Langage de programmation

Préface

Ce document présente **Deko**, le langage de programmation intégré dans **SYNAPXIS** (version 2.7.6).

Le Locle, août 2015 HE-Arc/IHC/UR-LPR

Table des matières

CHAPITRE 1

Introduction

1.1 Description

Deko est le langage de programmation intégré au logiciel **SYNAPXIS**. Il observe les propriétés suivantes :

Langage interprété L'interprétation est faite une seule et unique fois au chargement des fichiers programmes (démarrage de l'application) ainsi qu'après une modification d'une *fonction* dans l'éditeur.

Multi-tâche Le langage permet de réaliser des applications *multi-tâches*. Chaque tâche est identifiée par un nom unique et comporte une *fonction* principale ; la fin de l'exécution de celle-ci provoque l'arrêt de la tâche.

Réentrance et récursivité Les *fonctions* sont réentrantes et peuvent exécutée sur plusieurs tâche simultanément. Une *fonction* peut aussi se rappeler elle-même.

Variables locales Les variables locales, ou *auto* ont une instance propre à chaque instance d'exécution d'une fonction.

Variables et constantes globales Les variables et les constantes globales sont visibles depuis toutes les procédures et pour chaque tâche.

1.2 Principes de base

1.2.1 Noms des éléments

Le nom de chaque élément doit être unique dans le cadre de l'application, et ce indépendamment de son type. Cela implique par exemple qu'une variable globale ne peut pas avoir le même nom qu'une fonction déjà existante, même dans un autre module. Pour tous les noms utilisés, **Deko** est *case sentive* et n'accepte que des caractères alphanumériques et le *underscore*.

1.2.2 Instances et références

Les variables et constantes globales sont instanciées lors du chargement du module et existent [1] jusqu'à sa fermeture.

Les variables *locales* sont créées lors de la création de l'instance d'exécution de la fonction [2] et existent jusqu'à sa sortie.

Les variables *locales* et *globales* sont utilisées par *référence*. Que ce soit pour le passage de paramètres à une fonction ou le retour d'une valeur par un fonction, c'est toujours la *référence* de la variable qui est considérée. Ainsi, si par exemple une variable *global* est passé en paramètre à une fonction qui la modifie avant de la retourner comme valeur de sortie, c'est toujours elle qui est considérée. Si la référence vers une variable doit être rompue lors du traitement, il est nécessaire de faire une copie de sa valeur, par exemple en réalisant une opération *neutre* :

```
function(a+0, b*1, pos+trans(0), txt+"")
```

Lors de l'accès à un élément d'un tableau (*array*), c'est également la référence sur l'élément qui est utilisée.

1. Les variables et constantes globales sont maintenue par le *taskManager* de **SYNAPXIS**.
2. Une instance d'exécution d'une fonction est créée pour chaque tâche dans laquelle elle est utilisée et pour chacune de ses occurrence dans le cas d'un appel récursif.

CHAPITRE 2

Modules et fonctions

Il existe 2 types de modules : le module *programme*, qui regroupe les différentes *fonctions*, ainsi que les *variables* et *constantes globales*. Il est encodé dans le fichier au format texte (**.mip*). Le module *data* ne peut lui que contenir que des *variables* et *constantes globales*. Il est encodé au format binaire (**.dip*), et lui seul permet l'enregistrement de données *globales* de type structuré *class*.

Il est ainsi possible d'organiser les *variables* et *constantes globales* en fonction de leur utilisation :

— *module programme* : fonctionnement multi-tâche, gestion de l'application, etc.

— *module data* : données de l'application (dont les valeurs peuvent évoluer ou non), données de grande taille, etc.

2.1 Modules programmes

Les *modules programme* permettent d'organiser et de regrouper les fonctions d'une application. Chaque *module* correspond à un fichier.

Un *module programme* regroupe :

— des fonctions

— des variables globales

— des constantes globales

La visibilité de ces éléments est globale, et n'est donc pas limitée au module lui-même. Ainsi, une fonction d'un module peux accéder aux variables et constantes globales d'un autre module.

2.2 Fonctions

Une fonction est définie par son nom, les paramètres qui lui sont passés, ses variables locales et son type de retour.

2.2.1 Nom

Le nom d'une fonction est l'identifiant par lequel l'appel de la fonction est fait [1]. Au lancement d'une nouvelle tâche, c'est également le nom de la fonction principale qui doit être spécifié.

Le nom est toujours suivi d'une paire de parenthèses, qui contient les éventuels paramètres.

2.2.2 Paramètres

Le nombre de paramètres d'une fonction n'a pas de limite théorique, et peut également être nul. Dans la mesure où ils sont passés par *référence*, ils peuvent aussi bien être utilisés comme paramètres d'*entrée* ou de *sortie*. Tous les paramètres déclarés doivent être définis lors de l'appel à une fonction, il n'y a pas de valeur par défaut. Chaque paramètre est défini par son type et son nom.

2.2.3 Variables locales

Une fonction peut utiliser autant de variables locales que nécessaire. Elles sont créées lors de l'appel à la fonction et détruites à la fin de son exécution (après un *return* ou la dernière ligne). Chaque variable locale est définie par son type et son nom.

2.2.4 Type de retour

Une fonction peut retourner une valeur, et ainsi être imbriquée dans l'appel d'une autre fonction ou instruction.

```
va = atan2(func_A(t), func_B(t))
...
vb = func_C(t)
```

Dans le cas ou la fonction possède un type de retour, il est nécessaire d'utiliser l'instruction *return* suivie de la valeur à retourner.

```
return val
...
return a*b
```

La valeur est retournée par *référence* ; si une variable locale est retournée, celle-ci n'est pas détruite à la sortie de l'exécution de la fonction et subsiste tant qu'elle est utilisée.

S'il n'y a pas de type de retour, l'utilisation du mot clé *return* n'est pas obligatoire. Ce type de retour vierge est appelé *void*.

1. A l'image du langage C, **Deko** réalise l'appel à une fonction directement à partir de son nom et ne nécessite pas d'instruction spécifique (*call*).

2.3 Variables globales

Les variables globales sont visibles depuis toutes les fonctions, indifféremment du module auquel elles appartiennent. Elles sont créées en mémoire au chargement du module et détruite à sa fermeture. La valeur d'une variables globales est toujours nulle après son chargement en mémoire (valeur par défaut).

Si une variables est accédée depuis plusieurs tâches simultanément, un mécanisme de verrouillage peut être mis en place afin d'assurer son *accès exclusif*.

Chaque variable globale est définie par son type et son nom.

2.4 Constantes globales

Les constantes globales fonctionnent de la même manière que les variables globales, à la différence près qu'elles sont enregistrées dans le fichier module avec leur valeur, et permettent de sauvegarder différentes données. De cette manière, après le chargement, la valeur d'une constante peut être non-nulle. Si une constante globale est modifiée et que le fichier module est réenregistré, c'est la nouvelle valeur qui sera sauvegardée. Un *module programme* ne peut pas contenir de constante globales de type *class* ou tableau de *class*.

Il est possible de renommer les variables et les constantes, ainsi que de faire le changement variable <-> constante.

2.5 Modules data

Un module data ne peut contenir que des *variables* et *constantes globales* de tous type, y compris des *constantes globales* de type structurées *class*, ainsi que des tableau d'éléments structurés. Leur gestion est identique à celle d'un module *programme*.

CHAPITRE 3

Les types primitifs

3.1 Introduction

Les types primitifs du langage **Deko** sont les suivants :

— *bool*
— *int*
— *real*
— *string*
— *locc*
— *locj*
— *array*
— *class*

3.2 Application

Ces types sont applicables aux éléments suivants :

— Paramètres (instructions)
— Valeur retournée (instructions)
— Paramètres (fonctions)
— Variables locales (fonctions)
— Valeur retournée (fonctions)
— Variables globales
— Constantes globales

3.3 Le type *bool*

Le type *bool* est le type binaire et correspond aux états *true* et *false*.

3.4 Le type *int*

Le type *int* est le type *entier signé*, codé sur 32 bits. Il peut représenter les valeurs entière de -2'147'483'648 à +2'147'483'647.

3.5 Le type *real*

Le type *real* est le type réel en virgule flottante, codé sur 64 bits (*double*). Il peut représenter les valeurs réelles sur la plage ±5.0 x 10^-324 .. ±1.7 x 10^308, avec 15 digits significatifs.

Remarque : Dans un programme, "123" est interprété comme un entier signé, alors que "123." ou "123.0" est interprété comme un réel.

3.6 Le type *string*

Le type *string* correspond aux chaînes de caractères ASCII. La longueur d'un string n'a pas de limite théorique. Le premier caractère est à l'index 0. La fin d'une chaîne de caractères est marquée par le caractère nul '\0'. L'occupation mémoire d'un string est donc de (nombre_caractères+1) octets.

3.7 Le type *locc*

Le type *locc* utilise une matrice de position (translations et rotations) et permet de réaliser toutes les opérations sur les positions dans un système cartésien 3D.

La forme matricielle n'étant pas visible pour l'utilisateur, les variables de type *locc* sont interfacées (initialisation et lecture) avec 6 coordonnées : x, y, z, rx, ry, rz. Les rotations sont effectuées successivement sur les axes x, y' et z" (repère mobile).

Le type *locc*est également appelé *position cartésienne*.

3.8 Le type *locj*

Le type *locj* est une ensemble de valeur réelle correspondant aux positions des axes mécaniques d'un robot (par exemple, translation en mm et rotations en degrés). Elles sont aussi appelées positions *articulaires* ou *joint*.

3.9 Le type *array*

Le type *array* permet de créer des tableaux sur la base des types primitifs. Il observe les règles suivantes :

— *array* est un tableau à une seule et unique dimension.

— Sa grandeur n'a pas de limite théorique ; la limite réelle est donnée par le système informatique.

— Les éléments du tableau peuvent être de n'importe quel type parmi les types primitifs.

— Tous les éléments du tableau doivent être du même type (un tableau peut être vidé et ensuite rempli avec des éléments d'un autre type).

— Les éléments d'un tableau pouvant être eux-mêmes des tableaux, il est possible de créer des tableaux à dimensions multiples (pas de limite théorique). Par exemple, dans le cas d'un tableau à 2 dimensions, un premier tableau contient pour chaque élément un tableau représentant une "ligne". L'index de l'élément dans une ligne correspond alors au numéro de la colonne. Les lignes peuvent avoir des types différents entre elles, ce qui permet d'obtenir des tableaux structurés.

Ligne/Colonne	C0	C1	C2	C3	type de valeur
L0	"A"	"B"	"C"	"D"	nom (*string*)
L1	2.01	1.99	2.03	2.04	diamètre (*real*)
L1	10.01	10.02	9.97	10.07	hauteur (*real*)
L2	*true*	*false*	*true*	*true*	état (*bool*)

Table 3.1: Exemple de tableau à 2 dimensions.

3.10 Le type *class*

Le type *class* est à l'heure actuelle un simple type structuré. Son évolution vers un type *objet* avec des *méthodes* et notions d'*héritage* n'est pas une priorité de développement (2013).

Ses propriété sont les suivantes :

— une variable de type *class* peut avoir un ou plusieurs *membres*.

— chaque *membre* est identifié par sa *clé* unique.

— chaque *membre* peut être de type différent, parmi tous les types primitifs existants.

— un membre pouvant lui-même être de type *class*, les éléments structurés peuvent être imbriqués sans limite.

— l'accès à un membre se fait par l'opérateur point ".", qui retourne la *référence* sur la valeur du membre.

— il n'y a pas d'éditeur de type structuré. La création d'élément structuré se fait directement au sein du programme, et est complètement libre (pas de vérification syntaxique).

— lors de son écriture, un membre est créé s'il n'existe pas déjà.

— lors de sa lecture, si un membre n'existe pas, il est créé avec le type indéfini *none*, qui engendre une erreur d'exécution ultérieure.

— pour chaque type de structure de l'application, il est conseillé de faire une fonction *constructeur* qui retourne une structure de type *class*, en ayant initialisé chacun de ses membres avec les valeurs par défaut.

Des instructions dédiée au type *class* permettent de définir un membre, le supprimer, tester son existence.

```
// Exemple de constructeur pour un type structuré. person est une
// variable auto, qui est retournée et utilisée plus loin (par
// exemple ajoutée à un tableau global
// arrEmpty est une variable tableau vide, servant à initialiser
// un membre vide dans la structure (copie)

// constructeur PersonNew()
person.firstName = "John"
person.lastName = "Doe"
person.birth.year = 1979
person.birth.month = 3
person.birth.day = 13
person.size = 183
person.weight = 81.5
person.single = true
person.pets = arrEmpty
arrayAdd(person.pets, "Milou")
arrayAdd(person.pets, "Leika")
return person

...

// création d'une nouvelle personne. la variable auto newPerson
// pointe sur la structure nouvellement créée.
valuePointer("newPerson", PersonNew())

// ajout de la nouvelle personne dans le tableau global gFamily
arrayAddRef(gFamily, newPerson) // ajout par référence, pas de copie!

// édition des données de la nouvelle personne
edit(newPerson.firstName, "Prénom")
edit(newPerson.lastName, "Nom de famille")

...
// utilisation d'une structure person via le pointeur auto aPerson
valuePointer("aPerson", gFamily[idx])
print(aPerson.firstName + " " + aPerson.lastName)
print("Pets count: ", arraySize(aPerson.pets))
print("Size [m]:", aPerson.size/100.0)
...
```

3.11 Conversions

3.11.1 Conversions implicites

Deko effectue des conversions implicites entre certains types primitifs. Celles-ci sont effectuées dans les cas suivants :

— Lors des tests (*if, while, until, for)*, c'est la valeur booléenne de l'expression qui est considérée, même si la condition est par exemple la valeur d'un compteur.

— Lors de l'affichage d'une valeur (**print**, **alert**), c'est la valeur *string* de l'expression qui est considérée.

— Lors de l'assignement (=), si les variables de part et d'autre de l'opérateur sont de types différents, il est possible d'assigner la valeur d'un entier vers un réel, et celle d'un entier ou d'un réel vers une chaîne de caractères. L'assignement provoque une erreur d'exécution dans les autres cas.

— Lors de l'évaluation d'une puissance (x^y) , ce sont les valeurs réelles des variables qui sont considérées, même si il s'agit de variables entières. L'opérateur *modulo* utilise lui la valeur entière des arguments.

Le tableau suivant présente les conversions implicites qui sont réalisées sans l'appel à un opérateur spécifique :

→	*bool*	*int*	*real*	*string*	*locc*	*locj*	*array*
bool		-	-	x	-	-	-
int	x(1)		x(2)	x	-	-	-
real	x(1)	-(2)		x	-	-	-
string	-	-	-	(3)	-	-	-
locc	-	-	-	x		-	-
locj	-	-	-	x	-		-
array	-	-	-	x	-	-	

Table 3.2: Conversions implicites.

Explications :

1. La valeur booléenne peut être évaluée pour un entier ou un réel. Le résultat est *false* si la valeur est égale à 0 (-1 $*$ 10e − 9 $<$ valeur $<$ 1 $*$ 10e − 9 pour les réels), et *true* dans le cas contraire.
2. Un réel ne peut pas être converti directement en entier (perte de données). Il existe une instruction permettant d'obtenir la partie entière d'un nombre réel (**integer**).

   Un entier peut être converti directement en réel.
3. Tous les types peuvent être converti vers une chaine de caractère. Dans le cas du type entier et réel, la chaîne de caractère prend la valeur du nombre. Dans le cas d'une variable de type *locc* la chaine de caractère vaut "dx, dy, dz, rx, ry, rz". Dans le cas d'une variable de type *locj* la chaine de caractère vaut "j1, j2, j3, ..., jn". Dans le cas d'une variable de type *array* la chaine de caractère vaut "[taille]".

3.11.2 Conversions lors du passage d'argument à une fonction

Dans le cas du passage d'argument à une fonction lors de son appel, aucune conversion n'est effectuée, à l'exception de la conversion entier → réel. Cette exception permet d'écrire dans le code des constantes entières (32, -12) et qu'elles soient interprétées en tant que réels.

```
// funcA posède deux paramètres d'entrées réels,
// ce qui implique une conversion entier -> réel
...
funcA(2, 3)
...
```

3.11.3 Conversions avec instructions

Des instructions spécifiques permettent de réaliser les conversions suivantes :

→	*bool*	*int*	*real*	*string*	*locc*	*locj*	*array*
bool		-	-	i	-	-	-
int	i		i	i	-	-	-
real	i	-		i	-	-	-
string	-	(1)	(2)		-	-	-
locc	-	-	(3)	i		-	-
locj	-	-	(4)	i	-		-
array	-	-	-	i	-	-	

Table 3.3: Conversions avec instructions.

1. L'instruction **strToInt** permet de convertir une chaîne de caractère en valeur entière.
2. L'instruction **strToReal** permet de convertir une chaîne de caractère en valeur réelle.
3. Les instructions **dx**, **dy**, **dz**, **drx**, **dry**, **drz** permettent d'extraire les différentes coordonnées d'une position cartésienne *locc*.
4. L'instruction **jointValue** permet d'extraire une la coordonnée d'un axe d'une position articulaire.

3.12 Opérations

Le tableau ci-dessous présente les opérations standards supportées par les types primitifs :

Opération	*bool*	*int*	*real*	*string*	*locc*	*locj*	*array*	**Priorité**
$+$	-	x(1)	x(1)	x(2)	x(3)	-	x(4)	3
$-$	-	x(1)	x(1)	-	-	-	-	3
$*$	-	x(1)	x(1)	-	x(5)	-	-	2
$/$	-	x(1)	x(1)	-	-	-	-	2
$\hat{}$ (puissance)	-	x(1)	x(1)	-	-	-	-	1
$\%$ (modulo)	-	x(6)	-	-	-	-	-	1
$-$ (négation)	-	x	x	-	-	-	-	0
$==, != $	x	x	x(7)	x(8)	x	x	x(9)	4
$\leq, <, >\geq$	-	x	x	-	-	-	-	4
$!(inversion)$	x	-	-	-	-	-	-	0
and	x	-	-	-	-	-	-	5
or	x	-	-	-	-	-	-	6
&	-	x	-	-	-	-	-	5
|	-	x	-	-	-	-	-	6

Table 3.4: Opérations sur les types primitifs.

Explications :

1. Pour les opérations de base (+ - * / $\hat{}$), si les deux arguments sont de types entiers, le résultat retourné est lui aussi un entier. Par contre, si un des deux arguments est de type réel (ou les deux), le résultat retourné est un réel. Une exception subsiste pour l'opérateur division (/) : si le résultat de la division est entier, le nombre retourné est lui aussi un entier. Si le résultat à une partie fractionnelle non-nulle, le nombre retournée est dans ce cas réel.

   Remarque : Dans un programme, "123" est interprété comme un entier signé, alors que "123." ou "123.0" est interprété comme un réel. Ainsi, "6/3" est une *division entière* dont le résultat vaut 2. Par contre, "1/2" est interprété comme une *division réelle* dont le résultat vaut 0.5. Il faut utiliser l'opérateur*integer* pour forcer une division entière : "integer(1/2)".
2. Lorsqu'une variable de type *string* est additionnée à une variable d'un autre type, la valeur de celle-ci est automatiquement convertie en chaîne de caractère ; le résultat de l'addition est alors obligatoirement de type *string*.
3. L'opérateur + utilisé avec deux variables de type *locc* réalise le *produit matriciel ordinaire* des deux matrices de positions.
4. L'addition de deux tableaux copie les éléments du premier tableau puis les éléments du deuxième tableau vers la valeur cible.
5. Une variable *locc* multipliée par un réel ($0 \leq r \leq 1$) permet d'obtenir une transformée intermédiaire. La translation est linéaire, et la rotation est réalisée

grâce aux *quaternions* (rotation unique autour d'un axe orienté). Celle-ci permet de réaliser une interpolation linéaire sphérique, connue sous le nom de SLERP (Spherical Linear intERPolation).

6. Le reste de la division entière est fait à partir de la valeur entière des 2 arguments. Si des nombres réels sont passés à cet opérateur, ce sont leur valeur entière qui sont considérées (conversion implicite).
7. L'égalité de deux réels est faite avec une précision de $1 * 10e - 9$.
8. L'égalité de deux chaînes de caractères est sensible à la *casse*.
9. L'égalité entre deux tableaux est faite à condition qu'ils aient le même nombre d'éléments et que ceux-ci soient identique entre eux.

3.13 Assignement

L'assignement d'une variable est réalisé à l'aide de l'instsruction =. Afin de simplifier des notations tels que `i = i+1` , l'assignement peut être accompagné d'une opération simple : `i += 1` . Le tableau ci-dessous présente à quels type sont applicables les différents assignements combinés à un opérateurs.

Assignement	*bool*	*int*	*real*	*string*	*locc*	*locj*	*array*
$=$	x	x	x	x	x	x	x
$+=$	-	x	x	x	x	-	x
$-=$	-	x	x	-	-	-	-
$*=$	-	x	x	-	-	-	-
$/=$	-	x	x	-	-	-	-

Table 3.5: Assignement.

CHAPITRE 4

Instructions internes

Les instructions internes constituent la base du langage : gestion des structures (boucles, tests), opérations sur les valeurs, interaction avec le système du PC, etc.

Remarques :

— Une *expression* représente une valeur unique. Elle peut consister en une simple constante ou variable, une instruction, un opérateur ou un appel à une fonction retournant une valeur.

— Certaines instructions peuvent avoir un ou plusieurs paramètres d'entrées *optionnels*. Ils sont identifiés par le symbole $*$. Si un paramètre optionnel n'est pas défini, c'est sa valeur par défaut qui est considérée lors de l'exécution de l'instruction. Il ne peut pas y avoir de *trou* dans l'implémentation des paramètres optionnels : tous les paramètres non-définis sont obligatoirement à la fin de l'implémentation de l'instruction.

— Une fonction ayant un nombre de paramètre indéfinis est indiquée comme suit : *instruction(type param1, ...)*

4.1 Structures

Ces instructions permettent de régir le fonctionnement du code du programme : ce sont les tests (**`if`**, **`switch`**) et les boucles (**`while`**, **`do...until`**, **`for`**). La fin d'une structure est toujours marquée par l'instruction **`end`**, à l'exception de la boucle **`do...until`**.

if

```
if(bool condition)
```

condition	*bool*	expression booléenne

La valeur *true* de la condition provoque l'exécution du code conditionnel inséré entre le *if* et le *end*. Si le *if* est combiné avec un *else*, la condition *false* provoque l'exécution du code inséré entre le *else* et le *end*.

```
cond = a > b
if (cond)
  ... // code exécuté si cond == true
end

...

if (func(a, b))
  ... // code exécuté si func(a, b) retourne true
else
  ... // code exécuté si func(a, b) retourne false
end
```

switch

```
switch(all value)
```

value	*all*	expression comparée aux différentes expressions des *cases*

L'instruction *switch* permet d'exécuter sélectivement certaines portions de code. L'expression *value* est exécutée avant que sa valeur soit comparée aux différentes possibilités de la structure. Si un des cas contient une expression dont la valeur est équivalente, l'exécution se poursuit avec le code compris entre le *case* correspondant et le *case* suivant. Si aucun cas ne contient une expression dont la valeur est identique, l'exécution se poursuit, le cas échéant, avec le cas *default*. Le cas *default* étant optionnel, il se peut que pour certaines valeur de *value*, aucun code ne soit exécuté dans la structure *switch*. Dans le cas ou plusieurs cas ont une valeur identique à *value*, c'est le premier qui est considéré.

```
switch (count)
  case 1
     ...
  case 2, 2*100, 3+100
     ...
  case errorCode()// -1
     ...
  case 0
     ...
  default
     ...
end
```

repeat

```
repeat(int value)
```

value	*int*	expression de type entier dont la valeur indique le nombre d'exécution de la boucle

L'instruction *repeat* est utile pour créer une boucle qui n'a pas besoin de compteur. L'expression *value* est exécutée une seule et unique fois, puis la boucle est répétée en fonction de la valeur entière de *value*. Si *value* change au cours de l'exécution de la boucle, la nouvelle valeur n'est jamais considérée.

```
repeat(n)
    ...// code répété n fois
end
```

for

```
for(<initialisation>; bool condition; <increment>)
```

<initialisation>		instruction d'initialisation, peut également consister à l'appel à une fonction
condition	*bool*	expression booléenne dont la valeur *true* autorise l'entrée dans la boucle
<increment>		instruction d'incrémentation (ou d'évolution de la condition), peut également consister à l'appel à une fonction

Le code *<initialisation>* est exécuté une seule fois avant le début de la boucle *for*. Le test de la *condition* est exécuté avant l'entrée de la boucle, à chaque itération. La valeur *true* provoque l'entrée dans la boucle. A la fin de la boucle, le code *<increment>* est exécuté. Le processus se répète jusqu'à ce que la condition ait une valeur *false*.

```
for(i = 0; i < n; i+=1)
  ... // code répété n fois
end

...

for(funcInit(i); funcCond(i), funcInc(i))
  ... // code répété
end
```

while

while(*bool* **condition**)

condition	*bool*	expression booléenne

La valeur *true* de la condition provoque l'entrée dans la boucle ; le code inséré entre le *while* et le *end* est exécuté. La boucle recommence avec à chaque itération l'exécution de la condition ; dès le moment où celle-ci prend une valeur *false* la boucle n'est plus répétée. L'exécution du programme se poursuit à la ligne qui suit le *end*. Étant donnée que le test est effectué *avant* l'entrée dans la boucle, il se peut que le code de la boucle ne soit pas exécuté si la condition est initialement fausse.

```
while (cond)
  ... // code répété tant cond == true
end

... // suite du programme
```

do...until

do...until(*bool* **condition**)

condition	*bool*	expression booléenne

L'entrée dans la boucle est systématique, et le code compris entre le *do* et le *until* est de toute manière exécuté au moins une fois. La condition est exécutée à chaque itération ; la valeur *false* provoque la répétition de la boucle. Dès le moment où la condition prend la valeur *true*, la boucle n'est plus répétée et l'exécution du programme se poursuit à la ligne qui suit le *until*.

```
do
  ... // code répété tant cond == false
until (cond)

... // suite du programme
```

exit

exit

L'instruction *exit* provoque la sortie de la boucle dans laquelle elle est immédiatement insérée.

```
for(i = 0; i < n; i+=1)
  ...
  error = func(i)
  if (error)
     exit // sortie de la boucle for en cas d'erreur
  end
  ... // suite du code de la boucle
end

... // suite du programme, exécuté à la sortie
... // de la boucle après n répétition ou une erreur.
```

continue

continue

L'instruction *continue* provoque un saut au début de la boucle dans laquelle elle est immédiatement insérée. Elle permet de simplifier le code en évitant par exemple l'imbrication de tests *if*. Il ne faut pas utiliser cette instruction dans une boucle **do...until** car elle devient une boucle infinie.

```
for(i = 0; i < n; i+=1)
  ...
  itemKind = func(i)
  if (itemKind != kind)
     continue // saut au sommet de la boucle
  end
  ... // suite du code de la boucle;
  ... // n'est pas exécuté si itemKind != kind
end
...
```

return

return *all* **value**

L'instruction *return* provoque la sortie immédiate de la fonction. Si un type de retour est déclarée pour celle-ci, le *return* doit être accompagné d'une valeur dans le type correspondant. C'est cette valeur qui est transmise à la fonction appelante. Dans le cas d'une fonction sans type de retour (*void*), le *return* est utilisé seul. Si un *return* est utilisé dans le programme principal d'une tâche, il provoque son arrêt immédiat.

```
...
error = func()
if (error != 0)
  return // sortie de la fonction (void)
end
...
```

```
...
return a+b
```

```
...
if (a > b)
  if (a < 100)
    return a
  else
    return 100
  end
end

return abs(b)
```

4.2 Valeurs

Ces instructions permettent de réaliser des conversions, des opérations mathématiques et différentes transformations sur les types primitifs.

bmask

```
int bmask(int bit_N,  ...)
```

bit_N	*int*	numéro du bit mis à 1 dans le mask

Retourne un entier dont la valeur est fonction des bits spécifiés en paramètres. L'instruction peut avoir au maximum 32 paramètres d'entrée, leur valeur devant être comprise entre $0 \leq bit \leq 32$. La valeur retournée vaut $2^{bit\_i} + 2^{bit\_j} + 2^{bit\_k} + ...$

```
print(bmask(0)) // "1"
print(bmask(0,1,2,3)) // "15"
print(bmask(1, 10)) // "1026"
print(bmask(1, 10) | bmask(2)) // "1030"
print(bmask(1, 2, 5, 7, 10) & bmask(3, 4, 7)) // "128"
```

abs

```
int/real abs(int/real value)
```

value	*int/real*	expression entière ou réelle

Retourne la valeur absolue du paramètre *value*. Cette instruction supporte les 2 types entier et réel. Le type de la valeur retournée est identique à celui de la valeur passée en paramètre.

integer

```
int integer(real value)
```

value	*real*	expression réelle

Retourne la partie entière du paramètre *value*.

real

```
real real(real value)
```

value	*real*	expression réelle

Retourne la partie réelle du paramètre *value*.

ln

```
real ln(real value)
```

value	*real*	expression réelle, différente de 0

Retourne le logarithme naturel de *value*.

cos

```
real cos(real value)
```

value	*real*	angle en degrés

Retourne le cosinus de l'angle *value*.

sin

```
real sin(real value)
```

value	*real*	angle en degrés

Retourne le sinus de l'angle *value*.

tan

```
real tan(real value)
```

value	*real*	angle en degrés

Retourne la tangente de l'angle *value*.

atan2

```
real atan2(real y, real x)
```

y	*real*	opposée de l'angle
x	*real*	adjacente de l'angle

Retourne l'angle, en degré, en fonction de l'opposée et de l'adjacente. L'angle est considéré sur 360 degrés : $-180 < angle \leq 180$.

hypot

```
real hypot(real a, real b)
```

a	*real*	premier côté du triangle rectangle
b	*real*	deuxième côté du triangle rectangle

Retourne l'hypoténuse du triangle rectangle dont a et b sont la longueur des deux côtés droits.

strLen

```
int strLen(string string)
```

string	*string*	chaîne de caractères

Retourne la longueur de la chaîne de caractères *string*, c'est à dire le nombre de caractères qu'elle contient.

strPos

```
int strPos(string string, string subString, int startPos*)
```

string	*string*	chaîne de caractère dans laquelle est cherchée la position de la chaîne *subString*
subString	*string*	chaîne de caractère cherchée
startPos*	*int*	position pour le départ de la recherche. Par défaut, la valeur 0 est considérée, la recherche commence au premier carractère

Retourne la position de la chaîne *subString* dans la chaîne *string*. La recherche commence au premier caractère (index 0). Si *string* ne contient pas *subString*, l'instruction retourne -1. Si *string* contient plusieurs fois *subString*, le paramètre *startPos* peut être utilisé pour commencer la recherche après le premier caractère.

```
...
str = "Hello World" //
pos = strPos(str, "World") // pos == 6
pos = strPos(str, "o") // pos == 4
pos = strPos(str, "o", pos+1) // pos == 7
```

strMid

string **strMid**(*string* **string**, *int* **start**, *int* **length**)

string	*string*	chaîne de caractère à partir de laquelle est générée la chaîne retournée
start	*int*	index du caractère à partir duquel commence la copie des caractères vers la chaîne retournée
length	*int*	nombre de caractère qui sont copiés vers la chaîne retournée

Retourne une chaîne de caractère dans laquelle sont copiés les caractères du paramètre d'entrée *string*, à partir de l'index *start* et sur une longueur de *length* caractères.

```
...
strA = "Hello World"
pos = strPos(strA, "World") // pos == 6
strB = strMid(strA, pos, strLen(strA)-pos)
print(strB)// "World"
```

strToInt

int **strToInt**(*string* **string**)

string	*string*	chaîne de caractère de laquelle est décodée une valeur entière

Retourne la valeur entière qui est codée dans la chaîne de caractère *string*. Si la chaîne de caractère ne commence pas par un nombre entier signé, l'instruction retourne 0.

```
...
valInt = strToInt("213") // valInt == 213
valInt = strToInt("-4567") // valInt == -4567
valInt = strToInt("a233") // valInt == 0
valInt = strToInt("65Hz) // valInt == 65
```

strToReal

real **strToReal**(*string* **string**)

string	*string*	chaîne de caractère de laquelle est décodée une valeur réelle

Retourne la valeur réelle qui est codée dans la chaîne de caractère *string*. Si la chaîne de caractère ne commence pas par un nombre réel, l'instruction retourne 0.0.

```
...
valReal = strToReal("0.285") // valReal == 0.285
valReal = strToReal("-45.67") // valReal == -45.67
valReal = strToReal("a28.3305") // valReal == 0.0
valReal = strToReal("-78.0548N") // valReal == -78.0548
valReal = strToReal("-4e-5") // valReal == -0.00004
```

char

string **char**(*int* **asciiCode**)

asciiCode	*int*	code ascii du caractère

Retourne une chaîne de caractère contenant uniquement le caractère correspondant au code *asciiCode*. Le code ascii doit être imprimable : $32 \leq asciiCode \leq 126$. Les caractères 10 (*new line*) et 13 (*carriage return)* sont également supportés.

```
...
str = char(72)
str += char(101)
str += char(108)
str += char(108)
str += char(111)
print(str) // str == "Hello"
```

asciiValue

int **asciiValue**(*string* **char**)

char	*string*	caractère

Retourne la valeur ASCII du caractère.

trans

```
locc trans(real x,
           real y*,
           real z*,
           real rx*,
           real ry*,
           real rz*)
```

x	*real*	translation selon l'axe x
y*	*real*	translation selon l'axe y. Par défaut, vaut 0.0.
z*	*real*	translation selon l'axe z. Par défaut, vaut 0.0.
rx*	*real*	rotation autour de l'axe rx, en degrés. Par défaut, vaut 0.0.
ry*	*real*	rotation autour de l'axe ry, en degrés. Par défaut, vaut 0.0.
rz*	*real*	rotation autour de l'axe rz, en degrés. Par défaut, vaut 0.0.

Retourne une position cartésienne initialisée avec les paramètres x, y, z, rx, ry, rz.

```
...
loc = trans(20, 50, 100)
print(loc) // 20, 50, 100, 0, 0, 0
loc += trans(0, 0, 40, 0, 0, 45)
print(loc) // 20, 50, 140, 0, 0, 45
loc += trans(100)
print(loc) // 90.711, 120.711, 140, 0, 0, 45
```

inverse

```
locc inverse(locc loc)
```

loc	*locc*	position cartésienne

Retourne l'inverse de la position cartésienne *loc*.

```
...
inv = inverse(loc)
inv += loc
print(inv) // 0, 0, 0, 0, 0, 0
```

deltaTo

```
locc deltaTo(locc locFrom, locc locTo)
```

locFrom	*locc*	position cartésienne de départ
locTo	*locc*	position cartésienne d'arrivée

Retourne la différence entre la position cartésienne de départ et d'arrivée. Si d est la différence entre A et B, la relation suivante peut être posée : $A * d = B$. L'instruction calcule et retourne la valeur de d : $d = A^{-1} * B$. Cette instruction est notamment très utile pour effectuer des *changement de repères*.

```
...
posA = trans(50, 100, 30, 0, 0, 90)
posB = trans(51, 102, 35)
posDelta = deltaTo(posA, posB)
print(posDelta) // 2, -1, 5, 0, 0, -90
```

distanceTo

```
real distanceTo(locc locFrom, locc locTo)
```

locFrom	*locc*	position cartésienne de départ
locTo	*locc*	position cartésienne d'arrivée

Retourne la distance entre la position cartésienne de départ et d'arrivée. Elle est exprimée dans la même unité que celle des translation x, y, z.

```
...
posA = trans(50, 100, 30, 0, 0, 90)
posB = trans(51, 102, 35)
dist = distanceTo(posA, posB)
print(dist) // 5.477226 ≈= (1^2 + 2^2 + 5^2)^1/2
```

alignTo

locc **alignTo**(*locc* **locToAlign**, *locc* **locReference**, *string* **alignMode***)

locToAlign	*locc*	position cartésienne à orienter
locReference	*locc*	position cartésienne de référence
alignMode*	*string*	mode d'alignement

Retourne une position cartésienne dont la translation est identique à celle de *locToAlign*, et dont les rotations *rx, ry, rz* sont calculées de manière à obtenir l'alignement par rapport à la position de référence *locReference*, selon le mode sélectionné *alignMode* :

ALIGN_Z_TO_NEAREST (valeur par défaut) : l'axe z de la position est aligné selon la plus proche des directions principales de *locReference* (il y a 6 directions principales : *-x, x, -y, y, -z, z*). C'est l'angle minimal entre ces 6 directions et l'axe z+ de *locToAlign* qui permet de déterminer la nouvelle orientation de l'axe z. L'axe Z de la position retournée est donc orienté différemment par rapport à *locToAlign*.

CONFINE_Y_TO_XY : l'axe z de la position retournée est identique à celui de *locToAlign*. Une rotation autour de celui-ci (rz) est réalisée de manière à ce que le vecteur y de la position retournée soit parallèle au plan x-y. Comme il y a 2 possibilités à cette condition, le vecteur y de *locReference* est utilisé de manière à orienter le vecteur y de la position ($angle < 90$). Si les 2 vecteurs z de *locToAlign* et *locReference* sont colinéaires, la position retournée et identique à *locToAlign*.

```
...
locToAlign = trans(50, 100, 30, -75, -10, -90)
locReference = trans(0)
locToAlign = alignTo(locToAlign, locReference)
print(locToAlign) // 50, 100, 30, -90, 0, -91.32
```

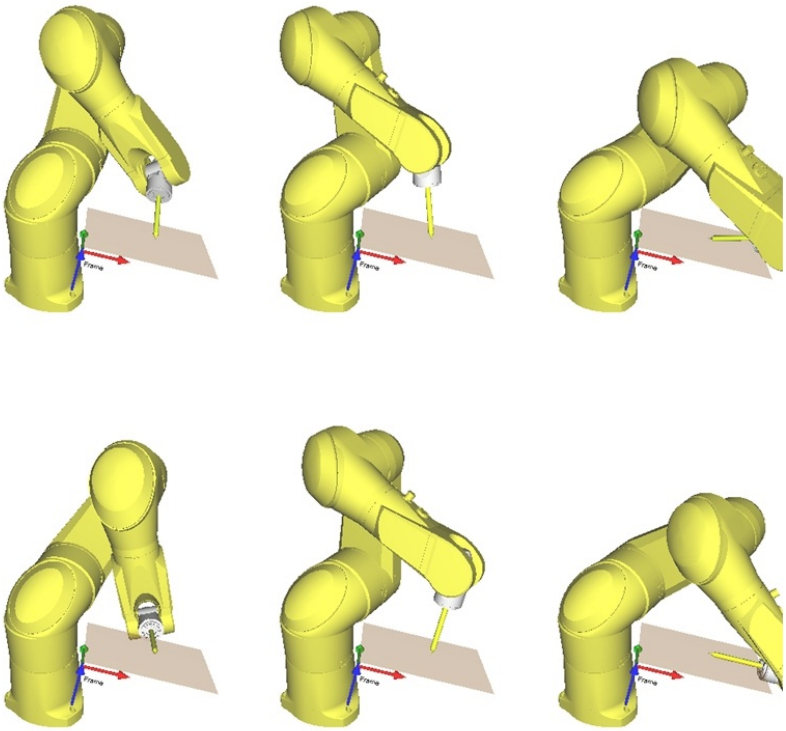

Figure 4.1: Exemples d'alignement *ALIGN_Z_TO_NEAREST*. L'image du haut montre la position avant alignement, celle du bas après. La position de référence est le frame représenté. Les directions d'alignements sont respectivement *-y, -z, -x*.

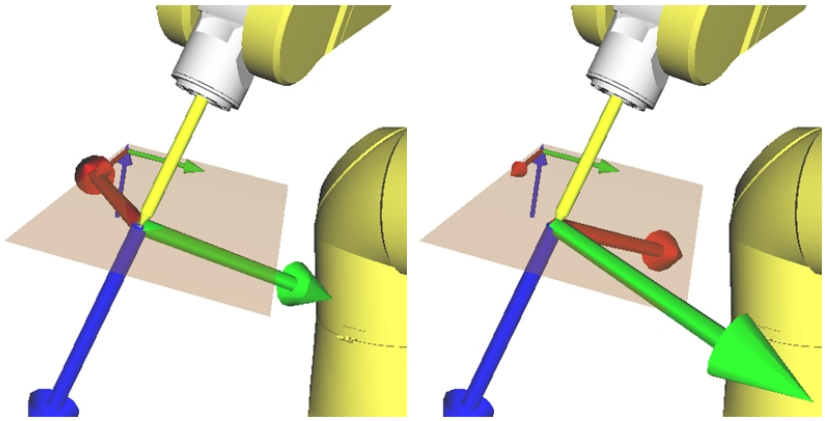

Figure 4.2: Exemple d'alignement *CONFINE_Y_TO_XY*. L'image de gauche montre la position avant alignement, celle de droite après. La position de référence est le frame représenté.

frameCompose

```
locc frameCompose(locc origin,
                  locc ptx,
                  locc pty,
                  locc position*)
```

origin	*locc*	position cartésienne correspondant à l'origine pour le calcul de l'orientation du frame
ptx	*locc*	correspondant à la direction x pour le calcul de l'orientation du frame
pty	*locc*	correspondant à la direction y pour le calcul de l'orientation du frame
position*	*locc*	correspondant à la position (translation) du frame. Si ce paramètre n'est pas spécifié, le frame est positionné à l'origine

Retourne une position cartésienne correspondant au *frame* calculé comme suit : les rotations rx et ry sont calculées à partir du plan passant par les points *origin*, *ptx* et *pty*, l'axe z étant normal à ce plan. La rotation rz est donnée par le vecteur passant par les points *origin* et *ptx*. Si le paramètre *position* est spécifié, c'est sa translation qui est appliquée au frame (x, y, z). Dans le cas contraire, la translation est celle du paramètre *origin*.

```
...
origin = trans(100, 100, 100)
ptx = trans(50, 100, 100)
pty = trans(105, 55, 100)// angle xoy != 90 degrés
frame = frameCompose(origin, ptx, pty)
print(frame) // 100, 100, 100, 0, 0, 180
position = trans(500, -250, -50)
frame = frameCompose(origin, ptx, pty, position)
print(frame) // 500, -250, -50, 0, 0, 180
```

dx

real **dx**(*locc* **loc**)

loc	*locc*	position cartésienne

Retourne la coordonnée x de la position cartésienne *loc*.

dy

real **dy**(*locc* **loc**)

loc	*locc*	position cartésienne

Retourne la coordonnée y de la position cartésienne *loc*.

dz

real **dz**(*locc* **loc**)

loc	*locc*	position cartésienne

Retourne la coordonnée z de la position cartésienne *loc*.

drx

```
real drx(locc loc)
```

loc	locc	position cartésienne

Retourne la coordonnée rx de la position cartésienne *loc*, en degrés.

dry

```
real dry(locc loc)
```

loc	locc	position cartésienne

Retourne la coordonnée ry de la position cartésienne *loc*, en degrés.

drz

```
real drz(locc loc)
```

loc	locc	position cartésienne

Retourne la coordonnée rz de la position cartésienne *loc*, en degrés.

decomposeValue

void **decomposeValue**(*all* **val**, *string* **kind**, *array* **outComponents**)

val	*all*	valeur à décomposer
kind	*string*	type de décomposition
outComponents	*array*	tableau contenant les composants après décomposition

L'instruction décompose la variable *val*, en fonction de son type, et du type de décomposition :

LOCC : LOCC_RXRYRZ (rotation successives rx, ry, rz sur le repère modifié), LOCC_YPR (rotation successives rz, ry, rz sur le repère modifié), LOCC_ABC (rotation rx, ry, rz sur le repère parent, inchangé), LOCC_QUATERNIONS (composantes x, y, z du vecteur de rotation, et rotation).

Les composantes sont retournées dans le tableau *outComponents*. Les translations sont exprimées en *mm* et les rotations en *degrés*.

transformMatrix

void **transformMatrix**(*locc* **loc**, *array* **outMatrix**)

loc	*locc*	position cartésienne
outMatrix	*array*	tableau contenant les 16 valeurs courantes de la matrice de transformation

Affecte le tableau *matrix* avec les 16 valeurs de la matrice de transformation de la position cartésienne *loc* :

$$\begin{pmatrix} x_x & y_x & z_x & t_x \\ x_y & y_y & z_y & t_y \\ x_z & y_z & z_z & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Les valeurs sont copiées dans le tableau selon les colonnes, à partir de l'élément 0 :

$$[x_x, x_y, x_z, 0, \ldots, t_x, t_y, t_z, 1]$$

setTransformMatrix

```
void setTransformMatrix(locc loc, array matrix)
```

loc	*locc*	position cartésienne
matrix	*array*	tableau contenant les 16 nouvelles valeurs de la matrice de transformation

Affecte les 16 valeurs de la matrice de transformation de la position cartésienne *loc* avec les valeurs du tableau *matrix*.

jointValue

```
real jointValue(locj jointPosition, int axisIndex)
```

jointPosition	*locj*	position articulaire
axisIndex	*int*	numéro de l'axe

Retourne la valeur de l'axe *axisIndex* de la position articulaire *jointPosition*. Pour un nombre n d'axes, le numéro de l'axe observe les limites suivantes : $0 \leq axisIndex < n$

jointSetValue

```
void jointSetValue(locj jointPosition,
                   int axisIndex,
                   real value)
```

jointPosition	*locj*	position articulaire
axisIndex	*int*	numéro de l'axe
value	*real*	nouvelle valeur de l'axe

Affecte la valeur de l'axe *axisIndex* de la position articulaire *jointPosition* à la valeur *value*. Pour un nombre n d'axes, le numéro de l'axe observe les limites suivantes : $0 \leq axisIndex < n$

arraySize

```
int arraySize(array array)
```

array	*array*	tableau

Retourne la taille (le nombre d'éléments) du tableau *array*.

arrayAdd

```
void arrayAdd(array array, all value)
```

array	*array*	tableau
value	*all*	valeur à ajouter à la fin du tableau

Ajoute la valeur *value* à la fin du tableau *array*. Si le tableau contient déjà des éléments, l'élément ajouté doit obligatoirement être de type identique. Une copie de *value* est faite lors de cette opération.

arrayAddRef

```
void arrayAddRef(array array, all value)
```

array	*array*	tableau
value	*all*	valeur à ajouter à la fin du tableau

Ajoute la valeur *value* à la fin du tableau *array*. Si le tableau contient déjà des éléments, l'élément ajouté doit obligatoirement être de type identique. Aucune copie de *value* est n'est faite, c'est la référence qui est considérée.

arrayInsert

```
void arrayInsert(array array, all value, int index)
```

array	*array*	tableau
value	*all*	valeur à insérer dans le tableau
index	*int*	index de l'emplacement où la valeur est ajoutée

Insert la valeur *value* dans le tableau *array*, à la position *index*. Si l'*index* est plus grand ou égal à la taille du tableau, l'élément est simplement ajouté à la fin du tableau. Sinon, l'élément est inséré à l'index spécifié, et les éléments suivants sont décalés d'un index vers la fin. Si le tableau contient déjà des éléments, l'élément inséré doit obligatoirement être de type identique.

arrayRemove

```
void arrayRemove(array array, int index)
```

array	*array*	tableau
index	*int*	index de l'emplacement à supprimer

Supprime l'élément à la position *index* du tableau *array*.

arrayClear

```
void arrayClear(array array)
```

array	*array*	tableau

Vide le tableau *array* de tous ses éléments.

bufferSize

int **bufferSize**(*buffer* **buffer**)

buffer	*buffer*	buffer de données

Retourne la taille courante du *buffer* en octets.

bufferClear

void **bufferClear**(*buffer* **buffer**)

buffer	*buffer*	buffer de données

Efface les données du *buffer* et affecte sa position courante à 0.

bufferPosition

int **bufferPosition**(*buffer* **buffer**)

buffer	*buffer*	buffer de données

Retourne la position courante du *buffer*. Elle correspond à l'index de l'octet auquel sera effectuée la prochaine lecture/écriture.

bufferSetPosition

```
void bufferSetPosition(buffer buffer, int position)
```

buffer	*buffer*	buffer de données
position	*int*	nouvelle position courante du buffer

Affecte la *position* courante du *buffer*.

bufferRead

```
bool bufferRead(buffer buffer,
                all readValue,
                int size,
                bool swap*)
```

buffer	*buffer*	buffer de données
readValue	*all*	valeur lue dans le buffer
size	*int*	nombre d'octets lus
swap*	*bool*	flag indiquant si un swap doit être fait sur la valeur lue

Lis la valeur *readValue* dans le *buffer*. Le type de donnée lu correspond au type de la variable *readValue* : *bool*, *int*, *real*, *string*. Dans le cas de type *int* ou *real*, le paramètre *size* permet de spécifier si c'est un *short* (2 octets) ou un *long* (4 octets), respectivement un *float* (4 octets) ou un *double* (8 octets) qui doit être considéré. Dans le cas d'un *string*, si *size* vaut -1, la lecture des caractères se fait jusqu'à ce que le caractère de fin `'\0'` soit lu.

Si le flag *swap* vaut *true*, la conversion big-endian →little-endian est exécutée dans le cas de la lecture d'un *int* ou d'un *real*.

bufferReadChar

```
bool bufferReadChar(buffer buffer, int readValue)
```

buffer	*buffer*	buffer de données
readValue	*int*	valeur lue dans le buffer

Lis un octet dans le *buffer* et affecte sa valeur au paramètre *readValue* ($0 \leq val \leq 255$).

bufferWrite

```
bool bufferWrite(buffer buffer,
                 all writeValue,
                 int size,
                 bool swap*)
```

buffer	*buffer*	buffer de données
writeValue	*all*	valeur écrite dans le buffer
size	*int*	nombre d'octets écris
swap*	*bool*	flag indiquant si un swap doit être fait sur la valeur écrite

Écrit la valeur *writeValue* dans le *buffer*. Le type de donnée écrit correspond au type de la variable *writeValue* : *bool*, *int*, *real*, *string*. Dans le cas de type *int* ou *real*, le paramètre *size* permet de spécifier si c'est un *short* (2 octets) ou un *long* (4 octets), respectivement un *float* (4 octets) ou un *double* (8 octets) qui doit être considéré. Dans le cas d'un *string*, le paramètre *size* indique le nombre de caractères de la chaîne qui doivent être copiés dans le buffer. S'il vaut -1, l'écriture des caractères se fait pour tous les caractères de la chaîne ; le caractère de fin `'\0'` est dans ce cas lui aussi écrit dans le buffer.

Si le flag *swap* vaut *true*, la conversion little-endian →big-endian est exécutée dans le cas de l'écriture d'un *int* ou d'un *real*.

bufferWriteChar

```
bool bufferWriteChar(buffer buffer, int writeValue)
```

buffer	*buffer*	buffer de données
writeValue	*int*	valeur écrite dans le buffer

Écris un octet *writeValue* dans le *buffer* ($0 \leq val \leq 255$).

classKeyDefined

```
bool classKeyDefined(class object, string key)
```

object	*class*	structure
key	*string*	nom du membre

Retourne *true* si le membre *key* est défini dans la structure *object*, *false* dans le cas contraire.

classKeyDelete

```
bool classKeyDelete(class object, string key)
```

object	*class*	structure
key	*string*	nom du membre

Supprime le membre *key* de la structure *object*, et retourne *true* en cas de succès.

classKeySet

void **classKeySet**(*class* **object**, *string* **key**, *all* **value**)

object	*class*	structure
key	*string*	nom du membre
value	*all*	valeur du membre

Créer le membre *key* en lui affectant la valeur *value* par référence. Si le membre existe déjà, la valeur est remplacée.

valuePointer

void **valuePointer**(*string* **autoPointerName**, *all* **value***)

autoPointerName	*string*	nom du pointeur auto
value*	*all*	variable cible

Transforme une variable auto en un pointeur vers la valeur *value*. Le pointeur ne peut être qu'une variable auto, de type identique à celui de la variable cible, identifiée par son nom *autoPointerName*. A partir de cette instruction, toute utilisation de la variable auto fera *référence* à la variable *value*. La valeur pointée peut être une constante ou une variable globale, l'élément d'un tableau, le membre d'une structure, etc. Si le paramètre *value* n'est pas défini, le pointeur est réinitialisé et redeviens une variable auto normale.

```
valuePointer("position", gPositionArray[3])
position = trans(100, 200)
...

valuePointer("entraxeX", gPaletArray[3].entraxes.x)
print("Entraxe colonnes: " entraxeX)
...

valuePointer("newItem", CreateItem())
newItem.creationDate = dateTimeString(dateTimeCurrent())
...
```

4.3 Système

Ces instructions donnent accès au système : affichages, gestion de tâches parallèles, gestion de fichiers, heure et temps système, etc.

print

```
void print(all value,  ...)
```

value	*all*	valeur à afficher

Affiche les paramètres sur une ligne, dans une fenêtre *console*. Celle-ci s'ouvre automatiquement lors du premier affichage. Le de paramètre à afficher n'est pas limité.

```
...
print(x)
print(2+3*5*x)
print(cos(alpha), sin(alpha), tan(alpha))
print("Nombre d'itération : ", i)
print("Nombre de " + itemName + " : ", n)
print("Position actuelle : ", loc)
```

alert

```
int alert(string title,
          string text*,
          string button0*,
          string button1*,
          string button2*,
          string button3*,
          string button4*)
```

title	*string*	titre de l'alerte
text*	*string*	texte de l'alerte
button0*	*string*	texte du 1er bouton
button1*	*string*	texte du 2ème bouton
button2*	*string*	texte du 3ème bouton
button3*	*string*	texte du 4ème bouton
button4*	*string*	texte du 5ème bouton

Affiche une boîte de dialogue ayant pour titre *title*. Le paramètre *text* peut être utilisé pour afficher une description. Si aucun bouton n'est spécifié, la boîte de dialogue contient un bouton *OK* par défaut. L'instruction retourne l'*index* du bouton sélectionné : $0 \leq index \leq (n-1)$.

```
...
alert("Opération terminée")
...

if (alert("Erreur. Continuer?", descr, "OK", "STOP"))
  taskStop()// arrêt de la tâche courante
end
... // suite du processus

hit = alert("Démarrer?", "", "Oui", "Non", "Info")
switch(hit)
  ...
end
```

alertProgress

```
bool alertProgress(string title,
                   string text,
                   real ratio*,
                   bool close*,
                   bool useCancel*)
```

title	*string*	titre de l'alerte
text	*string*	texte de l'alerte
ratio*	*real*	progression 0.0− > 1.0
close*	*bool*	flag provoquant la fermeture de l'alerte
useCancel*	*bool*	flag spécifiant si le bouton *annuler* est affiché

Affiche et gère une boîte de dialogue non-bloquante. Celle-ci peut être utilisée pour afficher des information au cours d'une opération dont la progression est connue (0-100%) ou non.
Si le paramètre *ratio* lors de l'affichage de l'alerte au premier appel de l'instruction est plus petit que 0 à l'affichage, l'alerte est indéfinie, définie dans les autres cas.
Le bouton *Annuler* peut être affiché ou non à l'aide du paramètre *useCancel* afin de donner la possibilité à l'utilisateur d'interrompre l'opération en cours.
L'instruction doit être rappelée à chaque fois que le *texte* ou la progression *ratio* doivent être remis à jour. L'instruction retourne *false* si l'opération est annulée par l'opérateur.
L'instruction doit être appelée avec le paramètre *close* à *true* afin de masquer l'alerte. Si le bouton *Annuler* n'est pas affiché, l'interface est figée pour l'utilisateur tout pendant qu'elle n'est pas refermée par le programme.

```
alertProgress("Indéfinie", "info", -1.0, false, true)
delay(0.1)
for (i = 0; i < 20; i+=1)
  if (!alertProgress("", clock()))
    alert("annulé")
    exit
  end
  delay(0.1)
end
alertProgress("", "", 0, true)
...

alertProgress("Définie", "info", 0.0, false, true)
delay(0.1)
for (i = 0; i < 20; i+=1)
  if (!alertProgress("", clock(), i/20))
    alert("annulé")
    exit
  end
  delay(0.1)
end
alertProgress("", "", 0, true)
```

edit

```
bool edit(all value, string title)
```

value	*all*	valeur à éditer
title	*string*	titre de l'éditeur

Affiche l'éditeur de la variable *value*, qui peut ainsi être modifiée. La boîte de dialogue a pour titre le paramètre *title*. L'instruction retourne *true* si l'édition de la valeur est validée (bouton *OK*), *false* dans le cas contraire.

```
...
speed = 1000
if (!edit(speed, "Vitesse de rotation"))
  taskStop() // action annulée, arrêt de la tâche courante
end
```

arraySelect

```
int arraySelect(array array, string title)
```

array	*array*	tableau de valeurs
title	*string*	titre du dialogue

Affiche un dialogue avec une liste contenant les descriptions des éléments du tableau *array*. La boîte de dialogue a pour titre le paramètre *title*. L'instruction retourne l'index de l'élément sélectionné : $0 \leq index \leq (n-1)$. Si aucun élément n'est sélectionné, la valeur -1 est retournée.

arraySelectMulti

bool **arraySelectMulti**(*array* **items**, *array* **selected**, *string* **title**)

items	*array*	tableau de valeurs
selected	*array*	tableau de sélection (booléen)
title	*string*	titre du dialogue

Affiche un dialogue avec une liste contenant les descriptions des éléments du tableau *items*. A chaque élément correspond une case à cocher, dont l'état est défini par le tableau de booléens *selected*. La boîte de dialogue a pour titre le paramètre *title*. L'instruction retourne *true* si le dialogue est validé (OK) ou *false* si annulé (Annuler). Dans le cas d'une validation, les valeurs du tableau *selected* prennent la valeur de sélection des case à cocher correspondantes.

```
// Exemple de code: les éléments du tableau names non-sélectionnés
// sont ensuite supprimé de ce tableau.

if (!arraySelectMulti(names, selected, title))
    return false
end
for (idx = arraySize(names); idx > 0; idx-=1)
    if (!(selected[idx-1]))
        arrayRemove(names, idx-1)
    end
end
```

random

```
real random()
```

Retourne un nombre réel aléatoire avec une résolution de 10^{-9} : $0 \leq r < 1$.

```
...
// calcul d'un index aléatoire dans le tableau gArray
size = arraySize(gArray)
randomIndex = integer(size*random())
...
```

delay

```
void delay(real time)
```

time	*real*	durée du délais

Provoque une attente d'une durée *time*, exprimée en secondes. La résolution de cette durée est de 1 milliseconde. Durant cette attente, la ressource processeur est libérée. Si la durée spécifiée est de 0 seconde, aucune attente n'est réalisée, mais la tâche est libérée jusqu'à son prochain séquencement, laissant la ressource processeur à disposition des autres tâche actives sur le système.

```
... // signal clignotant
frequency = 3 // 3Hz
halfPeriod = 0.5/frequency // 1/3/2 == 0.16666 s
while(run)
  signal =!signal
  ...
  delay(halfPeriod)
end
```

clock

real **clock**()

Retourne le temps système, en secondes. Le temps système 0 correspond au démarrage de l'OS.

```
...
t = clock() // surveillance d'un signal
error = false
timeout = 3 // [secondes]
while(true)
  signal = ...
  if (signal)
     break
  else
     if (clock()-t > timeout)
       error = true
       break
  end
  ...
  delay(0) // libération de la ressource processeur
end
```

dateTimeCurrent

real **dateTimeCurrent**()

Retourne la date et l'heure système. La partie réelle du nombre correspond à l'heure, considérée sur 24h : 12h correspond à 1/2, 1h correspond à 1/24, 1 minute correspond à $1/(24 * 60)$, 1 seconde correspond à $1/(24 * 3600)$.

La partie entière du nombre correspond au numéro du jour à compter de la date du 30/12/1899. Le 8 août 2008 correspond au jour 39668.

```
...
cycleTime = 78 // temps de cycle : 1min 18 secondes
cycleCount = 4000 // nombre de cycles
dateStart = dateTimeCurrent() // heure de début
dt = cycleTime * cycleCount // durée en secondes
dDate = dt/(86400) // 24*3600 secondes
dateEnd = dateStart + dDate
```

dateTimeString

string **dateTimeString**(*real* **dateTime**)

dateTime	*real*	date/heure système

Retourne une chaîne de caractère au format "dd.mm.yyyy hh :mm :ss", en fonction de la date et heure système passée en paramètre *dateTime*. Si la partie réelle est nulle, la chaîne de caractère retournée est au format "dd.mm.yyyy".

```
...
print(dateTimeString(0)) // "30.12.1899"
print(dateTimeString(39668)) // "08.08.2008"
print(dateTimeString(39668.505)) // "08.08.2008 12 :07 :12"
```

```
...
cycleTime = 78 // temps de cycle : 1 min 18 secondes
cycleCount = 1000 // nombre de cycles
dateStart = dateTimeCurrent() // heure de début
print(dateTimeString(dateStart)) // "08.08.2008 16 :40 :21"
dt = cycleTime * cycleCount // durée en secondes
dDate = dt/(86400) // 24*3600 secondes
print(strMid(dateTimeString(dDate), 11, 8)) // "21 :40 :00"
dateEnd = dateStart + dDate
print(dateTimeString(dateEnd)) //"09.08.2008 14 :20 :21"
```

taskExecute

taskExecute **<programName>**(...), *string* **taskName***, *bool* **displayError***, *string* **errorProgram***

<programName>		nom du programme principal de la nouvelle tâche
taskName*	*string*	nom de la nouvelle tâche
displayError*	*bool*	flag spécifiant l'affichage d'une éventuelle l'erreur
errorProgram*	*string*	nom du programme à appeler lorsqu'une erreur se produit

Lance l'exécution d'une nouvelle tâche. Le paramètre *<programName>* spécifie le nom du programme principal de cette tâche. Si celui-ci a des paramètres d'entrée, ils doivent être implémentés dans la paire de parenthèse (passage par *référence*).

Le paramètre *taskName* spécifie le nom de la nouvelle tâche. Si le nom spécifié correspond à celui d'une tâche déjà existante, une erreur est créée et la nouvelle tâche n'est pas lancée. Si ce paramètre n'est pas spécifié, la tâche prend comme valeur un numéro égal au nombre total de tâches existantes, y compris la nouvelle tâche.

Le fait de donner à une tâche un nom connu est important, car il permet par la suite d'obtenir des information sur celle-ci ainsi que de la piloter (pause, stop, ...).

Si le flag *displayError* vaut *false*, aucun message n'est affichée à l'écran lorsqu'une erreur survient dans la tâche exécutée (fenêtre rouge). Si rien n'est spécifié, l'erreur est affichée normalement.

Si un programme d'erreur est spécifié, celui-ci est appelé dans la tâche exécutée si une erreur se produit. La tâche est stoppée lorsque le programme d'erreur retourne. Celui-ci doit avoir un paramètre d'entrée *int*, correspondant au numéro d'erreur.

```
...
// variable globale initialisée avec
// le nom de la nouvelle tâche
gTaskName = "mainTask"
...

// lancement de la nouvelle tâche,
// avec comme programme principal mainProgram
taskExecute mainPrograme(1, true), gTaskName

...
if (taskExists(gTaskName))
  // mise en pause de l'exécution la tâche
  taskPause(gTaskName)
  ...
  // relancement de l'exécution de la tâche
  taskResume(gTaskName)
end

...
if (taskExists(gTaskName))
  taskStop(gTaskName) // arrêt de la tâche
end
```

taskSetPriority

void **taskSetPriority**(*int* **lines**)

lines	*int*	nombre de lignes exécutées à chaque séquencement de la tâche

Modifie la priorité de la tâche courante, en spécifiant le nombre de lignes exécutées à chaque séquencement de la tâche. Lors de son lancement, chaque tâche a une priorité identique (séquencement de 20 lignes par défaut. Max : 1000).

```
...
taskSetPriority(300) // priorité élevée pour traitement
rapide
while(...)
  ...
end
taskSetPriority(50)// priorité faible pour reste de la
tâche ...
```

taskName

string **taskName**()

Retourne le nom de la tâche courante.

```
...
if (error)
  if (taskName() == "mainTask")
     alert("Arrêt de la tâche principale")
  end
end
```

taskExists

```
bool taskExists(string taskName)
```

taskName	*string*	nom de la tâche

Retourne *true* si la tâche spécifiée existe, *false* dans le cas contraire. Si une tache existe, elle est soit un pause, soit en cours d'exécution.

```
...
if (!taskExists("auxTask"))
  alert("Problème avec la tâche auxiliaire")
  taskStop() // arrêt de la tâche courante
end
...
```

taskStop

```
bool taskStop(string taskName*)
```

taskName*	*string*	nom de la tâche à stopper

Stoppe la tâche spécifiée par le paramètre *taskName*. Si aucun nom n'est spécifié, c'est la tâche courante qui est considérée, une tâche pouvant se stopper elle-même.

Une fois stoppée, la tâche est détruite et ne peut pas être reprise à l'endroit où elle à été stoppée. Si la tâche spécifiée est déjà en pause, elle est détruite.

L'instruction retourne *true* en cas de succès.

```
...
if (error)
  taskStop("auxTask") // arrêt de la tâche auxiliaire
  taskStop() // arrêt de la tâche courante
end
...
```

taskPause

```
bool taskPause(string taskName*)
```

taskName*	*string*	nom de la tâche à mettre en pause

Met en pause la tâche spécifiée par le paramètre *taskName*. Si aucun nom n'est spécifié, c'est la tâche courante qui est considérée, une tâche pouvant se mettre en pause elle-même.

Une fois mise en pause, la tâche peut par la suite être relancée (depuis une autre tâche) afin qu'elle reprenne son exécution à l'endroit où elle à été suspendue.

L'instruction retourne *true* en cas de succès.

```
...
if (intervention)
  ...
  taskPause() // mise en pause de la tâche courante
end
... // code exécuté à la reprise de la tâche
```

taskPaused

```
bool taskPaused(string taskName)
```

taskName	*string*	nom de la tâche

Retourne *true* si la tâche spécifiée est couramment en pause, *false* dans le cas contraire.

taskResume

```
bool taskResume(string taskName)
```

taskName	*string*	nom de la tâche

Relance la tâche spécifiée par le paramètre *taskName*, à condition que celle-ci existe et soit en pause. L'instruction retourne *true* en cas de succès.

```
...
if (taskPaused("auxTask"))
  if (alert("Reprendre le cycle?", "", "Oui", "Stop"))
     // arrêt de la tâche auxiliaire (déjà en pause)
     taskStop("auxTask")
     taskStop() // arrêt de la tâche courante
  else
     // Reprise de la tâche auxiliaire
     taskResume("auxTask")
  end
end
...
```

taskMutex

```
bool taskMutex(bool ressource, bool state)
```

ressource	*bool*	variable globale
state	*bool*	état

Effectue une affectation indivisible de l'état *state* vers la variable globale *ressource*. Cette instruction permet de créer un mécanisme de verrouillage entre tâche. L'instruction retourne *true* quand elle a pu effectuer l'affectation indivisible.

taskCallStack

void **taskCallStack**(*array* **outStack**)

outStack	*array*	pile d'appel

Remplit le tableau *outStack* avec l'état actuel de la pile d'appel de la tâche courante (nom de la fonction et numéro de ligne). Le premier élément du tableau correspond à la fonction d'entrée de la tâche.

identifierExists

bool **identifierExists**(*string* **identifier**)

identifier	*string*	nom du module, du programme, de la variable ou de la constante globale

L'instruction retourne *true* si un module, un programme, une variable ou une constante globale dont le nom coïncide avec *identifier* existe, *false* dans le cas contraire. Le système considère les données qui existent en mémoire, en fonction des différents *modules* chargés.

programCreate

```
string programCreate(string moduleName, string newProgramName,
bool editName, string programToDuplicate*)
```

moduleName	*string*	nom du module dans lequel le nouveau programme est créé
newProgramName	*string*	nom du nouveau programme
editName	*bool*	flag indiquant si le nom du nouveau programme doit être édité
programToDuplicate*	*string*	nom du programme à copier pour créer le nouveau programme

Créer un nouveau programme, qui est ajouté au module *moduleName*. Le nom du nouveau programme est *newProgramName*. Si ce nom est déjà utilisé, ou si le flag *editName* vaut *true*, le nom est édité à l'écran. Si le paramètre *progamToDuplicate* est défini, le programme créé sera une copie de celui-ci. L'instruction retourne le nom du nouveau programme.

programEdit

```
bool programEdit(string programName, bool restrictedInstructions)
```

programName	*string*	nom du programme à éditer
restrictedInstructions	*string*	édition en mode restreint

Ouvre l'éditeur de programme pour le programme *programName*, en mode *Touch Screen Edition*. Si le flag *restrictedInstructions* vaut *true*, le programme ne peut comporter qu'un nombre restreint d'instructions et d'appels. L'instruction retourne *true* si le programme à été modifié, *false* s'il n'a pas été modifié ou que les modifications ont été annulées. Le programme exécutant cette instruction doit obligatoirement appartenir à un module différent de celui du programme édité.

programDelete

bool **programDelete**(*string* **programName**, *bool* **prompt**)

programName	*string*	nom du programme à supprimer
prompt	*string*	confirmation

Supprime le programme *programName*. Si le flag *prompt* vaut *true*, une confirmation est demandée.

programDescriptionRW

void **programDescriptionRW**(*string* **programName**, *array* **desrcLines**, *bool* **readOrWrite**)

programName	*string*	nom du programme
desrcLines	*array*	tableau des lignes de description
readOrWrite	*bool*	flag spécifiant la lecture ou l'écriture

L'instruction permet de lire ($readOrWrite = true$) ou écrire ($readOrWrite = false$) les lignes de description (*descrLines*) du programme *programName* .

moduleLoad

string **moduleLoad**(*string* **moduleFilePath**)

moduleFilePath	*string*	chemin du fichier du module

L'instruction charge en mémoire le module correspondant au fichier *moduleFilePath*, et retourne le nom de ce nouveau module.

Les programmes ainsi que les variables et constantes globales du modules sont dès lors disponibles pour le fonctionnement de l'application. Si un programme comporte faute de syntaxe ou de structure, une erreur est générée lors de son appel.

moduleClose

string **moduleClose**(*string* **moduleName**)

moduleName	*string*	nom du module

L'instruction ferme le module *moduleName*. Aucune sauvegarde n'est effectuée si le module a été modifié. Retourne *true* en cas de succès.

moduleSave

string **moduleSave**(*string* **moduleName**, *bool* **build***)*

moduleName	*string*	nom du module
build*	*string*	build des programmes avant enregistrement

L'instruction effectue l'enregistrement du module *moduleName* si celui-ci a été modifié. Retourne *true* en cas de succès. Si le paramètre *build* vaut *true*, les programmes du modules sont compilés avant l'enregistrement. En cas d'erreur de compilation, l'instruction retourne *false*, et l'enregistrement n'est pas effectué.

```
file = "C :\Temp \moduleDemo.mip"
gModName = moduleLoad(file)
print("module name : ", gModName)

...

if (identifierExists(gModName))
  moduleSave(gModName)
  moduleClose(gModName)
end
```

modulePrograms

void **modulePrograms**(*string* **moduleName**, *array* **outProgramNames**)

moduleName	*string*	nom du module
outProgramNames	*array*	tableau des noms de programmes du module

L'instruction rempli le tableau *outProgramNames* avec les noms des programmes du module *moduleName*.

fileRead

```
bool fileRead(string filePath, array lines)
```

filePath	*string*	fichier
lines	*array*	lignes de texte lues dans le fichier

Lit le contenu d'un fichier (en mode texte) et remplit le tableau *lines* ; chaque élément du tableau (de type *string*) correspond à une ligne du fichier. Le tableau est initialement vidé.

Le paramètre *filePath* correspond au chemin d'accès du fichier. L'instruction retourne *false*si le fichier spécifié n'existe pas ou ne peut être lu, *true* en cas de succès.

```
...
filePath = "C :\temp\data.cvs"
fileRead(filePath, lines)
n = arraySize(lines)
for (i = 0; i < n; i += 1)
  print(lines[i]) //impression des lignes du fichier
end
...
```

fileWrite

bool **fileWrite**(*string* **filePath**, *array* **lines**)

filePath	*string*	fichier
lines	*array*	lignes de texte écrites dans le fichier

Écrit le contenu du tableau *lines* dans un fichier (en mode texte) ; chaque élément du tableau (de type *string*) correspond à une ligne du fichier. Si le fichier existe déjà au moment de l'écriture, son contenu est écrasé par le contenu du tableau.

Le paramètre *filePath* correspond au chemin d'accès du fichier. Si l'arborescence des dossiers spécifiée n'existe pas ou si le fichier ne peut être créé, l'instruction retourne *false*, *true* en cas de succès.

```
...
arrayAdd(errorStringArray, "Erreur 1")
...
arrayAdd(errorStringArray, "Erreur 2")
...
arrayAdd(errorStringArray, "Erreur 3")
...
filePath = "C :\temp\error.log"
fileWrite(filePath, errorStringArray)
...
```

fileAdd

bool **fileAdd**(*string* **filePath**, *array* **lines**)

filePath	*string*	fichier
lines	*array*	lignes de texte écrites dans le fichier

Ajoute le contenu du tableau *lines* à la suite d'un fichier (en mode texte) ; chaque élément du tableau (de type *string*) correspond à une nouvelle ligne du fichier. Si le fichier n'existe pas au moment de l'écriture, il est créé.

Le paramètre *filePath* correspond au chemin d'accès du fichier. Si l'arborescence des dossiers spécifiée n'existe pas ou si le fichier ne peut être créé, l'instruction retourne *false*, *true* en cas de succès.

```
...
arrayClear(errorStringArray)
...
arrayAdd(errorStringArray, "Erreur N")
...
filePath = "C :\temp\error.log"
fileAdd(filePath, errorStringArray)
...
```

fileBrowse

bool **fileBrowse**(*string* **initialPath**, *string* **fileExt**, *string* **outFilePath**)

initialPath	*string*	Chemin du dossier initial
fileExt	*string*	Extension du fichier (filtre)
outFilePath	*string*	Chemin du fichier sélectionné

Ouvre une fenêtre de sélection de fichier à l'endroit spécifié par *initialPath*. Le chemin complet du fichier sélectionné est écrit dans le paramètre *outFilePath*. Si *initialPath* vaut "", la fenêtre s'ouvre à la racine du répertoire de travail (workspace). *fileExt* permet de spécifier quel type de fichier (extension) est recherché. L'instruction retourne *true* si un fichier a bien été sélectionné, *false* sinon.

```
...
if (fileBrowse("C :\Temp\", "txt", l_sOutFilePath))
  print("Fichier texte sélectionné :", l_sOutFilePath)
else
  print("Aucun fichier texte sélectionné")
end
...
```

xmlParse

bool **xmlParse**(*string* **filePath**, *class* **outData**)

filePath	*string*	Chemin du fichier xml
outData	*class*	Variable de type *class* recevant les données analysées

Extrait les données du fichier xml *filePath* dans la variable *outData*. Les données sont représentées en arborescence où chaque noeuds est une clé (numéro) qui contient au minimum les champs suivant : NAME, VALUE, TYPE, ATTRIBUTES. Le champ NAME contient le nom d'une balises du fichier xml. L'instruction retourne *true* si l'opération s'est bien déroulée, *false* sinon.

fileAppLog

void **fileAppLog**(*string* **text**)

text	*string*	texte

Ajoute le texte *text* à la suite du fichier *log* de Synapxis.

osCommand

```
all osCommand(string command, all params...)
```

command	*string*	sélecteur de la commande à exécuter
params	*all*	paramètres de la commande

Exécute une opération en fonction du sélecteur :

- "OPEN" : ouvre le fichier *param_1* (*string*) à l'aide du programme spécifié *param_2* (*string*).
- "PROCESS_COUNT" : retourne le nombre de processus en cours d'exécution (*int*) dont le nom coïncide avec *param_1* (*string*). La comparaison du nom du processus est *case sensitive*. Le nom spécifié peut ne représenter qu'une partie du nom du processus.
- "WINDOW_SEND_MESSAGE" : envoie un message à la fenêtre spécifiée par *param_1* (window class name) et *param_2* (window title). Le type du message est le *param_3*, et ses éventuels arguments *wParam* et *lParam* les paramètres 4 et 5 :
  - "WM_SYSCOMMAND" : modifie la disposition d'une fenêtre. Le paramètre *wParam* peut prendre les valeurs "SC_MINIMIZE", "SC_MAXIMIZE", "SC_RESTORE".
  - "WM_SETFOCUS" : applique le focus à la fenêtre.
  - "WM_CLOSE" : ferme la fenêtre/application.
- "SWITCH_TO_WINDOW" : sélectionne l'application liée à fenêtre spécifiée par *param_1* (window class name) et *param_2* (window title).
- "EXECUTE" : exécute une ligne de commande (*param_1*), et retourne le résultat (*int*) une fois que l'exécution de la ligne est effectuée.

```
...
pCount = osCommand("PROCESS_COUNT", "not")
osCommand("OPEN", "notepad++", "C :\temp\demo.txt")
delay(0.2)
while (osCommand("PROCESS_COUNT", "not") > pCount)
  delay(0.1)
end
...

osCommand("WINDOW_SEND_MESSAGE", "", "Calculatrice",
"WM_SYSCOMMAND", "SC_MINIMIZE")
...
//fermeture de Synapxis : osCommand("WINDOW_SEND_MESSAGE",
"", "Synapxis", "WM_CLOSE")
```

directoryWorkspace

string **directoryWorkspace**()

Retourne le chemin absolu du dossier du *workspace* courant.

4.4 Interfaces

SYNAPXIS permet de réaliser des *frames*, ou interfaces, dédiées à une application. Ceux-ci existent soit sous la forme de fenêtre, soit sous la forme d'onglet. Ils sont composés d'éléments tels que boutons, labels, check-boxes ou text-edit. Les instruction décrites ici permettent d'interagir avec ces interfaces depuis un programme : ouvrir et fermer une fenêtre, lire et modifier les propriétés d'un élément de frame.

interfaceShow

```
void interfaceShow(string name, bool modal)
```

name	*string*	nom de la fenêtre à afficher
modal	*bool*	option d'affichage

Affiche la fenêtre correspondant au nom *name*. Si le paramètre *modal* vaut *true*, l'instruction est bloquante : l'exécution de la tâche courante reprend au moment où la fenêtre ouverte est refermée. Si ce paramètre vaut *false*, la fenêtre est affichée et l'exécution du programme se poursuit normalement.

interfaceClose

```
void interfaceClose(string name)
```

name	*string*	nom de la fenêtre à fermer

Ferme la fenêtre correspondant au nom *name*.

interfaceProperty

```
all interfaceProperty(string frame,
                      string component,
                      string property)
```

frame	*string*	nom de l'interface
component	*string*	nom de l'élément de l'interface
property	*string*	nom de la propriété de l'élément

Retourne la valeur de la propriété *property* de l'élément graphique *component* de l'interface *frame*.

```
...
// vérifie l'état "checked" du contrôle check-box
// "Mesure" de la fenêtre "Maintenance"
if (intefaceProperty("Maintenance", "Mesure", "Checked"))
  ...
end
...
```

interfaceSetProperty

```
bool interfaceSetProperty(string frame,
                          string component,
                          string property,
                          all value)
```

frame	*string*	nom de l'interface
component	*string*	nom de l'élément de l'interface
property	*string*	nom de la propriété de l'élément
value	*all*	nouvelle valeur de la propriété

Modifie la valeur de la propriété *property* de l'élément graphique *component* de l'interface *frame*. L'instruction retourne *true* en cas de succès.

```
...
// mise à jour du texte du label "CycleTime"
// de la fenêtre "Main"
tCycle = ...
intefaceSetProperty("Main", "CycleTime", "Text", tCycle)
...
```

4.5 Interface GUI

Des interface graphiques spécifiques peuvent être créées dans des *dll* d'extensions. L'interaction entre elles et le programme est faite avec les instructions ci-dessous. Les composants graphiques, ou *contrôles* avec lesquels il est possible d'interagir peuvent être de type *bouton*, *champ éditable*, *liste-view*, *liste-box*, *combo-box*, *check-box*, *label*, etc.. Certaines instructions sont valables pour plusieurs type de contrôles, d'autres spécifiques à un type.

guiDisplay

```
void guiDisplay(string guiName, bool modal)
```

guiName	*string*	nom de l'interface à afficher
modal	*bool*	option d'affichage

Affiche la fenêtre correspondant au nom *guiName*. Si le paramètre *modal* vaut *true*, l'instruction est bloquante : l'exécution de la tâche courante reprend au moment où la fenêtre ouverte est refermée. Si ce paramètre vaut *false*, la fenêtre est affichée et l'exécution du programme se poursuit normalement.

guiCtrlEnable

```
void guiCtrlEnable(string guiName, string ctrlName, bool state)
```

guiName	*string*	nom de l'interface
ctrlName	*string*	nom du composant graphique
state	*bool*	état activé

Modifie l'état activé du composant graphique *ctrlName* de la fenêtre *guiName*.

guiCtrlSetFocus

void **guiCtrlSetFocus**(*string* **guiName**, *string* **ctrlName**)

guiName	*string*	nom de l'interface
ctrlName	*string*	nom du composant graphique

Active le focus sur le composant graphique *ctrlName* de la fenêtre *guiName*.

guiCtrlSetText

void **guiCtrlSetText**(*string* **guiName**, *string* **ctrlName**, *string* **text**)

guiName	*string*	nom de l'interface
ctrlName	*string*	nom du composant graphique
text	*string*	text du composant

Modifie le texte du composant graphique *ctrlName* de la fenêtre *guiName*.

guiCtrlText

string **guiCtrlText**(*string* **guiName**, *string* **ctrlName**)

guiName	*string*	nom de l'interface
ctrlName	*string*	nom du composant graphique

Retourne le texte du composant graphique *ctrlName* de la fenêtre *guiName*.

guiButtonChecked

bool **guiButtonChecked**(*string* **guiName**, *string* **buttonName**)

guiName	*string*	nom de l'interface
buttonName	*string*	nom du bouton check-box

Retourne l'état sélectionné du bouton check-box *buttonName* de la fenêtre *guiName*.

guiButtonCheck

void **guiButtonCheck**(*string* **guiName**, *string* **buttonName**, *bool* **state**)

guiName	*string*	nom de l'interface
buttonName	*string*	nom du bouton check-box
state	*bool*	état sélectionné

Modifie l'état sélectionné du bouton check-box *buttonName* de la fenêtre *guiName* selon *state*.

guiListFill

void **guiListFill**(*string* **guiName**, *string* **listName**, *array* **items**)

guiName	*string*	nom de l'interface
listName	*string*	nom de la liste
items	*array*	éléments à afficher dans la liste

Remplit la liste *listName* de la fenêtre *guiName* avec le contenu du tableau *items*. Le tableau doit être à 1 dimension dans le cas d'une liste à une seule colonne, et à 2 dimensions pour un tableau à plusieurs colonnes. Dans ce cas, la première dimensions correspond aux colonnes, et la deuxième au lignes. Le nombre de colonnes du tableau doit correspondre à celui de la liste, et chaque ligne doit être complète. Les éléments doivent être uniquement de type *string*.

guiListAdd

void **guiListAdd**(*string* **guiName**, *string* **listName**, *all* **item**)

guiName	*string*	nom de l'interface
listName	*string*	nom de la liste
item	*all*	éléments à ajouter dans la liste

Ajoute à la liste *listName* de la fenêtre *guiName* l'élément *item* à la fin de la liste. Dans le cas d'un tableau à une seule colonne, l'élément ajouté est considéré comme un *string* unique. Dans le cas d'une liste à plusieurs colonnes, *item* doit être un tableau à deux dimensions, et peut contenir une ou plusieurs lignes (*string*).

guiListItemSelected

int **guiListItemSelected**(*string* **guiName**, *string* **listName**)

guiName	*string*	nom de l'interface
listName	*string*	nom de la liste

Retourne l'index de l'élément couramment sélectionné de la liste *listName* de la fenêtre *guiName*. Si aucun élément n'est sélectionné, l'instruction retourne -1.

guiListItemSelect

void **guiListItemSelect**(*string* **guiName**, *string* **listName**, *int* **index**)

guiName	*string*	nom de l'interface
listName	*string*	nom de la liste
index	*int*	index à sélectionner

Sélectionne l'élément de la liste *listName* de la fenêtre *guiName* en fonction de *index*.

guiListClear

void **guiListClear**(*string* **guiName**, *string* **listName**)

guiName	*string*	nom de l'interface
listName	*string*	nom de la liste

Vide la liste *listName* de la fenêtre *guiName*.

guiCtrlSetColor

void **guiCtrlSetColor**(*string* **guiName**, *string* **ctrlName**, *int* **color**)

guiName	*string*	nom de l'interface
ctrlName	*string*	nom du composant
ctrlName	*int*	couleur RVB

Modifie la couleur du composant *ctrlName* de la fenêtre *guiName*. La couleur appliquée est codée en RGB256 : $0x00BBGGRR$.

guiSendMsg

void **guiSendMsg**(*string* **guiName**, *string* **msgName**, ...*)

guiName	*string*	nom de l'interface
msgName	*string*	nom du message

Envoie le message *ctrlName* à la fenêtre *guiName*. D'autre paramètre additionnels peuvent être envoyé en fonction du message.

4.6 tcp/ip

SYNAPXIS permet d'ouvrir des connexions tcp/ip *clients* ou *serveur*, pouvant être gérée à l'aide des instructions suivantes.

tcpConnect

bool **tcpConnect**(*string* **tcpDeviceName**)

tcpDeviceName	*string*	nom de la connexion

Ouvre la connexion *tcpDeviceName* en fonction des paramètres qui lui sont propres : adresse, numéro de port. L'instruction retourne *true* en cas de succès.

tcpDisconnect

bool **tcpDisconnect**(*string* **tcpDeviceName**)

tcpDeviceName	*string*	nom de la connexion

Ferme la connexion *tcpDeviceName*. L'instruction retourne *true* en cas de succès.

tcpIsConnected

bool **tcpIsConnected**(*string* **tcpDeviceName**)

tcpDeviceName	*string*	nom de la connexion

Retourne *true* si la connexion *tcpDeviceName* est couramment ouverte, *false* dans le cas contraire.

tcpSend

```
bool tcpSend(string tcpDeviceName, all data, int ticket*)
```

tcpDeviceName	*string*	nom de la connexion
data	*all*	données envoyées
ticket*	*int*	identifiant du message envoyé

Envoie les données *stringData* (*string* ou *buffer*) via la connexion *tcpDeviceName*. Dans le cas d'un fonctionnement en mode serveur, l'identifiant du message *ticket* permet de faire correspondre la réponse envoyée à la requête courante. L'instruction retourne *true* en cas de succès.

tcpSendAndWait

```
bool tcpSendAndWait(string tcpDeviceName,
                    all data,
                    all outData,
                    real timeOut*)
```

tcpDeviceName	*string*	nom de la connexion
data	*all*	données envoyées
outData	*all*	données reçues
timeOut*	*real*	délais de réponse

Envoie les données *data* via la connexion *tcpDeviceName*. Une réponse est attendue puis copiée dans le paramètre *outData*. Le paramètre *timeOut* permet de spécifier la durée de l'attente de la réponse, au delà de laquelle une erreur est générée. La valeur par défaut du timeout est de 2 secondes. L'instruction retourne *true* en cas de succès.

tcpPopMessage

```
bool tcpPopMessage(string tcpDeviceName,
                   all outData,
                   int outTicket*)
```

tcpDeviceName	*string*	nom de la connexion
outStringData	*all*	données reçues
outTicket*	*int*	identifiant du message reçu

Extrait la dernière réponse de la pile de réception de la connexion *tcpDeviceName* et copie son contenu dans le paramètre *outData*. Si le paramètre *outTicket* est spécifié, il est affecté avec la valeur de l'identifiant du message. Si la pile est vide, l'instruction retourne *false*, *true* dans le cas contraire.

CHAPITRE 5

Instructions externes

Les instructions externes sont publiées dans **Deko** par les différents modules de **SYNAPXIS**. Ces instructions permettent par exemple d'accéder aux données des *références* ou de la *machine*, d'interagir avec l'interface de *production*, de commander un *robot*, etc.

5.1 Références

Les instructions liées au module *références* permettent d'accéder aux données maintenues par les références elles-même.

Deko ne peut accéder uniquement aux données des référence qui sont *ouvertes* en mémoire. Pour utiliser les données d'une référence, la premier étape consiste à *sélectionner* une référence pour la tâche courante. Il est possible de travailler simultanément avec plusieurs références depuis plusieurs tâches. Une référence est également accessible depuis plusieurs tâches simultanément.

refListNames

```
bool refListNames(string directoryName, array outNames)
```

directoryName	*string*	nom du dossier de références
outNames	*array*	noms des références

L'instruction copie les noms des références présentes dans le dossier de référence *directoryName* dans le tableau *outNames*. L'instruction retourne *false* si le dossier de référence n'existe pas, *true* en cas de succès.

refOpen

bool **refOpen**(*string* **refName**)

refName	*string*	nom de la référence à ouvrir

L'instruction ouvre la référence *refName* et retourne *true* en cas de succès.

```
directoryName = "LOCAL"

if (!refListNames(directoryName, refNames))
  print("directory indéfini :", directoryName)
  return
end

refIndex = arraySelect(refNames, "Ouvrir référence...")
if (refIndex < 0)
  return
end

refName = refNames[refIndex]
refOpen(refName)
refSelect(refName)
...
```

refClose

bool **refClose**(*string* **refName***)

refName*	*string*	nom de la référence à fermer

L'instruction ferme la référence *refName* et retourne *true* en cas de succès. Si aucun nom de référence n'est spécifié, toutes les références ouvertes sont fermées.

refSave

```
bool refSave(string refName)
```

refName	*string*	nom de la référence à enregistrer

L'instruction enregistre les données de la référence *refName* et retourne *true* en cas de succès.

refSelect

```
bool refSelect(string refName*)
```

refName*	*string*	nom de la référence

Sélectionne la référence *refName* pour la tâche courante. La référence spécifiée doit être ouverte en mémoire.

Si le paramètre *refName* n'est pas être spécifié, 3 possibilités peuvent se présenter :

— aucune référence n'est ouverte ; l'instruction retourne *false*, aucune référence n'est sélectionnée pour la tâche courante.

— une seule référence est ouverte ; celle-ci est automatiquement sélectionnée pour la tâche courante, et l'instruction retourne *true*.

— plusieurs références sont ouvertes ; un dialogue permet à l'utilisateur de choisir la référence à sélectionner pour la tâche courante. L'instruction retourne *true* si une référence est sélectionnée, *false* si l'action est annulée.

Les instructions qui accèdent aux données des références décrites ci-dessous **considèrent la référence sélectionnée pour la tâche courante**.

refSelected

string **refSelected**(*string* **taskName***)

taskName*	*string*	nom de la tâche

Retourne le nom de la référence sélectionnée pour la tâche courante si aucun nom de tâche n'est spécifié, ou pour la tâche correspondante si le nom est spécifié. Si aucune référence n'est sélectionnée, l'instruction retourne une chaîne de caractères vide.

refApplyConfig

void **refApplyConfig**(*bool* **state**, *string* **configName**)

state	*bool*	flag indiquant si la configuration est appliquée ou enlevée
configName	*string*	nom de la configuration

Applique la configuration *configName* si le flag *state* vaut *true*, et l'enlève s'il vaut *false*.

Une référence peut avoir une ou plusieurs configuration, chacune d'entre elle définissant les paramètres nécessaires aux opérations d'usinage (tool, frames, outils). L'application d'une configuration affecte le tool au robot (un seul robot utilisé par configuration), les outils aux frames, etc. L'enlèvement d'une configuration restaure le tool par défaut au robot, enlève les outils aux frames, etc. Certaines de ces affectations sont visibles sur la simulation.

```
...
if (!refSelect())
  taskStop() // erreur
end

// application de la configuration "usinage" si la
// référence sélectionnée n'est pas la référence "demo"
if (refSelected != "demo")
  refApplyConfig(true, "usinage")
end
...
```

refToolName

string **refToolName**(*string* **configName***)

configName*	*string*	nom de la configuration

Retourne le nom du tool défini pour la configuration *configName*. Si la configuration spécifiée n'existe pas, une erreur est générée. Si aucune nom de configuration n'est spécifié et qu'une seule configuration est définie pour la référence sélectionnée, c'est celle-ci qui est considérée. Une erreur est générée dans le cas contraire.

refPaletName

string **refPaletName**(*string* **configName***, *string* **paletType***)

configName*	*string*	nom de la configuration
paletType*	*string*	type de palette

Retourne le nom de la palette définie pour la configuration *configName*, en fonction du type de palette *paletType*.

Si la configuration spécifiée n'existe pas, une erreur est générée. Si aucune nom de configuration n'est spécifié et qu'une seule configuration est définie pour la référence sélectionnée, c'est celle-ci qui est considérée. Une erreur est générée dans le cas contraire.

Si le type de palette n'est pas spécifié et qu'un seul type est défini pour la référence sélectionnée, c'est celui-ci qui est considéré. Une erreur est générée dans le cas contraire.

```
...
toolName = refToolName("usinage")
// deux type de palettes : "charge" et "décharge"
paletName = refPaletName("usinage", "charge")
...
```

refValue

all **refValue**(*string* **paramName**)

paramName	*string*	nom du paramètre de la référence

Retourne la valeur du paramètre *paramName* de la référence sélectionnée.

refSetValue

bool **refSetValue**(*string* **paramName**, *all* **value**)

paramName	*string*	nom du paramètre de la référence
value	*all*	nouvelle valeur du paramètre

Affecte la nouvelle valeur *value* au paramètre *valueName* de la référence sélectionnée.

```
...
method = refValue("calculMethod")
switch(method)
  case "linear"
     refSetValue("delta", a*x)
  case "cubic"
     refSetValue("delta", a*x*x)
end
```

Paramètres internes Ces 2 instructions peuvent également être utilisées pour accéder aux paramètres internes. Il faut spécifier le nom du paramètres interne en utilisant le caractère % comme identifiant (premier caractère). Si le chemin du paramètre interne comporte plusieurs éléments, ils sont également séparés par ce caractère :

%GenericToolCompFixing%configStateName Transformée du composant *fixing* du tool générique. "aConfigName" représente le nom de l'état de configuration voulu.

%GenericToolCompOffset%configStateName Transformée du composant *offset* du tool générique.

5.2 Machine

Les instructions liées au module *machine* permettent d'accéder aux fonctionnalités et aux données liées à la machine.

5.2.1 Device - utilisation des appareils depuis **Deko**

Le module *machine* gère les différents appareils déclarés dans la configuration de **SYNAPXIS**. Afin de pouvoir interagir avec ces appareils, des instructions spécifiques sont publiées pour chaque type (robot, MCP [1]). Grâce aux mécanismes de *sélection* et d'*attachement*, le service machine gère les liens entre les appareils et les différentes tâches en respectant les notions suivantes :

— la configuration peut déclarer plusieurs appareils de type identique (par exemple deux robot A et B).

— pour certaines fonctionnalités, un appareil peut être utilisé depuis plusieurs tâches simultanément. L'appareil doit être *sélectionné* pour la tâche courante.

— pour certaines fonctionnalités, un appareil ne peut être utilisé que depuis une seule tâche exclusivement. L'appareil doit être *attaché* à la tâche courante. Il peut ensuite être *détaché* pour pouvoir être attaché par une autre tâche.

— une tâche ne peut utiliser qu'un seul et unique appareil par type. Par exemple, elle peut utiliser en même temps un robot et un MCP, mais pas plusieurs robots.

— lorsqu'une tâche est créée, elle ne comporte aucun appareil sélectionné ou attaché.

— une tâche ne peut pas attacher plusieurs fois le même appareil.

— lorsqu'une tâche meurt (à la fin de son exécution ou après un *taskStop()*, les appareils attachés sont automatiquement détachés.

— lorsqu'un appareil est *attaché* à une tâche, il est implicitement *sélectionné* pour cette même tâche.

1. Manuel Control Pendant, ou boîtier de commande lié à un robot

deviceSelect

bool **deviceSelect**(*string* **deviceName***)

deviceName*	*string*	nom de l'appareil

Sélectionne l'appareil *deviceName* pour la tâche courante. Si le paramètre *deviceName* n'est pas spécifié et que la configuration de **SYNAPXIS** ne comporte qu'un seul et unique robot, c'est celui-ci qui est considéré. Dans le cas contraire, une erreur est générée. L'instruction retourne *true* en cas de succès.

```
...
deviceSelect() // selection du robot
actualPos = here()
...
```

deviceAttach

bool **deviceAttach**(*string* **deviceName***)

deviceName*	*string*	nom de l'appareil

Attache l'appareil *deviceName* à la tâche courante. Si le paramètre *deviceName* n'est pas spécifié et que la configuration de **SYNAPXIS** ne comporte qu'un seul et unique robot, c'est celui-ci qui est considéré. Dans le cas contraire, une erreur est générée.

L'instruction retourne *true* en cas de succès.

```
...
deviceAttach("A") // attachement du robot
deviceAttach("MCP_A") // attachement du MCP
// message au MCP
if (mcpAlert("Mise en puissance?", "OUI", "NON") == 0)
  power(true) // mise en puissance du robot
end
...
```

deviceDetach

bool **deviceDetach**(*string* **deviceName***)

deviceName*	*string*	nom de l'appareil

Détache l'appareil *deviceName* de la tâche courante. Si le paramètre *deviceName* n'est pas spécifié et que la configuration de **SYNAPXIS** ne comporte qu'un seul et unique robot, c'est celui-ci qui est considéré. Dans le cas contraire, une erreur est générée.

L'instruction retourne *true* en cas de succès.

```
...

// mise en puissance des 2 robots par la même tâche

deviceAttach("A") // attachement du robot A
power(true) // mise en puissance du robot A
deviceDetach("A")// detachement du robot A

deviceAttach("B") // attachement du robot B
power(true) // mise en puissance du robot B
deviceDetach("B")
...
```

deviceSelected

string **deviceSelected**(*string* **deviceKind**, *bool* **deviceAttached**)

deviceKind	*string*	type d'appareil sélectionné ou attaché à la tâche courante : ROBOT, PENDANT
deviceAttached	*bool*	flag indiquant si l'instruction retourne l'appareil sélectionné ou attaché à la tâche courante

Retourne le nom de l'appareil sélectionné à la tâche courante si *deviceAttached* vaut *false*, le nom de l'appareil attaché s'il vaut *true*. Si aucun appareil n'est sélectionné, respectivement attaché à la tâche courante, l'instruction retourne une chaîne de caractère vide.

```
...

// changement de robot attaché pour la tâche courante

currentRobot = deviceAttached("ROBOT", true)
deviceDetach(currentRobot)
if (currentRobot == "A")
  currentRobot = "B"
else
  currentRobot = "A"
end

deviceAttach(currentRobot)
...
```

deviceAttachedTask

string **deviceAttachedTask**(*string* **deviceName**)

deviceName	*string*	nom de l'appareil pour lequel le nom de la tâche à laquelle il est attaché est retourné

Retourne le nom de la tâche à laquelle l'appareil *deviceName* est attaché. Si il n'est attaché à aucune tâche courante, l'instruction retourne une chaîne de caractère vide.

machineEstopRetry

void **machineEstopRetry**(*bool* **retry**)

retry	*bool*	flag indiquant si le processus interrompu est repris ou non

L'instruction permet d'indiquer au système si le processus interrompu par un *arrêt d'urgence* doit être repris ou non au travers du paramètre *retry*. Cette instruction est utilisée uniquement dans l'*événement machine* **<functionEstopOccur>**.

5.2.2 Frames

Le module machine gère les différents frames de **SYNAPXIS**. Ils peuvent être de type plan, cylindrique, palette ou axe externe. Le module machine offre différentes instructions qui permettent d'obtenir les informations liées à ces frames, ainsi que de les modifier (dans le cas notamment de programmes d'apprentissage semi-automatiques).

Chaque frame a un nom unique et est dédié à un robot donné. Lors de l'accès aux donnée du frame à l'aide des instructions décrites ci-dessous, le nom du robot peut être spécifié afin d'éviter des erreurs dans le cas d'application multi-robots. Dans ce cas, il doit correspondre au robot pour lequel est défini le frame. Si celui-ci n'est pas spécifié et que la configuration ne défini qu'un seul et unique robot, c'est celui-ci qui est considéré. Si plusieurs robots sont définis, c'est le robot *attaché* à la tâche qui est considéré.

machineFrame

```
locc machineFrame(string frameName, string robotName*)
```

frameName	*string*	nom du frame
robotName*	*string*	nom du robot lié au frame

Retourne la position cartésienne du frame *frameName*.

machineFrameNames

bool **machineFrameNames**(*string* **frameKind**, *array* **outNames** , *string* **robotName***)

frameKind	*string*	type du frame
outNames	*array*	tableau rempli avec les noms des frames
robotName*	*string*	nom du robot lié au frame

Remplit le tableau *outNames* avec les noms des frames de type *frameKind* : *PLAN, CYLINDRICAL, PALET*.

machineFrameEdit

bool **machineFrameEdit**(*string* **frameName**, *string* **frameKind**, *string* **frameToCopy**, *string* **robotName***)

frameName	*string*	nom du frame
frameKind	*string*	type du frame
frameToCopy	*string*	nom du frame à copier
robotName*	*string*	nom du robot lié au frame

Ouvre l'interface d'édition graphique du frame *frameName*. Celui-ci doit être du type *frameKind*. Si le frame spécifié n'existe pas, il est créé dans le type spécifié. Le nom de ce nouveau frame est édité à l'écran, et sa valeur retournée par la variable *frameName*.

machineFrameDelete

bool **machineFrameDelete**(*string* **frameName**, *string* **frameKind**, *bool* **prompt**)

frameName	*string*	nom du frame à supprimer
frameKind	*string*	type du frame
prompt	*bool*	message de validation

Supprime le frame *frameName*. Celui-ci doit être du type *frameKind*. Si le frame spécifié n'existe pas, une erreur est produite. La variable *prompt* permet de spécifier si un message de validation est affiché ou non.

machinePaletCount

int **machinePaletCount**(*string* **paletName**, *string* **robotName***)

paletName	*string*	nom de la palette
robotName*	*string*	nom du robot lié à la palette

Retourne le nombre d'emplacements de la palette *paletName*.

machinePaletPosition

```
locc machinePaletPosition(string paletName,
                          int positionIndex,
                          string robotName*)
```

paletName	*string*	nom de la palette
positionIndex	*int*	index de l'emplacement de la palette
robotName*	*string*	nom du robot lié à la palette

Retourne la position de l'emplacement *positionIndex* de la palette *paletName*. Pour une palette de capacité n, l'index respecte la règle suivante : $0 \leq index < n$.

```
...
// paletName = "thePalet"
n = machinePaletCount(paletName)
for (i = 0; i < n; i += 1)
  pos = machinePaletPosition(paletName, i)
  ...
end
```

machineGetFrameOffset

```
locc machineGetFrameOffset(string frameName,
                           string robotName*,
                           string offsetName*)
```

frameName	*string*	nom du frame
robotName*	*string*	nom du robot lié au frame
offsetName*	*string*	nom de l'offset

retourne la valeur de décalage du frame *frameName*. Le décalage est sélectionné par le paramètre *offsetName* :

"USER_OFFSET" (valeur par défaut) décalage utilisateur.

"PREVIOUS_OFFSET" décalage *avant* du frame.

"NEXT_OFFSET" décalage *après* du frame.

setMachineFrameOffset

```
bool setMachineFrameOffset(string frameName,
                           locc frameOffset,
                           string robotName*,
                           string offsetName*)
```

frameName	*string*	nom du frame
frameOffset	*locc*	décalage appliqué au frame
robotName*	*string*	nom du robot lié au frame
offsetName*	*string*	nom de l'offset

Affecte la valeur de décalage *offsetName* du frame *frameName*. Le décalage est sélectionné par le paramètre *offsetName* :

"USER_OFFSET" (valeur par défaut) décalage utilisateur : permet de modifier temporairement un frame (par exemple décalage gauche/droite sur frame cylindrique de polissage) ; cet offset est par défaut nul et n'est pas enregistrée lors de la fermeture de l'application.

"PREVIOUS_OFFSET" décalage *avant* du frame : permet de modifier la position d'un frame ; la valeur est enregistrée lors de l'appel de l'instruction.

"NEXT_OFFSET" décalage *après* du frame : permet de modifier la position d'un frame ; la valeur est enregistrée lors de l'appel de l'instruction.

L'instruction retourne *true* en cas de succès.

machineFrameData

```
bool machineFrameData(string frameName,
                      array frameData,
                      string robotName*)
```

frameName	*string*	nom du frame
frameData	*array*	tableau dans lequel sont écrites les données courantes du frame
robotName*	*string*	nom du robot lié au frame

Affecte les données courantes du frame *frameName* dans le tableau *frameData*. Cette instruction permet de connaître toutes les positions définissant le frame afin de les utiliser par exemple dans un programme d'apprentissage/modification semi-automatique du frame. Les données sont retournées dans le tableau selon l'ordre suivant dans le cas d'un *frame plan* :

1. origine du frame
2. direction x du frame
3. direction y du frame
4. position du frame

Pour plus d'information, se référer à l'instruction **frameCompose**.

Dans le cas d'un *frame cylindrique* :

1. plan A
2. plan B
3. plan C
4. périmètre A
5. périmètre B
6. périmètre C

machineSetFrameData

```
bool machineSetFrameData(string frameName,
                         array frameData,
                         string robotName*)
```

frameName	*string*	nom du frame
frameData	*array*	tableau contenant les nouvelles données du frame
robotName*	*string*	nom du robot lié au frame

Modifie le frame *frameName* en fonction des nouvelles valeurs contenues dans le tableau *frameData*. L'organisation des données dans le tableau est identique que dans le cas de l'instruction **machineFrameData**.

machineFrameTransitionPointName

```
string machineFrameTransitionPointName(string frameName,
                                       string robotName*)
```

frameName	*string*	nom du frame
robotName*	*string*	nom du robot lié au frame

Retourne le nom du point de transition associé au frame *frameName*.

5.2.3 Tools

Depuis un programme macro, il est possible de connaître la transformée géométrique d'un tool en fonction de son nom, ainsi que la transformée géométrique d'un composant de tool uniquement.

machineTool

```
locc machineTool(string toolName, string robotName*)
```

toolName	*string*	nom du tool
robotName*	*string*	nom du robot lié au tool

Retourne une position cartésienne correspondant à la transformée géométrique tu tool *toolName*. Les tools sont définis pour un robot donnés. Si le paramètre *robotName* est défini, il doit être identique au nom du robot pour lequel le tool est déclaré. Si le nom n'est pas spécifié et que la configuration ne définit qu'un seul robot, c'est celui-ci qui est considéré. Si plusieurs robot sont définis, c'est le robot *attaché* à la tâche qui est considéré.

machineToolPartTrans

```
locc machineToolPartTrans(string partName)
```

partName	*string*	nom du composant de tool

Retourne une position cartésienne correspondant à la transformée géométrique du composant de tool *partName*.

machineToolSetPartTrans

```
void machineToolSetPartTrans(string partName,
                             locc trans,
                             bool saveData* )
```

partName	*string*	nom du composant de tool
trans	*locc*	transformée géométrique
saveData*	*bool*	enregistrement des données

Affecte la transformée géométrique du composant de tool *partName* avec la position cartésienne *trans*. La nouvelle valeur est enregistrée si *saveData* vaut *true* (valeur par défaut : *false*).

5.2.4 Outils

Dans le cas d'une application *pièce portée*, les outils sont montés sur différents frame. Les instructions ci-dessous permettent d'obtenir et modifier différentes information sur les outils couramment utilisés.

machineOutilNameForFrame

```
string machineOutilNameForFrame(string frameName,
                                string robotName*)
```

partName	*string*	nom du composant de tool
robotName*	*string*	nom du robot lié au frame

Retourne le nom de l'outil couramment monté sur le frame *frameName*. Si aucun outil n'est défini pour le frame, l'instruction retourne une chaîne de caractère vide. Si le frame spécifié n'existe pas, une erreur est produite.

machineOutilRadius

```
real machineOutilRadius(string outilName)
```

outilName	*string*	nom de l'outil

Retourne le rayon de l'outil *outilName*. Si l'outil spécifié n'est pas de type cylindrique, une erreur est produite.

machineSetOutilRadius

```
bool machineSetOutilRadius(string outilName, real radius)
```

outilName	*string*	nom de l'outil
radius	*real*	nouveau rayon de l'outil

Affecte le nouveau rayon *radius* à l'outil *outilName*. Si l'outil spécifié n'est pas de type cylindrique, une erreur est produite. La nouvelle valeur du rayon est enregistrée par l'application. L'instruction retourne *true* en cas de succès.

```
...
// diminution du diamètre de l'outil de 1/10ème
outilName = machineOutilNameForFrame("meulage")
radius = machineOutilRadius(outilName)
machineSetOutilRadius(outilName, radius*0.9)
...
```

machineSetOutilRadiusOffset

```
bool machineSetOutilRadiusOffset(string outilName,
                                 real radiusOffset)
```

outilName	*string*	nom de l'outil
radiusOffset	*real*	offset du rayon

Affecte l'offset *radiusOffset* à l'outil *outilName*. Si l'outil spécifié n'est pas de type cylindrique, une erreur est produite. Cet offset est par défaut nul et n'est pas mémorisé lors de la fermeture de l'application. L'instruction retourne *true* en cas de succès.

machineOutilData

```
int/real machineOutilData(string outilName,
                          string outilData)
```

outilName	*string*	nom de l'outil
outilData	*string*	nom de la donnée

Retourne la donnée de l'outil *outilName* correspondant au sélecteur *outilData*, celui-ci pouvant prendre les valeurs suivantes :

— "NominalDiameter" : diamètre nominal de l'outil cylindrique (*real*)

— "Height" : hauteur de l'outil cylindrique (*real*)

— "UseWithoutOffset" : usure courante, sans tenir compte de l'offset au rayon (*real*)

— "Use" : usure courante totale (*real*)

— "MaxUse" : usure maximale de l'outil (*real*)

— "PartsNumber" : nombre de pièces usinées sur l'outil (*int*)

— "MaxPartsNumber" : nombre maximal de pièce usinées sur l'outil (*int*)

machineOutilApplyPart

```
bool machineOutilApplyPart(int partCount,
                           string configName*,
                           string outilName*)
```

partCount	*int*	nombre de pièce à comptabiliser
configName*	*string*	nom de la configuration
outilName*	*string*	nom de l'outil

Comptabilise le nombre de pièce *partCount*. Si plusieurs configurations existent, le paramètre *configName* doit être défini. Si le paramètre *outilName* est défini, le nombre de pièce est comptabilisé uniquement pour l'outil correspondant. Si ce paramètre n'est pas défini, le nombre de pièce est comptabilisé pour tous les outils utilisés dans la configuration spécifiée.

L'instruction retourne *true* en cas de succès.

machineOutilGetSpeed

```
real machineOutilGetSpeed(string outilName,
                          real nominalSpeed)
```

outilName	*string*	nom de l'outil
nominalSpeed	*real*	vitesse nominale de l'outil

Retourne la vitesse effective pour l'outil *outilName*, en fonction de la vitesse nominale désirée *nominalSpeed*. La vitesse effective est calculée en fonction de l'usure de l'outil (nombre de pièces, diamètre effectif) et la règle de calcul appliquée à l'outil (vitesse tangentielle constante, tables, etc).

machineOutilHasAlarm

```
bool machineOutilHasAlarm(string configName,
                          string outilName*)
```

configName	*string*	nom de la configuration
outilName*	*string*	nom de l'outil

Retourne *true* si au moins un des outils de la configuration *configName* à atteint le niveau d'alarme pour son usure. Si le paramètre *outilName* est spécifié, seul l'outil correspondant est considéré. L'instruction retourne *false* s'il n'y a pas d'alarme.

Lorsqu'un outil est en alarme, il peut être utilisé jusqu'à ce que sa *limite* soit atteinte.

machineOutilLimitReached

```
bool machineOutilLimitReached(string configName,
                              string outilName*)
```

configName	*string*	nom de la configuration
outilName*	*string*	nom de l'outil

Retourne *true* si au moins un des outils de la configuration *configName* a atteint sa limite d'usure. Si le paramètre *outilName*, seul l'outil correspondant est considéré. L'instruction retourne *false* s'il n'y a pas de limite atteinte.

Un outil ne peut plus être utilisé lorsque sa limite d'usure est atteinte, et une erreur est généré lors du prochain décompte de pièces.

machineOutilReset

```
void machineOutilReset(string configName,
                       string outilName)
```

configName	*string*	nom de la configuration
outilName	*string*	nom de l'outil

Remet aux valeur par défaut l'usure, le nombre de pièce, l'usure courante, l'offset usure et l'offset vitesse pour l'outil *outilName*.

machineOutilSelect

```
bool machineOutilSelect(string configName,
                        string unitName,
                        string outilName)
```

configName	*string*	nom de la configuration
unitName	*string*	nom de l'unité
outilName	*string*	nom de l'outil

Sélectionne l'outil *outilName* pour l'unité *unitName*. En pièce portée, l'unité correspond à un *frame* et à un *tool* en pièce fixe.

Les modifications ainsi apportées à la configuration de la référence sont enregistrées à la fermeture de la référence.

machineUnits

```
bool machineUnits(string configName,
                  array unitNames)
```

configName	*string*	nom de la configuration
unitNames	*array*	tableau contenant le nom des unités

Remplit le tableau *unitNames* avec le nom des unités pour la configuration *configName*.

machineSetUnitsToDisplay

```
bool machineSetUnitsToDisplay(string configName,
                              array unitNames)
```

configName	*string*	nom de la configuration
unitNames	*array*	tableau contenant le nom des unités

Spécifie les noms des unités *unitNames* qui doivent être affichés dans l'interface Synapxis pour la configuration *configName*.

5.2.5 Variables machines

Le module machine permet de maintenir des grandeurs qui sont propres à la machine (délais d'attente, longueurs, positions, etc).

machineVar

```
all machineVar(string varName)
```

varName	*string*	nom de la variable machine

Retourne la variable machine correspondant au nom *varName*.

machineSetVar

```
bool machineSetVar(string varName, all value)
```

varName	*string*	nom de la variable machine
value	*all*	nouvelle valeur de la variable machine

Affecte la valeur de *value* à la variable machine *varName*. L'instruction retourne *true* en cas de succès. Le fichier lié aux variables machines (*MachineVariables.dat*) est mis à jour lors de l'exécution de cette instruction, de manière à assurer l'enregistrement de la nouvelle valeur (par exemple en cas de rupture d'alimentation).

Lorsqu'une variable machine de type *array* est modifiée, elle est accédée par *référence* à l'aide de l'instruction **machineVar** ; l'enregistrement de la nouvelle valeur n'est donc pas reporté dans le fichier. Il faut par exemple modifier une autre valeur (*bool*, *int*, ...).

```
...
delay = machineVar("delay_fermeture")
if (edit(delay, "durée de fermeture"))
  machineSetVar("delay_fermeture", delay)
end
...
```

5.2.6 Divers

Le module machine offre d'autre fonctionnalités permettant d'améliorer l'accès à différentes données et fonctionnalités de **SYNAPXIS** depuis un programme.

machineDisplay

```
void machineDisplay(string name)
```

name	*string*	nom de l'interface à afficher

Affiche l'interface correspondant à *name*, celui-ci pouvant prendre les valeurs suivantes :

TeachAndTest Configuration et test de la machine.

Outils Configuration des outils d'usinage.

MachineVariables Edition des variables machines.

IOs Visualisation et modification de l'état des entrées/sorties.

Macros Edition des programmes.

RefCycle Edition du cycle d'usinage.

RefParameters Edition des paramètres de la référence.

5.3 Robot

Les instructions liées au module *robot* permettent d'accéder aux différentes fonctionnalités du robot considéré. En fonction de son type, certaines instructions sont indisponible.

Le robot étant un appareil géré par le module machine, toutes les instructions décrites dans ce qui suit implique que le robot considéré est *sélectionné* ou *attaché* à la tâche courante.

isConnected

bool **isConnected**()

Retourne *true* si la connexion est établie entre **SYNAPXIS** et le contrôleur du robot sélectionné, *false* dans le cas contraire.

appareil sélectionné

```
...
if (!isConnected())
  alert("Robot non connecté, impossible de continuer")
  taskStop()
end
...
```

ensure

```
bool ensure(bool state)
```

state	*bool*	flag indiquant si le robot est assuré ou dé-assuré

Prépare le robot pour une utilisation en mode automatique si *state* vaut *true*. L'instruction contrôle et réalise les conditions suivantes :

— robot connecté

— robot calibré

— robot en mode déporté

— robot en puissance

Si toutes ces conditions peuvent être réalisées, l'instruction retourne *true*, *false*dans le cas contraire.

SYNAPXIS permet de définir une macro qui est appelée automatiquement lorsque l'instruction *ensure* est exécutée. Si celle-ci est définie, elle est appelée au moment de la mise en puissance, et reçoit *state* en paramètre d'entrée (voir chapitre 6).

appareil attaché

power

```
bool power(bool state)
```

state	*bool*	flag indiquant si le robot est mis en puissance ou hors-puissance

Réalise la mise en puissance ou hors-puissance si *state* vaut *true*, respectivement *false*. L'instruction retourne *true* en cas de succès.

appareil attaché

hasPower

bool **hasPower**()

Retourne l'état de la puissance du robot.

appareil sélectionné

remoteMode

bool **remoteMode**()

Retourne *true* si le robot est en mode déporté, *false* dans le cas contraire.

appareil sélectionné

manualMode

bool **manualMode**()

Retourne *true* si le robot est en mode manuel, *false* dans le cas contraire.

appareil sélectionné

here

```
locc here()
```

Retourne la position cartésienne du robot. Le *tool* courant du robot est considéré pour le calcule de la position courante du robot.

appareil sélectionné

herej

```
locj herej()
```

Retourne la position articulaire du robot.

appareil sélectionné

inrangej

```
bool inrangej(locj pos)
```

pos	*locj*	position articulaire

Retourne *true* si la position articulaire *pos* est accessible par le robot, *false* dans le cas contraire.

appareil sélectionné

inrange

```
bool inrange(locc pos, locc tool, locj config)
```

pos	*locc*	position cartésienne
tool	*locc*	tool
config	*locj*	configuration du bras

Retourne *true* si la position cartésienne *pos* est accessible par le robot, *false* dans le cas contraire. Le calcul de l'accessibilité d'une position cartésienne nécessite la connaissance de la transformé géométrique du *tool* ainsi que la configuration mécanique du bras (*lefty/righty, above/below, flip/noflip*). Le paramètre *tool* spécifie le tool considéré, et la position articulaire *config* spécifie la configuration du bras à partir de laquelle la position cartésienne doit être atteinte.

appareil sélectionné

```
pos = ...

// erreur si pos n'est pas accessible avec
// le tool courant et la configuration mécanique courante

if (!inrange(pos, tool(), herej())
  alert("Position inaccessible")
  taskStop()
end
```

solveJointToCartesian

```
locc solveJointToCartesian(locj pos, locc tool)
```

pos	*locj*	position articulaire
tool	*locc*	tool

Retourne la position cartésienne correspondant à la position articulaire *pos* et au tool *tool*.

appareil sélectionné

solveCartesianToJoint

```
bool solveCartesianToJoint(locc pos,
                           locc tool,
                           locj config,
                           locj outSolvedPos)
```

pos	*locc*	position cartésienne
tool	*locc*	tool
config	*locj*	configuration du bras
outSolvedPos	*locj*	position calculée

Calcule la position articulaire correspondant à la position cartésienne *pos*, au tool *tool* et à la configuration du bras *config*. L'instruction retourne *false* si la position ne peut être déterminée ou n'est pas atteignable par le bras. Si la position peut être déterminée, sa valeur est affectée à *outSolvedPos* et l'instruction retourne *true*.

appareil sélectionné

tool

```
locc tool()
```

Retourne une position cartésienne correspondant à la transformée géométrique du tool courant.

appareil sélectionné

setTool

```
bool setTool(locc toolTrans)
```

toolTrans	*locc*	transformée du tool

Affecte le tool courant du robot avec la valeur du paramètre *tool*. L'instruction retourne *true* en cas de succès.

appareil attaché

speedMonitor

```
real speedMonitor()
```

Retourne la vitesse *monitor* courante.

appareil sélectionné

speed

```
real speed()
```

Retourne la vitesse *programme* courante.

appareil sélectionné

speedForOperation

```
real speedForOperation(string operation)
```

operation	*string*	nom de l'opération

Retourne la vitesse correspondant à l'*opération*, celle-ci pouvant prendre les valeurs suivantes :

— "Default"
— "Approach"
— "Depart"
— "Teach"
— "Security"

Ces vitesses sont éditées depuis le *gestionnaire de vitesse* du robot.

appareil sélectionné

setSpeed

```
bool setSpeed(real speed)
```

speed	*real*	nouvelle valeur de la vitesse programme

Affecte la vitesse *programme* courante du robot avec la valeur du paramètre *tool*. L'instruction retourne *true* en cas de succès.

appareil attaché

```
...
setSpeed(speedForOperation("Default"))
move(pos_A)
setSpeed(speedForOperation("Approach"))
move(pos_B)
setSpeed(speedForOperation("Default"))
move(pos_A)
...
```

setSpeedLinear

bool **setSpeedLinear**(*real* **speed**, *real* **speedInMMPS**)

speed	*real*	nouvelle valeur de la vitesse programme
speedInMMPS	*real*	nouvelle valeur de la vitesse linéaire

Affecte la vitesse *linéaire* courante du robot avec la valeur du paramètre *speedInMMPS*. La vitesse programme spécifiée *speed* doit être suffisamment élevée afin de permettre le respect de la vitesse linéaire. L'instruction retourne *true* en cas de succès.

appareil attaché

```
...
setSpeed(speedForOperation("Default"))
move(pos_A)
setSpeedLinear(100, 5.5) // 5.5 mm/s
moves(pos_B) // mouvement linéaire
setSpeed(speedForOperation("Default"))
move(pos_A)
...
```

setAccel

bool **setAccel**(*real* **accel**, *real* **decel***)

accel	*real*	nouvelle valeur de l'accélération
decel*	*real*	nouvelle valeur de la décélération

Affecte l'accélération et la décélération courantes avec les paramètres *accel* et *decel*. Si *decel* n'est pas défini, la valeur de décélération considérée est identique à la valeur d'accélération. L'instruction retourne *true* en cas de succès.

appareil attaché

setBlending

```
bool setBlending(real leave, real reach)
```

leave	*real*	valeur de blending lors du départ du point
reach	*real*	valeur de blending lors du de l'approche du point

Affecte les valeurs de blending courantes avec les paramètres *leave* et *reach*. Le blending, **actif uniquement en mode *désynchronisé***, autorise au robot de ne pas passer exactement par le point spécifié lors d'un mouvement, et de respecter uniquement une distance d'approche *leave* et une distance de départ *leave* entre le point et l'endroit où il quitte ou rejoint la trajectoire théorique. En fonction de la disposition des points, le robot peut de cette manière conserver une vitesse non-nulle lors de l'enchaînement des mouvements. Le blending n'a aucun effet si il n'y a pas au moins 3 mouvements successifs dans la pile de mouvement. L'instruction retourne *true* en cas de succès.

appareil attaché

isOnTransition

```
bool isOnTransition()
```

Retourne *true* si la position articulaire courante correspond à un des *points de transition*, *false* dans le cas contraire.

appareil sélectionné

isAtTransitionPoint

bool **isAtTransitionPoint**(*string* **transitionPointName**)

transitionPointName	*string*	nom du point de transition

Retourne *true* si la position articulaire courante correspond au point *transitionPointName*, *false* dans le cas contraire.

appareil sélectionné

transitionPoint

locj **transitionPoint**(*string* **transitionPointName**)

transitionPointName	*string*	nom du point de transition

Retourne la position articulaire correspondant au point de transition *transitionPointName*.

appareil sélectionné

nearestTransitionPointName

string **nearestTransitionPointName**(*locj* **pos**)

pos	*locj*	position articulaire

Retourne le nom du point de transition le plus proche de la position articulaire *pos*.

appareil sélectionné

transitionMove

bool **transitionMove**(*string* **transitionPointName**)

transitionPointName	*string*	nom du point de transition

Exécute la transition vers le point *transitionPointName*. L'instruction retourne *true* en cas de succès.

appareil attaché

transitionReach

bool **transitionReach**(*string* **transitionPointName**)

transitionPointName	*string*	nom du point de transition

Exécute le mouvement direct (move) depuis la position courante hors transition vers le point de transition *transitionPointName*. L'instruction retourne *true* en cas de succès.

appareil attaché

```
...
if (!isOnTransition())
  nearest = nearestTransitionPointName(herej())
  title = "Robot hors transition"
  text = "Deplacement au point " + nearest + "?"
  if (alert(title, text, "OK", "STOP"))
     taskStop()
  else
     setSpeed(speedForOperation("Security"))
     transitionReach(nearest)
     setSpeed(speedForOperation("Default"))
  end
end
...

transitionMove("...")
...
```

setSynchronizedMove

bool **setSynchronizedMove**(*bool* **state**)

state	*bool*	état du mode synchronisé

Active le mode de mouvement *synchronisé* ou *désynchronisé* en fonction de l'état *true* ou *false* du paramètre *state*. Par défaut, le mode synchronisé est activé, ce qui implique que chaque instruction de mouvement (**movej**, **move**, **moves**, **movec**) attend automatiquement la fin de son exécution par le robot. Le programme macro est ainsi toujours *synchronisé* avec les mouvements du robot.

Certains robots permettant de gérer une pile de mouvement, le mode désynchronisé permet au programme de prendre de l'avance et d'envoyer plusieurs instructions de mouvements dans la pile de mouvement du robot. Celui-ci peut ainsi calculer une suite de mouvement, avec arrêt aux différents points ou non, en fonction du paramètre *break*. Lorsque la pile de mouvement atteint sa taille maximale, les instructions de mouvements deviennent bloquantes, ce qui provoque une erreur d'exécution du programme (timeout). La resynchronisation est faite à l'aide de l'instruction **waitEndMove**.

L'instruction retourne *true* en cas de succès.

appareil attaché

setPendantMode

bool **setPendantMode**(*bool* **state**)

state	*bool*	état du mode pendant

Active et désactive le mode *pendant* pour les mouvements en fonction de l'état *true* ou *false* du paramètre *state*. Le mode pendant implique que le robot soit en mode manuel et que l'utilisateur de valide sur le MCP tous les mouvements.

L'instruction retourne *true* en cas de succès.

appareil attaché

movej

```
bool movej(locj pos, bool break*)
```

pos	*locj*	position articulaire
break*	*bool*	arrêt au point

Exécute un mouvement vers la position articulaire *pos*. En fonction de la position courante du robot, le mouvement est généré de manière à ce que chaque axe réalise sa variation de position en un temps identique. Le paramètre *break* indique si un arrêt doit être marqué au point en mode désynchronisé (**setSynchronizedMove**), et vaut *false* par défaut. L'instruction retourne *true* en cas de succès.

appareil attaché

move

```
bool move(locc pos, real approch*, bool break*)
```

pos	*locc*	position cartésienne
approch*	*real*	décalage selon -z (approche)
break*	*bool*	arrêt au point

Exécute un mouvement vers la position cartésienne *pos*. En fonction de la position courante du robot, le mouvement est généré de manière à ce que chaque axe réalise sa variation de position en un temps identique. La configuration de la position finale est identique à celle de la position initiale. Le paramètre *break* indique si un arrêt doit être marqué au point en mode désynchronisé (**setSynchronizedMove**), et vaut *false* par défaut. L'instruction retourne *true* en cas de succès.

appareil attaché

moves

bool **moves**(*locc* **pos**, *real* **approch***, *bool* **break***)

pos	*locc*	position cartésienne
approch*	*real*	décalage selon -z (approche)
break*	*bool*	arrêt au point

Exécute un mouvement vers la position cartésienne *pos*. En fonction de la position courante du robot, le mouvement est généré de manière à ce que l'extrémité du *tool* réalise une translation rectiligne et une rotation sphérique linéaire (*SLERP*). La configuration de la position finale est identique à celle de la position initiale. Si le paramètre *approch* est spécifié, la position de destination est décalée d'autant selon -z. Par défaut, ce paramètre vaut 0. Le paramètre *break* indique si un arrêt doit être marqué au point en mode désynchronisé (**setSynchronizedMove**), et vaut *false* par défaut. L'instruction retourne *true* en cas de succès.

appareil attaché

movec

bool **movec**(*locc* **intermediatePos**, *locc* **endPos**, *bool* **break***)

intermediatePos	*locc*	position intermédiaire
endPos	*locc*	position finale
break*	*bool*	arrêt au point

Exécute un mouvement circulaire vers la position cartésienne *endPos*, et passant par *intermediatePos*. En fonction de la position courante du robot, le mouvement est généré de manière à ce que l'extrémité du *tool* réalise un arc de cercle. La configuration de la position finale est identique à celle de la position initiale. Le paramètre *break* indique si un arrêt doit être marqué au point en mode désynchronisé (**setSynchronizedMove**), et vaut *false* par défaut. L'instruction retourne *true* en cas de succès.

appareil attaché

waitEndMove

bool **waitEndMove**()

Effectue une resynchronisation du programme macro avec les mouvement du robot en attendant que la pile de mouvement soit vide et que le robot soit en position stable. Cette instruction est utilisée en mode *désynchronisé* uniquement. L'instruction retourne *true* en cas de succès.

appareil attaché

```
...
deviceAttach()
if (!ensure(true)
  taskStop()
end

// mouvements en mode synchronisé : chaque mouvement
// provoque une attente et un arrêt au point
transitionMove("START")
movej(posj1) // position joint
move(pos1, 20) // position cartésienne, approche
moves(pos1) // déplacement linéaire
movec(pos2, pos3) // déplacement circulaire
moves(pos3, 20) // dégagement de 20 mm

// mouvements en mode desynchronisé avec blending = 5 mm
// depuis le point courant, le robot passe en mode
// blending les points A, B, C et s'arrête au point D

setBlending(5, 5) // leave et reach == 5 mm
setSynchronizedMove(false) // mode désynchronisé
move(pos_A, 0, false)// pas d'arrêt au point
move(pos_B, 0, false)// pas d'arrêt au point
move(pos_C)// arrêt au point
move(pos_D, 0, false)// pas d'arrêt au point
move(pos_E, 0, false)
waitEndMove()// arrêt au point E
setSynchronizedMove(true) // mode synchronisé
...
```

moveStop

```
bool moveStop()
```

Les instructions *moveStop*, *moveReset* et *moveRestart* permettent de piloter l'exécution de la pile des mouvements envoyés au robot. Par définition, elles doivent être utilisées depuis une tâche Synapxis différente que celle qui envoie les mouvements au robot :

- la tâche (A) qui envoie les commande de mouvement doit avoir le robot attaché.
- cette tâche (A), en mode synchronisé ou désynchronisé, envoie les commande de mouvement, et attend aux endroits voulu la fin d'un mouvement.
- La tâche (B) qui veut suspendre l'exécution des mouvement du robot doit avoir le robot sélectionné.
  - *moveStop* permet d'interrompre l'exécution du mouvement en cours (pause). La pile des mouvements enregistrés est inchangé.
  - *moveRestart* permet de reprendre l'exécution du mouvement interrompu, puis l'exécution des autres mouvement de la pile.
  - *moveReset* annule le mouvement en cours et vide la pile des mouvements.

Le mode de marche (manuel/déporté) ainsi que la puissance doivent être gérés de manière à ce que le mouvement puisse reprendre au moment de l'exécution du moveRestart. Ces instructions de pilotage de la pile de mouvement ne fonctionnent qu'en mode réel, et provoquent un erreur en mode simulé.

appareil sélectionné

moveReset

```
bool moveReset()
```

Vide la pile de mouvement du robot. L'instruction retourne *true* en cas de succès.

appareil sélectionné

moveRestart

```
bool moveRestart()
```

Provoque la reprise de la séquence de mouvement.

appareil sélectionné

resetTurn

```
void resetTurn()
```

Réinitialise la position angulaire du dernier axe (6ème axe pour un robot anthropomorphe) à la valeur correspondante comprise dans le premier tour, en degrés : ($-180 < angle \leq 180$).

appareil attaché

reactiReset

```
bool reactiReset(int ioIndex)
```

ioIndex	*int*	numéro de l'entrée

Active le mécanisme d'interruption de mouvement par I/O. L'instruction retourne *true* en cas de succès.

Pendant l'exécution d'un mouvement, l'entrée du robot *ioIndex* est scannée, et si elle passe à l'état 1, le mouvement en cours est interrompu. L'instruction de mouvement correspondante, en mode *synchronisé*, se termine. L'instruction **reactiOccur** permet de déterminer après l'instruction de mouvement si celui-ci a été interrompu.

appareil attaché

reactiOccur

bool **reactiOccur**()

Retourne *true* si le mécanisme d'interruption de mouvement par I/O est activé pour une entrée et que celle-ci est passé à l'état 1 pendant l'exécution d'un mouvement, *false* dans le cas contraire.

appareil attaché

```
...
deviceAttach()
if (!ensure(true)
  taskStop()
end
...
move(pos, 100) // approche de la position
setSpeed(5)// vitesse lente

// activation du reacti pour detection collision if
(!reactiReset(0))
  alert("Impossible d'activer le reacti sur l'entree 0")
  taskStop()
end

do
  moves(pos, 0)
  if (reactiOccur())
     if (alert("Collision", "Réessayer?", "OK", "STOP"))
       taskStop()
     else
       moves(pos, 100) // approche de la position
       reactiReset(0)
     end
  else
     exit
   end
until(false)
...
```

5.4 MCP

Le module *Pendant* donne la possibilité d'interagir avec l'utilisateur via l'interface du MCP.

mcpAlert

```
int mcpAlert(string text,
             string button0*,
             string button1*,
             ... )
```

text	*string*	texte du message
button0*	*string*	texte du 1er bouton
button1*	*string*	texte du 2eme bouton

Affiche le message *text* sur l'écran du MCP. Les paramètres *button0, button1, ..., buttonN* permettent de spécifier le texte des boutons à afficher. Si aucun bouton n'est spécifié, un bouton *OK* est créé par défaut. L'instruction retourne l'index du bouton sélectionné par l'opérateur : $0 \leq index \leq (n-1)$.

appareil sélectionné

```
...
switch(mcpAlert("Message", "OK", "Annuler", "Info"))
  ...
end
```

mcpAlertN

int **mcpAlertN**(*string* **text**, *array* **buttons**)

text	*string*	texte du message
buttons	*array*	texte des boutons

Affiche le message *text* sur l'écran du MCP. Le tableau *buttons* contient les textes des boutons à afficher avec le message. Si aucun bouton n'est spécifié, un bouton *OK* est créé par défaut. L'instruction retourne l'index du bouton sélectionné par l'opérateur : $0 \leq index \leq (n-1)$.

appareil sélectionné

```
...
arrayAdd(buttons, "Annuler")
arrayAdd(buttons, "5 mm")
arrayAdd(buttons, "10 mm")
arrayAdd(buttons, "15 mm")
arrayAdd(buttons, "30 mm")
arrayAdd(buttons, "50 mm")
switch(mcpAlert("Message", buttons))
  ...
end
```

5.5 IOs

Les entrées/sorties de la machines peuvent être lues et affectées depuis un programme macro.

Les instructions *ioRead, ioWrite, ioToggle* sont sensibles au mode activé ou désactivé de la simulation de **SYNAPXIS**. Aucun accès à la couche matériel n'est réalisé lorsque la simulation est activée ; les instructions d'écriture n'ont aucune action, et l'instruction de lecture retourne une valeur par défaut (*false*ou 0). Les instructions *ioReadF, ioWriteF, ioToggleF* forcent l'accès à la couche matériel, même lorsque la simulation est activée. Cela permet par exemple à une tâche de surveillance d'être active en tout temps. Ces instructions observent la même syntaxe que les 3 instructions décrites ci-dessous.

ioRead

bool/real **ioRead**(*string* **ioName**)

ioName	*string*	nom de l'entrée/sortie

Retourne la valeur de l'entrée/sortie *ioName*. Le type de la valeur retournée correspond à celui de l'entrée/sortie considérée : *bool* pour une e/s binaire, *real*dans le cas d'une e/s analogique.

```
...
// lecture de la valeur d'un laser de mesure
// si le capteur n'est pas en erreur
if (ioRead("laserInrange"))
  laserDist = ioRead("laserDistance")
else
  laserDist = -1
end
...
```

ioWrite

```
bool ioWrite(string ioName, bool/real value)
```

ioName	*string*	nom de la sortie
value	*bool/real*	nouvelle valeur de la sortie

Affecte la sortie *ioName* avec la valeur de *value*. Le type de la valeur doit correspondre au type de la sortie : . L'instruction retourne *true* en cas de succès.

ioToggle

```
bool ioToggle(string ioName)
```

ioName	*string*	nom de la sortie

Inverse l'état de la sortie binaire *ioName*. L'instruction retourne *true* en cas de succès.

5.6 Production

Le module *Production* donne la possibilité d'interagir l'interface de production depuis les programmes.

prodParam

```
all prodParam(string paramName)
```

paramName	*string*	nom du paramètre de production

Retourne la valeur du paramètre de production *paramName*.

prodSetParam

```
bool prodSetParam(string paramName, all value)
```

paramName	*string*	nom du paramètre de production
value	*all*	nouvelle valeur du paramètre de production

Affecte le paramètre de production *paramName* avec la valeur *value*. L'instruction retourne *true* en cas de succès.

prodSetInfo

```
bool prodSetInfo(string infoName, all value)
```

infoName	*string*	nom de l'information de production
value	*all*	nouvelle valeur de l'information de production

Affecte l'information de production *paramName* avec la valeur *value*. L'instruction retourne *true* en cas de succès.

prodSetOperation

```
void prodSetOperation(string value)
```

value	*string*	opération de production courante

Affecte l'information de l'opération de production courante.

prodBatchCount

```
int prodBatchCount()
```

Retourne le nombre de lots de production présents dans la pile des *lots en attente*.

prodBatchPop

```
bool prodBatchPop()
```

Sélectionne le premier lot de production en attente, celui-ci devenant le *lot de production courant* ; il est sorti de la pile des lots en attente et l'instruction retourne *true*. Si la pile des lots en attente est vide, l'instruction retourne *false*. L'ancien lot courant est mis dans la pile des *lots produits*.

prodBatchClear

```
bool prodBatchClear()
```

Efface le lot courant. Celui-ci n'est pas mis dans la pile des *lots produits*. L'instruction retourne *true* en cas de succès.

prodBatchID

```
int prodBatchID()
```

Retourne l'ID du lot courant. S'il n'y aucun lot courant, l'instruction retourne -1.

prodBatchRefName

```
string prodBatchRefName()
```

Retourne le nom de la *référence* du lot courant.

prodBatchWaitingRefNames

void **prodBatchWaitingRefNames**(*array* **outRefNames**)

Rempli le tableau passé en paramètre avec les noms des *références* des lots chargés avec l'instruction prodBatchAdd()

prodBatchPartCount

int **prodBatchPartCount**()

Retourne le nombre de pièces du lot courant.

prodBatchPartState

string **prodBatchPartState**(*int* **partIndex**)

partIndex	*int*	index de la pièce du lot de production

Retourne le nom de l'état de la pièce *partIndex* du lot courant : $0 \leq partIndex \leq (n-1)$. Lorsque la production utilise une palette, un état associé à une couleur peut être spécifié pour chaque pièce, et peut consulté et modifié au cours du cycle de production depuis le programme.

prodBatchSetPartState

bool **prodBatchSetPartState**(*int* **partIndex**, *string* **stateName**)

partIndex	*int*	index de la pièce du lot de production
stateName	*string*	nouvel état de la pièce

Affecte la pièce *partIndex* du lot courant avec l'état *stateName*. L'instruction retourne *true* en cas de succès.

prodCycleGroupData

bool **prodCycleGroupData**(*array* **outData**)

outData	*array*	tableau contenant les données des groupes d'usinage sélectionnés pour la production

Copie les données des groupes d'usinage sélectionnés pour la production dans le tableau *outData*. Celui-ci est initialement vidé, puis rempli selon 2 dimensions : la première dimension accède au type de donnée (index = 0 pour le nom du groupe, et ensuite les données configurées dans l'interface de production) et la deuxième dimension pour l'index du groupe. L'instruction retourne *true* en cas de succès.

```
...
while(prodBatchPop())
  refName = prodBatchRefName()
  refSelect(refName)
  paletName = refPaletName()
  partCount = prodBatchPartCount()
  for (i = 0; i < partCount; i+=1)
     pos = machinePaletPosition(paletName, i)
     switch(prodBatchPartState(partIndex))
       case "brut"
         getPiece(pos) // fonction de chargement
         processPiece() // fonction du processus
         putPiece(pos) // fonction de déchargement
         prodBatchSetPartState(i, "processed")
       ...
     end
  end
end
...
```

5.7 Trajectoire

Le module *Trajectoire* donne accès aux fonctionnalités nécessaire à l'exécution les trajectoires.

trajEnable

```
bool trajEnable(bool state)
```

state	*bool*	état activé ou désactivé

Active et désactive le mode *trajectoire* en fonction de l'état *true* ou *false* du flag *state*. Le mode trajectoire doit être activé pour pouvoir utiliser les instructions suivantes. L'instruction retourne *true* en cas de succès.

trajEditedRefName

```
string trajEditedRefName()
```

L'instruction retourne le nom de la référence couramment éditée dans l'interface de réglage des trajectoires.

trajRefCycleLoad

string **trajRefCycleLoad**(*string* **refName**)

refName	*string*	nom de la référence

Charge en mémoire le *cycle d'usinage* de la référence *refName* et retourne le nom du cycle d'usinage.

trajCycleGroupNames

bool **trajCycleGroupNames**(*string* **cycleName**, *array* **outNames**)

cycleName	*string*	nom du cycle
outNames	*array*	tableau contenant les noms de groupes

Remplit le tableau *outNames* avec les nom des groupes *activés* du cycle d'usinage *cycleName*. L'instruction retourne *true* en cas de succès.

trajCyclePrepare

bool **trajCyclePrepare**(*string* **cycleName**, *array* **groupNames***)

cycleName	*string*	nom du cycle d'usinage
groupNames*	*array*	noms des groupes à préparer

Prépare l'exécution des trajectoires pour les groupes du cycle *cycleName* spécifiés dans le tableau *groupNames*. Si le tableau n'est pas défini, tous les groupes actifs du cycle sont considérés. L'instruction retourne *true* en cas de succès.

trajCycleBegin

bool **trajCycleBegin**(*string* **cycleName**, *string* **groupName**)

cycleName	*string*	nom du cycle d'usinage
groupName	*string*	nom du groupe à exécuter

Sélectionne le groupe *groupName* du cycle *cycleName* pour l'exécution. L'instruction retourne *true* en cas succès.

trajCycleRun

bool **trajCycleRun**()

Exécute les différentes étapes du groupe sélectionné. Cette instruction doit être appelé en boucle en combinaison avec l'instruction de fin d'exécution **trajCycleEnd**. L'instruction retourne *true* en cas de succès.

trajCycleEnd

bool **trajCycleEnd**()

Retourne *false* tant que la fin de l'exécution du groupe sélectionné n'est pas terminée.

```
...
trajEnable(true)

while(prodBatchPop())
  refName = prodBatchRefName()
  refSelect(refName)
  paletName = refPaletName()
  partCount = prodBatchPartCount()
  cycleName = trajRefCycleLoad(refName)
  trajCyclePrepare(cycleName) // tous les groupes
  // usinage des pièces
  for (i = 0; i < partCount; i+=1)
    prodCycleGroupData(groupData)

    ... // exécution des goupes de trajectoire
    for (gp = 0; gp < arraySize(groupData[0]); gp += 1)
      groupName = groupData[0][gp]
      tracjCycleBegin(cycleName, groupName)
      while(!trajCycleEnd())
        trajCycleRun()
      end
    end
    ...
  end
end
...
trajEnable(false)
```

trajTryCurrentLinearPosition

```
real trajTryCurrentLinearPosition()
```

Retourne la position linéaire courante sur la trajectoire. Cette instruction est utilisé en combinaison avec l'instruction *trajTryPositionAtLinearPosition* en mode *réglage pendant* uniquement.

trajTryPositionAtLinearPosition

```
bool trajTryPositionAtLinearPosition(real linearPosition,
                                     locc outFramePosition,
                                     locc outPartPosition)
```

linearPosition	*real*	position linéaire sur la trajectoire
outFramePosition	*locc*	position du point de contact
outPartPosition	*locc*	position de la pièce par rapport au point de contact

Calcul la position cartésienne de la trajectoire courante à la position linéaire *linearPosition*. La position cartésienne est retournée en 2 parties : *outFramePosition* et *outPartPosition*, représentant respectivement la position du point de contact par rapport au robot et la position de la pièce par rapport au point de contact. Cette instruction n'est utilisable qu'en mode *réglage pendant*, et retourne *true* en cas de succès.

trajTryPositionAtLinearPositionForGST

```
bool trajTryPositionAtLinearPositionForGST(string groupName,
                                           int stepIndex,
                                           string tryName,
                                           real linearPosition,
                                           locc outFramePosition,
                                           locc outPartPosition)
```

groupName	*string*	nom du groupe
stepIndex	*int*	index du step trajectoire
tryName	*string*	nom du try
linearPosition	*real*	position linéaire sur la trajectoire
outFramePosition	*locc*	position du point de contact
outPartPosition	*locc*	position de la pièce par rapport au point de contact

Calcul la position cartésienne de la trajectoire correspondant au *groupName*, *stepIndex* et *tryName*, à la position linéaire *linearPosition*. La position cartésienne est retournée en 2 parties : *outFramePosition* et *outPartPosition*, représentant respectivement la position du point de contact par rapport au robot et la position de la pièce par rapport au point de contact. Un cycle doit être couramment sélectionné. L'instruction retourne *true* en cas de succès.

```
...
trajEnable(true)
cycleName = trajRefCycleLoad("ref")
trajCyclePrepare(cycleName)

// calcul de la position pour le groupe "Groupe 1",
// le premier step et le try "Try 1"
trajTryPositionAtLinearPositionForGST("Groupe 1", 0, "Try
1", 0, fPos, pPos)
...
```

trajTrajectoryPoints

```
void trajTrajectoryPoints(string cycleName,
                          string trajectoryName,
                          array outPoints)
```

cycleName	*string*	nom du cycle d'usinage
trajectoryName	*string*	nom de la trajectoire
outPoints	*array*	tableau contenant les points

Copie les points de la trajectoire *trajectoryName* dans le tableau *outPoints*.

trajTrajectorySetPoint

```
void trajTrajectorySetPoint(string cycleName,
                            string trajectoryName,
                            int pointIndex,
                            locc position)
```

cycleName	*string*	nom du cycle d'usinage
trajectoryName	*string*	nom de la trajectoire
pointIndex	*int*	index du points modifié
position	*locc*	nouvelle position du point

Modifie le point *pointIndex* de la trajectoire *trajectoryName*.

trajTrajectorySetPoints

```
void trajTrajectorySetPoints(string cycleName,
                             string trajectoryName,
                             array points)
```

cycleName	*string*	nom du cycle d'usinage
trajectoryName	*string*	nom de la trajectoire
points	*array*	tableau de *locc*

Modifie la trajectoire *trajectoryName* avec les nouveaux points *points*. Le tableau *points* doit contenir au moins deux *locc*. Cette instruction met à jour les steps utilisant la trajectoire *trajectoryName*.

trajTryEditOperation

```
bool trajTryEditOperation(string cycleName,
                          string groupName,
                          int stepIndex,
                          string tryName,
                          int operation,
                          all p1,
                          all p2*,
                          all p3*)
```

cycleName	*string*	nom du cycle
groupName	*string*	nom du groupe
stepIndex	*int*	index du step
tryName	*string*	nom du try
operation	*int*	operation
p1	*all*	p1
p2*	*all*	p2
p3*	*all*	p3

Exécute les opérations suivantes sur le try spécifié, en fonction du paramètre *operation* :

1 retourne les informations du try courant : position linéaire de départ (*p1*), position linéaire de fin (*p2*) et durée du try (*p3*). L'instruction retourne *true* si la configuration du try est correcte, *false* dans le cas contraire.

10 retourne les valeurs d'abscisses (*p2*) et d'ordonnées (*p3*) des ancres du paramètre *p1*. L'ordonnée correspond à la position linéaire.

11 retourne les valeurs d'abscisses *p2* et d'ordonnées (*p3*) des ancres du paramètre (*p1*). L'ordonnée correspond au temps.

12 créer les ancres du paramètre *p1* avec les valeurs d'abscisses (*p2*) et d'ordonnées (*p3*). L'ordonnée correspond à la position linéaire.

13 créer les ancres du paramètre *p1* avec les valeurs d'abscisses (*p2*) et d'ordonnées (*p3*). L'ordonnée correspond au temps.

En mode édition, c'est le try courant qui est considéré. Le nom du paramètre doit contenir la famille (*SPEED*, *TRAJ*, *TOOL*, *FRAME*) et le nom de la coordonnée (*DX*, *DY*, *DZ*, *RX*, *RY*, *RZ*, *ELEVATION*).

5.8 Sécurité

Le module *Sécurité* donne la possibilité de connaître et modifier le niveau d'accès courant de **SYNAPXIS**.

accessCheckLevel

bool **accessCheckLevel**(*string* **levelName**)

levelName	*string*	nom du niveau

Retourne *true* si le nom du niveau d'accès courant correspond à *levelName*, *false* dans le cas contraire.

```
...
if (simEnabled())
  print("IO" ,ioName, state)
else
  ioWrite(ioName, state)
end
...
```

accessSetLevel

bool **accessSetLevel**(*string* **levelName**)

levelName	*string*	nom du niveau

Sélectionne le niveau d'accès *levelName* en tant que niveau d'accès courant. L'instruction retourne *true* en cas de succès.

5.9 Simulation

Le module *Simulation* donne la possibilité d'interagir avec la simulation 3D.

simEnabled

```
bool simEnabled()
```

Retourne *true* si la simulation set activée, *false* dans le cas contraire. Si la simulation est activée, c'est vers les robots et MCP virtuels que sont redirigées les instructions liées au robots et au MCP.

simEnable

```
bool simEnable(bool state)
```

state	*bool*	état activé ou désactivé de la simulation

Active ou désactive la simulation en fonction de la valeur *true* ou *false* de *state*. L'instruction retourne *true* en cas de succès.

```
...
if (simEnabled())
  print("IO" ,ioName, state)
else
  ioWrite(ioName, state)
end
...
```

simAsmClear

```
void simAsmClear(string assembly)
```

assembly	*string*	chemin de l'assemblage

Supprime tous les composants de l'assemblage *assembly*.

simAsmObjectAdd

```
void simAsmObjectAdd(string assembly,
                     string libObjectPath,
                     string objectName,
                     locc position)
```

assembly	*string*	chemin de l'assemblage
libObjectPath	*string*	objet de la librairie
objectName	*string*	nom de l'objet dans l'assemblage
position	*locc*	position de l'objet dans l'assemblage

Ajout l'objet de la librairie *libObjectPath* à l'assemblage *assembly*. L'objet prend le nom *objectName* dans cet assemblage, et sa *position* est relative au frame de l'assemblage.

```
...
partName = "Jeton_"
libPath = "Internal\Parts\Jeton_A"
simAsmClear(asm)

...
for (i = 0; i < n; i+=1)
  simAsmObjectAdd(asm, libPath, partName+i, trans(10*i))
end
...
```

simObjectSetLink

```
bool simObjectSetLink(string childObject,
                      string parentObject,
                      bool state)
```

childObject	*string*	chemin de l'objet enfant
parentObject	*string*	chemin de l'objet parent
state	*bool*	état du lien entre l'objet enfant et l'objet parent

Active ou désactive le lien entre l'objet parent et l'objet enfant. Un objet dont le lien avec son parent est rompu devient positionné de manière *absolue*. Un objet ne peut être lié relativement à un parent que s'il est positionné de manière absolue. Lors d'un changement de lien, les position absolues des 2 objets ne varient pas. L'instruction retourne *true* en cas de succès.

```
...
simObjectSetLink("Cellule/Piece", "Cellule/Palette", false)
simObjectSetLink("Cellule/Piece", "Cellule/Pince", true)
...
```

simObjectPosition

```
locc simObjectPosition(string childObject,
                       string parentObject)
```

childObject	*string*	chemin de l'objet enfant
parentObject	*string*	chemin de l'objet parent

Retourne la position relative de l'objet enfant *childObject* par rapport à l'objet parent *parentObject*.

simObjectSetPosition

```
bool simObjectSetPosition(string childObject,
                          string parentObject,
                          locc position)
```

childObject	*string*	chemin de l'objet enfant
parentObject	*string*	chemin de l'objet parent
position	*locc*	nouvel position relative

Affecte la position relative de l'objet enfant par rapport à l'objet parent avec la valeur *position*. L'instruction retourne *true* en cas de succès.

```
...
parent = "Cellule/Piece"
enfant = "Cellule/Piece"
deltaRZ = 0.5
for (angleZ = 0; angleZ <= 360; angleZ += deltaRZ)
  pos = simObjectPosition(enfant, parent)
  pos += trans(0, 0, 0, 0, 0, deltaRZ)
  pos = simObjectSetPosition(enfant, parent, pos)
end
...
```

simObjectLibModify

```
void simObjectLibModify(string libObjectPath,
                        string property,
                        all value)
```

libObjectPath	*string*	chemin de l'objet dans la librairie
property	*string*	nom de la propriété à modifier
value	*all*	nouvel valeur de la propriété

Modifie la valeur d'une propriété d'un objet de la librairie *libObjectPath*. A l'heure actuelle, seule la propriété *property* = "TRANSFORMATION" est supportée, pour modifier la transformée d'un composant *Tool Simple*.

```
libPath = "Internal/comp/obj"
simObjectLibModify(libPath, "transformation", trsf)
```

simAddPointGroup

```
void simAddPointGroup(string name,
                      string assembly,
                      array points,
                      locc parentPosition,
                      bool showParent*,
                      bool showTrace*,
                      bool showNames*,
                      real markSize*,
                      string style*)
```

name	*string*	nom du groupe de point
assembly	*string*	chemin de l'assemblage
points	*array*	tableau contenant les positions (*locc*) des points
parentPosition	*locc*	position du groupe de points
showParent*	*bool*	affichage du repère parent
showTrace*	*bool*	affichage de la trace des points
showNames*	*bool*	affichage du nom des points
markSize*	*real*	rayon de la sphère de représentation des points
markSize*	*string*	style de dessins des points

Affiche un groupe de points dans l'assemblage *assembly*. Le groupe peut être constitué d'un seul ou plusieurs points. Les positions du tableau *points* sont définis par une variable cartésienne, et sont représenté par une sphère ainsi qu'un système d'axe orthonormé (X, Y, Z = Red, Green, Blue). La position du groupe de points est spécifiée par le paramètre *parentPosition*. L'affichage d'un repère parent à cette position est spécifié par *showParent*. Le paramètre *showTrace* permet d'afficher un traçage des coordonnées X-Y-Z afin d'améliorer la visibilité de la position des points par rapport au repère parent. *markSize* permet de spécifier le rayon de la sphère [mm] qui est dessinée à la position du point. *showNames* permet d'afficher le nom de chaque point. S'il y a plus d'un point dans le groupe, le nom vaut *name[i]*. Le paramètre *style* permet de spécifier la représentation des points :

"SPHERES" sphères et système d'axes (représentation par défaut)

"POINTS-RED" points simples rouges

"LINES-GREEN" suite de lignes vertes

"TILES-BLUE" tuiles bleues

```
// affichage d'un groupe de point généré aléatoirement

asmName = "Cellule/ASM_Frame"
n = 10
box = 100
t = 0

for (i = 0; i < n; i+=1)
    arrayAdd(points, trans(random()*box, random()*box, random()*box))
end

pos = trans(200, -800, 50, 0, 0, 0) // position parent
simAddPointGroup("rand", asmName, points, pos, true, true, true, 5)
```

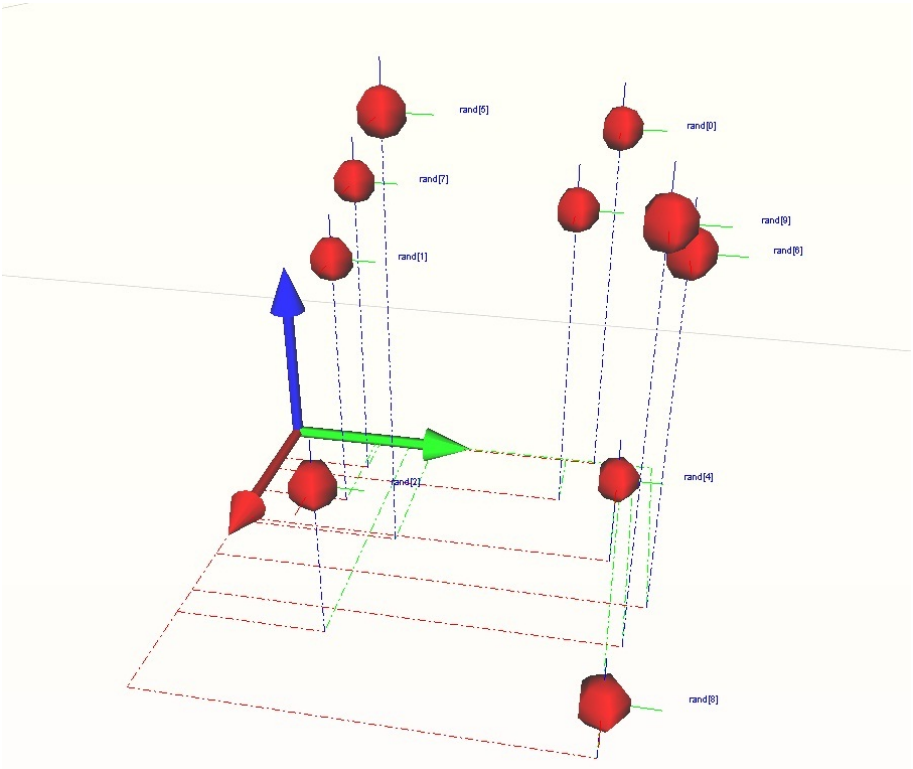

Figure 5.1: Exemple de représentation d'un groupe de points

simEditPoints

```
bool simEditPoints(string object,
                   array points,
                   string title,
                   int maxPointsCount*)
```

object	*string*	chemin de l'objet sur lequel les points doivent être sélectionnés
points	*array*	tableau de points
title	*string*	titre du dialogue
maxPointsCount*	*int*	nombre maximal de points à sélectionner

Permet d'éditer une liste de points, et de sélectionner leur position via la vue 3D. Le paramètre *object* spécifie le nom de le fichier 3D de l'objet sur lequel la sélection est faite (il peut ne pas être défini - string vide - et le système identifie l'objet à la sélection). Le tableau *points* contient les points sélectionnés. Le titre du dialogue est défini par *title*. Le paramètres *maxPointsCount* permet de fixer une limite aux nombre de sélections.

```
// sélections de points sur la pièce fixée sur le robot

deviceAttach()
setTool(machineTool("joint_chuck"))
h = here()

simEditPoints("", pts, "Select points")

// calcul de la position relative par rapport au tool
n = arraySize(pts)
for (i = 0; i < n; i+=1)
    arrayAdd(pts2, deltaTo(h, pts[i]))
end

// représentation des points sélectionnés
simAddPointGroup("Points", "CyberPolish/ASM_Tool", pts2, tool())
```

simAxisControllerSetPosition

```
void simAxisControllerSetPosition(string axisController,
                                  locj position)
```

axisController	*string*	nom du contrôleur d'axes
position	*locj*	nouvelle position articulaire du contrôleur d'axes

Affecte la position articulaire courante du contrôleur d'axes *axisController* avec la position articulaire *position*. L'instruction retourne *true* en cas de succès.

simViewSelect

```
void simViewSelect(string view)
```

view	*string*	nom de la vue

Sélectionne la vue *view* pour la représentation 3D.

simViewApply

```
void simViewApply(locc position)
```

view	*locc*	position de l'origine de la vue par rapport à l'écran

Applique la *position* à la représentation 3D.

CHAPITRE 6

Les événements macros

Les *événements* permettent aux différents modules **SYNAPXIS** d'appeler une fonction afin de réaliser une tâche paramétrable en fonction de l'application. Si le nom de la fonction liée à un événement n'est pas imposé, ses paramètres et le type de retour doivent respecter les déclarations ci-dessous.

6.1 Événements Production

Le module *Production* publie 3 événements liés au démarrage, mise en pause et arrêt de la production.

<functionProdStart>

void **<functionProdStart>**()

Fonction appelée lors du lancement de la production.

```
prodTaskName = "prodTask"
if (taskExists(prodTaskName))
  if (taskPaused(prodTaskName))
    taskResume(prodTaskName)
  end
else
  if (alert("Demarrage du cycle?", "", "OK", "NON") == 1)
    return
  end
  taskExecute prodCycleContinu(), prodTaskName
end
```

<functionProdPause>

void **<functionProdPause>**()

Fonction appelée lors de la mise en pause de la production.

<functionProdStop>

void **<functionProdStop>**()

Fonction appelée lors de l'arrêt de la production.

```
prodTaskName = "prodTask"
hit = alert("Arret du cycle de production", "",
"Programme", "Immediat", "Annuler")
if (hit == 2)
  return
end
if (hit == 1)
  taskStop(prodTaskName)
  return
end

hit = alert("Arret du cycle", "", "Fin de piece", "Fin de
groupe", "Annuler")
if (hit == 2)
  return
end

// variable globales pour pilotage
// de la tâche de production
if (hit == 1)
  gProdStopGroup = true
end
if (hit == 0)
  gProdStopPiece = true
end
```

6.2 Événements Machine

Le module *Machine* publie 5 événements liés au démarrage/fermeture de l'application et au *estop*.

<functionStartup>

void **<functionStartup>**()

Fonction appelée au démarrage de **SYNAPXIS**, après l'initialisation de tous les modules.

<functionShutdown>

void **<functionShutdown>**()

Fonction appelée à la fermeture de **SYNAPXIS**.

<functionInterfaceOpenClose>

```
void <functionInterfaceOpenClose>(string interfaceName,
                                  bool openOrClose)
```

interfaceName	*string*	nom de l'interface
openOrClose	*bool*	flag indiquant si l'interface est ouvert ou fermé

La fonction est appelée lors de l'ouverture de certaine fenêtre d'interface de **SYNAPXIS**. Le paramètre *interfaceName* permet d'identifier l'interface en question, et le paramètre *openOrClose* indique s'il s'agit d'une ouverture (*true*) ou d'une fermeture (*false*). Le paramètre *interfaceName* peut prendre les valeurs suivantes :

Production Interface de production.

RefCycle Interface de réglage du cycle de la référence.

RefParameters

Outils Interface de gestion des outils.

MachineVariables Éditeur des variables machine.

TeachAndTest Interface d'apprentissage des références.

Macros Éditeur de programmes.

IOs Interface de gestion des entrées/sorties.

<functionEstopOccur>

```
void <functionEstopOccur>(int reason)
```

reason	*int*	raison du déclenchement de l'estop

Fonction appelée lorsqu'un *estop* intervient. Un *estop* est déclenché au niveau de **SYNAPXIS** par un module *robot* lorsque celui-ci est couramment *attaché* à une tâche et qu'il détecte un arrêt d'urgence.

Cette fonction permet notamment d'agir sur la machine, d'informer l'utilisateur et de lui soumettre si nécessaire la possibilité de reprendre le processus. La réponse de reprise doit obligatoirement faite à l'aide de l'instruction **machineEstopRetry**.

Le paramètre *reason* indique la cause qui est à l'origine de l'estop :

0 Raison indéterminée.

1 Arrêt d'urgence.

2 Enlèvement de la puissance robot.

3 Changement de mode d'opération du robot (manuel, déporté).

4 Erreur d'enveloppe (position ou vitesse inatteignable).

5 Erreur interne.

6.3 Événements Robot

Le module *Robot* publie 2 événements liés à la mise en puissance et l'exécution de mouvements.

<functionRobotEnsure>

bool **<functionRobotEnsure>**(*string* **robotName**, *bool* **ensureState**)

robotName	*string*	nom du robot
ensureState	*bool*	flag indiquant si le robot est assuré ou dé-assuré

Fonction appelée par l'instruction **ensure**. Elle permet en fonction de l'état *true* ou *false* du flag *ensureState* de sécuriser la cellule et mettre en puissance les différents éléments, ou enlever la puissance et dé-sécuriser la cellule. Si l'instruction retourne *false* une erreur est créée et la tâche est arrêtée. Si cet événement n'est pas défini, l'instruction **ensure** est exécutée directement par le robot.

<functionRobotMove>

```
void <functionRobotMove>(string robotName,
                         int moveKind,
                         string transitionPointName,
                         locc pos,
                         locj posj,
                         real speed,
                         real approch,
                         bool straight)
```

robotName	*string*	nom du robot
moveKind	*int*	type de mouvement
transitionPointName	*string*	point de transition
pos	*locc*	position cartésienne
posj	*locj*	position articulaire
speed	*real*	vitesse programme
approch	*real*	distance d'approche
straight	*bool*	approche linéaire

Fonction appelée par **SYNAPXIS** : édition de point, test des transition, etc. En fonction de la valeur du paramètre *moveKind*, la fonction doit réaliser l'opération suivante :

— 0 : mouvement sur une position articulaire

— 1 : mouvement sur une position cartésienne

— 2 : transition

— 3 : transition et mouvement sur une position articulaire

— 4 : transition et mouvement sur une position cartésienne

Les paramètres *speed*, *approch* et *straight* permettent de spécifier le mouvement.

```
if (!deviceAttach(robotName))
  print("Attach failed :", robotName)
  return
end

ensure(true)
setSpeed(speedProgram)

switch(moveKind)
  case 0
    movej(locj)
  case 1
    if (straight)
      moves(locc, appro)
    else
      move(locc, appro)
    end
  case 2
    transitionMove(transitionName)
  case 3
    transitionMove(transitionName)
    movej(locj)
  case 4
    transitionMove(transitionName)
    if (straight)
      moves(locc, appro)
    else
      move(locc, appro)
    end
end

deviceDetach(robotName)
```

6.4 Événements Trajectoire

Le module *Trajectoire* publie 3 événements liés à l'exécution des trajectoires en mode réglage et l'exécution des *connecteurs*.

<functionEditRun>

```
void <functionEditRun>(string robotName,
                       locc toolTrans,
                       bool editPendant,
                       string transitionPointName,
                       real startLimit,
                       real entryPos,
                       real endLimit,
                       real approchDistance)
```

robotName	*string*	nom du robot
toolTrans	*locc*	transformée du tool
editPendant	*bool*	exécution ou réglage au MCP
transitionPointName	*string*	point de transition
startLimit	*real*	départ de la trajectoire
entryPos	*real*	position d'entrée sur la trajectoire
endLimit	*real*	fin de la trajectoire
approchDistance	*real*	distance d'approche

Fonction appelée lors de l'édition d'un try en *mode réglage pendant* (*editPendant* == *true*), ainsi que lors de l'exécution d'un *groupe*, d'un *step* ou d'un *try* depuis l'interface de réglage (*editPendant* == *false*).

Les paramètres *robotName* et *tool* permettent de préparer le robot.

En mode *réglage pendant*, le paramètre *transitionPointName* permet de générer un déplacement sécurisé vers le frame considéré par la trajectoire. Par rapport à la *longueur linéaire* de la trajectoire, le *try* considéré débute à *startLimit* et se termine à *endLimit*, et ne peut être exécutée en dehors de ces limites. Le robot commence l'édition en mode pendant à la position linéaire *entryPos*.

Le paramètre *approchDistance* correspond à la valeur d'approche courante définie dans les préférence du module *Trajectoire*.

<functionEditMultiGroupRun>

```
void <functionEditMultiGroupRun>(string cycleName,
                                 string robotName,
                                 locc toolTrans,
                                 array groupNames)
```

cycleName	*string*	nom du cycle trajectoire
robotName	*string*	nom du robot
toolTrans	*locc*	transformée du tool
groupNames	*bool*	tableau contenant les noms des groupes à exécuter

Fonction appelée lors de l'exécution complète (tous les groupes) ou partielle (groupes sélectionné uniquement) du cycle trajectoire. Le paramètre *cycleName* permet de préparer le cycle. Les paramètres *robotName* et *tool* permettent de préparer le robot. Le paramètre *groupNames* contient les noms des groupes à exécuter.

```
deviceAttach("MCP")// MCP correspondant
deviceAttach(robotName)
ensure(true)
setTool(toolTrans)

if (!trajCyclePrepare(cycleName))
  print("prepare error", cycleName)
  taskStop()
end

if (!simEnabled())
  if (mcpAlert("Executer?", "OK", "Annuler") == 1)
    taskStop()
  end
end

for (i = 0; i < arraySize(groupNames); i+=1)
  groupName = groupNames[i]
  if (!trajCycleBegin(cycleName, groupName))
    taskStop()
  end
  while (!trajCycleEnd())
    trajCycleRun()
  end
end
```

<functionTryConnectorExecute>

void **<functionTryConnectorExecute>**(*array* **steps**)

steps	*array*	étapes du connecteurs

Fonction appelée avant et après l'exécution d'un *try*. Elle permet d'amener le robot en position avant l'exécution de la trajectoire, et de le repositionner après. Le paramètre *connectorSteps* est un tableau à deux dimensions. Chaque colonne correspond à une étape du connecteur, et chaque ligne à un type de donnée :

— 0 : type de l'étape (*int*)

— 1 : nom du point de transition (*string*)

— 2 : position articulaire

— 3 : position cartésienne

— 4 : valeur réelle (vitesse programme)

Les types d'étape sont les suivants :

— 1 : retour au point de transition

— 2 : transition au point de transition

— 3 : mouvement articulaire vers position articulaire

— 4 : mouvement articulaire vers position joint

— 5 : mouvement linéaire vers position cartésienne

— 6 : affectation de la vitesse

```
stringIndex = 0
locjIndex = 0
loccIndex = 0
realIndex = 0
stepCount = arraySize(steps[0])

for (step = 0; step < stepCount; step += 1)
  switch(steps[0][step])
   case 1
    transitionReach(steps[1][stringIndex])
    stringIndex += 1
   case 2
    transitionMove(steps[1][stringIndex])
    stringIndex += 1
   case 3
    movej(steps[2][locjIndex])
    locjIndex += 1
   case 4
    move(steps[3][loccIndex])
    loccIndex += 1
   case 5
    moves(steps[3][loccIndex])
    loccIndex += 1
   case 6
    setSpeed(steps[4][realIndex])
    realIndex += 1
   end
end
```

<functionEditStepMacro>

```
bool <functionEditStepMacro>(string stepMacroName,
                             int paramIndex,
                             array parameters)
```

stepMacroName	*string*	nom du step macro courant
paramIndex	*int*	index du step macro courant
parameters	*array*	paramètres du step macro courant)

Fonction appelée depuis l'interface de réglage du cycle d'usinage lors de l'édition d'un paramètre de step macro. Si le paramètre est édité par la fonction, celle-ci doit retourner *true*. Si la fonction retourn *false*, l'édition du paramètre se fait normalement. Les paramètres *stepMacroName* et *paramIndex* permettent d'identifier le step macro et le paramètre. Le tableau *parameters* permet d'accéder au(x) paramètre(s) à éditer depuis la fonctions.

<functionApplyFrameView>

void **<functionApplyFrameView>**(*string* **frameName**,
int **viewId**)

frameName	*string*	nom du frame courant
viewId	*int*	index de la vue (0, 1, 2)

Fonction appelée depuis l'interface de réglage du cycle d'usinage (boutons des vues F1, F2, F3), permettant de positionner la vue 3D en fonction du frame correspondant au *step trajectoire* ou *try* couramment sélectionné.

```
fi = inverse(machineFrame(frameName))
s = gViewScaleFactor
fi = trans(s*dx(fi), s*dy(fi), s*dz(fi), drx(fi), dry(fi),
drz(fi))

viewDz = 500
switch(viewId)
  case 0
    viewRz = -90
  case 1
    viewRy = -90
    viewRz = -90
  case 2
    viewRx = 90
    viewRy = -90
    viewRz = -90
end
simViewApply(trans(0, 0, -viewDz*s, viewRx, viewRy,
viewRz)+fi)
```

CHAPITRE 7

Options

7.1 Statistiques

Cette option permet de connaître le nombre de modules, de programmes, de lignes, etc.

7.2 Préférences éditeur

Cette option permet de modifier la taille de la police de l'éditeur de programmes.

7.3 Modules chargés automatiquement

Cette option permet de définir la liste des modules qui sont chargés automatiquement au démarrage de l'application. L'ordre de leur chargement, ainsi que celui de leur représentation dans l'arborescence, correspond à l'ordre de cette liste. Celui-ci peut être modifié par *drag-drop*.

7.4 Entête programme par défaut

Cette option permet de définir les lignes de code qui sont copiée dans chaque nouveau programme lors de sa création.

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// Application:     App / Synapxis, HE-ARC 2013
// Module:          -
//
// creation:        10.01.2013, XYz
// modification:    10.01.2013, XYz
//
// description:     -
// inputs:          /
// return:          /
// * * * * * * * * * * * * * * * * * * * * * * * * * * * *

return
```

7.5 Comparaison de fichier module

Cette option permet de comparer les fichiers modules courants avec une version antérieur. Le dossier contenant la version antérieur des fichiers modules doit être sélectionné, hors du *worskpace* courant. Le programme utilisé pour cette comparaison est un programme tiers, par exemple *Compare It !*. Son exécution est lancée à partir d'une commande dos (le point d'exclamation doit absolument être supprimé de tous les chemins) :

```
""C:\Program Files\CompareIt\wincmp.exe" "%s" "%s" /1 /r "
```

La combinaison de touche *Ctrl+D* permet de lancer la comparaison pour le module courant, et *Ctrl+Alt+D* pour tous les modules ouverts. Dans ce dernier cas, la comparaison n'est faite que pour les paires fichiers dont la date+heure sont différentes.

7.6 Touch Screen Edition

Cette option permet d'activer le mode *Touch Screen* pour l'édition facilité d'un programme sans clavier physique. Sur la partie gauche de l'éditeur, un onglet supplémentaire présente des boutons pour l'ajout de ligne prédéterminées (appel à des fonctions), duplication de ligne, suppression, etc.

Liste des tableaux