num sioSet(sio siSortie, num& nDonnées[])

Fonction

Cette instruction écrit un caractère ou un tableau de caractères sur **siSortie** et renvoie le nombre de caractères écrits.

Les valeurs numériques sont converties avant transmission en entier entre **0** et **255**, en prenant l'entier le plus proche modulo **256**.

Pour une connexion sur embase Ethernet, **sioSet** essaie d'abord d'ouvrir une connexion s'il n'en existe pas une ouverte. Lorsque le délai d'attente de communication de sortie est atteint, **sioSet** retourne **-1**. Le nombre de caractères écrits peut être inférieur à la taille de **nDonnées** si un erreur de communication est détectée.

Une erreur d'exécution est générée si **siSortie** n'est pas liée à un port série du système ou à une embase Ethernet.

num sioCtrl(sio siCanal, string nParamètre, *valeur)

Fonction

Cette instruction modifie un paramètre de communication de l'entrée/sortie série/Ethernet siCanal spécifiée.

Pour les lignes série, le matériel peut ne pas accepter certains paramètres ou certaines valeurs de paramètres : se reporter au manuel du contrôleur.

L'instruction renvoie :

0	Le paramètre a été modifié
-1	Le paramètre n'est pas défini
-2	La valeur du paramètre n'a pas le type attendu
-3	La valeur du paramètre n'est pas prise en charge
-4	Le canal série n'est pas prêt à appliquer le changement de paramètre (il doit d'abord être arrêté)
-5	Le paramètre n'est pas défini pour ce type de canal

Les paramètres pris en charge sont indiqués dans le tableau ci-dessous :

Nom du paramètre	Type de paramètre	Description
"port"	num	(Pour client ou serveur TCP) Port TCP
"target"	string	(Pour client TCP) Adresse IP du serveur TCP à atteindre, par exemple "192.168.0.254"
"clients"	num	(Pour serveur TCP) Nombre maximal de clients simultanés sur le serveur
"endOfString"	num	(Pour ligne série, client et serveur TCP) Code ASCII pour le caractère de fin de chaîne à utiliser avec les opérateurs '=' (dans la plage [0, 255])
"timeout"	num	(Pour ligne série, client et serveur TCP) Temps de réponse maximal pour le canal de communication. 0 signifie : pas de limite de délai.
"baudRate"	num	(Pour la ligne série) Vitesse de communication
"parity"	string	(Pour la ligne série) Contrôle de parité : "none", "even" ou "odd"
"bits"	num	(Pour la ligne série) Nombre de bits par octet (5, 6, 7 ou 8)
"stopBits"	num	(Pour la ligne série) Nombre de bits d'arrêt par octet (1 ou 2)
"mode"	string	(Pour la ligne série) Mode de communication : "rs232" ou "rs422"
"flowControl"	string	(Pour la ligne série) Contrôle du débit : "none" ou "hardware"
"nagle"	bool	(Pour le client ou serveur TCP) Activation (par défaut) ou désactivation de l'optimisation de nagle. Si l'optimisation de nagle est désactivée, le temps de réponse est amélioré mais la charge sur le système est accrue.

Exemple

Ce programme configure les paramètres principaux d'une ligne série.

```
sioCtrl(siPortSerial1, "baudRate", 115200)
sioCtrl(siPortSerial1, "bits", 8)
sioCtrl(siPortSerial1, "parity", "none")
sioCtrl(siPortSerial1, "stopBits", 1)
sioCtrl(siPortSerial1, "timeout", 0)
sioCtrl(siPortSerial1, "endOfString", 13)
```

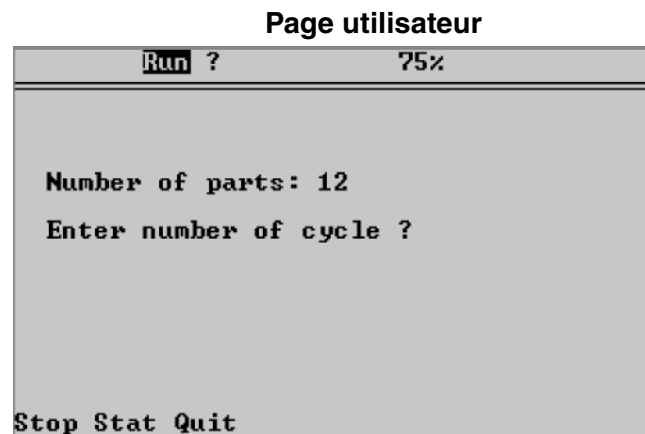

CHAPITRE 4

INTERFACE UTILISATEUR

4.1. PAGE UTILISATEUR

Les instructions d'interface utilisateur du langage **VAL 3** permettent :

- l'affichage de messages sur une page du pendant de commande manuelle (MCP) réservée à l'application
- acquisition de séquences de touches sur le clavier du **MCP**



La page utilisateur comprend 14 lignes de 40 colonnes. La dernière ligne sert souvent à créer des menus à l'aide de la touche associée. Une ligne supplémentaire est disponible pour l'affichage d'un titre.

4.2. TYPE D'ÉCRAN

Le contexte de la page utilisateur (informations affichées et séquences de touches) peut être associé à une variable de type **screen**. Il est ainsi plus facile de créer et d'entretenir plusieurs écrans dans une activation, par exemple un écran de production, un écran de maintenance et un écran de débogage.

Les touches 'User-Down' et 'User-Up' permettent de passer facilement d'un écran à l'autre. La touche 'User-Shift' appelle le premier écran et 'User-Shift-Up' le dernier.

Le type **screen** occupe une place importante en mémoire et ne peut donc pas être utilisé pour les variables locales, à moins d'augmenter significativement la taille de la mémoire d'exécution de l'application.

4.2.1. SÉLECTION DE L'ÉCRAN UTILISATEUR

L'instruction **userPage()** peut être définie avec une variable **screen** facultative comme paramètre. L'écran affiché par le MCP passe alors à l'écran spécifié. Si aucun paramètre **screen** n'est défini, l'écran affiché passe à l'écran utilisateur par défaut, qui est toujours défini.

4.2.2. ECRIRE DANS UN ÉCRAN UTILISATEUR

Les instructions pour écrire dans l'écran (**title()**, **gotoxy()**, **cls()**, **put()**, **putln()**) peuvent être spécifiées avec une variable **screen** facultative comme premier paramètre. L'opération d'écriture s'effectue ensuite sur l'écran spécifié, sans affecter les autres écrans. L'écran modifié peut ne pas être celui qui est affiché. Dans ce cas, les modifications restent cachées jusqu'à ce que l'on change l'écran actif à l'aide de l'instruction **userPage()**. Si aucun paramètre **screen** n'est défini, l'écran modifié est l'écran utilisateur par défaut, qui est toujours défini.

4.2.3. LECTURE DANS UN ÉCRAN UTILISATEUR

Les instructions d'acquisition des séquences de touches (**get()**, **getKey()**, **isKeyPressed()**) peuvent être spécifiées avec une variable **screen** facultative comme premier paramètre. L'opération d'entrée s'effectue alors dans l'écran spécifié : une séquence de touches ne peut être lue que par l'écran affiché sur le MCP. Les autres écrans ne sont pas affectés. Il est donc possible que plusieurs écrans différents attendent au même moment des séquences de touches différentes : seul l'écran actif (affiché) sera informé des séquences de touches effectives. Si aucun paramètre **screen** n'est défini, l'écran concerné est l'écran utilisateur par défaut, qui est toujours défini.

4.3. INSTRUCTIONS

**void userPage(), void userPage(screen scPage),
void userPage(bool bFixe)**

Fonction

Cette instruction affiche sur l'écran du **MCP** la page utilisateur spécifiée, le cas échéant, ou la page utilisateur par défaut.

Si le paramètre **bFixe** est **true**, seule la page utilisateur sera accessible à l'opérateur, à l'exception de la page de changement de profil accessible par le raccourci clavier "Shift User". Lorsque cette page est affichée, il est possible de stopper l'application par la touche "Stop" si le profil utilisateur courant l'autorise.

Si le paramètre est **false**, les autres pages du contrôleur **MCP** redeviennent accessibles.

**void gotoxy(num nX, num nY),
void gotoxy(screen scPage, num nX, num nY)**

Fonction

Cette instruction place le curseur aux coordonnées (**nX, nY**) sur la page utilisateur spécifiée, le cas échéant, ou sur la page utilisateur par défaut. Le coin en haut à gauche a pour coordonnées (**0,0**) et le coin en bas à droite (**39, 13**).

Le nombre de colonnes **nX** est en modulo **40**. Le nombre de lignes **nY** est en modulo **14**.

Voir aussi

void cls(), void cls(screen scPage)

void cls(), void cls(screen scPage)

Fonction

Cette instruction vide la page utilisateur spécifiée ou par défaut et place le curseur sur (**0,0**).

Voir aussi

void gotoxy(num nX, num nY), void gotoxy(screen scPage, num nX, num nY)

**void setTextMode(num nMode),
void setTextMode(screen scPage, num nMode)**

Fonction

Cette instruction modifie le mode d'affichage de l'écran spécifié, le cas échéant, ou de l'écran par défaut. Le nouveau mode d'affichage n'affecte pas l'affichage en cours mais est appliqué à tous les nouveaux textes, jusqu'à ce qu'un nouveau mode de texte soit défini. Les modes pris en charge sont définis ci-dessous :

0	mode texte standard (noir sur fond blanc)
1	mode vidéo inverse (blanc sur fond noir)
2	mode texte standard clignotant
3	mode vidéo inverse clignotant

Voir aussi

void put(string sTexte), void put(screen scPage, string sTexte) void put(num nValeur), void put(screen scPage, num nValeur), void putln(string sTexte), void putln(screen scPage, string sTexte), void putln(num nValeur), void putln(screen scPage, num nValeur),

`num getDisplayLen(string sTexte)`

Fonction

Cette instruction renvoie la longueur de **sTexte** sur l'écran du **MCP** (nombre de colonnes nécessaire pour afficher **sTexte**).

Pour les chaînes de caractères **ASCII**, la longueur affichée est le nombre de caractères de la chaîne ; `getDisplayLen()` est alors identique à l'instruction `len()`.

Certains caractères (chinois) sont affichés sur deux colonnes adjacentes de l'écran ; `getDisplayLen()` est alors plus grand que la longueur de **sTexte** et peut être utilisé pour contrôler l'alignement de **sTexte** à l'écran.

Voir aussi

`num len(string sTexte)`

```
void put(string sTexte), void put(screen scPage, string sTexte)
void put(num nValeur), void put(screen scPage, num nValeur),
void putln(string sTexte), void putln(screen scPage, string sTexte),
void putln(num nValeur), void putln(screen scPage, num nValeur),
```

Fonction

Cette instruction affiche le **sTexte** ou **nValeur** spécifié (à 3 décimales) à la position du curseur dans la page utilisateur éventuellement spécifiée ou dans la page utilisateur par défaut. Le curseur est ensuite positionné sur le caractère suivant le dernier caractère du message affiché (instruction **put**), ou sur le premier caractère de la ligne suivante (instruction **putln**).

En fin de ligne, l'affichage se poursuit sur la ligne suivante.

En fin de page, l'affichage de la page utilisateur se décale d'une ligne vers le haut.

Voir aussi

void popUpMsg(string sTexte)

bool logMsg(string sTexte)

void title(string sTexte), void title(screen scPage, string sTexte)

```
void title(string sTexte), void title(screen scPage, string sTexte)
```

Fonction

Cette instruction change le titre de la page utilisateur spécifiée, le cas échéant, ou de la page utilisateur par défaut.

La position courante du curseur n'est pas modifiée par l'instruction **title()**.

```
num get(string& sTexte), num get(screen scPage, string& sTexte),
num get(num& nValeur), num get(screen scPage, num& nValeur),
num get(), num get(screen scPage)
```

Fonction

Cette instruction acquiert une chaîne, un nombre ou une clé de panneau de contrôle.

Le paramètre **sTexte** ou **nValeur** est affiché à la position actuelle du curseur et peut être modifié par l'utilisateur. Pour terminer la saisie, on appuie sur une touche du menu ou sur les touches **Return** ou **Esc**.

L'instruction renvoie le code de la touche ayant terminé la saisie.

Sur appui de **Return** ou d'un menu, la variable **sTexte** ou **nValeur** est mise à jour. Sur appui de **Esc**, elle reste inchangée.

Si aucun paramètre n'est passé, l'instruction **get()** attend un appui touche quelconque et retourne son code. La touche appuyée n'est pas affichée.

Dans tous les cas, la position courante du curseur n'est pas modifiée par l'instruction **get()**.

Sans Shift

3	Caps	Space		
283	-	32		
			Ret.	
2	Shift	Esc	Help	
282	-	255	-	270
	Menu	Tab	Up	Bksp
	-	259	261	263
1	User	Left	Down	Right
281	-	264	266	268

Avec Shift

3	Caps	Space		
283	-	32		
			Ret.	
2	Shift	Esc	Help	
282	-	255	-	270
	Menu	UnTab	PgUp	Bksp
	-	260	262	263
1	User	Home	PgDn	End
281	-	265	267	269

Menus (avec ou sans **Shift**)

F1	F2	F3	F4	F5	F6	F7	F8
271	272	273	274	275	276	277	278

Pour les touches standards, le code retourné est le code **ASCII** du caractère correspondant :

Sans Shift									
q	w	e	r	t	y	u	i	o	p
113	119	101	114	116	121	117	105	111	112
a	s	d	f	g	h	j	k	l	<
97	115	100	102	103	104	106	107	108	60
z	x	c	v	b	n	m	.	,	=
122	120	99	118	98	110	109	46	44	61

Avec Shift									
7	8	9	+	*	;	()	[]
55	56	57	43	42	59	40	41	91	93
4	5	6	-	/	?	:	!	{	}
52	53	54	45	47	63	58	33	123	125
1	2	3	0	"	%	-	.	,	>
49	50	51	48	34	37	95	46	44	62

Avec double Shift									
Q	W	E	R	T	Y	U	I	O	P
81	87	69	82	84	89	85	73	79	80
A	S	D	F	G	H	J	K	L	}
65	83	68	70	71	72	74	75	76	125
Z	X	C	V	B	N	M	\$	\	=
90	88	67	86	66	78	77	36	92	61

Exemple

Ce programme lit une valeur numérique validée avec la touche Retour :

```
do
    nKey = get (nValeur)
until (nKey == 270)
```

Voir aussi

`num getKey()`, `num getKey(screen scPage)`

num getKey(), num getKey(screen scPage)

Fonction

Cette instruction acquiert une séquence de touches sur le clavier du panneau de contrôle. Elle renvoie le code de la dernière touche enfoncée depuis le dernier appel `getKey()` ou -1 si aucune touche n'a été actionnée depuis. Une séquence de touches ne peut être détectée que quand la page utilisateur sélectionnée, le cas échéant, ou la page utilisateur par défaut est affichée.

Contrairement à l'instruction `get()`, `getKey()` retourne immédiatement.

La touche actionnée n'est pas affichée et la position du curseur reste inchangée.

Exemple

Ce programme actualise l'affichage de l'horloge système jusqu'à ce qu'on appuie sur une touche :

```
// Réinitialiser d'abord le code de la dernière touche enfoncée
getKey()
while (getKey() == -1)
    gotoxy(0,0)
    put(toString("", clock() * 10))
    delay(0)
endWhile
```

Voir aussi

num get(string& sTexte), num get(screen scPage, string& sTexte), num get(num& nValeur), num
get(screen scPage, num& nValeur), num get(), num get(screen scPage), void userPage(), void
userPage(screen scPage), void userPage(bool bFixe)
bool isKeyPressed(num nCode), bool isKeyPressed(screen scPage, num nCode)

**bool isKeyPressed(num nCode),
bool isKeyPressed(screen scPage, num nCode)**

Fonction

Cette instruction renvoie le statut de la touche indiqué par son code (voir `get()`), **true** si la touche est enfoncée ou **false** dans le cas contraire. Une séquence de touches ne peut être détectée que quand la page utilisateur éventuellement spécifiée ou la page utilisateur par défaut est affichée, sauf dans le cas des touches (1), (2) et (3) qui sont toujours détectées.

Voir aussi

num get(string& sTexte), num get(screen scPage, string& sTexte), num get(num& nValeur), num
get(screen scPage, num& nValeur), num get(), num get(screen scPage), void userPage(), void
userPage(screen scPage), void userPage(bool bFixe)

void popUpMsg(string sTexte)

Fonction

Cette instruction affiche **sTexte** dans une fenêtre "popup" au-dessus de la fenêtre actuelle du **MCP**. Cette fenêtre reste affichée jusqu'à ce qu'elle soit validée par le menu **Ok** ou la touche **Esc**.

Voir aussi

void userPage(), void userPage(screen scPage), void userPage(bool bFixe)
void put(string sTexte), void put(screen scPage, string sTexte) void put(num nValeur), void put(screen
scPage, num nValeur), void putln(string sTexte), void putln(screen scPage, string sTexte), void
putln(num nValeur), void putln(screen scPage, num nValeur),

bool logMsg(string sTexte)

Fonction

Cette instruction inscrit **sTexte** dans l'historique du système (journal d'erreurs). Le message sera enregistré avec la date et l'heure courante. "USR" est ajouté au début de la chaîne pour signaler qu'il s'agit d'un message de l'utilisateur. Certains messages du fichier journal peuvent se perdre si de nombreux messages sont enregistrés pendant la même seconde. L'instruction renvoie alors false de sorte que les messages sont enregistrés plus tard si cela est important.

Exemple

Ce programme assure l'enregistrement du message :

```
while logMsg(sMessage) == false
delay(0)
endWhile
```

Voir aussi

void popUpMsg(string sTexte)

string getProfile()

Fonction

Cette instruction renvoie le nom du profil utilisateur actuel.

Voir aussi

num setProfile(string sLoginUtilisateur, string sMotDePasseUtilisateur)

num **setProfile**(string sLoginUtilisateur,
string sMotDePasseUtilisateur)

Fonction

Cette instruction change le profil utilisateur actuel (avec effet immédiat).

La fonction renvoie :

- 0 : Le profil d'utilisateur spécifié prend effet
- 1 : Le profil d'utilisateur spécifié n'est pas défini
- 2 : Le mot de passe d'utilisateur spécifié est incorrect
- 3 : 'staubli' n'est pas un profil d'utilisateur autorisé avec cette instruction
- 4 : Le profil d'utilisateur en cours est 'staubli' et ne peut pas être changé avec cette instruction

Voir aussi

string getProfile()

string getLanguage()

Fonction

Cette instruction renvoie la langue actuelle du contrôleur du robot.

Exemple

```
switch (getLanguage())  
  case "français"  
    sMessage="Attention!"  
  break  
  case "english"  
    sMessage="Warning!"  
  break  
  case "deutsch"  
    sMessage="Achtung!"  
  break  
  case "italiano"  
    sMessage="Avviso!"  
  break  
  case "español"  
    sMessage="¡Advertencia!"  
  break  
  default  
    sMessage="Warning!"  
  break  
endSwitch
```

Voir aussi

bool setLanguage(string sLangue)

bool setLanguage(string sLangue)

Fonction

Cette instruction modifie la langue actuelle du contrôleur du robot : le nom de la langue spécifiée sLangue doit correspondre au nom d'un fichier de traduction sur le contrôleur. Voir le manuel du contrôleur pour supprimer ou installer une langue dans le contrôleur du robot.

Exemple

Ce programme fait passer la langue du robot au chinois :

```
if (setLanguage ("chinois") == false)
    putln ("La langue chinoise n'est pas disponible sur le contrôleur du robot")
endIf
```

Voir aussi

string getLanguage()

string getDate(string sFormat)

Fonction

Cette instruction renvoie la date et/ou l'heure actuelles du contrôleur du robot. Le paramètre sFormat spécifie le format de la date à renvoyer. Dans cette chaîne, chaque occurrence de certains mots-clés est remplacée par la valeur de date ou d'heure correspondante. Les mots-clés de format acceptés sont indiqués dans le tableau ci-dessous :

Mot-clé	Description
%y	Année en 2 chiffres (00-99), sans le siècle
%Y	Année en 4 chiffres, par exemple 2007
%m	Mois (00-12)
%d	Jour (00-31)
%H	Heure au format 24 (00-23)
%I	Heure au format 12 (01-12)
%p	Indicateur A.M./P.M. pour l'horloge sur 12 heures
%M	Minutes (00-59)
%S	Secondes (00-59)

Exemple

Ce programme affiche la date et l'heure au format "January 01, 2007 13:45:23"

```
switch (getDate ("%m"))
    case "01"
        sMois="Janvier"
        break
    case "02"
        sMois="Février"
        break
    case "03"
        sMois="Mars"
        break
    case "04"
        sMois="Avril"
        break
    case "05"
        sMois="Mai"
        break
    case "06"
        sMois="Juin"
```

```

break
case "07"
    sMois="Juillet"
break
case "08"
    sMois="Août"
break
case "09"
    sMois="Septembre"
break
case "10"
    sMois="October"
break
case "11"
    sMois="Novembre"
break
case "12"
    sMois="Décembre"
break
default
    sMois="???"
break
endSwitch

// Afficher date et heure sous la forme : "January 01, 2007 13:45:23"
println(getDate(sMonth+" %d, %Y %H:%M:%S"))

```


CHAPITRE 5

TÂCHES

5.1. DÉFINITION

Une tâche est un programme en cours d'exécution. Plusieurs tâches peuvent s'exécuter dans une même application, et c'est habituellement le cas.

Dans une application, on trouvera typiquement une tâche pour les déplacements du bras, une tâche automate, une tâche pour l'interface utilisateur, une tâche pour le suivi des signaux de sécurité, des tâches de communication...

Une tâche est caractérisée par les éléments suivants :

- un nom : un identifiant de tâche unique à l'intérieur de la librairie ou de l'application
- une priorité ou une période : paramètre pour le séquençement des tâches
- un programme : le point d'entrée (et de sortie) de la tâche
- un état : actif ou arrêté
- la prochaine instruction à exécuter (et son contexte)

5.2. REPRISE APRÈS UNE ERREUR D'EXÉCUTION

Quand une instruction provoque une erreur d'exécution, la tâche est interrompue. L'instruction `taskStatus()` sert à diagnostiquer l'erreur d'exécution. Il est alors possible de redémarrer la tâche avec l'instruction `taskResume()`. Si l'erreur d'exécution peut être corrigée, la tâche peut reprendre à partir de la ligne d'instruction où elle s'est arrêtée. Dans le cas contraire, il est nécessaire de repartir soit avant, soit après cette ligne d'instruction.

Démarrage et arrêt d'application

Lorsqu'une application est démarrée, son programme `start()` est exécuté dans une tâche du nom de l'application suivi de '~', et de priorité **10**.

Lorsqu'une application se termine, son programme `stop()` est exécuté dans une tâche du nom de l'application précédé de '~', et de priorité **10**.

Si l'arrêt d'une application **VAL 3** est provoqué depuis l'interface utilisateur du contrôleur **MCP**, la tâche de démarrage, si elle existe encore, est immédiatement détruite. Le programme `stop()` est exécuté ensuite, puis les tâches de l'application éventuellement restantes sont supprimées dans l'ordre inverse de celui de leur création, après quoi les librairies sont déchargées de la mémoire.

5.3. VISIBILITÉ

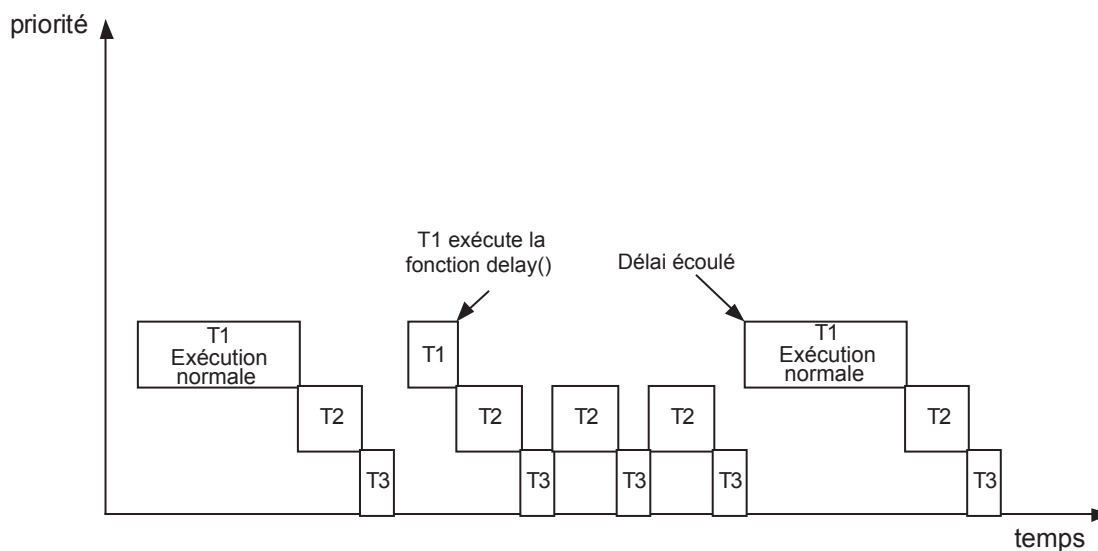
Une tâche n'est visible que de l'intérieur du programme ou de la librairie qui l'a créée. Les instructions `taskSuspend()`, `taskResume()`, `taskKill()` et `taskStatus()` agissent sur une tâche créée par une autre librairie comme si la tâche n'existait pas. Deux librairies différentes peuvent donc créer des tâches ayant le même nom.

5.4. SÉQUENCEMENT

Lorsque qu'une application a plusieurs tâches en cours d'exécution, celles-ci semblent s'exécuter simultanément et indépendamment. Ceci est vrai si l'on observe l'application globalement sur des intervalles de temps suffisamment larges (de l'ordre de la seconde), mais n'est pas exact lorsque l'on s'intéresse au comportement précis sur un intervalle de temps réduit.

En effet, le système n'ayant qu'un seul processeur, il ne peut exécuter qu'une seule tâche à la fois. La simultanéité de l'exécution est simulée par un séquençement rapide des tâches, qui exécutent à tour de rôle quelques instructions avant que le système ne passe à la tâche suivante.

Séquencement



Le séquençement des tâches **VAL 3** est fait suivant les règles suivantes :

1. Les tâches sont séquençées suivant leur ordre de création
2. A chaque séquence, le système essaye d'exécuter un nombre de lignes d'instructions **VAL 3** égal à la priorité de la tâche.
3. Quand une ligne d'instruction ne peut pas être terminée (erreur d'exécution, attente d'un signal, tâche arrêtée, etc.), le système passe à la tâche **VAL 3** suivante.
4. Lorsque toutes les tâches **VAL 3** sont terminées, le système conserve du temps disponible pour les tâches système de priorité inférieure (par ex. communication réseau, rafraîchissement de l'écran utilisateur, accès aux fichiers) avant de démarrer un nouveau cycle.
L'attente maximum entre deux cycles consécutifs est égale à la durée du dernier cycle de séquençement ; mais le plus souvent, cette attente est nulle car le système n'en a pas besoin.

Les instructions **VAL 3** pouvant provoquer le séquençement immédiat de la tâche suivante sont :

- **watch()** (attente temporisée d'une condition)
- **delay()** (temporisation)
- **wait()** (attente d'une condition)
- **waitEndMove()** (attente de l'arrêt du bras)
- **open()** et **close()** (attente de l'arrêt du bras, puis temporisation)
- **get()** (attente d'un appui touche)
- **taskResume()** (attend que la tâche soit prête à être relancée)
- **taskKill()** (attend que la tâche soit effectivement tuée)
- **disablePower()** (attend que la puissance soit effectivement coupée)
- Les instructions accédant au contenu du disque (**libLoad**, **libSave**, **libDelete**, **libList**, **setProfile**)
- Les instructions de lecture/écriture de sio (opérateur =, **sioGet()**, **sioSet()**)
- **setMutex()** (attend que le mutex booléen soit faux)

5.5. TÂCHES SYNCHRONES

La séquence décrite ci-dessus est la séquence des tâches normales, appelées tâches asynchrones, dont le système programme l'exécution la plus rapide possible. Il est parfois nécessaire de programmer les tâches à intervalles réguliers, que ce soit pour l'acquisition de données ou pour le contrôle de périphérique : on parle alors de tâches synchrones.

Celles-ci sont exécutées dans le cycle de séquence en interrompant la tâche asynchrone en cours entre deux lignes de **VAL 3**. Lorsque les tâches synchrones sont terminées, la tâche asynchrone reprend.

Le séquençement des tâches synchrones **VAL 3** obéit aux règles suivantes :

1. Chaque tâche synchrone est séquencée exactement une fois pour chaque période définie lors de la création de la tâche (par exemple toutes les 4 ms).
2. Lors de chaque séquençement, le système exécute jusqu'à 3000 lignes d'instructions **VAL 3**. Il passe à la tâche suivante lorsqu'une ligne d'instructions ne peut être terminée immédiatement (erreur d'exécution, attente d'un signal, tâche arrêtée, ...).
En pratique, une tâche synchrone se termine souvent explicitement par l'instruction `delay(0)` qui oblige le système à passer au séquençement de la tâche suivante.
3. Les tâches synchrones de même période sont séquencées suivant leur ordre de création.

5.6. ERREUR DE CADENCE

Si la durée d'exécution d'une tâche synchrone **VAL 3** est supérieure à la période définie, le cycle en cours se termine normalement mais le cycle suivant est annulé. Cette erreur de cadence est signalée à l'application **VAL 3** en mettant à **true** la variable booléenne spécialement prévue à cet effet lors de la création de la tâche. Ainsi, cette variable booléenne indique au début de chaque cycle si le séquençement précédent s'est exécuté entièrement ou non.

5.7. RAFRAÎCHISSEMENT DES ENTRÉES/SORTIES

Les entrées sont rafraîchies avant l'exécution des tâches synchrones et des tâches asynchrones. De même, les sorties sont rafraîchies après l'exécution des tâches synchrones et des tâches asynchrones.

ATTENTION :

Il n'est pas possible de définir quelles sont les entrées/sorties que doit utiliser une tâche donnée. Chaque rafraîchissement porte, par conséquent, sur l'ensemble des entrées/sorties.

Le rafraîchissement des entrées/sorties sur Modbus, BIO board, MIO board, CIO board ou AS-i bus n'est pas contrôlé par le programmeur VAL 3. Elles peuvent être rafraîchies à tout moment pendant le séquençement d'une tâche VAL 3.

5.8. SYNCHRONISATION

Il est parfois nécessaire de synchroniser plusieurs tâches avant qu'elles ne poursuivent leur exécution.

Si l'on connaît a priori la durée d'exécution de chacune des tâches, cette synchronisation peut se faire par simple attente d'un signal émis par la tâche la plus lente. Mais lorsqu'on ne sait pas quelle tâche sera la plus lente, il faut utiliser un mécanisme de synchronisation plus complexe dont un exemple de programmation **VAL 3** est donné ci-dessous.

Exemple

// Programme de synchronisation pour N tâches

Le programme `synchro(num& n, bool& bSynch, num nN)` ci-après doit être appelé dans chaque tâche à synchroniser.

La variable `n` doit être initialisée à 0, `bSynch` à `false` et `nN` au nombre de tâches à synchroniser.

```
begin
  n = n + 1
  // Synchronisation de tâches en attente d'instructions
  // vérifiez que toutes les tâches sont en attente ici pour reprendre l'opération
  wait((n==nN) or (bSynch==true))
  bSynch = true
  n = n - 1
  // Libération de tâches en attente d'instructions
  // vérifiez que toutes les tâches ont repris pour supprimer le contexte de synchronisation
  wait((n==0) or (bSynch == false))
  bSynch = false
end
```

5.9. PARTAGE DE RESSOURCE

Quand plusieurs tâches utilisent le même système ou la même ressource de cellule (données globales, écran, clavier, robot, etc.), il est important d'éviter tout conflit entre elles.

On peut pour cela utiliser un mécanisme d'exclusion mutuelle (**'mutex'**) qui protège une ressource en autorisant son accès à une seule tâche à la fois. Un exemple de programmation de mutex en **VAL 3** est donné ci-dessous.

Exemple

Cet affichage de programme (num c) remplit l'écran des mêmes caractères, en vérifiant qu'aucune autre tâche n'écrit sur l'écran en même temps avec le même programme. bScreen doit être initialisé à **false**.

```
begin
  // vérifiez qu'une seule tâche accède à l'écran à la fois
  setMutex(bEcran)
  c=c%10
  // remplissez l'écran de caractères
  for y=0 to 13
    gotoxy(x,y)
    put(c)
  endFor
endFor
// attente de l'actualisation de l'écran
delay(0.2)
// laissez les autres tâches accéder à l'écran
bEcran=false
end
```

5.10. INSTRUCTIONS

void taskSuspend(string sNom)

Fonction

Cette instruction suspend l'exécution de la tâche **sNom**.

Si la tâche est déjà dans l'état **STOPPED**, l'instruction est sans effet.

Une erreur d'exécution est générée si **sNom** ne correspond à aucune tâche **VAL 3** ou correspond à une tâche **VAL 3** créée par une autre librairie.

Voir aussi

void taskResume(string sNom, num nSaut)

void taskKill(string sNom)

void taskResume(string sNom, num nSaut)

Fonction

Cette instruction reprend l'exécution de la tâche **sNom** sur la ligne située à **nSaut** lignes d'instructions avant ou après la ligne courante.

Si **nSaut** est négatif, l'exécution reprend avant la ligne courante. Si la tâche n'est pas dans l'état **STOPPED**, l'instruction est sans effet.

Une erreur d'exécution est générée si **sNom** ne correspond pas à une tâche **VAL 3**, correspond à une tâche **VAL 3** créée par une autre librairie, ou s'il n'y a pas de ligne d'instruction à l'endroit du **nSaut** spécifié.

Voir aussi

void taskSuspend(string sNom)

void taskKill(string sNom)

void taskKill(string sNom)

Fonction

Cette instruction suspend puis supprime la tâche **sNom**. Après l'exécution de cette instruction, la tâche **sNom** n'est plus présente dans le système.

S'il n'existe pas de tâche **sNom** ou si la tâche **sNom** a été créée par une autre librairie, l'instruction n'a aucun effet.

Voir aussi

void taskSuspend(string sNom)

void taskCreate string sNom, num nPriorité, programme(...)

void setMutex(bool& bMutex)

Fonction

Cette instruction attend que la variable **bMutex** soit false puis la règle à true.

Cette instruction est nécessaire pour utiliser une variable booléenne comme mécanisme d'exclusion mutuelle en vue de la protection de ressources partagées (voir chapitre 5.9).

string help(num nCodeErreur)

Fonction

Cette instruction renvoie la description du code d'erreur d'exécution spécifié avec le paramètre nCodeErreur. La description est donnée dans la langue actuelle du contrôleur.

Exemple

Ce programme vérifie s'il y a une erreur dans la tâche "robot" et affiche le code d'erreur pour l'opérateur s'il y en a une.

```
nCodeErreur=taskStatus("robot")
if (nCodeErreur > 1)
  gotoxy(0,12)
  put(help(nCodeErreur))
endIf
```

num taskStatus(string sNom)

Fonction

Cette instruction renvoie le statut actuel de la tâche **sNom** ou le code d'erreur de temps d'exécution de la tâche si elle est en erreur :

Code	Description
-1	Aucune tâche sNom n'a été créée par la librairie ou l'application actuelle
0	La tâche sNom est suspendue sans erreur de temps d'exécution (instruction taskSuspend() ou mode de débogage)
1	La tâche sNom créée par la librairie ou l'application actuelle est en cours d'exécution
10	Calcul numérique non valide (division par zéro).
11	Calcul numérique non valide (par exemple ln(-1))
20	Accès à un tableau ayant un index supérieur à la taille du tableau.
21	Accès à un tableau ayant un index négatif.
29	Nom de tâche invalide. Voir instruction taskCreate() .
30	Le nom spécifié ne correspond à aucune tâche VAL 3 .
31	Une tâche de même nom existe déjà. Voir instruction taskCreate .
32	Seules 2 périodes différentes pour les tâches synchrones sont supportées. Modifier la période d'exécution.
40	Pas assez de place mémoire système.
41	Pas assez de mémoire d'exécution pour la tâche. Voir Taille de mémoire d'exécution.
60	Temps maximal d'exécution de l'instruction dépassé.
61	Erreur interne à l'interpréteur VAL 3
70	Valeur de paramètre d'instruction invalide. Voir l'instruction correspondante.
80	Utilisation d'une donnée ou d'un programme d'une librairie non chargée en mémoire.
81	Cinématique incompatible : Utilisation d'une point/joint/config incompatible avec le bras cinématique.
82	Le Frame ou Tool de référence d'une variable fait partie d'une librairie qui n'est pas accessible depuis l'application (librairie non déclarée dans l'application de la variable ou variable avec attribut privée).
90	La tâche ne peut pas reprendre à l'endroit spécifié. Voir instruction taskResume() .
100	La vitesse spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
101	L'accélération spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
102	La décélération spécifiée dans le descripteur de mouvement est invalide (négative, trop grande, ou inférieure à la vitesse).
103	La vitesse de translation spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
104	La vitesse de rotation spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
105	Le paramètre reach spécifié dans le descripteur de mouvement est invalide (négatif).
106	Le paramètre leave spécifié dans le descripteur de mouvement est invalide (négatif).
122	Tentative d'écriture sur une entrée du système.
123	Utilisation d'une entrée-sortie dio, aio ou sio non reliée à une entrée-sortie du système.
124	Tentative d'accès à une entrée-sortie protégée du système
125	Erreur de lecture ou d'écriture sur une dio , aio ou sio (erreur sur le bus de terrain)
150	Impossible d'exécuter cette instruction de mouvement : un mouvement demandé précédemment n'a pas pu être terminé (point hors d'atteinte, singularité, problème de configuration...)
153	Commande de mouvement non supportée
154	Instruction de mouvement invalide : cible hors de portée, ou vérifier le descripteur de mouvement.
160	Coordonnées de l'outil flange non valides
161	Coordonnées du repère world non valides
162	Utilisation d'un point sans repère de référence. Voir Définition.
163	Utilisation d'un repère sans repère de référence. Voir Définition.
164	Utilisation d'un outil sans outil de référence. Voir Définition.
165	Repère ou outil de référence invalide (variable globale liée à une variable locale)
250	Pas de licence d'exécution (runtime) pour cette instruction, ou validité de la licence démo dépassée.

Voir aussi

void taskResume(string sNom, num nSaut)

void taskKill(string sNom)

`void taskCreate string sNom, num nPriorité, programme(...)`

Fonction

Cette instruction crée et lance la tâche **sNom**.

sNom doit avoir **1** à **15** caractères parmi "**a..zA..Z0..9_**". Aucune autre tâche créée par la même librairie ne doit porter le même nom.

L'exécution de **sNom** commence par l'appel de **programme** avec les paramètres spécifiés. Il n'est pas possible d'utiliser une variable locale pour un paramètre transmis par référence, afin d'assurer que la variable ne sera pas effacée avant la fin de la tâche.

La tâche se terminera par défaut avec la dernière ligne d'instructions de **programme**, ou plus tôt si elle est détruite explicitement.

nPriorité doit être compris entre **1** et **100**. A chaque séquençement de la tâche, le système exécutera un nombre de ligne d'instructions égal à **nPriorité**, ou moins si une instruction bloquante est rencontrée (voir le chapitre Séquençement).

Une erreur d'exécution est générée si le système n'a pas assez de mémoire pour créer la tâche, si **sNom** n'est pas valide ou déjà utilisé dans la même librairie, ou si **nPriorité** n'est pas valide.

Exemple

```
// lancer une nouvelle tâche pour lire un message
taskCreate "t1", 10, read(sMessage)
// attente de la fin de la tâche t1
wait(taskStatus("t1") == -1)
// Utiliser le message
putln(sMessage)
```

Voir aussi

void taskSuspend(string sNom)

void taskKill(string sNom)

num taskStatus(string sNom)

void taskCreateSync string sNom, num nPériode,
bool& bOverrun, programme(...)

Fonction

Cette instruction crée et lance une tâche synchrone.

L'exécution de la tâche commence par l'appel du programme spécifié avec les paramètres spécifiés. Il n'est pas possible d'utiliser une variable locale pour un paramètre transmis par référence, afin d'assurer que la variable ne sera pas effacée avant la fin de la tâche.

Une erreur d'exécution est générée si le système ne possède pas la mémoire nécessaire pour pouvoir créer la tâche ou si un ou plusieurs paramètres ne sont pas valides.

Pour une présentation détaillée des tâches synchrones (voir chapitre 5.5).

Paramètres

string sNom	Nom de la tâche à créer. Il doit contenir de 1 à 15 caractères choisis dans la plage suivante "_a..zA..Z0..9". Deux tâches appartenant à la même application ou librairie ne peuvent porter le même nom.
num nPériode	Période d'exécution en secondes de la tâche à créer (s). La valeur indiquée est arrondie au multiple de 4 ms immédiatement inférieur (0.004 seconde). Le système accepte toutes les durées positives, mais pas plus de deux durées différentes de tâches synchrones à la fois.
bool& bOverrun	Variable booléenne indiquant les erreurs de cadence. Seules les variables globales sont supportées, afin de garantir que la variable ne sera pas supprimée avant la tâche.
programme	Nom du programme VAL 3 à appeler lorsque la tâche est lancée, les paramètres correspondants apparaissent entre parenthèses.

Exemple

```
// Créer une tâche de supervision programmée toutes les 20 ms
taskCreateSync "supervisor", 0.02, bSupervisor, supervisor()
```

`void wait(bool bCondition)`

Fonction

Cette instruction met la tâche courante en attente jusqu'à ce que **bCondition** soit **true**.

La tâche reste **RUNNING** pendant la durée de l'attente. Si **bCondition** est **true** lors de la première évaluation, l'exécution se poursuit immédiatement sur la même tâche (il n'y a pas séquençement de la tâche suivante).

Voir aussi

`void delay(num nSecondes)`

`bool watch(bool bCondition, num nSecondes)`

`void delay(num nSecondes)`

Fonction

Cette instruction met la tâche courante en attente pendant **nSecondes**.

La tâche reste **RUNNING** pendant la durée de l'attente. Si **nSecondes** est négative ou nulle, le système procède immédiatement au séquençement de la tâche **VAL 3** suivante.

Exemple

Ce programme se met en boucle pour obtenir une clé, en veillant à ne pas utiliser trop de ressources de CPU :

```
// Réinitialiser d'abord le code de la dernière touche enfoncée
getKey()
while(getKey() == -1)
  gotoxy(0,0)
  put(toString("", clock()* 10))
  // laisser une autre tâche s'exécuter immédiatement
  delay(0)
endWhile
```

Voir aussi

`num clock()`

`bool watch(bool bCondition, num nSecondes)`

num clock()

Fonction

Cette instruction renvoie la valeur actuelle de l'horloge système interne, exprimée en secondes.

La précision de l'horloge interne du système est la milliseconde. Elle est initialisée à **0** lors du démarrage du contrôleur, et n'a donc aucune relation avec l'heure calendaire.

Exemple

Pour calculer le délai d'exécution entre deux instructions, enregistrer la valeur d'horloge avant la première instruction :

```
nDebut=clock()
```

Après la dernière instruction, calculer le délai d'exécution avec :

```
nDelay = clock()-nDebut
```

Voir aussi

void delay(num nSecondes)

bool watch(bool bCondition, num nSecondes)

bool watch(bool bCondition, num nSecondes)

Fonction

Cette instruction met la tâche courante en attente jusqu'à ce que **bCondition** soit **true** ou jusqu'à ce que **nSecondes** secondes soient écoulées.

Renvoie **true** si l'attente se termine lors que **bCondition** est **true**, sinon **false** lorsque l'attente se termine parce que le délai est écoulé.

La tâche reste **RUNNING** pendant la durée de l'attente. Si **bCondition** est **true** à la première évaluation, la tâche est exécutée immédiatement ; dans le cas contraire, le système séquence les **VAL 3** autres tâches (même si **nSecondes** est inférieur ou égal à **0**).

Exemple

Ce programme attend un signal et affiche un message d'erreur après 20 s.

```
if (watch (diSignal==true, 20)) == false
    popUpMsg ("Erreur : en attente de signal")
    wait (diSignal==true)
endIf
```

Voir aussi

void delay(num nSecondes)

void wait(bool bCondition)

num clock()

CHAPITRE 6

LIBRAIRIES

6.1. DÉFINITION

Une librairie **VAL 3** est une application **VAL 3** contenant des variables ou programmes qui peuvent être réutilisés par une autre application ou par d'autres librairies **VAL 3**.

Comme la librairie **VAL 3** est une application **VAL 3**, elle contient les composants suivants :

- un ensemble de **programmes** : les instructions **VAL 3** à exécuter
- un ensemble de **variables globales** : les données de la librairie
- un ensemble de **librairies** : les instructions externes et variables utilisées par la librairie

Lorsqu'une librairie est en cours d'exécution, elle peut contenir également :

- un ensemble de tâches : les programmes propres à la librairie en cours d'exécution

Toute application peut être utilisée comme librairie, et toute librairie peut être utilisée comme application, si les programmes **start()** et **stop()** y sont définis.

6.2. INTERFACE

Les programmes et variables globales d'une librairie peuvent être publics ou privés. Seuls les programmes et variables publics sont accessibles en dehors de la librairie. Les programmes et variables globales privés ne peuvent être utilisés que par les programmes de la librairie.

Tous les programmes et variables globaux publics d'une librairie constituent l'interface de celle-ci : plusieurs librairies différentes peuvent avoir la même interface, tant que leurs programmes et variables publics utilisent les mêmes noms.

Les tâches créées par un programme d'une librairie sont toujours privées, c'est-à-dire qu'elles ne sont accessibles que par cette librairie.

6.3. IDENTIFIANT D'INTERFACE

Pour pouvoir utiliser une librairie, une application doit d'abord déclarer un identifiant qui lui soit affecté, puis demander, dans un programme, de charger la librairie dans la mémoire en utilisant cet identifiant.

L'identifiant est affecté à l'interface de la librairie et non pas à la librairie elle-même. Toute librairie possédant la même interface peut ensuite être chargée en utilisant cet identifiant. Ce mécanisme peut être utilisé, par exemple, pour définir une librairie pour toutes les parties d'une application, puis ne charger que la partie utilisée actuellement par chaque cycle.

6.4. CONTENU

Une librairie n'a pas de contenu obligatoire : elle peut ne contenir que des programmes, que des variables, ou les deux.

L'accès au contenu d'une librairie se fait en écrivant le nom de l'identifiant suivi de ':' avant le nom du programme ou de la donnée de la librairie, par exemple :

```
// Charger la librairie "article_7" sous l'identifiant "article"
article:libLoad("article_7")
// Afficher comme titre le contenu de la variable 'sNom' de la librairie article_7
title(article:sNom)
// Appeler le programme init() pour l'article en cours
call article:init()
```

L'accès au contenu d'une librairie qui n'a pas encore été chargée en mémoire entraîne une erreur d'exécution.

6.5. CRYPTAGE

VAL 3 peut utiliser des librairies cryptées, sur la base des outils de compression ZIP et de cryptage courants.

Une librairie cryptée est un fichier ZIP standard du contenu du répertoire de la librairie (attention : le cryptage 128 avancé et 256 AES n'est pas possible). Le nom du fichier ZIP doit être suivi de l'extension ".zip" et doit comporter moins de 15 caractères (y compris l'extension). Pour obtenir un cryptage robuste, le mot de passe du ZIP doit comporter plus de 10 caractères et il ne doit pas apparaître dans un dictionnaire.

Mot de passe secret, mot de passe public

L'interpréteur **VAL 3** doit avoir accès au mot de passe ZIP secret pour charger une librairie cryptée. Un outil pour PC est prévu pour cela dans la **Stäubli Robotics Suite** pour coder le mot de passe ZIP secret dans un mot de passe **VAL 3** public. Le mot de passe **VAL 3** public permet d'utiliser la bibliothèque cryptée dans un programme **VAL 3**. Le contenu de la librairie reste toutefois secret puisque le mot de passe ZIP ne peut pas être calculé à partir du mot de passe **VAL 3**.

Cryptage de projet

L'application de démarrage ne peut pas être cryptée directement sur le contrôleur. Une application complète peut être cryptée :

- en déclarant son programme start() comme public.
- en créant une autre application qui charge simplement l'application cryptée sous la forme d'une librairie et appelle son programme start().

6.6. CHARGEMENT ET DÉCHARGEMENT

Lorsqu'une application **VAL 3** est ouverte, toutes les librairies déclarées sont analysées pour construire les interfaces correspondantes. Cette étape ne charge pas les librairies en mémoire.

ATTENTION :

Il ne peut pas y avoir de références circulaires entre les librairies. Si la librairie A utilise la librairie B, la librairie B ne peut pas utiliser la librairie A.

Lors du chargement d'une librairie, ses variables globales sont initialisées et ses programmes vérifiés pour détecter d'éventuelles erreurs de syntaxe. Quand plusieurs identifiants de librairie différents chargent la même librairie sur le disque, ils partagent la même librairie en mémoire. La librairie est ainsi chargée une seule fois et réutilisée par tous les identifiants. Dans l'exemple ci-dessous, lib1 et lib2 utilisent la même variable en mémoire.

```
lib1:libLoad("appData")
lib1:sText = "lib1"
lib2:libLoad("appData")
// Le changement de lib2:sText s'applique aussi à lib1:sText dans ce cas
lib2:sText = "lib2"
```

Le déchargement d'une librairie n'est pas nécessaire, il se fait automatiquement lorsque l'application se termine ou lorsqu'une nouvelle librairie est chargée à la place d'une autre.

Lorsqu'une application **VAL 3** est arrêtée depuis l'interface utilisateur du contrôleur **MCP**, le programme **stop()** est d'abord exécuté, puis toutes les tâches de l'application et de ses librairies, s'il en reste, sont détruites.

Chemin d'accès

Les instructions **libLoad()**, **libSave()** et **libDelete()** utilisent un chemin d'accès à une librairie, spécifié sous forme de chaîne de caractères. Un chemin d'accès contient une racine (facultative), un chemin (facultatif), et un nom de librairie, suivant le format :

racine://Chemin/Nom

La racine spécifie le support de fichier : **"Floppy"** pour une disquette, **"USB0"** pour un dispositif sur un port **USB** (clé, disquette), **"Disk"** pour le disque Flash du contrôleur, ou le nom d'une connexion **Ftp** défini dans le contrôleur pour un accès au réseau.

Par défaut, la racine est **"Disk"** et le chemin vide.

Exemples

```
// charger la librairie "article_1" sur le disque équivaut à "Disk://article_1"
article:libLoad("article_1")
// Sauvegarder la librairie sur un périphérique USB
article:libSave("USB0://articles/article_1")
// Charger l'article par défaut défini dans l'application courante
article:libLoad("./defaultArticle")
```

Codes d'erreur

Les fonctions **VAL 3** de manipulation de librairie ne génèrent jamais d'erreur d'exécution, mais renvoient un code d'erreur permettant de vérifier le résultat de l'instruction et de résoudre les problèmes éventuels.

Code	Description
0	Pas d'erreur
10	L'identifiant de librairie n'a pas été initialisé par libLoad() .
11	Librairie chargée, mais l'interface publique ne correspond pas. Une erreur d'exécution 80 se produit si le programme VAL 3 tente d'accéder aux éléments manquants. Voir l'instruction libExist ou les instructions isDefined() .
12	Chargement de librairie impossible : la librairie contient des données ou des programmes invalides, ou, si elle est cryptée, le mot de passe spécifié n'est pas correct.
13	Déchargement de librairie impossible : La librairie est en cours d'exécution par une autre tâche.
14	Déchargement de librairie impossible : La librairie comprend une tâche VAL 3 en cours. Toutes les tâches créées par les programmes VAL 3 à partir de la librairie doivent être terminées avant le déchargement de la librairie.
20	Erreur d'accès fichier : la racine du chemin est invalide.
21	Erreur d'accès fichier : le chemin est invalide.
22	Erreur d'accès fichier : le nom est invalide.
23	Librairie cryptée attendue. La librairie n'est pas ou pas correctement cryptée.
>=30	Erreur de lecture/écriture sur un fichier.
31	Impossible d'enregistrer la librairie : le chemin spécifié contient déjà une librairie. Pour remplacer une librairie sur le disque, commencer par l'effacer avec libdelete() .
32	Message du pilote de périphérique "Dispositif introuvable"
33	Message du pilote de périphérique "Erreur de dispositif"
34	Message du pilote de périphérique "Fin de temporisation du dispositif"
35	Message du pilote de périphérique "Dispositif protégé en écriture"
36	Message du pilote de périphérique "Pas de disque"
37	Message du pilote de périphérique "Disque non formaté"
38	Message du pilote de périphérique "Disque plein"
39	Message du pilote de périphérique "Fichier introuvable"
40	Message du pilote de périphérique "Fichier en lecture seule"
41	Message du pilote de périphérique "Connexion refusée"
42	Message du pilote de périphérique "Pas de réponse du serveur FTP"
43	Message du pilote de périphérique "Erreur du kernel FTP"
44	Message du pilote de périphérique "Erreur de paramètres FTP"
45	Message du pilote de périphérique "Erreur d'accès FTP"
46	Message du pilote de périphérique "Disque FTP plein"
47	Message du pilote de périphérique "Nom d'utilisateur FTP invalide"
48	Message du pilote de périphérique "Pas de connexion FTP définie"

6.7. INSTRUCTIONS

num identifiant:**libLoad**(string sChemin)

num identifiant:**libLoad**(string sChemin, string sMot de passe)

Fonction

Cette instruction initialise l'identifiant de librairie en chargeant le programme et les variables de librairie dans la mémoire après le **sChemin** spécifié. Le paramètre **sMot de passe** spécifié (facultatif) est utilisé comme clé de déchiffrement pour les librairies cryptées. Le **sMot de passe** spécifié doit être le mot de passe **VAL 3** public calculé à partir du mot de passe ZIP secret de la librairie cryptée (voir chapitre 6.5, page 98).

L'instruction renvoie **0** après un chargement réussi ou un code d'erreur de chargement de librairie si des tâches créées par la librairie sont encore en cours d'exécution, si le chemin d'accès à la librairie est incorrect, si la librairie contient des erreurs de syntaxe ou si elle ne correspond pas à l'interface déclarée pour l'identifiant.

Voir aussi

num identifiant:**libSave**(), **num** libSave()

num identifiant:**libSave**(), **num** libSave()

Fonction

Cette instruction sauvegarde les variables et programmes attribués à l'identifiant de la librairie. Si **libSave()** est appelé sans identifiant, l'application contenant l'instruction **libSave()** est sauvegardée. Si un paramètre est spécifié, la sauvegarde se fait au **sChemin** indiqué. Sinon, la sauvegarde se fait au chemin spécifié lors du chargement.

L'instruction renvoie **0** si le contenu a été sauvegardé ou un code d'erreur si l'identifiant n'a pas été initialisé, si le chemin est incorrect, si une erreur d'écriture s'est produite ou si le chemin indiqué contient déjà une librairie.

ATTENTION :

Certains appareils tels que le disque Flash du contrôleur n'acceptent qu'un nombre limité d'accès en écriture. Si **libSave()** est utilisé fréquemment dans un programme (une fois ou plus par minute), il doit l'être sur un appareil acceptant cette fréquence.

Voir aussi

num libDelete(string sChemin)

num libDelete(string sChemin)

Fonction

Cette instruction supprime la librairie située dans le **sChemin** indiqué.

L'instruction renvoie **0** si la librairie indiquée n'existe pas ou a été supprimée et un code d'erreur si l'identifiant n'a pas été initialisé, si le chemin est incorrect ou si une erreur d'écriture se produit.

Voir aussi

num identifiant:**libSave**(), **num** libSave()

string identifiant:**libPath**(), **string** libPath()

string identifiant:libPath(), string libPath()

Fonction

Cette instruction renvoie le chemin d'accès de la librairie associé à l'identifiant, ou de l'application ou librairie appelante si aucun identifiant n'est spécifié.

Voir aussi

bool libList(string sChemin, string& sContenu[])

bool libList(string sChemin, string& sContenu[])

Fonction

Cette instruction donne la liste du contenu du chemin **sChemin** spécifié dans le tableau **sContenu**. Renvoie **true** si le tableau **sContenu** contient le résultat complet ou **false** si le tableau est trop petit pour contenir toute la liste.

Tous les éléments du tableau **sContenu** sont d'abord initialisés à "" (chaîne vide). Après l'exécution de **libList()**, la fin de la liste est recherchée dans la première chaîne vide du tableau **sContenu**.

Si **sContenu** est une variable globale, le tableau est automatiquement agrandi autant que nécessaire pour permettre l'enregistrement du résultat complet.

Voir aussi

string identifiant:libPath(), string libPath()

bool identifier:libExist(string sNomSymbole)

Fonction

L'instruction **libExist** vérifie si un symbole (variable globale ou programme) est défini dans une librairie. Elle renvoie un résultat "true" si le symbole existe et s'il est accessible (public), sinon "false".

Le nom de symbole d'un programme doit se terminer par "()" : "mySymbol" désigne un nom de variable, tandis que "mySymbol()" est un nom de programme.

L'instruction **libExist** est utile pour tester si une entrée/sortie est définie sur un contrôleur ; elle est également utile pour gérer l'évolution de l'interface d'une librairie et adapter son utilisation selon s'il s'agit d'une version récente ou ancienne de l'interface.

Exemple

L'exemple suivant teste l'interface d'une librairie.

```
// Charger librairie de pièces
nLoadCode = part:libLoad(sPartPath)
// part:sVersion n'a pas été défini dans la première version de la librairie
// Tester si cette librairie le définit
if (nLoadCode==0) or (nLoadCode==11)
  if (part:libExist("sVersion")==false)
    // version initiale
    sLibVersion = "v1.0"
  else
    sLibVersion = part:sVersion
  endif
endif
```

Ce programme appelle le programme "init" dans la librairie "protocol", si elle existe :

```
if(protocol:libExist("init()")==true)
  call protocol:init()
endif
```

Voir aussi

bool isDefined(*)

CHAPITRE 7

TYPE D'UTILISATEUR

7.1. DÉFINITION

Un type d'utilisateur est un type structuré, défini dans une application VAL 3 dans laquelle il peut être utilisé comme type standard. Un type d'utilisateur combine des types simples, structurés ou même d'autres types d'utilisateur dans un nouveau type de données. Les types d'utilisateur élèvent le niveau d'abstraction des programmes et les rendent plus facile à comprendre, à développer et à entretenir. Ils nécessitent toutefois davantage de travail de conception pour identifier les types convenant le mieux aux contraintes de l'application.

Un type d'utilisateur est un ensemble de champs dont chacun comprend :

- un nom : une chaîne de caractères
- un type de variable (simple, structuré ou type d'utilisateur)
- un conteneur de variables (élément, tableau ou collection)
- un ensemble de valeurs par défaut

Les champs des types standard de VAL 3 utilisent toujours un conteneur élément (avec une seule valeur). Les champs des types d'utilisateur peuvent utiliser des conteneurs tableau ou collection et peuvent donc contenir plusieurs valeurs. La valeur par défaut du champ définit le nombre par défaut d'éléments dans le conteneur de champ et la valeur par défaut de chacun de ces éléments. Dans une variable définie avec un type d'utilisateur, on peut changer à tout moment les valeurs des champs, mais aussi le nombre d'éléments dans les conteneurs de champs.

7.2. CRÉATION

Les champs d'un type d'utilisateur ont les mêmes caractéristiques que les variables globales d'une application VAL 3. C'est la raison pour laquelle la création d'un nouveau type d'utilisateur se résume à sélectionner une application VAL 3 et à l'associer à un nom.

- L'ensemble des variables globales publiques dans l'application sélectionnée définit l'ensemble des champs du type d'utilisateur avec les valeurs par défaut de ceux-ci.
- Le nom définit le nom à utiliser pour le nouveau type d'utilisateur dans l'application dans laquelle il est défini.

Les variables privées et les programmes de l'application utilisés comme définition du type ne sont pas pris en considération dans le type d'utilisateur.

Une fois qu'un nouveau type d'utilisateur est défini dans une application, il est possible de créer des variables de ce type. L'application ainsi obtenue peut aussi être utilisée comme définition de type.

7.3. DOMAINE D'UTILISATION

Les champs d'une variable de type d'utilisateur sont accessibles à l'aide d'un '.' suivi du nom du champ : `userVariable.field1.field2` désigne la valeur du champ 'field2' du champ 'field1' des données `userVariable`. La création ou la suppression d'éléments dans un conteneur de champ sont possibles, comme pour une variable, à l'aide des instructions `insert()`, `delete()`, `append()` ou `resize()`. Quand un nouvel élément est créé, chaque champ reçoit une valeur par défaut composée des éléments et de leurs valeurs définies dans l'application utilisée comme définition du type.

ATTENTION :

Les champs de type point, outil ou repère ne sont pas liés par défaut.

L'opérateur '=' est toujours défini entre deux variables du même type d'utilisateur. Après l'exécution de l'opérateur '=', la variable de gauche est une copie de celle de droite : le conteneur des champs a le même nombre d'éléments et les éléments ont la même valeur.

CHAPITRE 8

CONTRÔLE DU ROBOT

Ce chapitre liste les instructions donnant accès à l'état des différentes parties du robot.

8.1. INSTRUCTIONS

`void disablePower()`

Fonction

Cette instruction couple l'alimentation du bras et attend que la coupure soit effective.

Si le bras est en mouvement, un arrêt rapide sur trajectoire est effectué avant la coupure de puissance.

Voir aussi

`void enablePower()`
`bool isPowered()`

`void enablePower()`

Fonction

En mode déporté, cette instruction met le bras sous tension.

Cette instruction n'a aucun effet en mode local, manuel ou test, ou lorsque la puissance est en cours de coupure. Elle crée toutefois un message dans le fichier journal, afin d'éviter des tentatives répétées sans temporisation d'établir l'alimentation.

Exemple

```
// Met la puissance et attend qu'elle soit installée
enablePower()
if (watch(isPowered(), 5) == false)
    putln("Impossible d'activer l'alimentation du bras")
endIf
```

Voir aussi

`void disablePower()`
`bool isPowered()`

`bool isPowered()`

Fonction

Cette instruction renvoie l'état d'alimentation du bras :

`true` : le bras est sous puissance

`false` : le bras est hors tension ou est en cours de mise sous tension

bool isCalibrated()

Fonction

Cette instruction renvoie l'état de calibrage du robot :

true : tous les axes du robot sont calibrés

false : au moins un axe du robot n'est pas calibré

num workingMode(), num workingMode(num& nEtat)

Fonction

Cette instruction renvoie le mode de fonctionnement courant du robot :

Mode	Etat	Mode de marche	Etat
0	0	Invalide ou en transition	-
1	0	Manuel	Mouvement programmé
	1		Mouvement de connexion
	2		Joint jogging
	3		Cartésien (Frame jogging)
	4		Tool jogging
	5		Vers point (Point jogging)
	6		Hold
2	0	Test	Mouvement programmé (< 250 mm/s)
	1		Mouvement de connexion (< 250 mm/s)
	2		Mouvement programmé rapide (> 250 mm/s)
	3		Hold
3	0	Local	Move (mouvement programmé)
	1		Move (mouvement de connexion)
	2		Hold
4	0	Déporté	Move (mouvement programmé)
	1		Move (mouvement de connexion)
	2		Hold

num esStatus()

Fonction

Cette instruction renvoie l'état du circuit d'arrêt d'urgence :

Code	Etat
0	Pas d'arrêt d'urgence (chaîne d'arrêt d'urgence fermée).
1	Plus d'arrêt d'urgence, en attente de validation. En mode manuel, le MCP doit se trouver sur son support pour que l'alimentation soit établie.
2	Arrêt d'urgence ouvert ou en attente de correction d'un défaut matériel.

Voir aussi

num workingMode(), num workingMode(num& nEtat)

bool safetyFault(string& sNom du signal)

bool safetyFault(string& sNom du signal)

Fonction

Cette instruction renvoie true si un défaut matériel dans la chaîne de sécurité doit être corrigé et acquitté. Dans ce cas, sSignalName est actualisé avec le nom du signal défectueux.

Voir aussi

num esStatus()

num ioBusStatus(string& sDescription de l'erreur[])

Fonction

Cette instruction vérifie l'état des appareils de bus de terrain et renvoie le nombre d'appareils en erreur ou zéro s'il n'y a pas d'appareil en erreur. Une description en texte est ajoutée au tableau de chaînes sErrorDescription pour chaque appareil en erreur. Le format de la description d'erreur est : 'statusValue:deviceName\moduleName'.

Si sErrorDescription est une variable globale, le tableau est automatiquement agrandi autant que nécessaire pour permettre l'enregistrement du résultat complet.

La valeur d'état est un numéro d'erreur qui dépend de l'appareil de bus de terrain et du protocole.

Voir aussi

num ioStatus(dio diEntréeSortie, string& sDescription, string& sChemin physique)

num ioStatus(aio diEntréeSortie, string& sDescription, string& sChemin physique)

num getMonitorSpeed()

Fonction

Cette instruction renvoie la vitesse moniteur actuelle du robot (dans la plage [0, 100]).

Exemple

Ce programme, qui est appelé dans une tâche spécifique, vérifie si le premier cycle du robot est exécuté à petite vitesse :

```
while true
  if(nCycle < 2)
    if (getMonitorSpeed() > 10)
      stopMove()
      gotoxy(0,0)
      putln("La vitesse moniteur doit rester à 10% pour le premier cycle")
      wait(getMonitorSpeed() <= 10)
    endIf
    restartMove()
  endIf
  delay(0)
endWhile
```

Voir aussi

num setMonitorSpeed(num nVitesse)

num setMonitorSpeed(num nVitesse)

Fonction

Cette instruction modifie la vitesse moniteur actuelle du robot. [setMonitorSpeed\(\)](#) est toujours capable de réduire la vitesse moniteur. Pour augmenter celle-ci, [setMonitorSpeed\(\)](#) n'est efficace que si le robot fonctionne en mode déporté et si l'opérateur n'a pas accès à la vitesse moniteur (quand le profil utilisateur courant ne permet pas l'utilisation des boutons de vitesses ou quand le MCP a été déconnecté).

Elle renvoie 0 si la vitesse moniteur a été modifiée, un code d'erreur négatif dans le cas contraire :

Code	Description
-1	Le robot n'est pas en mode déporté
-2	La vitesse moniteur est contrôlée par l'opérateur : modifier le profil d'utilisateur actuel pour supprimer l'accès de l'opérateur à la vitesse moniteur
-3	La vitesse spécifiée n'est pas acceptée : elle doit se situer dans la plage [0, 100]

Voir aussi

num getMonitorSpeed()

string getVersion(string sComposant)

Fonction

Cette instruction renvoie la version de différents composants matériels et logiciels du contrôleur du robot. Le tableau ci-dessous donne la liste des composants pris en charge et le format de la valeur renvoyée pour chacun d'entre eux :

Composant	Description
"VAL 3"	Version du contrôleur VAL 3 , par exemple "s7.0 - Jun 18 2010 - 16:01:17"
"ArmType"	Type de bras fixé au contrôleur, par exemple "tx90-S1" ou "rs60-S1-D20-L200"
"Tuning"	Version du réglage du bras, par exemple "R3"
"Mounting"	Montage du bras, par exemple "sol", "mur" ou "plafond"
"ControllerSN"	Numéro de série du contrôleur, par exemple "F07_12R3A1_C_01"
"ArmSN"	Numéro de série du bras, par exemple "F07_12R3A1_A_01"
"Starc"	Version du firmware Starc (CS8C), par exemple "1.16.3 - Sep 27 2007 - 16:01:17"
Nom de la licence	Etat de la licence du logiciel du contrôleur : "" (non installée ou période de démo dépassée), "demo" ou "enabled" Les noms des licences de contrôleur installées (par exemple "alter", "compliance", "remoteMcp", "oemLicence", "plc", "testMode", "mcpMode"...) sont listés dans le tableau de contrôle du boîtier de commande manuelle du robot

Exemple

```
if getVersion("compliance") != "activé"
    putln("Pas de licence de compliance sur le contrôleur")
endif
```

Voir aussi

string getLicence(string sOemNomLicence, string sOemMot de passe)

CHAPITRE 9

POSITIONS DU BRAS

9.1. INTRODUCTION

Ce chapitre décrit les types de données **VAL 3** permettant de programmer les positions du bras occupées dans une application **VAL 3**.

Deux types de positions sont définies en **VAL 3** : des positions d'articulations (type articulation **joint**) qui indiquent la position angulaire de chaque angle de révolution et la position linéaire de chaque axe linéaire, et des points cartésiens (type **point**) qui donnent la position cartésienne du centre de l'outil à l'extrémité du bras par rapport à un repère de référence.

Le type **tool** décrit un outil avec sa géométrie, utilisé pour le positionnement et le contrôle de vitesse du bras ; il décrit également le mode d'activation de l'outil (sortie numérique, temporisation).

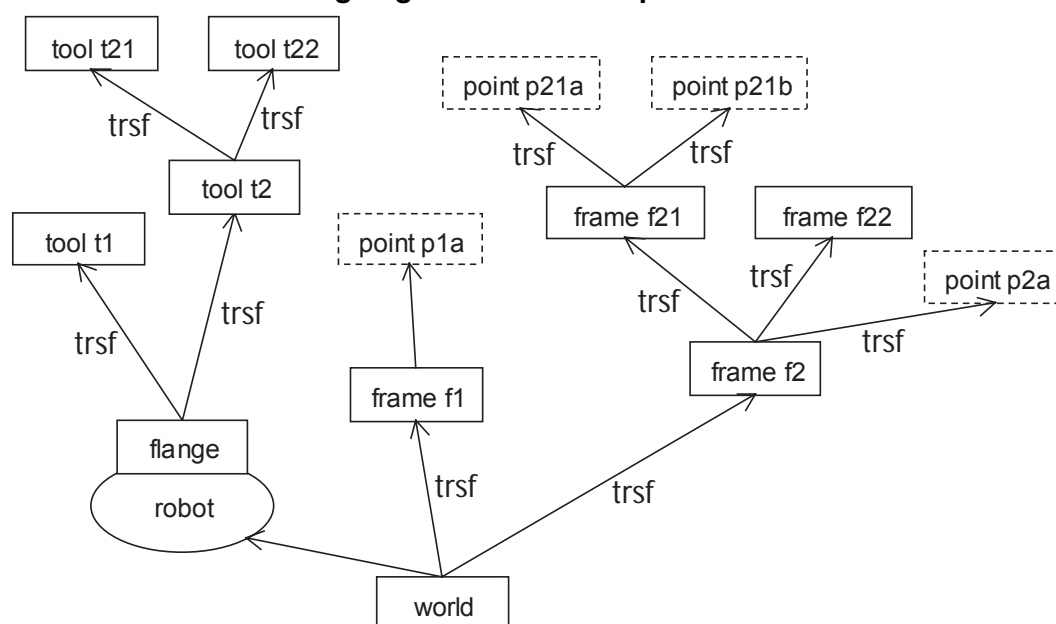
Le type **frame** décrit un repère géométrique. L'utilisation de repères rend en particulier les manipulations géométriques sur les points plus simples et intuitives.

Le type **trsf** décrit une transformation géométrique. Il est utilisé par les types **tool**, **point** et **frame**.

Enfin, le type **config** décrit la notion plus avancée de configuration de bras.

Les relations entre ces différents types peuvent être résumées ainsi :

Organigramme : frame / point / tool / trsf



9.2. TYPE JOINT

9.2.1. DÉFINITION

Une localisation d'articulation (type **joint**) définit la position angulaire de chaque angle de rotation et la position linéaire de chaque axe linéaire.

Le type **joint** est un type structuré dont les champs sont, dans l'ordre :

num j1	position Joint de l'axe 1
num j2	position Joint de l'axe 2
num j3	position Joint de l'axe 3
num j...	position Joint de l'axe ... (un champ par axe)

Ces champs sont exprimés en degrés pour les axes rotatifs, en millimètres ou en pouces pour les axes linéaires. L'origine de chaque axe est définie par le type de bras utilisé.

Par défaut, chaque champ d'une variable de type **joint** est initialisé à la valeur **0**.

9.2.2. OPÉRATEURS

Par ordre de priorité croissant :

joint < joint & jPosition1 > = < joint jPosition2 >	Affecte jPosition2 à la variable jPosition1 champ par champ et renvoie jPosition2 .
bool < joint jPosition1 > != < joint jPosition2 >	Renvoie true si un champ de jPosition1 n'est pas égal au champ correspondant de jPosition2 à la précision du robot près, false sinon.
bool < joint jPosition1 > == < joint jPosition2 >	Renvoie true si chaque champ de jPosition1 est égal au champ correspondant de jPosition2 à la précision du robot près, false sinon.
bool < joint jPosition1 > > < joint jPosition2 >	Renvoie true si chaque champ de jPosition1 est strictement supérieur au champ correspondant de jPosition2 , false sinon.
bool < joint jPosition1 > < < joint jPosition2 >	Renvoie true si chaque champ de jPosition1 est strictement inférieur au champ correspondant de jPosition2 , false sinon. Attention : jPosition1 > jPosition2 n'est pas l'équivalent de !(jPosition1 < jPosition2)
joint < joint jPosition1 > - < joint jPosition2 >	Renvoie la différence champ par champ de jPosition1 et jPosition2 .
joint < joint jPosition1 > + < joint jPosition2 >	Renvoie la somme champ par champ de jPosition1 et jPosition2 .

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions.

9.2.3. INSTRUCTIONS

joint abs(joint jPosition)

Fonction

Cette instruction renvoie la valeur absolue d'une articulation **jPosition**, champ par champ.

Détails

La valeur absolue d'un joint, avec les opérateurs joint ">" ou "<", permet de calculer facilement la distance séparant la position d'un joint de la position de référence.

Exemple

```
jReference = {90, 45, 45, 0, 30, 0}
jEcartMax = {5, 5, 5, 5, 5, 5}
j = herej()
// Vérifie que tous les axes ont moins de 5 degrés par rapport à la référence
if(!(abs(j - jReference) < jEcartMax))
    popUpMsg("Rapprochez vous des marques")
endif
```

Voir aussi

Operator < (joint)

Operator > (joint)

joint herej()

Fonction

Cette instruction renvoie la position actuelle de l'articulation du bras.

Lorsque le bras est sous tension, la valeur renvoyée correspond à la position transmise aux amplificateurs par le contrôleur, et non pas à la position relevée sur les encodeurs de l'axe.

Lorsque le bras est hors tension, la valeur renvoyée correspond à la position relevée sur les encodeurs de l'axe ; en raison du bruit dans les mesures des encodeurs, cette mesure peut varier légèrement même lorsque le bras est à l'arrêt.

La position du bras est rafraîchie toutes les 4 ms.

Exemple

```
//Attendre que le bras soit proche de la position de référence, avec un timeout de 60 s
bDebut = watch(abs(herej() - jReference) < jEcartMax, 60)
if bDebut==false
  popUpMsg("Rapprocher de la position de départ")
endIf
```

Voir aussi

```
point here(tool tOutil, frame fReference)
bool getLatch(joint& jPosition) (CS8C only)
bool isInRange(joint jPosition)
```

bool isInRange(joint jPosition)

Fonction

Cette instruction vérifie qu'une position d'articulation se trouve dans les limites d'articulation logicielles du bras.

Lorsque le bras est hors des limites logicielles de l'articulation (après une opération de maintenance), il ne peut pas être déplacé au moyen d'une application **VAL 3** ; seuls les déplacements manuels sont possibles (avec des directions de mouvement qui permettent uniquement de ramener le bras vers ses limites).

Exemple

```
// Vérifie que la position actuelle se situe à l'intérieur des limites articulaires
if isInRange(herej())==false
  putLn("Placez le bras dans son espace de travail")
endIf
```

Voir aussi

```
joint herej()
```

void setLatch(dio diEntrée) (CS8C only)

Fonction

Cette instruction active la capture de position du robot sur le flanc ascendant suivant du signal d'entrée.

Détails

La capture de la position du robot est une fonctionnalité électronique qui n'est supportée que par les entrées rapides du contrôleur CS8C (fln0, fln1).

La détection du front montant du signal d'entrée n'est garantie que si le signal reste bas pendant au moins 0.2 ms avant le front montant, et haut pendant au moins 0.2 ms après le front montant.

ATTENTION :

La capture n'est activée qu'au bout d'un certain temps (entre 0 et 0.2 ms) après l'exécution de l'instruction setLatch. Vous pouvez ajouter une instruction delay(0) après setLatch pour vous assurer que la capture est effective avant l'exécution de l'instruction VAL 3 suivante.

Une erreur d'exécution 70 (valeur de paramètre invalide) est générée si l'entrée digitale spécifiée ne supporte pas la capture de la position du robot.

Voir aussi

bool getLatch(joint& jPosition) (CS8C only)

bool getLatch(joint& jPosition) (CS8C only)

Fonction

Cette instruction lit la dernière position capturée du robot.

La fonction renvoie true si une capture de position valide peut être lue. Si une capture est en attente, ou si la capture n'a jamais été activée, la fonction renvoie false et la position n'est pas mise à jour.

getLatch renvoie la même capture de position tant qu'une nouvelle capture n'est pas activée par l'instruction setLatch.

La position du bras est rafraîchie dans le contrôleur CS8C toutes les 0.2 ms ; la capture de position correspond à la position du bras entre 0 et 0.2 ms après le front montant de l'entrée rapide.

Exemple

```
setLatch(diLatch)
// Wait for setLatch to be effective before using getLatch
delay(0)
// Attend une capture de position pendant 5 secondes.
bLatch = watch(getLatch(jPosition)==true, 5)
if bLatch==true
    putln("Capture de position réussie")
else
    putln("Aucun signal de capture détecté")
endif
```

Voir aussi

void setLatch(dio diEntrée) (CS8C only)
joint herej()

9.3. TYPE TRSF

9.3.1. DÉFINITION

Une transformation (type **trsf**) décrit un changement de position et/ou d'orientation. C'est l'assemblage mathématique d'une translation et d'une rotation.

La transformation ne représente pas elle-même une position dans l'espace, mais elle peut être interprétée comme la position et l'orientation d'un point ou d'un repère cartésien par rapport à un autre repère.

Le type **ttrsf** est un type structuré dont les champs sont, dans l'ordre :

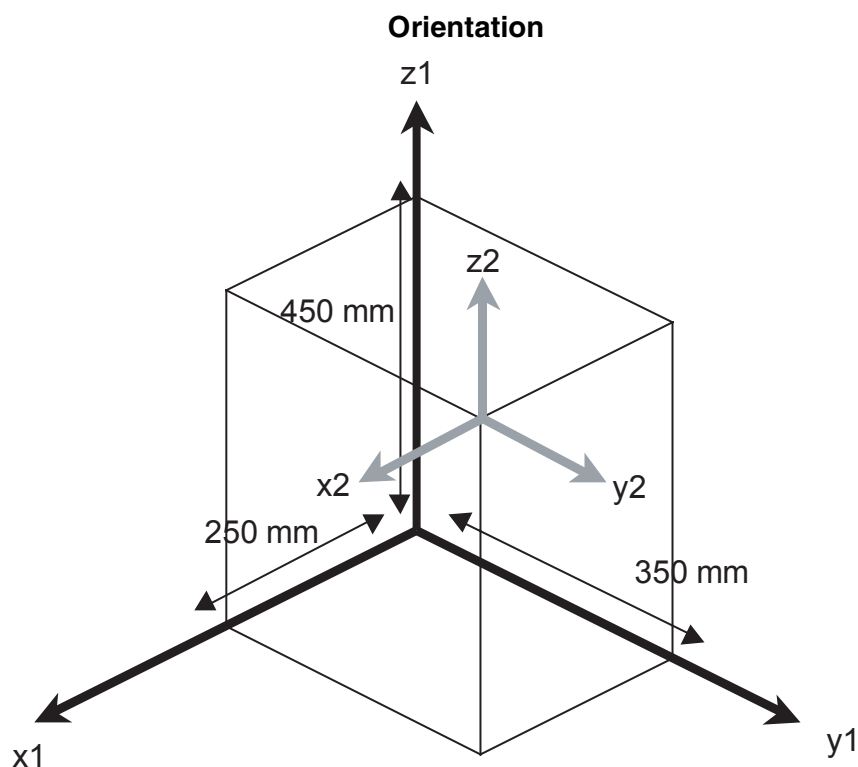
num x	Translation sur l'axe x
num y	Translation sur l'axe y
num z	Translation sur l'axe z
num rx	Rotation autour de l'axe x
num ry	Rotation autour de l'axe y
num rz	Rotation autour de l'axe z

Les champs **x**, **y** et **z** sont exprimés dans l'unité de longueur de l'application (millimètre ou inch, voir chapitre Unité de longueur). Les champs **rx**, **ry** et **rz** sont exprimés en degrés.

Les coordonnées **x**, **y** et **z** sont les coordonnées cartésiennes de la translation (ou la position d'un point ou d'un repère dans le repère de référence). Lorsque **rx**, **ry** et **rz** sont nuls, la transformation est une translation sans changement d'orientation.

Par défaut, une variable de type **trsf** est initialisée à la valeur **{0,0,0,0,0,0}**.

9.3.2. ORIENTATION

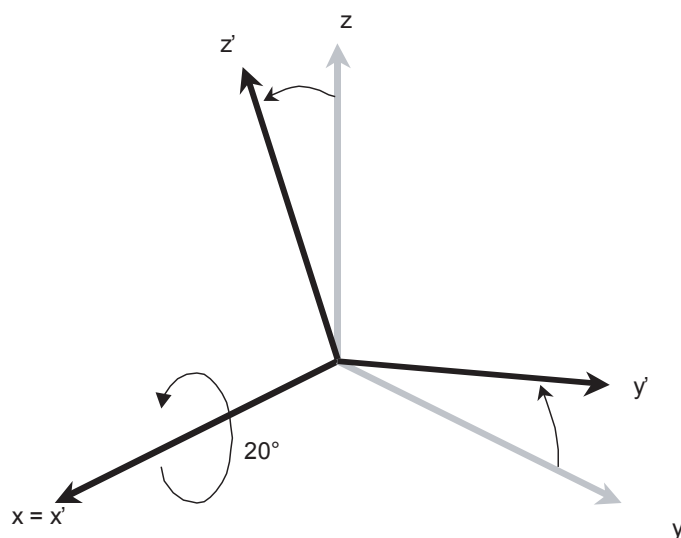


La position du repère **R2**(gris) par rapport à **R1**(noir) est :
 $x = 250 \text{ mm}$, $y = 350 \text{ mm}$, $z = 450 \text{ mm}$, $r_x = 0$, $r_y = 0$, $r_z = 0$

Les coordonnées **r_x** , **r_y** et **r_z** correspondent aux angles de rotation qui doivent être appliqués successivement autour des axes **x** , **y** et **z** pour obtenir l'orientation du repère.

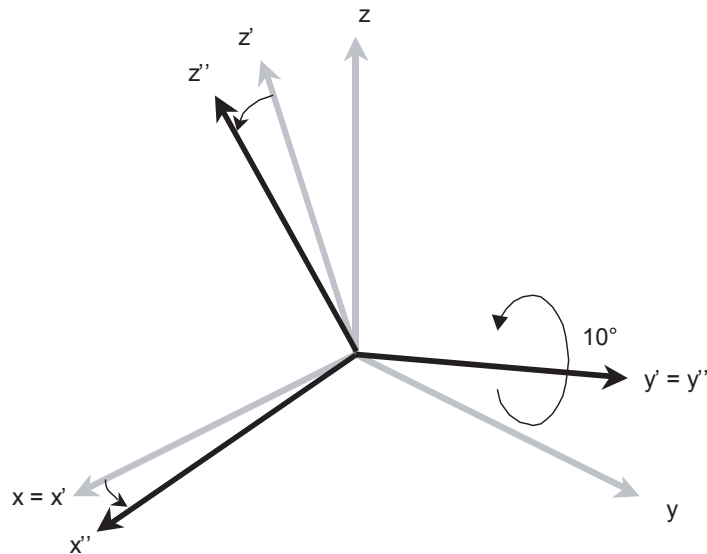
Par exemple, l'orientation **$r_x = 20$** , **$r_y = 10$** , **$r_z = 30$** est obtenue de la manière suivante. Le repère **(x, y, z)** est d'abord tourné de **20** autour de l'axe **x** . On obtient un nouveau repère **(x', y', z')** . Les axes **x** et **x'** sont confondus.

Rotation repère par rapport à l'axe : X



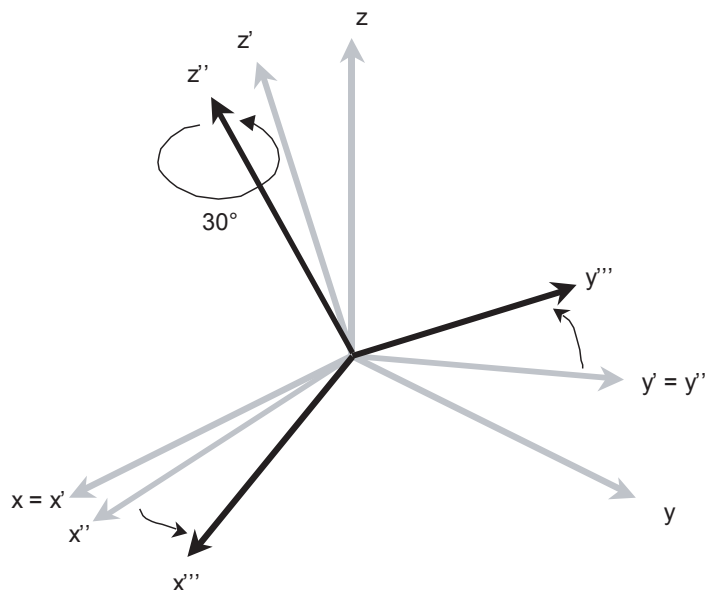
Le repère est ensuite tourné de **20** autour de l'axe **y'** du repère obtenu dans l'étape précédente. On obtient un nouveau repère **(x'', y'', z'')** . Les axes **y'** et **y''** sont confondus.

Rotation repère par rapport à l'axe : Y'



Enfin, le repère est tourné de **20** autour de l'axe **Z''** du repère qu'on a obtenu à l'étape précédente. Le nouveau repère (**X'''**, **Y'''**, **Z'''**) obtenu est celui dont l'orientation est définie par **rx**, **ry**, **rz**. Les axes **Z''** et **Z'''** sont confondus.

Rotation repère par rapport à l'axe : Z''



La position du repère **R2**(gris) par rapport à **R1**(noir) est :
 $x = 250\text{mm}$, $y = 350\text{ mm}$, $z = 450\text{mm}$, $rx = 20$, $ry = 10$, $rz = 30$

Les valeurs de **rx**, **ry** et **rz** sont définies modulo **360** degrés. Lorsque le système calcule **rx**, **ry** et **rz**, leurs valeurs sont toujours comprises entre **-180** et **+180** degrés. Il reste alors encore plusieurs valeurs possibles pour **rx**, **ry**, et **rz** : Le système garantit qu'au moins deux coordonnées se situent entre **-90** et **90** degrés (sauf si **rx** vaut **+180** et si **ry** vaut **0**). Lorsque **ry** vaut **90** degrés (**modulo 180**), **rx** est choisi nul.

9.3.3. OPÉRATEURS

Par ordre de priorité croissant :

trsf < trsf & trPosition1> = < trsf trPosition2>	Affecte trPosition2 à la variable trPosition1 champ par champ et renvoie trPosition2 .
bool < trsf trPosition1> != < trsf trPosition2>	Renvoie true si un champ de trPosition1 n'est pas égal au champ correspondant de trPosition2 , false sinon.
bool < trsf trPosition1> == < trsf trPosition2>	Renvoie true si chaque champ de trPosition1 est égal au champ correspondant de trPosition2 , false sinon.
trsf < trsf trPosition1> * < trsf trPosition2>	Renvoie la composition géométrique des transformations trPosition1 et trPosition2 . Attention trPosition1 * trPosition2 != trPosition2 * trPosition1 dans le cas général!
trsf ! < trsf trPosition>	Renvoie la transformation inverse de trPosition .

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions.

9.3.4. INSTRUCTIONS

num distance(trsf trPosition1, trsf trPosition2)

Fonction

Renvoie la distance entre **trPosition1** et **trPosition2**.

ATTENTION :

Pour que la distance soit valide, il faut que position1 et position2 soient définies par rapport au même repère de référence.

Exemple

Cette ligne calcule la distance entre deux outils :

```
distance(position(tTool1, flange), position(tTool2, flange))
```

Voir aussi

point appro(point pPosition, trsf trTransformation)

point compose(point pPosition, frame fReference, trsf trTransformation)

trsf position(point pPosition, frame fReference)

num distance(point pPosition1, point pPosition2)

trsf interpolateL(trsf trDémarrage, trsf trFin, num nPosition)

Fonction

Cette instruction renvoie une position intermédiaire alignée avec une position de départ **trDémarrage** et une position cible **trFin**. Le paramètre **nPosition** spécifie l'interpolation linéaire à appliquer selon l'équation pour la coordonnée x : $\text{trsf.x0} = \text{trDémarrage.x} + (\text{trFin.x} - \text{trDémarrage.x}) * \text{nPosition}$. La même équation est applicable pour les coordonnées Y et Z.

L'orientation rx, ry, rz est calculée selon une équation similaire mais plus complexe. L'algorithme utilisé par interpolateL est le même que celui utilisé par le générateur de mouvement pour calculer les positions intermédiaires lors d'une instruction movel.

interpolateL(trDémarrage, trFin, 0) renvoie **trDémarrage** ; interpolateL(trDémarrage, trFin, 1) renvoie **trFin** ; interpolateL(trDémarrage, trFin, 0.5) renvoie la position médiane entre **trDémarrage** et **trFin**. Une valeur négative du paramètre **nPosition** donne une position située "avant" **trDémarrage**. Une valeur supérieure à 1 donne une position située "après" **trFin**.

Une erreur d'exécution est générée si le paramètre nPosition ne se trouve pas dans la plage]-1, 2[.

Voir aussi

trsf position(point pPosition, frame fReference)

trsf position(frame fRepere, frame fReference)

trsf position(tool tOutil, tool tReference)

trsf interpolateC(trsf trDémarrage, trsf trIntermédiaire, trsf trFin, num nPosition)

trsf align(trsf trPosition, trsf Reference)

trsf interpolateC(trsf trDémarrage, trsf trIntermédiaire, trsf trFin,
num nPosition)

Fonction

Cette instruction renvoie une position intermédiaire sur l'arc d'un cercle défini par les positions **trDémarrage**, **trIntermédiaire** et **trFin**. Le paramètre **nPosition** indique l'interpolation circulaire à appliquer. L'algorithme utilisé par **interpolateC** est le même que celui utilisé par le générateur de mouvement pour calculer les positions intermédiaires dans une instruction **movec**.

interpolateC(trDémarrage, trIntermédiaire, trFin, 0) renvoie **trDémarrage** ;
interpolateC(trDémarrage, trIntermédiaire, trFin, 1) renvoie **trFin** ;
interpolateC(trDémarrage, trIntermédiaire, trFin, 0.5) renvoie la position médiane sur l'arc entre **trDémarrage** et **trFin**. Une valeur négative du paramètre **nPosition** donne une position située "avant" **trDémarrage**. Une valeur supérieure à 1 donne une position située "après" **trFin**.

Une erreur d'exécution est générée si l'arc n'est pas correctement défini (positions trop proches) ou si l'interpolation de rotation reste indéterminée (voir le chapitre "Contrôle des mouvements - Interpolation de l'orientation").

Voir aussi

trsf position(point pPosition, frame fReference)
trsf position(frame fRepere, frame fReference)
trsf position(tool tOutil, tool tReference)
trsf interpolateL(trsf trDémarrage, trsf trFin, num nPosition)
trsf align(trsf trPosition, trsf Reference)

trsf align(trsf trPosition, trsf Reference)

Fonction

Cette instruction renvoie l'entrée **trPosition** avec une orientation modifiée de telle sorte que l'axe Z de l'orientation renvoyée est aligné sur l'axe X, Y ou Z le plus proche de l'orientation de référence de **trReference**. Les coordonnées X, Y, Z de **trPosition** et **trReference** ne sont pas utilisées : les coordonnées x, y, z de la valeur renvoyée sont les mêmes que les coordonnées x, y, z de **trPosition**.

Voir aussi

trsf position(point pPosition, frame fReference)
trsf position(frame fRepere, frame fReference)
trsf position(tool tOutil, tool tReference)
trsf interpolateL(trsf trDémarrage, trsf trFin, num nPosition)
trsf interpolateC(trsf trDémarrage, trsf trIntermédiaire, trsf trFin, num nPosition)

9.4. TYPE FRAME

9.4.1. DÉFINITION

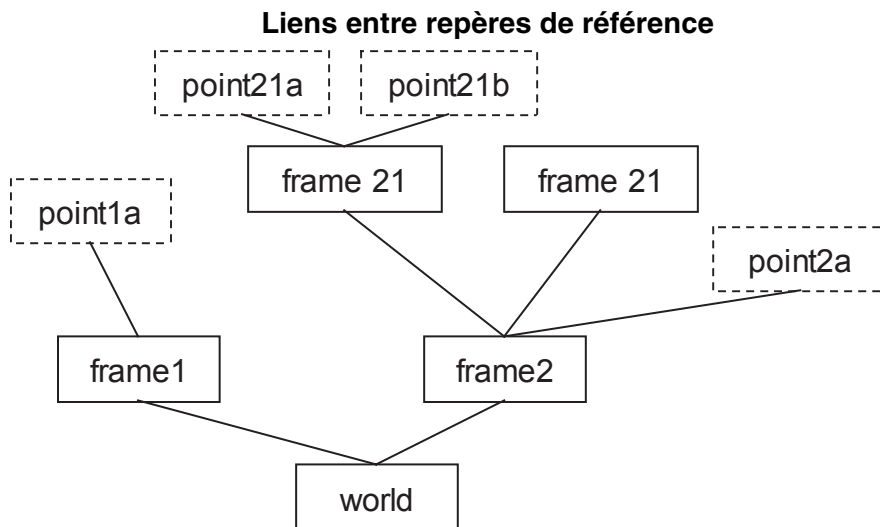
Le type frame permet de définir la position de repères de référence dans la cellule.

Le type frame est un type structuré avec un seul champ accessible :

trsf trsf position du repère dans son repère de référence

Le **repère de référence** d'une variable de type **frame** est défini quand il est initialisé (à l'aide de l'interface utilisateur, de l'opérateur = ou de l'instruction [link\(\)](#)). Le repère de référence **world** est toujours défini dans une application **VAL 3** : tout repère de référence est, directement ou via d'autres repères, lié au repère **world**.

Une erreur d'exécution est générée pendant un calcul géométrique si les coordonnées du repère **world** ont été modifiées.



Par défaut, les variables de repère local et les repères dans les variables de type d'utilisateur n'ont pas de repère de référence. Avant de pouvoir être utilisés, ils doivent être initialisés à partir d'un autre repère à l'aide de l'opérateur '=' ou de l'une des instructions [link\(\)](#) et [setFrame\(\)](#).

9.4.2. UTILISATION

L'utilisation de repères de référence dans une application robotique est vivement recommandée :

- **Pour donner une vue plus intuitive des points de l'application**

L'affichage des points de la cellule est structuré selon l'organisation hiérarchique des repères.

- **Pour mettre à jour rapidement la position d'un ensemble de points**

Dès qu'un point de l'application est lié à un objet, il est souhaitable de définir un repère pour cet objet, et de lier les points **VAL 3** à ce repère. Si l'objet est déplacé, il suffit de réapprendre le repère pour que tous les points qui y sont liés soient corrigés du même coup.

- **Pour reproduire une trajectoire à plusieurs endroits de la cellule**

On peut pour cela définir les points de la trajectoire par rapport à un repère de travail, et apprendre un repère pour chaque endroit où la trajectoire doit être reproduite. En affectant la valeur d'un repère appris au repère de travail, la trajectoire entière se "déplace" sur le repère appris.

- **Pour calculer facilement des déplacements géométriques**

L'instruction [compose\(\)](#) permet d'effectuer sur tout point des déplacements géométriques exprimés dans un repère de référence quelconque. L'instruction [position\(\)](#) permet de calculer la position d'un point dans un repère de référence quelconque.

9.4.3. OPÉRATEURS

Par ordre de priorité croissant :

frame < frame & fReference1 > = < frame fReference2 >	Affecte la position et le repère de référence de fReference2 à la variable fReference1 .
bool < frame fReference1 > != < frame fReference2 >	Renvoie true si fReference1 et fReference2 n'ont pas le même repère de référence ou n'ont pas la même position dans leur repère de référence.
bool < frame fReference1 > == < frame fReference2 >	Renvoie true si fReference1 et fReference2 ont la même position dans le même repère de référence.

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions.

9.4.4. INSTRUCTIONS

num **setFrame**(**point** pOrigine, **point** pAxeOx, **point** pPlanOxy, **frame**& fResultat)

Fonction

Cette instruction calcule les coordonnées de **fResultat** à partir de son origine **pOrigine**, d'un point **pAxeOx** sur l'axe (**Ox**) et d'un point **pPlanOxy** sur l'axe (**Oxy**).

Le point **pAxeOx** doit être du côté des **x** positifs. Le point **pPlanOxy** doit être du côté des **y** positifs.

La fonction renvoie :

- 0** Pas d'erreur.
- 1** Le point **pAxeOx** est trop proche de **pOrigine**.
- 2** Le point **pPlanOxy** est trop proche de l'axe (**Ox**).

Une erreur d'exécution est générée si l'un des points n'a pas de repère de référence.

trsf position(**frame** fRepere, **frame** fReference)

Fonction

Cette instruction renvoie les coordonnées du repère **fRepere** dans le repère de référence **fReference**.

Une erreur d'exécution est générée si **fRepere** ou **fReference** n'a pas de repère de référence.

Voir aussi

trsf position(**point** pPosition, **frame** fReference)
trsf position(**tool** tOutil, **tool** tReference)

void link(**frame** fRepere, **frame** fReference)

Fonction

Cette instruction change le repère de référence de **fRepere** et le règle à **fReference**. La position du repère dans le repère de référence reste inchangée.

Voir aussi

Operator frame <**frame**& **fFrame1**> = <**frame** **fFrame2**>

9.5. TYPE TOOL

9.5.1. DÉFINITION

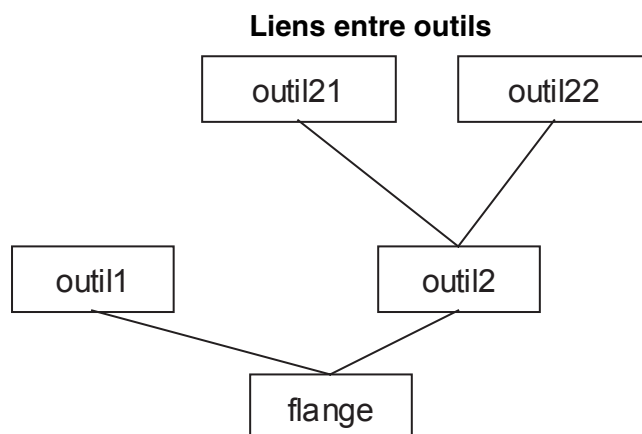
Le type **tool** permet de définir la géométrie et l'action d'un outil.

Le type **tool** est un type structuré avec comme champs, dans l'ordre :

trsf trsf	position du point du centre outil (TCP) dans son outil de base
dio gripper	sortie tout ou rien servant à actionner l'outil
num otime	Délai d'ouverture de l'outil (secondes)
num ctime	Délai de fermeture de l'outil (secondes)

L'**outil de référence** d'une variable de type **tool** est défini quand il est initialisé (à l'aide de l'interface utilisateur, de l'opérateur = ou de l'instruction [link\(\)](#)). L'outil **flange** est toujours défini dans une application **VAL 3** : tout outil est, directement ou via d'autres outils, lié à l'outil **flange**.

Une erreur d'exécution est générée pendant un calcul géométrique si les coordonnées de l'outil **flange** ont été modifiées.



Par défaut, la sortie d'un outil est la sortie **valve1** du système, les temps d'ouverture et de fermeture sont à **0**, et l'outil de base est **flange**. Les variables d'outil local et les outils dans les variables de type d'utilisateur n'ont pas d'outil de référence. Avant de pouvoir être utilisés, ils doivent être initialisés à partir d'un autre outil à l'aide de l'opérateur '=' ou de l'instruction [link\(\)](#).

9.5.2. UTILISATION

L'utilisation d'outils dans une application robotique est vivement recommandée :

- **Pour contrôler la vitesse de déplacement**
Lors de déplacement manuels ou programmés, le système contrôle la vitesse cartésienne en bout d'outil.
- **Pour aller aux mêmes points avec différents outils**
Il suffit de sélectionner l'outil **VAL 3** correspondant à l'outil physique en bout de bras.
- **Pour contrôler l'usure de l'outil ou un changement d'outil**
L'usure de l'outil peut simplement être compensée par l'actualisation des coordonnées géométriques de celui-ci.

9.5.3. OPÉRATEURS

Par ordre de priorité croissant :

tool < tool & tOutil1 > = < tool tOutil2 >	Affecte la position et l'outil de base de tOutil2 à la variable tOutil1 .
bool < tool tOutil1 > != < tool tOutil2 >	Renvoie true si tOutil1 et tOutil2 n'ont pas le même outil de base, la même position dans leur outil de base, la même sortie digitale, ou les mêmes temps d'ouverture et fermeture.
bool < tool tOutil1 > == < tool tOutil2 >	Renvoie true si tOutil1 et tOutil2 ont la même position dans le même outil de base, utilisent la même sortie digitale avec les mêmes temps d'ouverture et fermeture.

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions.

9.5.4. INSTRUCTIONS

void open(tool tOutil)

Fonction

Cette instruction active l'outil (ouverture) en réglant sa sortie numérique à **true**.

Avant d'actionner l'outil, **open()** attend que le robot soit au point en faisant l'équivalent d'un **waitEndMove()**.

Après l'activation, le système attend **otime** secondes avant d'exécuter l'instruction suivante.

Cette instruction n'assure pas que le robot soit stabilisé à sa position finale avant l'activation de l'outil. Lorsqu'il est nécessaire d'attendre la stabilisation complète du mouvement, il faut utiliser l'instruction **isSettled()**.

Une erreur d'exécution est générée si la **dio** de **tOutil** n'est pas définie ou n'est pas une sortie, ou si une commande de mouvement précédemment enregistrée ne peut être exécutée.

Exemple

```
// l'instruction open() est équivalente à :
waitEndMove()
tOutil.gripper=true
delay(tOutil.otime)
```

Voir aussi

void close(tool tOutil)

void waitEndMove()

void close(tool tOutil)

Fonction

Cette instruction active l'outil (fermeture) en réglant sa sortie numérique à **false**.

Avant d'actionner l'outil, **close()** attend que le robot soit arrêté au point en faisant l'équivalent d'un **waitEndMove()**. Après l'activation, le système attend ctime secondes avant d'exécuter l'instruction suivante.

Cette instruction n'assure pas que le robot soit stabilisé à sa position finale avant l'activation de l'outil. Lorsqu'il est nécessaire d'attendre la stabilisation complète du mouvement, il faut utiliser l'instruction **isSettled()**.

Une erreur d'exécution est générée si la **dio** de **tOutil** n'est pas définie ou n'est pas une sortie, ou si une commande de mouvement précédemment enregistrée ne peut être exécutée.

Exemple

```
// l'instruction close est équivalente à :
waitEndMove()
tOutil.gripper = false
delay(tOutil.ctime)
```

Voir aussi

Type tool

void open(tool tOutil)

void waitEndMove()

trsf position(tool tOutil, tool tReference)

Fonction

Cette instruction renvoie les coordonnées de l'outil **tOutil** dans l'outil **tReference**.

Une erreur d'exécution est générée si **tOutil** ou **tReference** n'a pas d'outil de référence.

Voir aussi

trsf position(point pPosition, frame fReference)

trsf position(frame fRepere, frame fReference)

void link(tool tOutil, tool tReference)

Fonction

Cette instruction change l'outil de référence de **tOutil** et le règle à **tReference**. La position de l'outil dans l'outil de référence reste inchangée.

Voir aussi

Operator tool <tool& tOutil1> = <tool tOutil2>

9.6. TYPE POINT

9.6.1. DÉFINITION

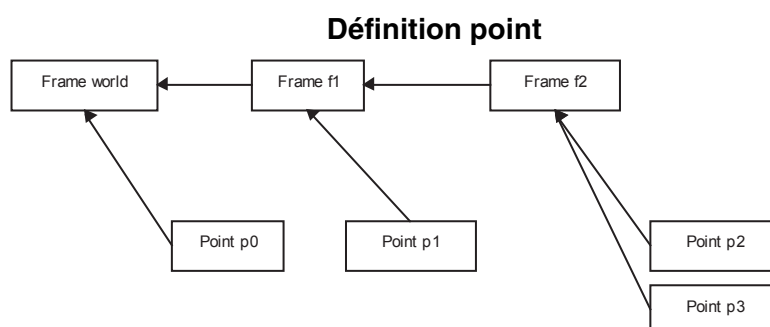
Le type **point** permet de définir la position et l'orientation de l'outil du robot dans la cellule.

Le type **point** est un type structuré avec comme champs, dans l'ordre :

trsf trTrsf position du point dans son repère de référence

config config configuration du bras pour atteindre la position

Le repère de référence d'un **point** est une variable de type **frame**, définie lors de son initialisation (depuis l'interface utilisateur, par l'opérateur = et les instructions **link()**, **here()**, **appro()** et **compose()**).



Une erreur d'exécution est générée si une variable de type **point** sans repère de référence est utilisée.

ATTENTION :

Par défaut, les variables de point local et les points dans les variables de type d'utilisateur n'ont pas de repère de référence. Avant de pouvoir être utilisés, ils doivent être initialisés à partir d'un autre point à l'aide de l'opérateur '=' ou de l'une des instructions **link()**, **here()**, **appro()** et **compose()**.

9.6.2. OPÉRATEURS

Par ordre de priorité croissant :

point <point& pPoint1> = <point pPoint2>	Affecte la position, la configuration et le repère de référence de pPoint2 à la variable pPoint1 .
bool <point pPoint1> != <point pPoint2>	Renvoie true si pPoint1 et pPoint2 n'ont pas le même repère de référence ou n'ont pas la même position dans leur repère de référence.
bool <point pPoint1> == <point pPoint2>	Renvoie true si pPoint1 et pPoint2 ont la même position dans le même repère de référence.

Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions.

9.6.3. INSTRUCTIONS

num distance(point pPosition1, point pPosition2)

Fonction

Cette instruction renvoie la distance entre **pPosition1** et **pPosition2**.

Une erreur d'exécution est générée si **pPosition1** ou **pPosition2** n'a pas de repère de référence défini.

Exemple

Ce programme attend que le bras se rapproche à moins de 10 mm de la position pCible

```
wait(distance(here(tTool,world),pTarget)< 10)
```

Voir aussi

point appro(point pPosition, trsf trTransformation)
point compose(point pPosition, frame fReference, trsf trTransformation)
trsf position(point pPosition, frame fReference)
num distance(trsf trPosition1, trsf trPosition2)

point compose(point pPosition, frame fReference, trsf trTransformation)

Fonction

Cette instruction renvoie le **pPosition** auquel la transformation géométrique **trTransformation** est appliquée par rapport au repère **fReference**.

ATTENTION :

La composante rotation de **tTransformation** modifie en général non seulement l'orientation de **pPosition**, mais aussi ses coordonnées cartésiennes (sauf si **pPosition** se situe sur l'origine de **fReference**).

Si l'on souhaite que **tTransformation** ne modifie que l'orientation de **pPosition**, il faut mettre à jour le résultat avec les coordonnées cartésiennes de **pPosition** (voir exemple).

Le repère de référence et la configuration du point renvoyé sont ceux de **pPosition**.

Une erreur d'exécution est générée si aucun repère de référence n'est défini pour **pPosition**.

Exemple

```
// modification de l'orientation sans modification de Position
pResultat = compose(pPosition,fReference,trTransformation)
pResultat.trsf.x = pPosition.trsf.x
pResultat.trsf.y = pPosition.trsf.y
pResultat.trsf.z = pPosition.trsf.z
// modification de Position sans modification d'orientation
trTransformation.rx = trTransformation.ry =trTransformation.rz = 0
pResultat = compose(pResultat,fReference,trTransformation)
```

Voir aussi

Operator trsf <trsf pPosition1> * <trsf pPosition2>
point appro(point pPosition, trsf trTransformation)

point appro(point pPosition, trsf trTransformation)

Fonction

Cette instruction renvoie un point modifié par transformation géométrique. La transformation est définie par rapport au même repère de référence que le point d'entrée.

Le repère de référence et la configuration du point renvoyé sont ceux du point d'entrée.

Une erreur d'exécution est générée si aucun repère de référence n'est défini pour **pPosition**.

Exemple

```
// Approche :rapprochement à 100 mm au-dessus du point (axe z)
movej(appro(pDestination,{0,0,-100,0,0,0}), flange, mNomDesc)
// Aller au point
move1(pDestination, flange, mNomDesc)
```

Voir aussi

Operator trsf <trsf trPosition1> * <trsf trPosition2>

point compose(point pPosition, frame fReference, trsf trTransformation)

point here(tool tOutil, frame fReference)

Fonction

Cette instruction renvoie la position actuelle de l'outil **tOutil** dans le cadre **fReference** (position commandée et non position mesurée). Le repère de référence du point renvoyé est **fReference**. La configuration du point renvoyé est la configuration courante du bras.

Voir aussi

joint herej()

config config(joint jPosition)

point jointToPoint(tool tOutil, frame fReference, joint jPosition)

point jointToPoint(tool tOutil, frame fReference, joint jPosition)

Fonction

Cette instruction renvoie la position de **tOutil** dans le repère **fReference** quand le bras est dans la position d'articulation **jPosition**.

Le repère de référence du point renvoyé est **fReference**. La configuration du point renvoyé est la configuration du bras dans la position de l'articulation **jPosition**.

Voir aussi

point here(tool tOutil, frame fReference)

bool pointToJoint(tool tOutil, joint jInitial, point pPosition, joint& jResultat)

bool pointToJoint(tool tOutil, joint jInitial, point pPosition, joint& jResultat)

Fonction

Cette instruction calcule la position d'articulation **jResultat** correspondant au point **pPosition** spécifié. Elle renvoie **true** si **jResultat** est actualisé ou **false** si aucune solution n'a été trouvée.

La position articulaire à calculer correspond à la localisation du **pPosition**. Les champs à la valeur **free** n'imposent pas la configuration. Les champs à la valeur **same** imposent la même configuration que **jInitial**.

Pour un axe capable d'effectuer plus d'un tour complet, il existe plusieurs solutions d'articulation ayant exactement la même configuration : la solution la plus proche de **jInitial** est alors prise.

Il peut ne pas y avoir de solution si **pPosition** est hors d'atteinte (bras trop court) ou hors des butées logicielles. Si **pPosition** spécifie une configuration, il peut être hors des butées pour cette configuration, mais dans les butées pour une autre configuration.

Une erreur d'exécution est générée si aucun repère de référence n'est défini pour **pPosition**.

Voir aussi

joint herej()

point jointToPoint(tool tOutil, frame fReference, joint jPosition)

trsf position(point pPosition, frame fReference)

Fonction

Cette instruction renvoie les coordonnées de **pPosition** dans le repère **fReference**.

Une erreur d'exécution est générée si **pPosition** n'a pas de repère de référence.

Exemple

La distance entre 2 points est la distance entre leurs positions dans world :

```
distance(position(pPoint1, world), position(pPoint2, world)) est distance(pPoint1, pPoint2)
```

Voir aussi

num distance(point pPosition1, point pPosition2)

trsf position(tool tOutil, tool tReference)

trsf position(frame fRepere, frame fReference)

void link(point pPoint, frame fReference)

Fonction

Cette instruction change le repère de référence de **pPoint** et le règle à **fReference**. La position du point dans le repère de référence reste inchangée.

Voir aussi

Operator point <point& pPoint1> = <point pPoint2>

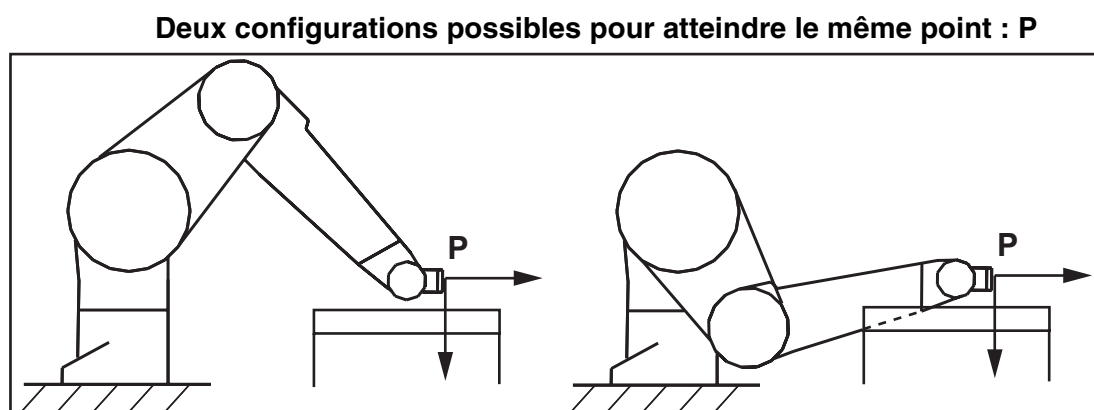
9.7. TYPE CONFIG

La notion de configuration d'un point cartésien est une notion "avancée" qui peut être omise en première lecture.

9.7.1. INTRODUCTION

En général, le robot a plusieurs possibilités pour atteindre une position cartésienne donnée.

Ces différentes possibilités sont appelées « configurations ». Deux configurations différentes sont représentées sur la figure suivante :



Dans certains cas, il est important de spécifier, parmi toutes les configurations possibles, celles qui sont valides et celles que l'on veut interdire. Pour résoudre ce problème, le type **point** permet de spécifier les configurations admises pour le robot grâce à son champ de type **config** défini ci-dessous.

9.7.2. DÉFINITION

Le type **config** permet de définir les configurations autorisées pour une positions cartésienne donnée.

Il dépend du type de bras utilisé.

Pour un bras **Stäubli RX/TX** le type **config** est un type structuré dont les champs sont, dans l'ordre :

shoulder	configuration de l'épaule
elbow	configuration du coude
wrist	configuration du poignet

Pour un bras **Stäubli RS/TS**, le type **config** se limite au champ **Shoulder** :

shoulder	configuration de l'épaule
-----------------	---------------------------

Les champs **shoulder**, **elbow** et **wrist** peuvent prendre les valeurs suivantes :

shoulder	righty	Configuration de l'épaule righty imposée
	lefty	Configuration de l'épaule lefty imposée
	ssame	Changement de configuration de l'épaule interdit
	sfree	Configuration de l'épaule libre

elbow	epositive	Configuration du coude epositive imposée
	enegative	Configuration du coude enegative imposée
	esame	Changement de configuration du coude interdit
	efree	Configuration du coude libre

wrist	wpositive	Configuration du poignet wpositive imposée
	wnegative	Configuration du poignet wnegative imposée
	wsame	Changement de configuration du poignet interdit
	wfree	Configuration du poignet libre

9.7.3. OPÉRATEURS

Par ordre de priorité croissant :

config <config& configuration1> = <config configuration2>	Affecte les champs shoulder , elbow et wrist de configuration2 à la variable configuration1 .
bool <config configuration1> != <config configuration2>	Renvoie true si configuration1 et configuration2 n'ont pas la même valeur de champ shoulder , elbow ou wrist .
bool <config configuration1> == <config configuration2>	Renvoie true si configuration1 et configuration2 ont les mêmes valeurs de champs shoulder , elbow et wrist .

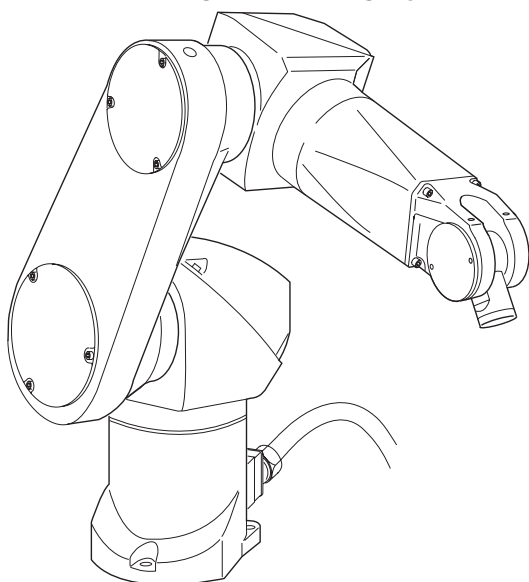
Afin d'éviter les confusions entre les opérateurs = et ==, l'opérateur = n'est pas autorisé dans les expressions de **VAL 3** servant de paramètre d'instructions.

9.7.4. CONFIGURATION (BRAS RX/TX)

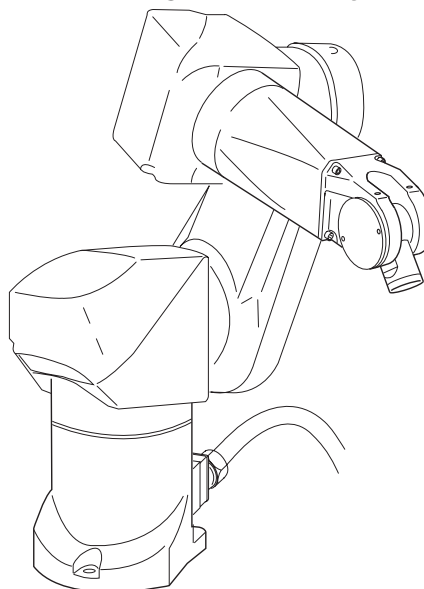
9.7.4.1. CONFIGURATION DE L'ÉPAULE

Pour atteindre un point cartésien, le bras du robot peut être à droite ou à gauche du point : ces deux configurations sont appelées **righty** et **lefty**.

Configuration : righty



Configuration : lefty

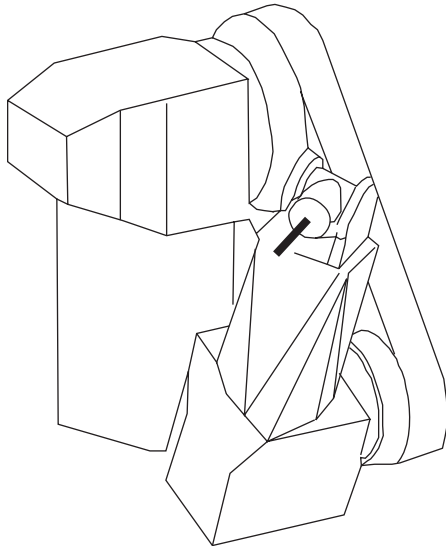


La configuration **righty** est définie par $(d1 * \sin(j2) + d2 * \sin(j2+j3) + \delta) < 0$, la configuration **lefty** est définie par $(d1 * \sin(j2) + d2 * \sin(j2+j3) + \delta) \geq 0$, où $d1$ est la longueur du bras du robot, $d2$ est la longueur de l'avant-bras, et δ est la distance entre l'axe 1 et l'axe 2, dans la direction x.

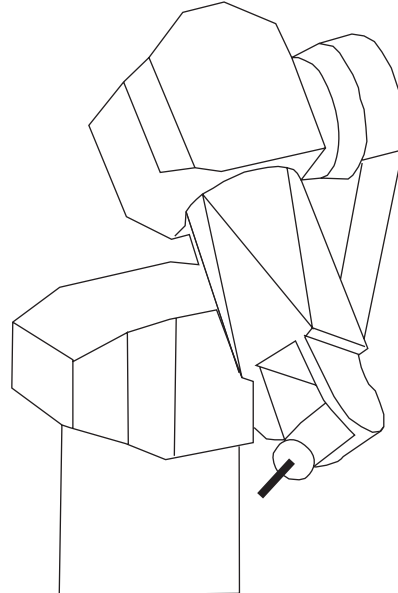
9.7.4.2. CONFIGURATION DU COUDE

En plus de la configuration de l'épaule, il y a deux possibilités pour le coude du robot : les configurations du coude sont appelées **epositive** et **enegative**.

Configuration : enegative



Configuration : epositive



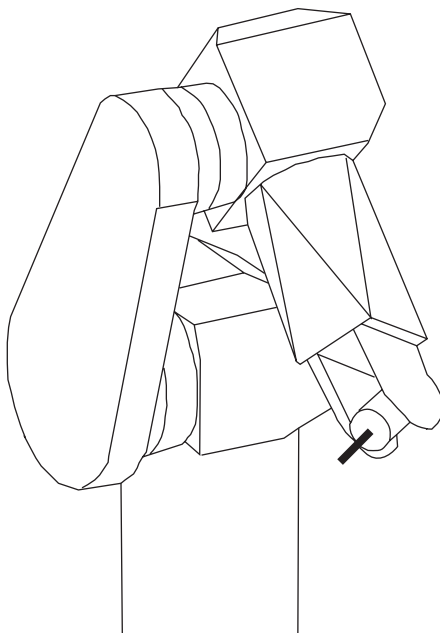
La configuration **epositive** est définie par $j3 \geq 0$.

La configuration **enegative** est définie par $j3 < 0$.

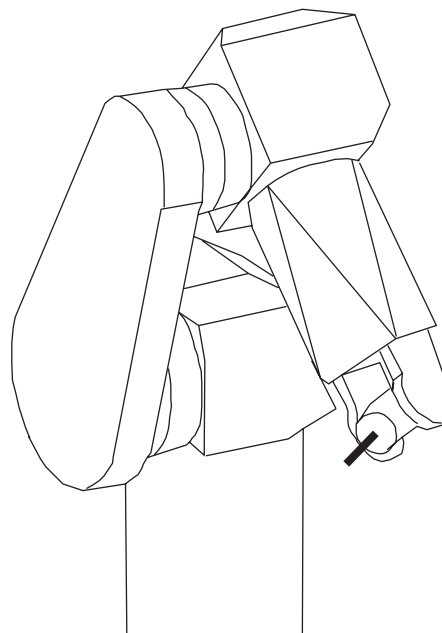
9.7.4.3. CONFIGURATION DU POIGNET

En plus de la configuration de l'épaule et du coude, il y a deux possibilités pour positionner le poignet du robot. Les deux configurations du poignet sont appelées **wpositive** et **wnegative**.

Configuration : wnegative



Configuration : wpositive



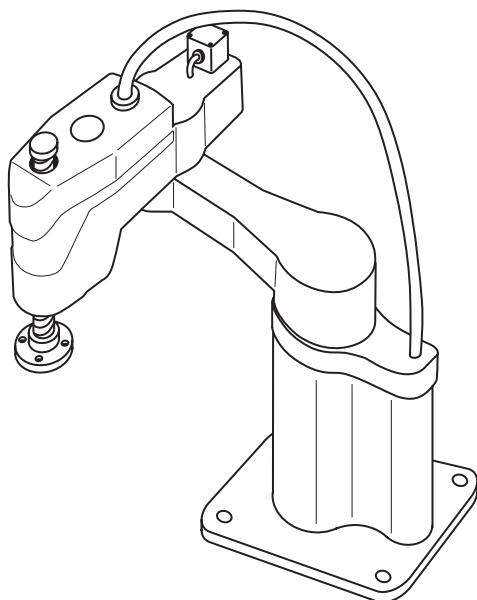
La configuration **wpositive** est définie par $j5 \geq 0$.

La configuration **wnegative** est définie par $j5 < 0$.

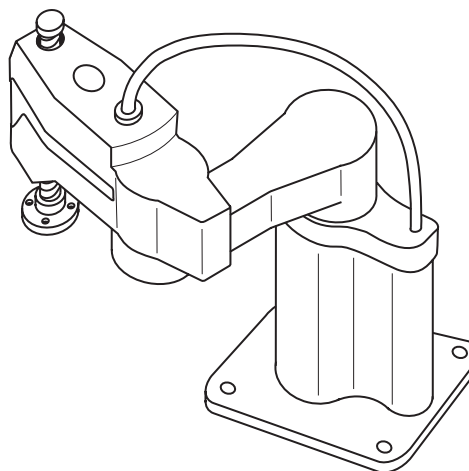
9.7.5. CONFIGURATION (BRAS RS/TS)

Pour atteindre un point cartésien, le bras du robot peut être à droite ou à gauche du point : ces deux configurations sont appelées **righty** et **lefty**.

Configuration : righty



Configuration : lefty



La configuration **righty** est définie par $\sin(j2) > 0$, la configuration **lefty** est définie par $\sin(j2) \leq 0$.

9.7.6. INSTRUCTIONS

config config(joint jPosition)

Fonction

Cette instruction renvoie la configuration du robot pour la position de l'articulation **jPosition**.

Voir aussi

point here(tool tOutil, frame fReference)

joint herej()

CHAPITRE 10

CONTRÔLE DES MOUVEMENTS

10.1. CONTRÔLE DE TRAJECTOIRE

Il ne suffit pas de donner une succession de points pour définir la trajectoire d'un robot. Il faut aussi indiquer le type de trajectoire utilisée entre les points (courbe ou ligne droite), spécifier comment ces trajectoires se raccordent entre elles, et enfin définir les paramètres se rapportant à la vitesse du mouvement. Ce paragraphe présente donc les différents types de mouvements (instructions **movej**, **movel** et **movec**) et explique comment utiliser les paramètres du descripteur de mouvement (type **mdesc**).

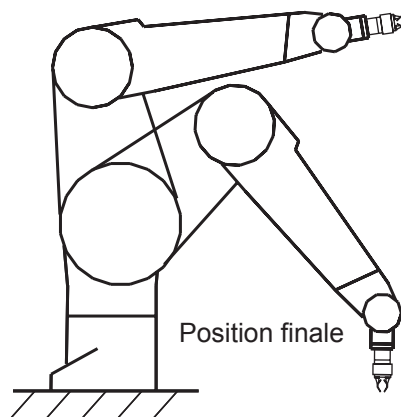
10.1.1. TYPES DE MOUVEMENT : POINT-À-POINT, LIGNE DROITE, CERCLE

Les mouvements du robot se programment essentiellement à l'aide des instructions **movej**, **movel** et **movec**. L'instruction **movej** permet de faire des mouvement point-à-point, **movel** s'utilise pour les mouvements en ligne droite et **movec** sur les mouvements circulaires.

Un mouvement de point à point est un mouvement dans lequel seule la destination finale (coordonnées cartésiennes ou position de l'articulation) est importante. Entre le point de départ et le point d'arrivée, le centre outil suit une courbe définie par le système de manière à optimiser la vitesse du mouvement.

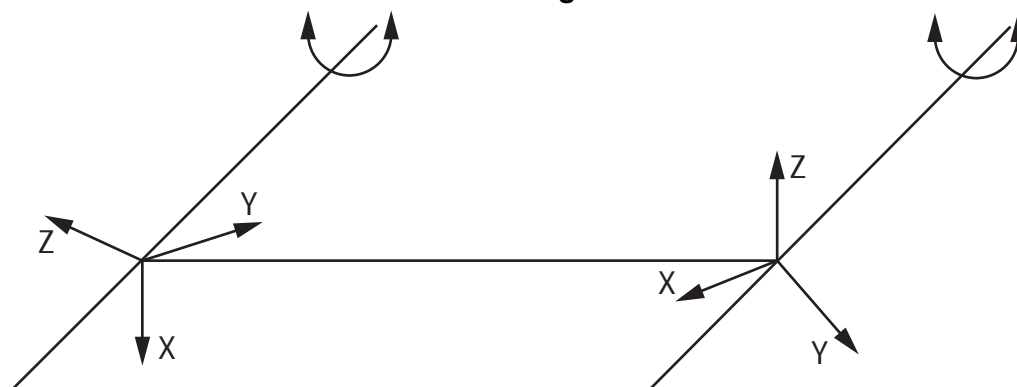
Position initiale et finale

Position initiale



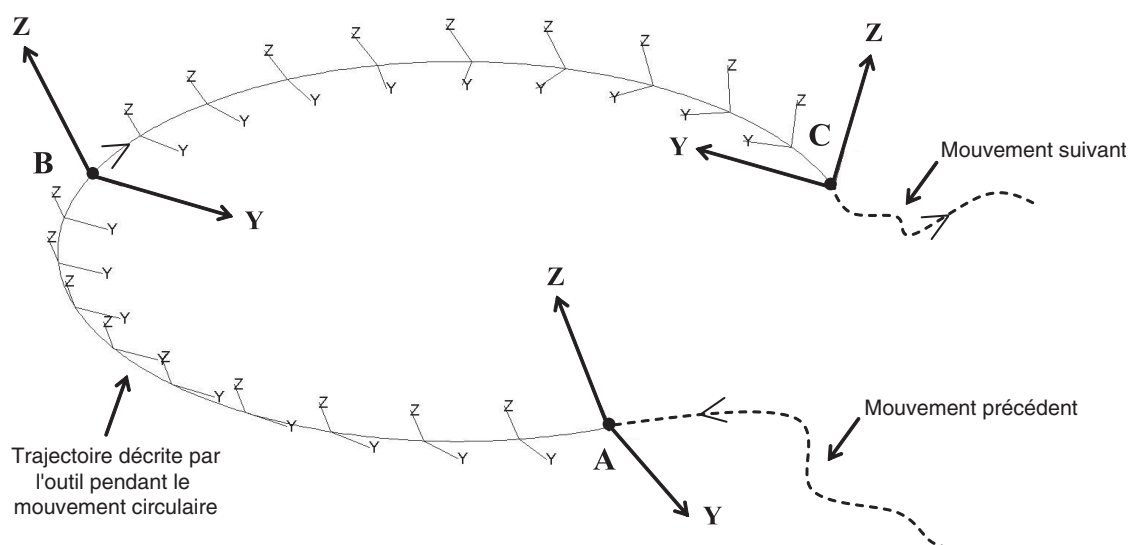
Au contraire, dans un mouvement en ligne droite, le centre outil se déplace le long d'une ligne droite. L'orientation est interpolée linéairement entre l'orientation de départ et l'orientation finale de l'outil.

Mouvement en ligne droite



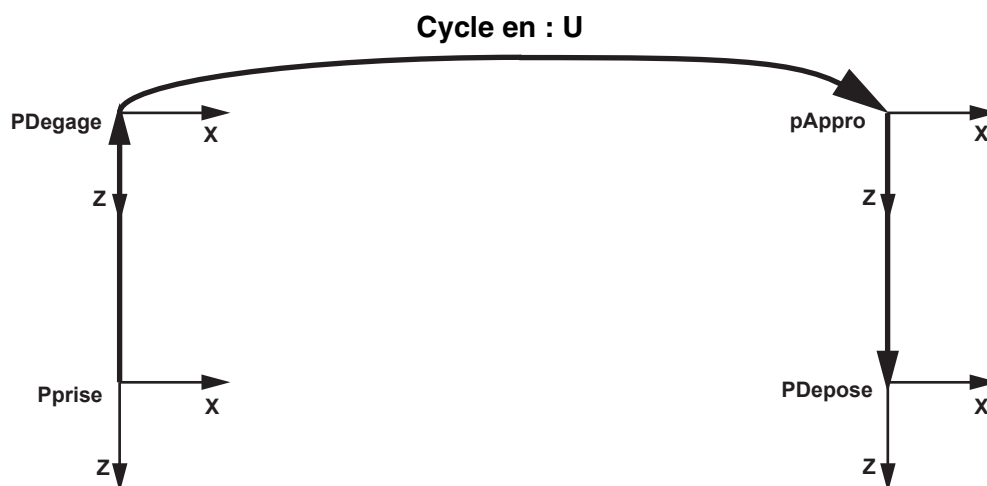
Dans un mouvement circulaire, le centre outil se déplace le long d'un arc de cercle défini par **3** points, et l'orientation outil est interpolée entre l'orientation de départ, l'orientation intermédiaire et l'orientation finale.

Mouvement circulaire



Exemple :

Une tâche de manipulation typique consiste à prendre des pièces à un endroit donné, et les déposer à un autre endroit. On suppose qu'on doit prendre les pièces au point **pPrise**, et les déposer au point **pDépose**. Pour aller du point **pPrise** au point **pDépose**, le robot doit passer par un point de dégagement **pDégage** et un point d'approche **pAppro**.



On suppose que le robot est initialement au point **pPrise**. Le programme pour exécuter le mouvement peut s'écrire :

```

movel(pDégage, tOutil, mDesc)
movej(pAppro, tOutil, mDesc)
movel(pDépose, tOutil, mDesc)
  
```


On utilise des mouvements en ligne droite pour le dégagement et l'approche. En revanche, le mouvement principal est un mouvement point à point car il n'est pas nécessaire de contrôler précisément la géométrie de cette portion de la trajectoire, l'objectif étant d'aller le plus vite possible.

Note :

Pour les deux types de mouvements, la géométrie de la trajectoire ne dépend pas de la vitesse à laquelle les mouvements sont exécutés. Le robot passe toujours au même endroit. Ceci est particulièrement important au moment du développement des applications. On peut commencer par exécuter les mouvements à faible vitesse, puis augmenter progressivement la vitesse sans déformer la trajectoire du robot.

10.1.2. ENCHAÎNEMENT DE MOUVEMENTS : LISSAGE

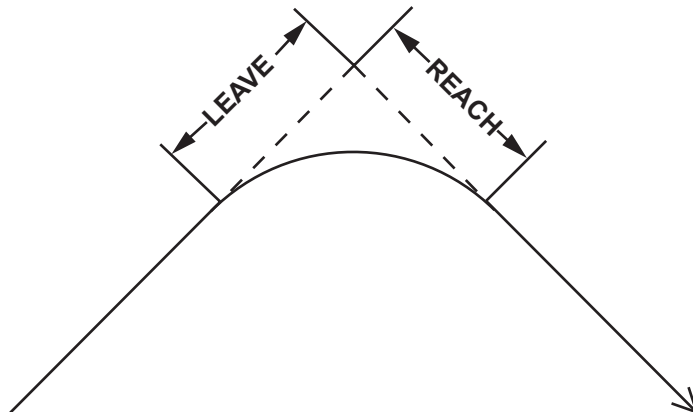
10.1.2.1. LISSAGE

Reprenons l'exemple de cycle en **U** du chapitre précédent. Sans gestion particulière de l'enchaînement des mouvements, le robot va s'arrêter aux points **pDegage** et **pAppro**, car la trajectoire présente des angles en ces points. Ceci augmente inutilement la durée de l'opération, et il n'est pas utile de passer exactement par ces points.

Il est possible de réduire de manière importante la durée du mouvement en "lissant" la trajectoire au voisinage des points **pDegage** et **pAppro**. Pour cela, on utilise le champ **blend** du descripteur de mouvement. Lorsque la valeur de ce champ est **off**, la trajectoire du robot s'arrête à chaque point. En revanche, quand le paramètre est réglé sur **joint** ou **Cartesian**, la trajectoire est lissée au voisinage de chaque point et le robot ne s'arrête plus aux points de passage.

Quand le champ **blend** a la valeur **joint** ou **Cartesian**, deux autres paramètres doivent être définis : **leave** et **reach**. Ces paramètres déterminent à quelle distance du point d'arrivée on quitte la trajectoire nominale (début du lissage), et à quelle distance du point d'arrivée on la rejoint (fin du lissage).

Définition des distances : 'leave' / 'reach'

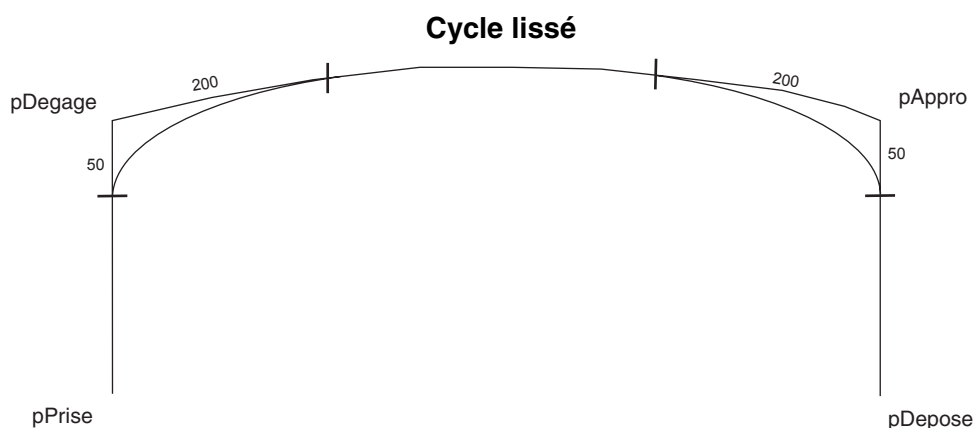


Exemple :

Reprenons le programme du paragraphe "Types de mouvement : point-à-point ou ligne droite". On peut modifier le programme de mouvement précédent :

```
mDesc.blend = joint
mDesc.leave = 50
mDesc.reach = 200
move1(pDegage, tOutil, mDesc)
mDesc.leave = 200
mDesc.reach = 50
movej(pAppro, tOutil, mDesc)
mDesc.blend = off
move1(pDepose, tOutil, mDesc)
```

On obtient alors la trajectoire suivante :



Le robot ne s'arrête plus aux points **pDegage** et **pAppro**. Le mouvement est donc plus rapide. Il sera d'ailleurs d'autant plus rapide que les distances **leave** et **reach** sont grandes.

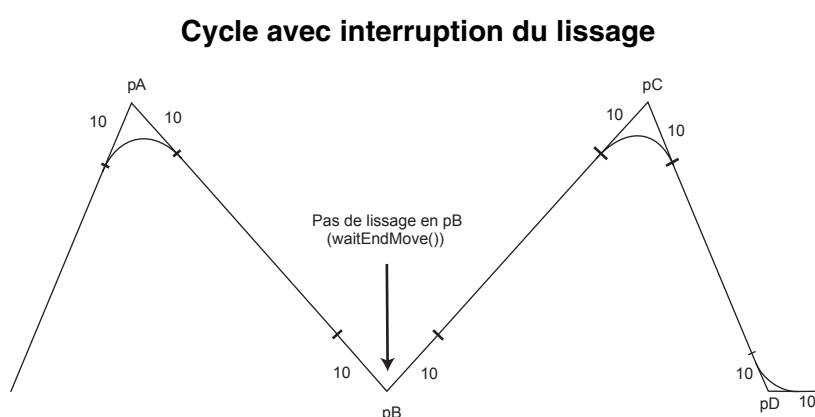
10.1.2.2. ANNULATION DU LISSAGE

L'instruction **waitEndMove()** permet, entre autre, d'annuler l'effet du lissage. Le robot termine alors complètement le dernier mouvement programmé jusqu'à son point d'arrivée, comme si le champ **blend** du descripteur de mouvement avait été mis à la valeur **off**.

Par exemple, considérons le programme suivant :

```
mDesc.blend = joint
mDesc.leave = 10
mDesc.reach = 10
movej(pA, tOutil, mDesc)
movej(pB, tOutil, mDesc)
waitEndMove()
movej(pC, tOutil, mDesc)
movej(pD, tOutil, mDesc)
etc.
```

La trajectoire suivie par le robot est alors la suivante :



10.1.2.3. LISSAGE DE L'ARTICULATION, LISSAGE CARTÉSIEN

Pour simplifier, un lissage d'articulation est comme un mouvement de point à point entre les points de départ et de destination et le lissage cartésien comme un déplacement circulaire entre ces points.

- Le lissage de l'articulation est habituellement plus rapide que le lissage cartésien. Il peut toutefois donner un trajet erratique en cas de changement complexe de l'orientation (habituellement un cercle dans un plan, suivi d'un cercle dans un plan perpendiculaire) ou pour les mouvements de rotation pure.

- Le contrôle de la vitesse et de l'accélération est plus précis avec le lissage cartésien. En outre, on est assuré qu'un lissage cartésien entre deux mouvements dans le même plan se situera dans ce plan.

La forme de lissage optimale dépend de l'application, mais l'interpréteur VAL 3 doit choisir cette forme automatiquement. Le résultat est rarement surprenant mais il peut arriver qu'il le soit...

Le choix peut donner une forme d'une complexité inattendue, avec des distances de départ et d'arrivée très différentes. La forme calculée réduit la courbure de la trajectoire afin d'optimiser la vitesse, mais le résultat peut ne pas être souhaitable pour certaines applications de processus. Quand les distances de départ et d'arrivée sont égales, le lissage cartésien donne toujours une forme simple.

Le lissage cartésien porte sur la position et sur l'orientation. Un changement d'orientation complexe peut influencer sur la forme du lissage et donner des résultats inattendus. Il existe également quelques restrictions concernant le changement d'orientation : dans un cercle, par exemple, les changements importants d'orientation peuvent donner une erreur de mouvement quand plusieurs solutions sont possibles mais le système n'a aucun critère pour en choisir une. Dans une telle situation, un ou plusieurs points intermédiaires supplémentaires sont nécessaires pour aider le système à trouver l'interpolation d'orientation correcte.

10.1.3. REPRISE DE MOUVEMENT

Lorsque la puissance du bras est coupée alors que le robot n'a pas fini ses mouvements, suite à un arrêt d'urgence par exemple, une reprise de mouvement doit se faire à la remise sous puissance. Si le bras a été déplacé manuellement pendant l'arrêt, il peut se trouver à une position éloignée de sa trajectoire normale. Il faut alors s'assurer que la reprise de mouvement peut se faire sans collision. Le contrôle de trajectoire du contrôleur **VAL 3** donne la possibilité de gérer la reprise de mouvement à l'aide d'un " mouvement de connexion ".

A la reprise du mouvement, le système s'assure que le robot est bien sur la trajectoire programmée : en cas d'écart, même minime, il enregistre automatiquement une commande de mouvement point-à-point vers la position exacte où le robot a quitté sa trajectoire : c'est le " mouvement de connexion ". Ce mouvement se fait à vitesse réduite. Il doit être validé par l'opérateur, sauf en mode automatique, où il peut se faire sans intervention humaine. L'instruction **autoConnectMove()** permet de préciser le comportement en mode automatique.

L'instruction **resetMotion()** permet d'annuler le mouvement en cours, et éventuellement de programmer un mouvement de connexion permettant de rejoindre une position à vitesse réduite et sous le contrôle de l'opérateur.

10.1.4. PARTICULARITÉS DES MOUVEMENTS CARTÉSIENS (LIGNE DROITE, CERCLE)

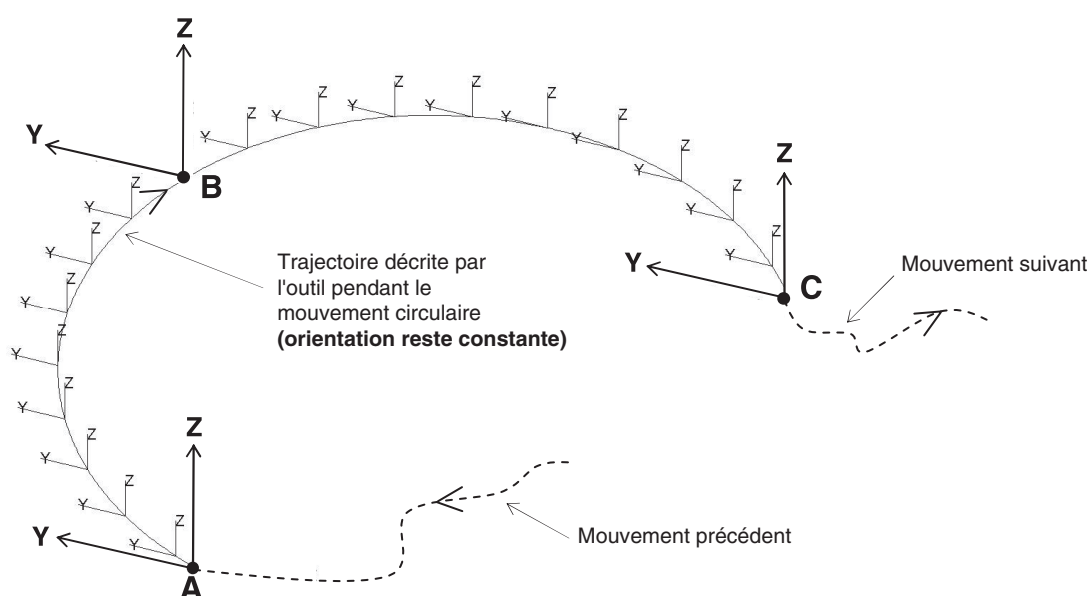
10.1.4.1. INTERPOLATION DE L'ORIENTATION

Le générateur de trajectoire de la **VAL 3** minimise toujours l'amplitude des rotations de l'outil pour passer d'une orientation à une autre.

Ceci permet, comme cas particulier, de programmer une orientation constante, en absolu, ou par rapport à la trajectoire, sur tous les mouvements en ligne droite ou circulaires.

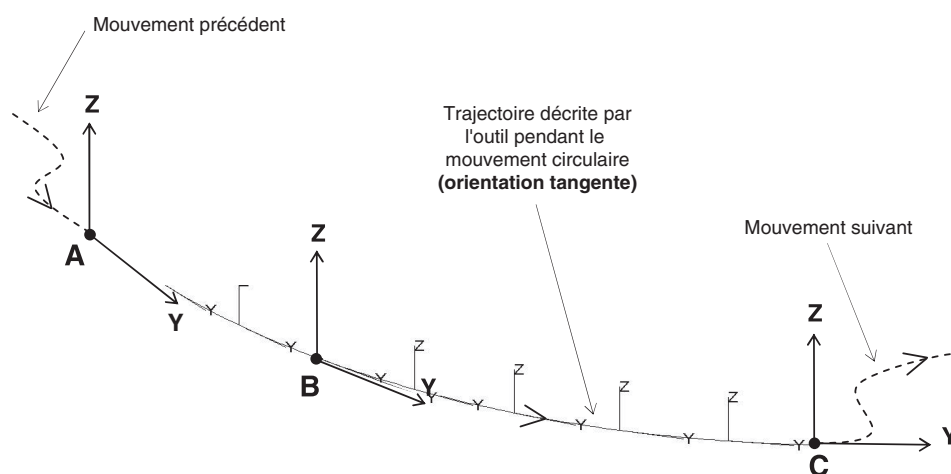
- Pour une orientation constante, les positions de départ, d'arrivée et la position intermédiaire pour un cercle, doivent avoir la même orientation.

Orientation constante en absolu



- Pour une orientation constante par rapport à la trajectoire (par exemple direction Y du repère outil tangente à la trajectoire), les positions de départ, d'arrivée et la position intermédiaire pour un cercle, doivent avoir la même orientation par rapport à la trajectoire.

Orientation constante par rapport à la trajectoire



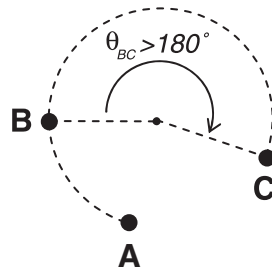
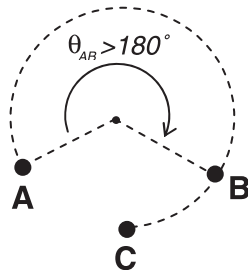
Il en découle une limitation pour les mouvements circulaires :

Si le point intermédiaire fait un angle de **180°** ou plus avec le point de départ ou le point d'arrivée, il y a plusieurs solutions d'interpolation de l'orientation, et une erreur est générée.

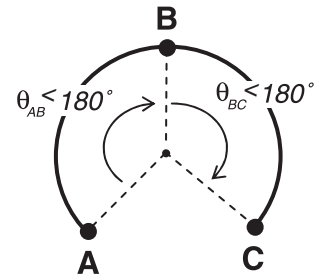
Il faut alors modifier la position du point intermédiaire pour lever l'ambiguïté sur les orientations intermédiaires.

Ambiguïté sur l'orientation intermédiaire

Erreur : mouvements circulaires



OK !

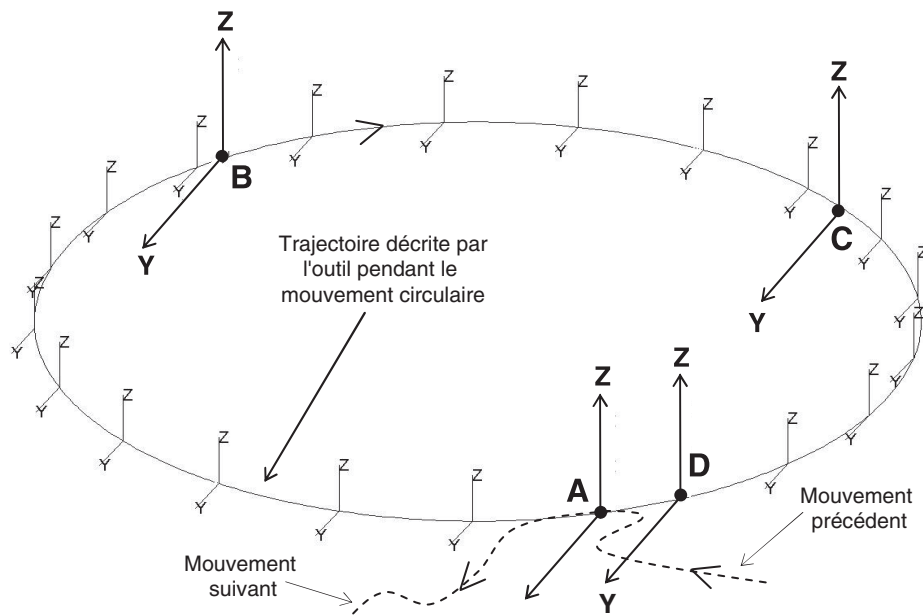


En particulier, la programmation d'un cercle complet nécessite **2** instructions **movec** :

```
movec (B, C, tOutil, mDesc)
```

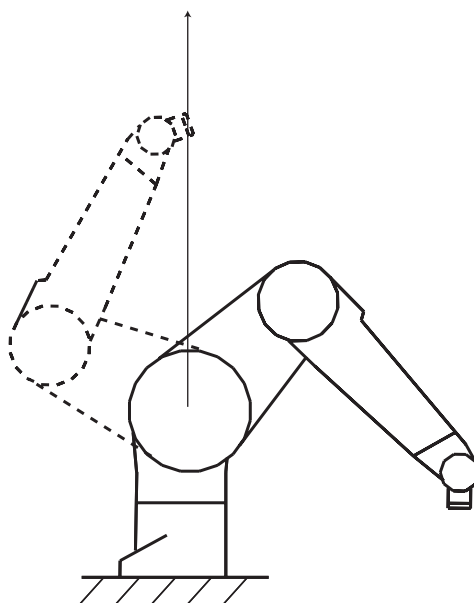
```
movec (D, A, tOutil, mDesc)
```

Cercle complet



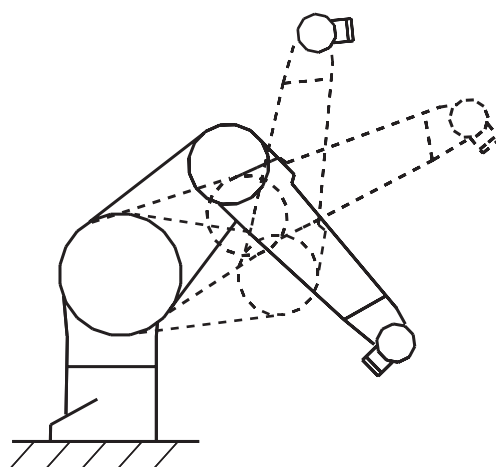
10.1.4.2. CHANGEMENT DE CONFIGURATION (BRAS RX/TX)

Changement : righty / lefty



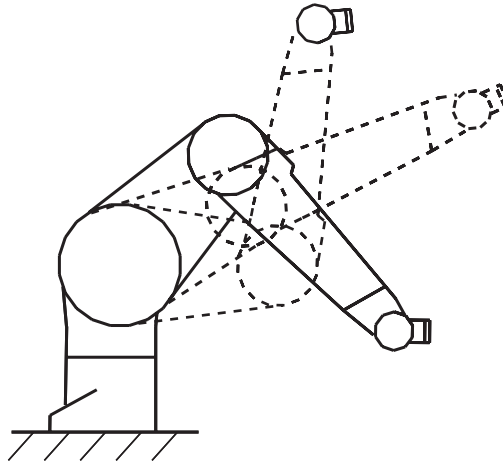
Lors d'un changement de la configuration de l'épaule, le centre du poignet du robot passe nécessairement à la verticale de l'axe 1 (pas exactement pour les robots avec déport).

Changement positive/negative du coude



Lors d'un changement de la configuration du coude, le bras passe nécessairement par la position bras tendu ($j_3 = 0$).

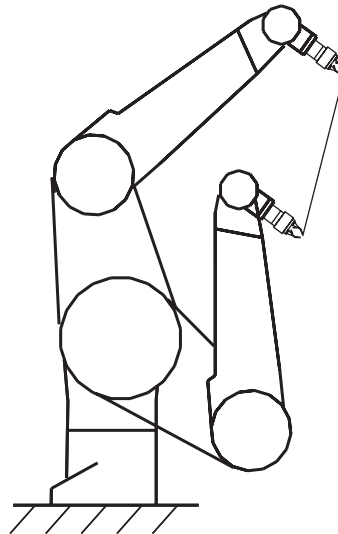
Changement positive/negative du poignet



Lors d'un changement de la configuration du poignet, le bras passe nécessairement par la position poignet tendu ($j_5 = 0$).

Ainsi pour changer de configuration, le robot doit obligatoirement passer par des positions particulières. Mais on ne peut pas imposer qu'un mouvement en ligne droite ou circulaire passe par ces positions si elles ne sont pas sur la trajectoire désirée! En conséquence, **on ne peut pas imposer un changement de configuration dans un mouvement en ligne droite ou circulaire.**

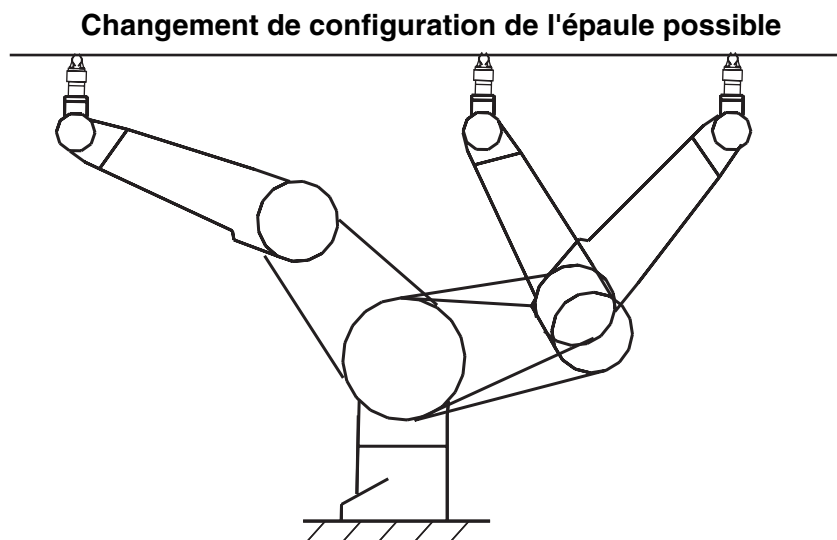
Changement de configuration du coude impossible



Autrement dit, dans un mouvement en ligne droite ou circulaire, on ne peut imposer une configuration que si elle est compatible avec la position de départ : on peut donc toujours spécifier une configuration libre, ou bien identique à celle de départ.

Dans certains cas exceptionnels, la ligne droite ou l'arc de cercle passe effectivement par une position où un changement de configuration est possible. Dans ce cas, si la configuration a été laissée libre, le système peut décider de changer de configuration dans un mouvement en ligne droite ou circulaire.

Pour un mouvement circulaire, la configuration du point intermédiaire n'est pas prise en compte. Seules comptent les configurations des positions de départ et d'arrivée.



10.1.4.3. SINGULARITÉS (BRAS RX/TX)

Les singularités sont une caractéristique inhérente à tous les robots **6** axes. Les singularités peuvent être définies comme les points où le robot change de configuration. Alors certains axes se trouvent alignés : deux axes alignés se comportent comme un seul axe, et le robot **6** axes se comporte donc localement comme un robot **5** axes. Certains mouvements de l'effecteur sont alors impossibles à réaliser. Cela n'est pas gênant dans un mouvement point-à-point : les mouvements générés par le système sont toujours possibles. En revanche, dans un mouvement en ligne droite ou circulaire, on impose la géométrie du mouvement. Si le mouvement est impossible, une erreur est générée lors du mouvement du robot.

10.2. ANTICIPATION DES MOUVEMENTS

10.2.1. PRINCIPE

Le système contrôle les mouvements du robot un peu comme un pilote qui conduit une voiture. Il adapte la vitesse du robot à la géométrie de la trajectoire. Aussi, plus la trajectoire est connue à l'avance, plus le système peut optimiser la vitesse du mouvement. C'est pourquoi le système n'attend pas que le mouvement en cours du robot soit terminé pour prendre en compte les prochaines instructions de mouvement.

Considérons les lignes de programme suivantes :

```
movej (pA, tOutil, mDesc)
movej (pB, tOutil, mDesc)
movej (pC, tOutil, mDesc)
movej (pD, tOutil, mDesc)
```

On suppose que le robot est à l'arrêt lorsque l'exécution du programme atteint ces lignes. Lorsque la première instruction est exécutée, le robot commence son mouvement vers le point **pA**. L'exécution du programme se poursuit alors immédiatement avec la deuxième ligne, bien avant que le robot n'atteigne le point **pA**.

Lorsque le système exécute la deuxième ligne, le robot est en train de commencer son mouvement vers **pA** et le système enregistre, qu'après le point **pA**, le robot devra aller au point **pB**. L'exécution du programme se poursuit alors avec la ligne suivante : alors que le robot continue toujours son mouvement vers **pA**, le système enregistre qu'après le mouvement vers **pB**, le robot devra se diriger en **pC**. Comme l'exécution du programme est beaucoup plus rapide que les mouvements réels du robot, il est probable que le robot soit toujours en train de se diriger vers **pA** au moment où la ligne suivante sera exécutée. Le système enregistrera ainsi les points successifs suivants.

Ainsi, au moment où le robot commence son mouvement vers **pA**, il 'sait' déjà qu'après **pA**, il devra aller successivement en **pB**, **pC** puis **pD**. Si le lissage a été activé, le système sait que le robot ne s'arrêtera pas avant le point **pD**. Il peut alors accélérer beaucoup plus que s'il devait se préparer à s'arrêter en **pB** ou **pC**.

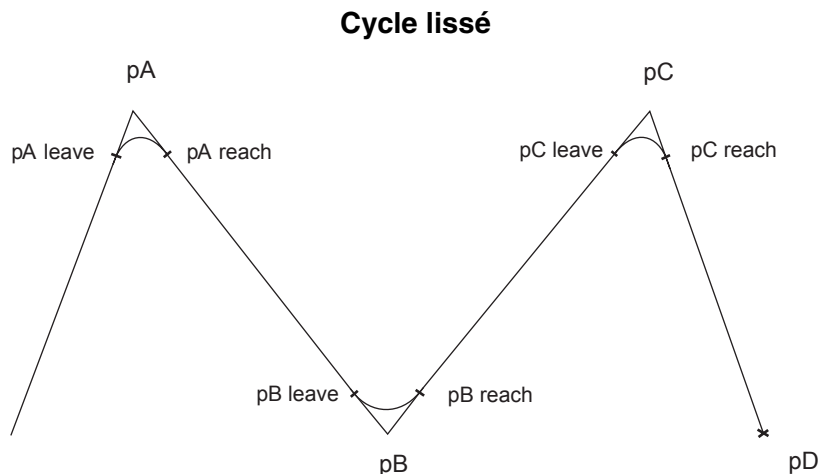
L'exécution des lignes de commande ne fait qu'enregistrer les commandes de mouvement successives. Le robot les effectue ensuite suivant ses possibilités. La mémoire dans laquelle les mouvements sont enregistrés est importante, pour permettre au système d'optimiser la trajectoire au maximum. Néanmoins, elle est limitée. Lorsqu'elle est pleine, l'exécution du programme s'arrête sur la prochaine instruction de mouvement. L'exécution du programme reprend dès que le robot a terminé le mouvement en cours, libérant ainsi de la place dans la mémoire du système.

10.2.2. ANTICIPATION ET LISSAGE

On revient dans ce paragraphe à ce qui se passe en détail lorsque des mouvements sont enchaînés. Reprenons l'exemple précédent :

```
movej (pA, tOutil, mDesc)
movej (pB, tOutil, mDesc)
movej (pC, tOutil, mDesc)
movej (pD, tOutil, mDesc)
```

On suppose que le lissage est activé dans le descripteur de mouvement **mDesc**. Lorsque la première ligne est exécutée, le système ne sait pas encore quel sera le mouvement suivant. Seul le mouvement entre le point de départ et le point **pA leave** est entièrement déterminé, le point **pA leave** étant déterminé par le système à partir de la donnée **leave** du descripteur de mouvements (voir figure suivante).



Tant que la deuxième ligne n'est pas exécutée, la portion de trajectoire de lissage au voisinage du point **pA** n'est pas entièrement déterminée, puisque le système n'a pas encore pris en compte le mouvement suivant. En mode pas-à-pas, tant que l'utilisateur ne passera pas à l'instruction suivante, le robot n'ira pas plus loin que le point **pA leave**. Au moment où l'instruction suivante est exécutée, la trajectoire de lissage au voisinage du point **pA** (entre **pA leave** et **pA reach**) peut être définie, ainsi que le mouvement jusqu'au point **pB leave**. Le robot peut alors avancer jusqu'à **pB leave**. En mode pas-à-pas, il ne dépassera pas ce point tant que l'utilisateur n'exécute pas la troisième instruction, et ainsi de suite.

L'intérêt de ce mode de fonctionnement est que le robot passe exactement au même endroit en mode pas-à-pas et en exécution normale du programme.

10.2.3. SYNCHRONISATION

Le mécanisme d'anticipation induit une désynchronisation entre les lignes d'instructions **VAL 3** et les mouvements correspondants du robot : le programme **VAL 3** est en avance sur le robot.

Lorsque l'on veut effectuer une action à une position donnée du robot, il faut que le programme attende que le robot ait terminé ses mouvements : l'instruction **waitEndMove()** permet cette synchronisation. On peut aussi utiliser l'instruction **getMoveId()** pour détecter la progression du bras sur la trajectoire (voir 10.4, contrôle du mouvement en temps réel).

Ainsi, dans le programme suivant :

```
movej (A, tOutil, mDesc)
movej (B, tOutil, mDesc)
waitEndMove ()
movej (C, tOutil, mDesc)
movej (D, tOutil, mDesc)
etc.
```

Les deux premières lignes seront exécutées alors que le robot commence son mouvement vers **A**. Puis, l'exécution du programme sera bloquée sur la troisième ligne jusqu'à ce que le robot soit stabilisé au point **B**. Une fois le mouvement du robot stabilisé en **B**, le programme reprendra son exécution.

Les instructions **open()** et **close()** attendent également la fin des mouvements du robot avant d'actionner l'outil.

10.3. CONTRÔLE DE VITESSE

10.3.1. PRINCIPE

Le principe du contrôle de vitesse sur une trajectoire est le suivant :

A chaque instant, le robot se déplace et accélère au maximum de ses capacités, tout en respectant les contraintes de vitesse et d'accélération données dans la commande de mouvement.

Les commandes de mouvement contiennent deux types de contraintes de vitesse, définies dans une variable de type **mdesc** :

1. Les contraintes de vitesse (de l'articulation), d'accélération et décélération
2. les contraintes de vitesses cartésiennes du centre outil

L'accélération détermine la rapidité avec laquelle la vitesse augmente au début d'une trajectoire. A l'inverse, la décélération définit la rapidité avec laquelle la vitesse diminue en fin de trajectoire. Lorsqu'on utilise des grandes valeurs d'accélération et de décélération, les mouvements sont plus rapides, mais plus saccadés. Avec des faibles valeurs, les mouvements prennent un peu plus de temps, mais sont plus doux.

10.3.2. RÉGLAGE SIMPLE

Lorsque l'outil et l'objet transportés par le robot ne nécessitent pas de précaution de manipulation particulière, les contraintes sur les vitesses cartésiennes sont inutiles. La mise au point de la vitesse sur la trajectoire se fera typiquement de la façon suivante :

1. Mettre les contraintes de vitesse cartésienne à de très grandes valeurs, par exemple les valeurs par défaut, pour qu'elles n'interviennent pas dans la suite du réglage.
2. Initialiser la vitesse, l'accélération et la décélération en utilisant les valeurs nominales (**100%**).
3. Régler ensuite la vitesse sur la trajectoire à l'aide du paramètre de vitesse.

Afin de conserver un comportement harmonieux du bras, l'accélération et la décélération doivent être modifiées avec la vitesse : les paramètres d'accélération et de décélération doivent correspondre approximativement au carré du paramètre de vitesse. Ainsi, la meilleure adaptation d'une vitesse de 120 % = 1.2 est obtenue avec une accélération et une décélération de $1.2 \times 1.2 = 1.44 = 144$ %. Les valeurs d'accélération et de décélération plus élevées donnent un comportement du bras plus agressif, mais aussi plus instable.

10.3.3. RÉGLAGE AVANCÉ

Lorsque l'on veut maîtriser la vitesse cartésienne au niveau de l'outil, par exemple pour obtenir une trajectoire effectuée à vitesse constante, on procédera plutôt de la manière suivante :

1. Mettre les contraintes de vitesses cartésiennes aux valeurs souhaitées à priori.
2. Initialiser la vitesse, l'accélération et la décélération en utilisant les valeurs nominales (**100%**).
3. Régler ensuite la vitesse sur la trajectoire à l'aide des seuls paramètres de vitesse cartésienne.
4. Si l'on n'arrive pas à atteindre la vitesse désirée, augmenter les paramètres d'accélération et de décélération.
Si l'on souhaite freiner automatiquement dans les parties à forte courbure, diminuer les paramètres d'accélération et de décélération.

10.3.4. ERREUR DE TRAÎNÉE

Les valeurs nominales de vitesse et d'accélération articulaires sont les valeurs de charge nominale supportées par le robot, quelle que soit la trajectoire.

Mais le robot peut bien souvent aller plus vite : les vitesses maximales atteignables par le robot dépendent de sa charge et de la trajectoire. Dans des cas favorables (faible charge, impact favorable de la gravité) le robot pourra dépasser ses valeurs nominales sans aucun dommage.

Lorsque le robot porte une charge plus lourde que sa charge nominale, ou si l'on utilise de trop grandes valeurs pour les paramètres de vitesse et d'accélération articulaires, le robot ne pourra dans certains cas plus suivre sa consigne de mouvement et s'arrêtera alors sur une erreur de traînée. En spécifiant des paramètres de vitesse et d'accélération articulaires plus faibles, on pourra éviter ces erreurs de traînée.

ATTENTION :

Dans les mouvements en ligne droite, près d'une singularité, un petit déplacement de l'outil nécessite de grands mouvements articulaires. Si la vitesse articulaire est réglée à une valeur trop élevée, le robot ne pourra pas suivre sa consigne et s'arrêtera sur une erreur de traînée.

10.4. CONTRÔLE DU MOUVEMENT EN TEMPS RÉEL

Les commandes de mouvement abordées jusqu'à présent n'ont pas un effet immédiat : au moment où chacune d'entre elle est exécutée, un ordre de mouvement est enregistré dans le système. Le robot exécute ensuite les mouvements enregistrés.

Il est possible d'intervenir immédiatement sur le mouvement du robot des manières suivantes :

- La vitesse moniteur modifie la vitesse de tous les déplacements. Elle peut être réglée avec effet immédiat à l'aide de l'instruction **setMonitorSpeed()**. Cette instruction ne peut cependant pas augmenter la vitesse quand l'opérateur peut aussi régler celle-ci à partir du MCP.
- Les instructions **stopMove()** et **restartMove()** permettent l'arrêt et la reprise de mouvement sur la trajectoire.
- L'instruction **resetMotion()** permet d'arrêter le mouvement en cours et d'annuler les commandes de mouvements enregistrés.
- L'instruction Alter (option) applique sur la trajectoire une transformation géométrique (translation, rotation, rotation au centre outil) qui est immédiatement effective.
- L'instruction **getMoveId()** permet de suivre, précisément et en temps réel, la position du robot sur sa trajectoire. Chaque instruction de mouvement est identifiée par une valeur numérique renvoyée par l'instruction. L'instruction **getMoveId()** renvoie une valeur numérique identifiant le mouvement courant (partie entière) et la progression de ce mouvement (partie décimale). Ainsi, un identifiant de mouvement de 17.572 signifie que le mouvement actuel est l'instruction de mouvement qui a renvoyé 17 et que la position du robot a atteint 57.2 % de ce mouvement.

10.5. TYPE MDESC

10.5.1. DÉFINITION

Le type **mdesc** permet de définir les paramètres d'un mouvement (vitesse, accélération, lissage).

Le type **mdesc** est un type structuré dont les champs sont, dans l'ordre :

num accel	Accélération articulaire maximale autorisée, en % de l'accélération nominale du robot.
num vel	Vitesse articulaire maximale autorisée, en % de la vitesse nominale du robot.
num decel	Décélération articulaire maximale autorisée, en % de la décélération nominale du robot.
num tvel	Vitesse maximale autorisée de translation du centre outil, en mm/s ou pouce/s selon l'unité de longueur de l'application.
num rvel	Vitesse maximale autorisée de rotation de l'outil, en degrés par seconde.
blend blend	Mode de lissage : off (pas de lissage), joint ou Cartesian (lissage cartésien).
num leave	Dans les modes de lissage joint et Cartesian , la distance entre le point cible auquel commence le lissage et le point suivant, en mm ou en pouces, selon l'unité de longueur de l'application.
num reach	Dans les modes de lissage joint et Cartesian , la distance entre le point cible auquel finit le lissage et le point suivant, en mm ou en pouces, selon l'unité de longueur de l'application.

L'explication détaillée de ces différents paramètres est donnée au début du chapitre "Contrôle des mouvements".

Par défaut, une variable de type **mdesc** est initialisée avec {100,100,100,9999,9999,joint,50,50}.

10.5.2. OPÉRATEURS

Par ordre de priorité croissant :

mdesc < mdesc & desc1 > = < mdesc desc2 >	Affecte chaque champ de desc2 au champ correspondant de la variable desc1 .
bool < mdesc desc1 > != < mdesc desc2 >	Renvoie true si desc1 et desc2 diffèrent par la valeur d'au moins un champ.
bool < mdesc desc1 > == < mdesc desc2 >	Renvoie true si desc1 et desc2 ont les mêmes valeurs de champs.

10.6. INSTRUCTIONS DE MOUVEMENT

```
num movej(joint jPosition, tool tOutil, mdesc mDesc)
```

```
num movej(point pPosition, tool tOutil, mdesc mDesc)
```

Fonction

Cette instruction enregistre une commande pour un mouvement de l'articulation vers les positions **pPosition** ou **jPosition** à l'aide des paramètres de mouvement **tOutil** et **mdesc**. Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement.

ATTENTION :

Le système n'attend pas la fin du mouvement pour passer à l'instruction VAL 3 suivante : plusieurs commandes de mouvements peuvent être enregistrées à l'avance. Lorsque le système n'a plus de mémoire disponible pour enregistrer une nouvelle commande, l'instruction attend jusqu'à ce que l'enregistrement puisse se faire.

L'explication détaillée des différents paramètres de mouvement est donnée au début du chapitre "Contrôle des mouvements".

Une erreur d'exécution est générée si **mdesc** contient des valeurs invalides, si **jPosition** se situe en dehors des butées logicielles, si **pPosition** ne peut pas être atteinte ou si une commande de mouvement précédemment enregistrée ne peut pas être exécutée (position hors d'atteinte).

Voir aussi

```
num movel(point pPosition, tool tOutil, mdesc mDesc)
bool isInRange(joint jPosition)
void waitEndMove()
num movec(point pIntermédiaire, point pCible, tool tOutil, mdesc mDesc)
```

```
num movel(point pPosition, tool tOutil, mdesc mDesc)
```

Fonction

Cette instruction enregistre une commande de mouvement linéaire vers le point **pPosition** à l'aide de l'outil **tOutil** et des paramètres de mouvement **mdesc**. Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement.

ATTENTION :

Le système n'attend pas la fin du mouvement pour passer à l'instruction VAL 3 suivante : plusieurs commandes de mouvements peuvent être enregistrées à l'avance. Lorsque le système n'a plus de mémoire disponible pour enregistrer une nouvelle commande, l'instruction attend jusqu'à ce que l'enregistrement puisse se faire.

L'explication détaillée des différents paramètres de mouvement est donnée au début du chapitre "Contrôle des mouvements".

Une erreur d'exécution est générée si **mdesc** contient des valeurs invalides, si **pPosition** ne peut pas être atteinte, si un mouvement en ligne droite vers **pPosition** n'est pas possible ou si une commande de mouvement précédemment enregistrée ne peut pas être exécutée (destination hors d'atteinte).

Voir aussi

```
num movej(joint jPosition, tool tOutil, mdesc mDesc)
void waitEndMove()
num movec(point pIntermédiaire, point pCible, tool tOutil, mdesc mDesc)
```

num movec(point pIntermédiaire, point pCible, tool tOutil,
mdesc mDesc)

Fonction

Cette instruction enregistre une commande pour un mouvement circulaire qui part de la destination du mouvement précédent et s'achève au point **pCible** en passant par le point **pIntermédiaire**. Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement.

L'orientation de l'outil est interpolée de sorte qu'il est possible de programmer une orientation constante en absolu, ou par rapport à la trajectoire.

ATTENTION :

Le système n'attend pas la fin du mouvement pour passer à l'instruction VAL 3 suivante : plusieurs commandes de mouvements peuvent être enregistrées à l'avance. Lorsque le système n'a plus de mémoire disponible pour enregistrer une nouvelle commande, l'instruction attend jusqu'à ce que l'enregistrement puisse se faire.

L'explication détaillée des différents paramètres de mouvement et de l'interpolation de l'orientation est donnée au début du chapitre "Contrôle des mouvements".

Une erreur d'exécution est générée si **mDesc** a des valeurs invalides, si point **pIntermédiaire** (ou point **pCible**) ne peut pas être atteint, si le mouvement circulaire n'est pas possible (voir le chapitre "Contrôle des mouvements - Interpolation de l'orientation") ou si une commande de mouvement précédemment enregistrée ne peut pas être exécutée (destination hors d'atteinte).

Voir aussi

num movej(joint jPosition, tool tOutil, mdesc mDesc)
num movel(point pPosition, tool tOutil, mdesc mDesc)
void waitEndMove()

void stopMove()

Fonction

Cette instruction arrête le bras sur la trajectoire et suspend l'autorisation du mouvement programmé.

ATTENTION :
Cette instruction retourne immédiatement : la tâche VAL 3 n'attend pas l'arrêt du bras pour exécuter l'instruction suivante.

Les descripteurs de mouvement utilisés pour exécuter l'arrêt sont ceux utilisés pour le mouvement en cours.

Les mouvements ne peuvent reprendre qu'après exécution d'une instruction **restartMove()** ou **resetMotion()**.

Les mouvements non programmés (déplacement manuels) restent possibles.

Exemple

```
// attente d'un signal
wait(diSignal==true)
// arrêt des mouvements sur la trajectoire
stopMove()
wait(diSignal==false)
// reprise des mouvements sur la trajectoire
restartMove()
```

Voir aussi

void restartMove()

void resetMotion(), void resetMotion(joint jdépart)

void resetMotion(), void resetMotion(joint jdépart)

Fonction

Cette instruction arrête le bras sur sa trajectoire et annule toutes les commandes de mouvement enregistrées. Elle remet à zéro l'identifiant de mouvement.

ATTENTION :
Cette instruction retourne immédiatement : la tâche VAL 3 n'attend pas l'arrêt du bras pour exécuter l'instruction suivante.

L'autorisation de mouvement programmé est restaurée si elle avait été suspendue par l'instruction **stopMove()**.

Si la position de l'articulation **jdépart** est spécifiée, la commande de mouvement suivante ne peut être exécutée qu'à partir de cette position : un mouvement de connexion devra d'abord être effectué pour rejoindre **jdépart**.

Si aucune position d'articulation n'est précisée, la commande de mouvement suivante est exécutée à partir de la position courante du bras, quelle que soit celle-ci.

Voir aussi

bool isEmpty()

void stopMove()

void autoConnectMove(bool bActif), bool autoConnectMove()

num setMoveld(num nldMvt)

joint resetTurn(joint jReference)

void restartMove()

Fonction

Cette instruction rétablit l'autorisation de mouvement programmée et relance la trajectoire interrompue par l'instruction **stopMove()**.

Si l'autorisation de mouvement programmé n'avait pas été interrompue par l'instruction **stopMove()**, cette instruction n'a aucun effet.

Voir aussi

void stopMove()

void resetMotion(), **void resetMotion(joint jdépart)**

void waitEndMove()

Fonction

Cette instruction annule le lissage de la dernière commande de mouvement enregistrée et attend l'exécution de la commande.

Cette instruction n'attend pas que le robot soit stabilisé dans sa position finale : elle attend seulement que la commande de position envoyée aux variateurs corresponde à la position finale souhaitée. Lorsqu'il est nécessaire d'attendre la stabilisation complète du mouvement, il faut utiliser l'instruction **isSettled()**.

Une erreur d'exécution est générée si un mouvement enregistré précédemment ne peut pas être exécuté (destination hors d'atteinte).

Exemple

(voir chapitre 10.2)

Voir aussi

bool isSettled()

bool isEmpty()

void stopMove()

void resetMotion(), **void resetMotion(joint jdépart)**

bool isEmpty()

Fonction

Cette instruction renvoie **true** si toutes les commandes de mouvement ont été exécutées, et **false** si au moins une commande est encore en cours d'exécution.

Exemple

Ce programme annule les mouvements enregistrés, s'il y en a :

```
// S'il y a des commandes en cours
if isEmpty() == false
    // arrêter le robot et annuler les commandes
    resetMotion()
    putln("Mouvements annulés")
endif
```

Voir aussi

void waitEndMove()

void resetMotion(), void resetMotion(joint jdépart)

bool isSettled()

Fonction

Cette instruction renvoie **true** si le robot est arrêté et **false** si sa position n'est pas encore stabilisée.

ATTENTION :

Le robot peut être arrivé pour différentes raisons et il peut donc être stabilisé avant que tous les mouvements enregistrés soient exécutés. Utilisez **isEmpty()** pour savoir si le robot est arrêté à la fin de son mouvement programmé.

La position est considérée comme stabilisée si l'erreur de position pour chaque articulation est inférieure à 1% de l'erreur maximale autorisée, pendant 50 ms.

Voir aussi

bool isEmpty()

void waitEndMove()

void autoConnectMove(bool bActif), bool autoConnectMove()

Fonction

En mode déporté, le mouvement de connexion est automatique si le bras est très proche de sa trajectoire (distance inférieure à l'erreur de traînée maximale autorisée). Si le bras est trop éloigné de sa trajectoire, le mouvement de connexion sera automatique ou sous contrôle manuel selon le mode défini par l'instruction **autoConnectMove** : automatique si **bActif** vaut **true**, sous contrôle manuel si **bActif** vaut **false**. Appelé sans paramètre, **autoConnectMove** retourne le mode de mouvement de connexion courant.

Par défaut, le mouvement de connexion en mode déporté est sous contrôle manuel.

ATTENTION :

Dans des conditions normales d'utilisation, le bras s'arrête sur sa trajectoire lors d'un arrêt d'urgence. Par conséquent, en mode déporté, le bras pourra repartir automatiquement quelque soit le mode de mouvement de connexion défini par l'instruction **autoConnectMove**.

Voir aussi

void resetMotion(), void resetMotion(joint jdépart)

num getSpeed(tool tOutil)

Fonction

Cette instruction renvoie la vitesse de translation cartésienne courante au TCPtOutil de l'outil tOutil spécifié. La vitesse est calculée à partir de la commande de vitesse de l'articulation et non du retour de vitesse de l'articulation.

Voir aussi

point here(tool tOutil, frame fReference)

joint getPositionErr()

Fonction

Cette instruction renvoie l'erreur de position d'articulation actuelle du bras. L'erreur de position de l'articulation est la différence entre la commande de position de l'articulation envoyée aux moteurs et le retour de position de l'articulation mesuré par les codeurs.

Voir aussi

void getJointForce(num& nForce)

void getJointForce(num& nForce)

Fonction

Cette instruction renvoie le couple actuel (N.m pour l'axe de révolution) ou la force (N pour l'axe linéaire) sur l'articulation calculés à partir des intensités du moteur.

La force sur l'articulation n'est pas une estimation directe des efforts externes. Elle inclut aussi la gravité, le frottement, la viscosité, l'inertie, le bruit et la précision des capteurs de courant, la relation entre le courant et le couple moteur. Elle ne peut être utilisée pour estimer les efforts externes qu'en enregistrant les forces dans des conditions de référence et en les comparant aux forces mesurées dans des conditions similaires avec des efforts externes supplémentaires.

Elle ne donne qu'un ordre de grandeur des forces. Sa précision n'est pas garantie et doit être évaluée pour chaque application.

Une erreur d'exécution est générée si le paramètre n'est pas un tableau de valeurs numériques d'une taille suffisante.

Voir aussi

joint getPositionErr()

num getMoveId()

Fonction

Cette instruction renvoie une valeur numérique indiquant la position actuelle du robot sur la trajectoire. La partie entière identifie le numéro de l'instruction de mouvement en cours d'exécution. Ce nombre entier est renvoyé quand l'instruction de mouvement est exécutée. La partie décimale donne le % de progression de ce mouvement.

Un identifiant de mouvement ne doit jamais être testé avec l'opérateur '==' mais avec l'opérateur '>=' : `wait(getMoveId()==12)` ne peut jamais être renvoyé parce que l'identifiant de mouvement peut augmenter en un incrément de 11.998 à 12.013 et ne jamais prendre exactement la valeur (12) attendue. Il est préférable d'écrire plutôt `wait(getMoveId())>=12`.

Exemple

Cet exemple montre comment l'identifiant de mouvement change sur une trajectoire simple :

```
nIdA = move1(pA, tOutil, mDesc)
nIdB = move1(pB, tOutil, mDesc)
waitEndMove()
nId = getMoveId()
```

Pendant l'exécution de ce programme :

Supposons que la valeur renvoyée nldA est 15. nldB est alors 16 : le mouvement est automatiquement incrémenté d'un à chaque instruction de mouvement.

- quand `getMoveId()` est 15.8, la position du robot se trouve à 80 % du mouvement vers le point pA.
- quand `getMoveId()` est 16.572, la position du robot se trouve à 57.2 % du mouvement vers le point pB.
- quand `getMoveId()` est 17, la position du robot se trouve à 100 % du mouvement 16, donc au point pB.

La valeur de nld, après `waitEndMove()`, est donc de $nldB+1=17$.

Voir aussi

`num movej(joint jPosition, tool tOutil, mdesc mDesc)`
`num movel(point pPosition, tool tOutil, mdesc mDesc)`
`num movec(point pIntermédiaire, point pCible, tool tOutil, mdesc mDesc)`

`num setMoveId(num nldMvt)`

Fonction

Cette instruction change l'identifiant de mouvement pour l'instruction de mouvement suivante. Elle est utile pour faire en sorte que la même trajectoire utilise toujours les mêmes valeurs d'identifiant de mouvement. Après un `resetMotion`, l'identifiant de mouvement est automatiquement réinitialisé à 0.

Après l'utilisation de `setMoveId()` ou `resetMotion()`, la relation entre un identifiant de mouvement et une instruction de mouvement peut devenir incertaine : plusieurs mouvements enregistrés peuvent alors avoir le même identifiant de mouvement. Il est donc préférable de ne pas attribuer à `setMoveId()` une valeur qui serait aussi l'identifiant de mouvement d'une commande de mouvement en cours.

Exemple

```
resetMotion()
nId1 = getMoveId()
setMoveId(1000)
nId2 = getMoveId()
nId3 = movel(pA, tOutil, mDesc)
nId4 = movel(pB, tOutil, mDesc)
waitEndMove()
nId5 = getMoveId()
```

Après l'exécution de ce programme, on a :

- nld1 est 0, parce que l'identifiant de mouvement est mis à 0 après `resetMotion()`
- nld2 est 0 : l'identifiant de mouvement vient d'être changé avec `setMoveId()`
- nld3 est 1000 : une instruction de mouvement renvoie l'identifiant de mouvement défini précédemment et l'incrémente pour le mouvement suivant
- nld4 est 1001 : l'identifiant de mouvement a été incrémenté par l'instruction de mouvement qui précède
- nld5 est 1002 : après `waitEndMove()`, 100 % du mouvement 1001 est exécuté et l'identifiant du mouvement est donc $1001+1 = 1002$

Voir aussi

`num getMoveId()`
`void resetMotion(), void resetMotion(joint jdépart)`

CHAPITRE 11

OPTIONS

11.1. MOUVEMENTS COMPLIANTS AVEC CONTRÔLE EN EFFORT

11.1.1. PRINCIPE

Dans une commande de mouvement standard, le robot se déplace pour rejoindre la position demandée avec une accélération et une vitesse programmée. Si le bras n'arrive pas à suivre la commande, un effort de plus en plus important va être demandé aux moteurs pour essayer de rejoindre la position désirée. Lorsque l'écart entre la position commandée et la position réelle est trop grand, ou lorsque l'effort demandé devient trop important, une erreur système est générée qui coupe la puissance dans le bras.

Le robot est dit 'compliant' lorsqu'il accepte certains écarts entre la position commandée et la position effective. Le contrôleur peut être programmé pour être compliant sur sa trajectoire, c'est-à-dire pour accepter un retard ou une avance sur la trajectoire programmée, en contrôlant l'effort exercé par le bras. Par contre, aucun écart par rapport à la trajectoire n'est toléré.

Concrètement, les mouvements compliants du contrôleur **VAL 3** peuvent permettre au bras de suivre une trajectoire en étant poussé ou tiré par une force extérieure, ou bien d'aller en contact avec un objet en contrôlant l'effort exercé par le bras sur cet objet.

11.1.2. PROGRAMMATION

Les mouvements compliants se programment comme des mouvements standards avec les instructions **movef()** et **movejf()**, un paramètre supplémentaire permettant de contrôler l'effort exercé par le bras. Pendant le mouvement compliant, des limitations de vitesse et d'accélération sont effectuées, comme sur un mouvement standard, à partir du descripteur de mouvement. Le mouvement peut se faire sur la trajectoire dans un sens comme dans l'autre.

Il est possible d'enchaîner des mouvements compliants, ou d'enchaîner mouvements compliants et mouvements standards : dès que la position de destination est atteinte, le robot enchaîne sur la commande de mouvement suivante. L'instruction **waitEndMove()** permet d'attendre la fin d'un mouvement compliant.

L'instruction **resetMotion()** annule tous les mouvements programmés qu'ils soient compliants ou non. Après le **resetMotion()**, le robot n'est plus compliant.

Les instructions **stopMove()** et **restartMove()** s'appliquent aussi aux mouvements compliants :

stopMove() force la vitesse du mouvement courant à zéro. S'il s'agit d'un mouvement compliant, celui sera donc stoppé, et le robot ne sera plus compliant jusqu'à l'exécution du **restartMove()**.

Enfin, l'instruction **isCompliant()** permet de s'assurer que le robot est bien en mode compliant avant, par exemple, d'autoriser un effort extérieur sur le bras.

11.1.3. CONTRÔLE DE L'EFFORT

Lorsque le paramètre d'effort spécifié est nul, le bras est passif, c'est-à-dire qu'il ne se déplace que sous l'action d'un effort extérieur.

Lorsque le paramètre d'effort est positif, tout se passe comme si un effort extérieur poussait le bras vers la position commandée : le bras se déplace alors tout seul, mais peut être retenu ou accéléré par une action extérieure qui s'ajoute à l'effort commandé.

Lorsque le paramètre d'effort est négatif, tout se passe comme si un effort extérieur tirait le bras vers sa position initiale : pour déplacer le bras vers la position commandée, il faut alors fournir un effort extérieur plus fort que l'effort commandé.

Le paramètre force est exprimé en tant que pourcentage de la charge nominale du bras. **100%** signifie que le bras applique une force vers la position demandée, qui est équivalente à la charge nominale. En rotation, **100%** correspond au couple nominal autorisé sur le bras.

Lorsque la vitesse ou l'accélération du bras atteignent les valeurs spécifiées dans le descripteur de mouvement, le robot s'oppose avec toute sa puissance à toute tentative d'augmentation de vitesse ou d'accélération.

11.1.4. LIMITATIONS

Les mouvements compliant nécessitent un ajustement spécifique du robot, qui n'est pas disponible sur tous les robots (consultez votre interlocuteur chez **Stäubli**).

Les mouvements compliant présentent les limitations suivantes :

- Il n'est pas possible d'utiliser du lissage au début ou à la fin d'un mouvement compliant : le bras marquera nécessairement un arrêt en début et fin de chaque mouvement compliant.
- Lors d'un mouvement compliant, le bras peut revenir en arrière vers son point de départ, mais il ne peut pas le dépasser : le bras s'arrête alors brutalement sur le point de départ.
- Le paramètre d'effort sur le bras ne peut pas dépasser **1000%**. La précision obtenue sur l'effort est limitée par les frottements internes. Elle dépend beaucoup de la position du bras et de la trajectoire commandée.
- Les longs mouvements compliant nécessitent beaucoup de mémoire interne. Une erreur d'exécution est générée si le système n'a pas assez de mémoire pour exécuter complètement le mouvement.

11.1.5. INSTRUCTIONS

num movejf(joint jPosition, tool tOutil, mdesc mDesc, num nForce)

Fonction

Cette instruction enregistre une commande de mouvement compliant de l'articulation vers la position **jPosition** avec l'outil **tOutil**, les paramètres **mDesc** et une commande de force **nForce**. Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement

La commande de force **nForce** est une valeur numérique représentant la force du bras, qui ne peut pas dépasser **±1000**. Une valeur de 100 correspond approximativement à la masse nominale du bras.

ATTENTION :

Le système n'attend pas la fin du mouvement pour passer à l'instruction VAL 3 suivante : plusieurs commandes de mouvements peuvent être enregistrées à l'avance. Lorsque le système n'a plus de mémoire disponible pour enregistrer une nouvelle commande, l'instruction attend jusqu'à ce que l'enregistrement puisse se faire.

L'explication détaillée des différents paramètres de mouvement est donnée au début du chapitre.

Une erreur d'exécution est générée si **mDesc** ou **nForce** a des valeurs invalides, si **jPosition** est en dehors des butées logicielles, si le mouvement précédent demande un lissage, ou si une commande de mouvement précédemment enregistrée ne peut être exécutée (destination hors d'atteinte).

Voir aussi

num movelf(point pPosition, tool tOutil, mdesc mDesc, num nForce)

bool isCompliant()

num movelf(point pPosition, tool tOutil, mdesc mDesc, num nForce)

Fonction

Cette instruction enregistre une commande de mouvement linéaire vers la position **pPosition** avec l'outil **tOutil**, les paramètres de mouvement **mDesc** et la commande de force **nForce**. Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement.

La commande de force **nForce** est une valeur numérique représentant la force du bras, qui ne peut pas dépasser **±1000**. Une valeur de 100 correspond approximativement à la masse nominale du bras.

ATTENTION :

Le système n'attend pas la fin du mouvement pour passer à l'instruction VAL 3 suivante : plusieurs commandes de mouvements peuvent être enregistrées à l'avance. Lorsque le système n'a plus de mémoire disponible pour enregistrer une nouvelle commande, l'instruction attend jusqu'à ce que l'enregistrement puisse se faire.

L'explication détaillée des différents paramètres de mouvement est donnée au début du chapitre.

Une erreur d'exécution est générée si **mDesc** ou **nForce** a des valeurs invalides, si **pPosition** n'est pas atteignable, si le mouvement vers **pPosition** est impossible en ligne droite, si le mouvement précédent demande un lissage, ou si une commande de mouvement précédemment enregistrée ne peut être exécutée (destination hors d'atteinte).

Voir aussi

num movejf(joint jPosition, tool tOutil, mdesc mDesc, num nForce)
bool isCompliant()

bool isCompliant()

Fonction

Cette instruction renvoie **true** si le robot est en mode compliant, sinon **false**.

Exemple

```
movelf(pPosition, tTool, mDesc, 0)
// Attend que le robot soit effectivement en mode compliant
wait(isCompliant())
// Commande l'éjection de la presse
diEjection = true
// Attend la fin du mouvement compliant
waitEndMove()
// redémarrer avec un mouvement standard
movej(jjDepart, tOutil, mDesc)
```

Voir aussi

num movelf(point pPosition, tool tOutil, mdesc mDesc, num nForce)
num movejf(joint jPosition, tool tOutil, mdesc mDesc, num nForce)

11.2. ALTER : CONTROLE EN TEMPS REEL D'UNE TRAJECTOIRE

Alter cartésien

11.2.1. PRINCIPE

L'altération cartésienne d'une trajectoire permet d'appliquer à la trajectoire une transformation géométrique (translation, rotation, rotation au centre outil) qui est immédiatement effective.

Cette fonction permet de modifier une trajectoire nominale au moyen d'un capteur externe, afin, par exemple, d'assurer un suivi précis de la forme d'une pièce, ou d'intervenir sur une pièce en mouvement.

11.2.2. PROGRAMMATION

La programmation consiste à définir d'abord la trajectoire nominale, puis à spécifier, en temps réel, une déviation sur cette trajectoire.

La trajectoire nominale est programmée de la même manière que les mouvements standards, au moyen des instructions `alterMoveI()`, `alterMoveJ()` et `alterMoveC()`. Plusieurs mouvements altérables peuvent se suivre, ou certains mouvements altérables peuvent alterner avec des mouvements non altérables. Nous définirons une trajectoire altérable comme la succession des commandes de mouvement altérables situées entre deux commandes de mouvement non altérables.

L'altération proprement dite est programmée au moyen de l'instruction `alter()`. Différents modes alter sont possibles en fonction de la transformation géométrique à appliquer ; le mode est défini au moyen de l'instruction `alterBegin()`. L'instruction `alterEnd()` sert, enfin, à indiquer de quelle manière doit se terminer l'altération, soit avant la fin du mouvement nominal, de manière à pouvoir séquencer le prochain mouvement non altérable sans provoquer d'arrêt ; soit après, de manière à pouvoir déplacer le bras au moyen d'alter pendant que le mouvement nominal est à l'arrêt.

Les autres instructions de commande de mouvement restent effectives en mode alter.

ATTENTION :

Les instructions `waitEndMove`, `open` et `close` attendent la fin du mouvement nominal, et non pas la fin du mouvement altéré. L'exécution du VAL 3 peut alors reprendre après un `waitEndMove` même si le bras est encore en mouvement en raison d'une modification de l'altération.

11.2.3. CONTRAINTES

Synchronisation, désynchronisation : Comme la commande alter est appliquée immédiatement, la variation de l'altération doit être contrôlée de manière à ce que la trajectoire du bras résultante ne présente aucune discontinuité et aucun bruit :

- Un changement important de l'altération ne peut être appliqué que progressivement avec une commande d'approche spécifique.
- La fin de la modification exige une vitesse d'altération nulle, obtenue progressivement grâce à une commande d'arrêt spécifique.

Commande synchrone : Le contrôleur envoie des commandes de position et de vitesse toutes les 4 ms aux amplificateurs. En conséquence, la commande alter doit être synchronisée avec cette période de communication de manière à ce que la vitesse d'altération reste sous contrôle. Pour ce faire, on utilisera une tâche **VAL 3** synchrone (voir chapitre Tâches). De la même manière, il faudra parfois filtrer préalablement une entrée capteur si les données sont bruitées ou si leur période d'échantillonnage n'est pas synchronisée avec la période du contrôleur.

Séquencement progressif : Le premier mouvement non altérable suivant une trajectoire altérable ne peut être calculé qu'après exécution de `alterEnd`. En conséquence, si `alterEnd` est exécuté trop peu de temps après la fin du mouvement altérable, le bras risque de ralentir voire de s'arrêter à proximité de ce point tant que le mouvement suivant n'est pas calculé.

De plus, la nécessité de calculer à l'avance le mouvement suivant impose des restrictions à la trajectoire altérée après exécution de `alterEnd` : Elle doit conserver la même configuration, et il faut s'assurer que toutes les articulations restent dans le même tour d'axe. Il est alors possible qu'une erreur soit générée pendant le mouvement, alors qu'elle n'aurait pas eu lieu si `alterEnd` n'avait pas été exécuté à l'avance.

11.2.4. SÉCURITÉ

A tout moment, l'altération de l'utilisateur peut être invalide : destination hors de portée, vitesse ou accélération trop élevée. Lorsque le système détecte des situations de ce type, une erreur est générée et le bras est arrêté brusquement sur la dernière position valide. Le générateur de mouvement doit être réinitialisé pour repartir.

Lorsque le mouvement du bras est désactivé au cours d'un mouvement (touche 'Move/Hold', demande d'arrêt ou arrêt d'urgence), la commande d'arrêt s'exerce sur les mouvements nominaux comme pour les mouvements standards. Au bout d'un certain temps, le mode alter est lui aussi automatiquement désactivé afin de garantir un arrêt complet du bras. Lorsque l'état d'arrêt disparaît, le mouvement reprend et le mode alter est automatiquement réactivé.

11.2.5. LIMITATIONS

Les mouvements nuls (lorsque la destination du mouvement se trouve sur la position de départ) ne sont pas pris en compte par le système. En conséquence, vous devez avoir recours à un mouvement non nul pour passer en mode alter. Une distance de mouvement de 0.001 mm est suffisante pour cela.

Il n'est pas possible de spécifier la configuration à appliquer à la trajectoire altérée ; le système utilise, en effet, toujours la même configuration. Il est donc impossible de modifier la configuration du bras à l'intérieur d'une trajectoire altérée (même en utilisant l'instruction alterMovej).

11.2.6. INSTRUCTIONS

num alterMovej(joint jPosition, tool tOutil, mdesc mDesc)

num alterMovej(point pPosition, tool tOutil, mdesc mDesc)

Fonction

Cette instruction enregistre une commande de mouvement d'articulation modifiable (une ligne dans l'espace de l'articulation). Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement.

Paramètres

jPosition/pPosition	Expression de type point ou joint définissant la position de fin du mouvement.
tOutil	Expression de type outil définissant le centre outil utilisé pendant le mouvement pour le contrôle de la vitesse cartésienne.
mdesc	Expression de type mdesc définissant les paramètres de contrôle de vitesse et de lissage pour le mouvement.

Détails

Cette instruction se comporte exactement comme l'instruction movej, à ceci près qu'elle autorise le mode alter pour le mouvement. Voir movej pour plus d'informations.

num alterMovel(point pPosition, tool tOutil, mdesc mDesc)

Fonction

Cette instruction enregistre une commande de mouvement linéaire modifiable (une ligne dans l'espace cartésien). Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement.

Paramètres

pPosition	Expression de type point définissant la position de fin du mouvement.
tOutil	Expression de type outil définissant le centre outil utilisé pendant le mouvement pour le contrôle de la vitesse cartésienne. A la fin du mouvement, le centre outil se trouve à la position cible spécifiée.
mDesc	Expression de type mdesc définissant les paramètres de contrôle de vitesse et de lissage pour le mouvement.

Détails

Cette instruction se comporte exactement comme l'instruction movel, à ceci près qu'elle autorise le mode alter pour le mouvement. Voir movel pour plus d'informations.

num alterMovec(point pIntermédiaire, point pCible, tool tOutil, mdesc mDesc)

Fonction

Cette instruction enregistre une commande de mouvement circulaire modifiable. Elle renvoie l'identifiant de mouvement attribué à ce mouvement et incrémente d'un l'identifiant de mouvement pour la prochaine commande de mouvement.

Paramètres

pIntermédiaire	Expression de type point définissant un point intermédiaire du cercle
pCible	Expression de type point définissant la position de fin du mouvement.
tOutil	Expression de type outil définissant le centre outil utilisé pendant le mouvement pour le contrôle de la vitesse cartésienne. A la fin du mouvement, le centre outil se trouve à la position cible spécifiée.
mDesc	Expression de type mdesc définissant les paramètres de contrôle de vitesse et de lissage pour le mouvement.

Détails

Cette instruction se comporte exactement comme l'instruction movec, à ceci près qu'elle autorise le mode alter pour le mouvement. Voir movec pour plus d'informations.

num alterBegin(frame fAlterReference, mdesc mVitesseMax)

num alterBegin(tool tAlterReference, mdesc mVitesseMax)

Fonction

Cette instruction initialise le mode alter pour le trajet modifiable en cours d'exécution.

Paramètres

fAlterReference/tAlterReference	Expression de type frame ou tool définissant la référence de la déviation alter.
mVitesseMax	Expression de type mdesc définissant les paramètres de vérification de sécurité de la déviation alter.

Détails

Le mode alter initié avec alterBegin ne se termine que par une commande alterEnd, ou par un resetMotion. Lorsque la fin d'une trajectoire altérable est atteinte, le mode alter demeure actif jusqu'à l'exécution de alterEnd.

L'expression trsf de la commande alter définit une transformation de la trajectoire entière autour de alterReference :

- La trajectoire pivote autour du centre du repère ou de l'outil en utilisant la partie de rotation de la trsf.
- La trajectoire est ensuite soumise à une translation par la partie de translation de la trsf.

Les coordonnées trsf de la commande alter sont définies dans le repère alterReference.

Lorsqu'un repère (frame) est utilisé comme référence, alterReference est fixe dans l'espace (World). Ce mode doit être utilisé lorsque l'altération de la trajectoire connue ou mesurée dans l'espace cartésien (pièce mobile telle que le suivi de convoyeur).

Lorsqu'un outil est utilisé comme référence, alterReference est fixe par rapport au centre outil. Ce mode doit être utilisé lorsque l'altération de la trajectoire est connue ou mesurée par rapport au centre outil (par exemple capteur de forme de pièce monté sur l'outil).

Le descripteur de mouvement est utilisé pour définir la vitesse articulaire et la vitesse cartésienne maximum sur la trajectoire altérée (à l'aide des champs vel, tvel et rvel du descripteur de mouvement). Une erreur est générée et le bras est arrêté sur la trajectoire si la vitesse altérée dépasse les limites spécifiées.

Les champs accel et decel du descripteur de mouvement contrôlent la durée de l'arrêt lorsqu'une condition d'arrêt survient (eStop, touche 'Move/Hold', VAL 3 [stopMove\(\)](#)) : L'altération de la trajectoire doit être arrêtée en utilisant ces paramètres de décélération (voir alterStopTime).

alterBegin renvoie une valeur numérique indiquant le résultat de l'instruction :

- 1 alterBegin a été exécuté avec succès
- 0 alterBegin attend le début du mouvement altérable
- 1 alterBegin n'a pas été pris en compte parce que le mode alter a déjà démarré
- 2 alterBegin a été refusé (l'option alter n'est pas activée)
- 3 alterBegin a été refusé parce que le mouvement est en erreur. Un resetMotion est requis.

Voir aussi

num alterEnd()
num alter(trsf trAltération)
num alterStopTime()

num alterEnd()

Fonction

Cette instruction quitte le mode alter et rend le mouvement en cours non modifiable.

Détails

Si alterEnd est exécuté alors que la fin de la trajectoire altérable est atteinte, le mouvement non altérable suivant (s'il en existe un) est démarré immédiatement.

Si alterEnd est exécuté avant la fin du mouvement altérable, la valeur actuelle de la déviation alter s'applique au reste de la trajectoire altérable, jusqu'au mouvement non altérable suivant. Il n'est pas possible de passer à nouveau en mode alter sur la même trajectoire altérable.

Le mouvement non altérable suivant, s'il en existe un, est calculé dès l'exécution de alterEnd afin que la transition entre la trajectoire altérable et le mouvement non altérable suivant se fasse sans arrêt.

alterEnd renvoie une valeur numérique indiquant le résultat de l'instruction :

- 1 alterEnd a été exécuté avec succès
- 1 alterEnd n'a pas été pris en compte parce que le mode alter n'a pas encore démarré
- 3 alterEnd a été refusé parce que le mouvement est en erreur. Un resetMotion est requis.

Voir aussi

num alterBegin(frame fAlterReference, mdesc mVitesseMax)

num alterBegin(tool tAlterReference, mdesc mVitesseMax)

num alter(trsf trAltération)

Fonction

Cette instruction définit une nouvelle modification de la trajectoire modifiable.

Détails

La transformation induite par la trsf d'altération dépend du mode alter sélectionné par l'instruction alterBegin. Les coordonnées de l'altération sont définies dans le repère ou dans l'outil spécifié par l'instruction alterBegin.

L'altération est appliquée par le système toutes les 4 ms : Lorsque plusieurs instructions alter sont exécutées en moins de 4 ms, c'est la dernière qui s'applique. Le plus souvent, l'instruction alter doit être exécutée dans une tâche synchrone pour forcer un rafraîchissement de l'altération toutes les 4 ms.

L'altération doit être calculée minutieusement de manière à ce que les commandes de position et de vitesse du bras qui en résultent soient exécutées sans discontinuité et sans bruit. Il faudra parfois filtrer préalablement l'entrée capteur de manière à atteindre la qualité recherchée sur la trajectoire et le comportement du bras.

Lorsque le mouvement est arrêté (touche 'Move/Hold', arrêt d'urgence, instruction [stopMove\(\)](#)), l'altération de la trajectoire est verrouillée jusqu'à ce que toutes les conditions d'arrêt soient annulées.

Lorsque l'altération de la trajectoire n'est pas valide (position hors d'atteinte, limites de vitesse dépassées), le bras s'arrête d'un coup sur la dernière position valide et le mode alter est verrouillé en position d'erreur. Un resetMotion est requis pour pouvoir repartir. Les limites de vitesse du mouvement alter sont définies par l'instruction alterBegin.

Alter renvoie une valeur numérique indiquant le résultat de l'instruction :

- 1 alter a été exécuté avec succès.
- 0 alter attend le redémarrage du mouvement (alterStopTime est nul).
- 1 alter n'a pas été pris en compte parce que le mode alter n'a pas démarré ou est déjà terminé.
- 2 alter a été refusé (l'option alter n'est pas activée).
- 3 alter a été refusé parce que le mouvement est en erreur. Un resetMotion est requis.

Voir aussi

num alterBegin(frame fAlterReference, mdesc mVitesseMax)

num alterBegin(tool tAlterReference, mdesc mVitesseMax)

void taskCreateSync string sNom, num nPériode, bool& bOverrun, programme(...)

num alterStopTime()

Fonction

Cette instruction renvoie le temps restant avant que la déviation alter soit verrouillée quand une condition d'arrêt se présente.

Détails

Lorsqu'une condition d'arrêt survient, le système évalue le temps nécessaire à l'arrêt du bras si les paramètres accel et decel du descripteur de mouvement spécifiés par alterBegin sont utilisés. Le minimum entre ce temps et le temps imposé par le système (généralement 0.5s lorsqu'un eStop survient) est renvoyé par alterStopTime.

Lorsque alterStopTime renvoie une valeur négative, il n'y a pas de condition d'arrêt active. Lorsque alterStopTime renvoie une valeur nulle, la commande alter est verrouillée jusqu'à ce que toutes les conditions d'arrêt aient été supprimées.

alterStopTime renvoie une valeur nulle lorsque le mode alter n'est pas activé.

Voir aussi

num alterBegin(frame fAlterReference, mdesc mVitesseMax)

num alterBegin(tool tAlterReference, mdesc mVitesseMax)

num alter(trsf trAltération)

11.3. CONTRÔLE DE LA LICENCE OEM

11.3.1. PRINCIPES

Une licence OEM est un code spécifique du contrôleur qui permet de limiter l'utilisation d'un projet **VAL 3** à certains contrôleurs de robots sélectionnés.

Un outil de **Stäubli Robotics Suite**(*) permet de coder un mot de passe OEM secret en licence OEM publique propre au contrôleur, qui peut ensuite être installée sur le contrôleur sous la forme d'une option logicielle. A l'aide de l'instruction **getLicence()**, une application ou une librairie peut vérifier si la licence OEM est installée et s'assurer ainsi qu'elle est utilisée uniquement par le robot choisi.

Pour tenir secret le mot de passe OEM et protéger le code pendant le test de la licence, l'instruction **getLicence()** doit être utilisée dans une bibliothèque cryptée.

Le mode de démonstration des licences OEM est pris en charge ; dans ce cas, le contrôleur est simplement configuré avec le code "demo" et l'instruction **getLicence()** en informe l'origine de l'appel. Avec l'émulateur **VAL 3**, il suffit de sélectionner le mode "demo" pour activer complètement la licence OEM.

L'instruction **getLicence()** est une option **VAL 3** qui nécessite l'installation d'une licence d'exécution (runtime) sur le contrôleur. Si cette licence d'exécution n'est pas définie, **getLicence()** renvoie un code d'erreur.

(*) Cet outil, un exécutable encrypttools.exe, ne peut être utilisé qu'avec une licence spécifique dans la **Stäubli Robotics Suite**.

11.3.2. INSTRUCTIONS

**string getLicence(string sOemNomLicence,
string sOemMot de passe)**

Fonction

Cette instruction renvoie le statut de la licence OEM spécifiée :

"oemLicenceDisabled"	La licence de temps d'exécution VAL 3 "oemLicence" n'est pas activée sur le contrôleur : la licence OEM ne peut pas être testée.
""	La licence OEM sOemNomLicence n'est pas activée (mot de passe non défini ou invalide).
"demo"	La licence OEM sOemNomLicence est activée en mode démonstration.
"enabled"	La licence OEM sOemNomLicence est activée.

Voir aussi
Cryptage

11.4. ROBOT ABSOLU

11.4.1. PRINCIPE

Un "robot absolu" est un robot qui utilise une identification des paramètres géométriques spécifiques au bras (souvent appelés "paramètres DH"). Ces paramètres sont spécifiques à chaque robot et correspondent à l'orientation et aux dimensions réelles de chaque articulation. Un robot absolu possède une précision accrue pour atteindre les positions cartésiennes définies par un outil de DAO ou calculées dans VAL 3 (par exemple des positions dans une palette). Les courbes cartésiennes (lignes longues, cercles) sont en outre plus précises. Le calibrage absolu ne modifie pas la répétabilité du bras.

Les paramètres DH se composent d'un ensemble de translations (a, b et d sur les axes X, Y, Z,) et de rotations (alpha, beta, theta autour des axes X, Y et Z)

La séquence de ces translations et rotations est définie de telle manière que la position de l'articulation {j1, j2, j3, j4, j5, j6} corresponde à la position cartésienne pCart au centre de la bride, avec :

```
pCart.trsf = {0,0,0,0,0, j1+theta[0]}
* {a[0], b[0], d[0], alpha[0], beta[0], j2+theta[1]}
* {a[1], b[1], d[1], alpha[1], beta[1], j3+theta[2]}
* {a[2], b[2], d[2], alpha[2], beta[2], j4+theta[3]}
* {a[3], b[3], d[3], alpha[3], beta[3], j5+theta[4]}
* {a[4], b[4], d[4], alpha[4], beta[4], j6+theta[5]}
* {a[5], b[5], d[5], alpha[5], beta[5], 0}
* {0, 0, d[6], 0, 0, 0}
```

Le paramètre d[6] n'est requis que pour certains poignets de bras.

11.4.2. EXPLOITATION

Les paramètres géométriques doivent être identifiés à l'aide d'un outil de mesure séparé (par exemple d'un appareil de poursuite laser). Le langage VAL 3 ne comprend pas d'outil permettant cette opération de mesure, mais il fournit un moyen pour appliquer les paramètres géométriques mesurés au robot, avec effet immédiat. Les paramètres sont en outre sauvegardés dans le fichier de configuration du bras, de sorte qu'ils sont automatiquement récupérés au redémarrage suivant.

Les paramètres géométriques peuvent être modifiés dans une application VAL 3 en cours d'exécution. On peut ainsi ajuster la géométrie du bras pendant le cycle de production, si la cellule comporte un outil de mesure adéquat.

11.4.3. LIMITATIONS

Les paramètres géométriques ne peuvent être modifiés dans une application VAL 3 en cours d'exécution que si le générateur de mouvements est vide (pas de mouvements en cours).

La géométrie modifiée d'un robot absolu a des propriétés mathématiques plus complexes qu'une géométrie standard. La notion de configuration du bras (type de configuration) ne peut plus être rigoureuse. La conversion d'une position cartésienne en position de l'articulation correspondante peut échouer près des limites de l'articulation ou dans des positions singulières.

11.4.4. INSTRUCTIONS

```
void getDH (num& theta[], num& d[], num& a[], num& alpha[],
            num& beta[])
```

```
void getDefaultDH(num& theta[], num& d[], num& a[], num& alpha[],
                 num& beta[])
```

Fonction

Ces instructions renvoient les paramètres DH du bras dans les tableaux spécifiés. Les paramètres d et a sont des translations en mm ; les paramètres alpha, beta et theta sont des angles en degrés. `getDH()` renvoie les paramètres DH actuels du bras connecté au contrôleur. `getDefaultDH()` renvoie les paramètres DH standard pour le type de bras.

Voir aussi

```
bool setDH(num& theta[], num& d[], num& a[], num& b[], num& alpha[], num& beta[])
```

```
bool setDH(num& theta[], num& d[], num& a[], num& b[], num& alpha[],
           num& beta[])
```

Fonction

Cette instruction n'est activée qu'avec la licence d'exécution spécifique. Elle modifie la géométrie du bras avec les paramètres DH. Les paramètres d, a et b sont des translations en millimètres ; les paramètres alpha, beta et theta sont des angles en degrés. La modification est immédiate et s'applique aussi au fichier de configuration du bras, de sorte que la géométrie modifiée prend effet au redémarrage suivant.

L'instruction renvoie true si le changement a bien été effectué ou false si la nouvelle géométrie n'a pas été appliquée parce qu'elle est trop différente des paramètres de la géométrie standard. `setDH()` attend que le mouvement soit vide avant d'effectuer son opération.

La taille des tableaux DH doit correspondre au nombre d'axes du robot. Une entrée supplémentaire peut être nécessaire dans le tableau d pour modifier la dimension de la bride : quand il est absent, `setDH()` renvoie false et un message de diagnostic est envoyé à l'enregistreur.

Voir aussi

```
void getDH (num& theta[], num& d[], num& a[], num& alpha[], num& beta[])
```

```
void getDefaultDH(num& theta[], num& d[], num& a[], num& alpha[], num& beta[])
```

11.5. AXE CONTINU

11.5.1. PRINCIPE

L'axe 6 (pour les bras RX/TX) ou 4 (pour les bras RS/TS) peut être "continu" dans une application robotique quand seule sa position dans un tour est importante : le nombre de tours qu'il a effectués au cours du cycle est sans importance. Ce principe s'applique, par exemple, aux applications dans lesquelles le robot tient une pièce qui est usinée par un outil fixe.

L'option continuousAxis permet de réinitialiser automatiquement le nombre de tours effectués dans le cycle précédent avant le début d'un nouveau cycle. Par exemple, si l'axe commence le cycle de l'application dans la position 0° et le finit dans la position 720° (2 tours), le cycle suivant peut remettre instantanément la position à 0° et commence un nouveau cycle, sans qu'il faille ramener l'axe de 720° à 0°.

11.5.2. INSTRUCTIONS

`joint resetTurn(joint jReference)`

Fonction

Cette instruction se comporte comme l'instruction `resetMotion()` standard, attend l'arrêt du bras et ajuste la position de l'axe continu afin de la rapprocher le plus possible de la position de référence spécifiée. Elle renvoie la position effective du bras après l'exécution de l'instruction. L'opération d'ajustement s'effectue indifféremment avec le bras sous tension ou hors tension et prend environ 50 ms. L'instruction `resetTurn()` modifie les données de calibration du bras. Au redémarrage suivant du contrôleur, la position du zéro sur l'axe est réinitialisée automatiquement (actualisation des données spécifiques du bras, dans le bras et sur le disque du contrôleur).

Voir aussi

`void resetMotion()`, `void resetMotion(joint jdépart)`

CHAPITRE 12

ANNEXES

12.1. CODES D'ERREUR D'EXÉCUTION

Code	Description
-1	Aucune tâche portant le nom spécifié n'a été créée par cette librairie ou application
0	La tâche est suspendue sans erreur d'exécution (instruction taskSuspend() ou mode de debuggage)
1	La tâche spécifiée est en cours d'exécution
10	Calcul numérique non valide (division par zéro).
11	Calcul numérique non valide (par exemple $\ln(-1)$)
20	Accès à un tableau ayant un index supérieur à la taille du tableau.
21	Accès à un tableau ayant un index négatif.
29	Nom de tâche invalide. Voir instruction taskCreate() .
30	Le nom spécifié ne correspond à aucune tâche VAL 3 .
31	Une tâche de même nom existe déjà. Voir instruction taskCreate .
32	Seules 2 périodes différentes pour les tâches synchrones sont supportées. Modifier la période d'exécution.
40	Pas assez de place mémoire système.
41	Pas assez de mémoire d'exécution pour la tâche. Voir Taille de mémoire d'exécution.
60	Temps maximal d'exécution de l'instruction dépassé.
61	Erreur interne à l'interpréteur VAL 3
70	Valeur de paramètre d'instruction invalide. Voir l'instruction correspondante.
80	Utilisation d'une donnée ou d'un programme d'une librairie non chargée en mémoire.
81	Cinématique incompatible : Utilisation d'une point/joint/config incompatible avec le bras cinématique.
82	Le Frame ou Tool de référence d'une variable fait partie d'une librairie qui n'est pas accessible depuis l'application (librairie non déclarée dans l'application de la variable ou variable avec attribut privée).
90	La tâche ne peut pas reprendre à l'endroit spécifié. Voir instruction taskResume() .
100	La vitesse spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
101	L'accélération spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
102	La décélération spécifiée dans le descripteur de mouvement est invalide (négative, trop grande, ou inférieure à la vitesse).
103	La vitesse de translation spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
104	La vitesse de rotation spécifiée dans le descripteur de mouvement est invalide (négative ou trop grande).
105	Le paramètre reach spécifié dans le descripteur de mouvement est invalide (négatif).
106	Le paramètre leave spécifié dans le descripteur de mouvement est invalide (négatif).
122	Tentative d'écriture sur une entrée du système.
123	Utilisation d'une entrée-sortie dio, aio ou sio non reliée à une entrée-sortie du système.
124	Tentative d'accès à une entrée-sortie protégée du système
125	Erreur de lecture ou d'écriture sur une dio, aio ou sio (erreur sur le bus de terrain)
150	Impossible d'exécuter cette instruction de mouvement : un mouvement demandé précédemment n'a pas pu être terminé (point hors d'atteinte, singularité, problème de configuration...)
153	Commande de mouvement non supportée
154	Instruction de mouvement invalide : vérifier le descripteur de mouvement.
160	Coordonnées de l'outil flange non valides
161	Coordonnées du repère world non valides
162	Utilisation d'un point sans repère de référence. Voir Définition.
163	Utilisation d'un repère sans repère de référence. Voir Définition.
164	Utilisation d'un outil sans outil de référence. Voir Définition.
165	Repère ou outil de référence invalide (variable globale liée à une variable locale)
250	Pas de licence d'exécution (runtime) pour cette instruction, ou validité de la licence démo dépassée.

12.2. CODES DES TOUCHES DU CLAVIER DU PUPITRE

Sans Shift

3	Caps	Space		
283	-	32		
			Ret.	
2	Shift	Esc	Help	
282	-	255	-	270
	Menu	Tab	Up	Bksp
	-	259	261	263
1	User	Left	Down	Right
281	-	264	266	268

Avec Shift

3	Caps	Space		
283	-	32		
			Ret.	
2	Shift	Esc	Help	
282	-	255	-	270
	Menu	UnTab	PgUp	Bksp
	-	260	262	263
1	User	Home	PgDn	End
281	-	265	267	269

Menus (avec ou sans Shift) :

F1	F2	F3	F4	F5	F6	F7	F8
271	272	273	274	275	276	277	278

Pour les touches standards, le code retourné est le code **ASCII** du caractère correspondant :

Sans Shift

q	w	e	r	t	y	u	i	o	p
113	119	101	114	116	121	117	105	111	112
a	s	d	f	g	h	j	k	l	<
97	115	100	102	103	104	106	107	108	60
z	x	c	v	b	n	m	.	,	=
122	120	99	118	98	110	109	46	44	61

Avec Shift

7	8	9	+	*	;	()	[]
55	56	57	43	42	59	40	41	91	93
4	5	6	-	/	?	:	!	{	}
52	53	54	45	47	63	58	33	123	125
1	2	3	0	"	%	-	.	,	>
49	50	51	48	34	37	95	46	44	62

Avec double Shift

Q	W	E	R	T	Y	U	I	O	P
81	87	69	82	84	89	85	73	79	80
A	S	D	F	G	H	J	K	L	}
65	83	68	70	71	72	74	75	76	125
Z	X	C	V	B	N	M	\$	\	=
90	88	67	86	66	78	77	36	92	61

ILLUSTRATIONS

Ambiguïté sur l'orientation intermédiaire	149
Cercle complet	149
Changement de configuration de l'épaule possible	152
Changement de configuration du coude impossible	151
Changement positive/negative du coude	150
Changement positive/negative du poignet	151
Changement : righty / lefty	150
Configuration : enegative	139
Configuration : epositive	139
Configuration : lefty	138
Configuration : lefty	140
Configuration : righty	138
Configuration : righty	140
Configuration : wnegative	139
Configuration : wpositive	139
Cycle avec interruption du lissage	146
Cycle en : U	144
Cycle lissé	146
Cycle lissé	153
Définition des distances : 'leave' / 'reach'	145
Définition point	132
Deux configurations possibles pour atteindre le même point : P	136
Liens entre outils	129
Liens entre repères de référence	127
Mouvement circulaire	144
Mouvement en ligne droite	143
Organigramme : frame / point / tool / trsf	117
Orientation constante en absolu	148
Orientation constante par rapport à la trajectoire	148
Orientation	122
Page utilisateur	71
Position initiale et finale	143
Rotation repère par rapport à l'axe : X	122
Rotation repère par rapport à l'axe : Y'	123
Rotation repère par rapport à l'axe : Z''	123
Séquencement	84

INDEX

A

abs (Instruction) 47, 118
 accel 156
 acos (Instruction) 46
 aio 28, 63
 aioGet (Instruction) 63
 aioLink (Instruction) 63
 aioSet (Instruction) 63, 64
 alter (Instruction) 174
 alterBegin (Instruction) 173
 alterEnd (Instruction) 174
 alterMoveec (Instruction) 172
 alterMovej (Instruction) 171
 alterMovel (Instruction) 172
 alterStopTime (Instruction) 175
 append (Instruction) 35
 appro (Instruction) 134
 asc (Instruction) 57
 asin (Instruction) 46
 atan (Instruction) 47
 autoConnectMove 147
 autoConnectMove (Instruction) 161

B

bAnd (Instruction) 50
 blend 145, 156
 bNot (Instruction) 50
 bool 28
 bOr (Instruction) 51
 bXor (Instruction) 51

C

call 22
 call (Instruction) 22
 chr (Instruction) 56
 clearBuffer (Instruction) 66
 clock (Instruction) 94
 close 84
 close (Instruction) 131
 cls (Instruction) 72
 codeAscii 56
 compose (Instruction) 133
 config 28, 117, 136
 config (Instruction) 140
 cos (Instruction) 46

D

decel 156
delay 84
delay (Instruction) 93
delete (Instruction) 33, 58
dio 28, 59
dioGet (Instruction) 60
dioLink (Instruction) 60
dioSet (Instruction) 61
disablePower (Instruction) 109
distance (Instruction) 124, 133
do ... until (Instruction) 24

E

elbow 136
enablePower (Instruction) 109
enegative 139
epositive 139
esStatus (Instruction) 111
exp (Instruction) 47

F

find (Instruction) 58
first (Instruction) 36
for (Instruction) 25
frame 28, 117
fromBinary (Instruction) 52

G

get 84
get (Instruction) 74
getData (Instruction) 33
getDate 79
getDefaultDH (Instruction) 178
getDH (Instruction) 178
getDisplayLen (Instruction) 73
getJointForce (Instruction) 162
getKey (Instruction) 76
getLanguage (Instruction) 78
getLatch (Instruction) 120
getLicence (Instruction) 176
getMonitorSpeed (Instruction) 112
getMoveld (Instruction) 162
getPosition (Instruction) 162
getProfile (Instruction) 77
getSpeed (Instruction) 162
getVersion (Instruction) 113
globale 30
gotoxy (Instruction) 72

H

help (Instruction) 89
here (Instruction) 134
herej (Instruction) 119

I

if (Instruction) 23
 insert (Instruction) 32, 58
 interpolateC (Instruction) 126
 interpolateL (Instruction) 125
 ioBusStatus (Instruction) 111
 ioStatus (Instruction) 61, 62, 64
 isCalibrated (Instruction) 110
 isCompliant 167
 isCompliant (Instruction) 169
 isDefined (Instruction) 31
 isEmpty (Instruction) 161
 isInRange (Instruction) 119
 isKeyPressed (Instruction) 76
 isPowered (Instruction) 109
 isSettled (Instruction) 161

J

joint 28
 jointToPoint (Instruction) 134

L

last (Instruction) 36
 leave 145, 156
 left (Instruction) 57
 lefty 138, 140
 len (Instruction) 58
 libDelete (Instruction) 101
 libList (Instruction) 102
 libLoad 100
 libLoad (Instruction) 101
 libPath (Instruction) 102
 libSave (Instruction) 101
 limit (Instruction) 49
 link (Instruction) 128, 131
 ln (Instruction) 48
 locale 30
 log (Instruction) 48
 logMsg (Instruction) 77

M

max (Instruction) 49
 mdesc 28, 143, 156
 mid (Instruction) 57
 min (Instruction) 49
 movec (Instruction) 158
 movej 143
 movej (Instruction) 157
 movejf 167
 movejf (Instruction) 168
 movel 143
 movel (Instruction) 157
 movelf 167
 movelf (Instruction) 169

N

next (Instruction) 36
num 28, 57

O

open 84
open (Instruction) 130

P

point 28
pointToJoint (Instruction) 135
popUpMsg (Instruction) 76
position (Instruction) 128, 131, 135
power (Instruction) 47
prev (Instruction) 36
put (Instruction) 74
putln (Instruction) 74

R

reach 145, 156
replace (Instruction) 58
resetMotion 147, 159, 167
resetMotion (Instruction) 159
resetTurn (Instruction) 179
resize (Instruction) 35
restartMove 159, 167
restartMove (Instruction) 160
return (Instruction) 22
right (Instruction) 57
righty 138, 140
round (Instruction) 48
roundDown (Instruction) 48
roundUp (Instruction) 48
RUNNING 93
rvel 156

S

safetyFault (Instruction) 111
sel (Instruction) 49
setDH (Instruction) 178
setFrame (Instruction) 128
setLanguage (Instruction) 79
setLatch (Instruction) 120
setMonitorSpeed (Instruction) 112
setMoveId (Instruction) 163
setMutex (Instruction) 89
setProfile (Instruction) 77
setTextMode (Instruction) 72
shoulder 136
sin (Instruction) 46
sio 28, 65
SioCtrl (Instruction) 67
sioGet (Instruction) 66
sioLink (Instruction) 66
sioSet (Instruction) 66
size (Instruction) 31, 35

sqrt (Instruction) 47
stopMove 167
stopMove (Instruction) 159
STOPPED 88
string 28
switch (Instruction) 23, 26

T

tan (Instruction) 46
taskCreate (Instruction) 91
taskCreateSync (Instruction) 92
taskKill (Instruction) 89
taskResume 83
taskResume (Instruction) 88
taskStatus 83
taskStatus (Instruction) 90
taskSuspend (Instruction) 88
title (Instruction) 74
toBinary (Instruction) 52
toNum (Instruction) 55
tool 28, 117
toString (Instruction) 54
trsf 28, 117
trsf align (Instruction) 126
tvel 156

U

userPage (Instruction) 72

V

vel 156

W

wait 84
wait (Instruction) 93
waitEndMove 84, 146, 167
waitEndMove (Instruction) 160
watch 84
watch (Instruction) 94
while (Instruction) 24
wnegative 139
workingMode (Instruction) 110
wpositive 139
wrist 136

