ROBOT KUKA KR C3

Jean-Louis Boimond
Université Angers

Table des matières

Ce document est largement inspiré des documents produits par KUKA intitulés :

- KUKA System Software 8.3 – Manuel de service et de programmation pour l'utilisateur final – Edition 17.04.2018, 269 pages,
- SOFTWARE - KR C2 / KR C3 - Expert Programming – KUKA System Software (KSS) - Realease 5.2, 178 pages.

1 DESCRIPTION DU ROBOT KUKA KR3

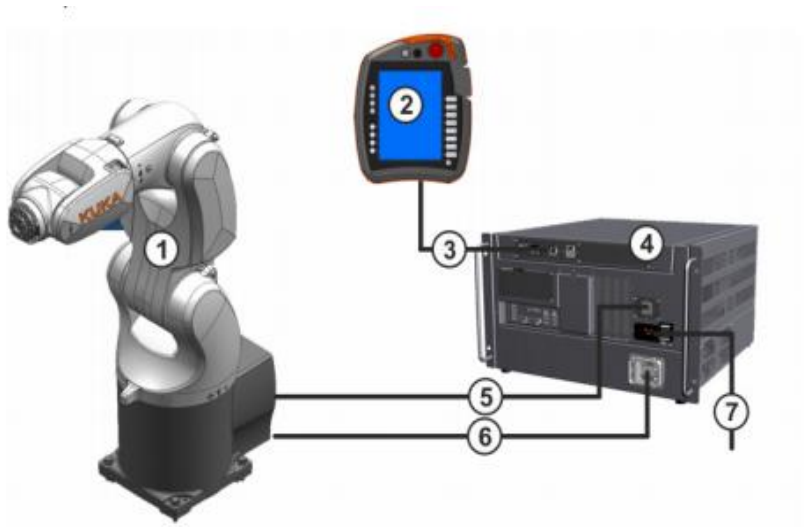
1.1 Description générale

Le KR 3 (AGILUS) est un robot 6 axes, compact (le plus petit de sa catégorie), offrant une flexibilité maximale de mouvement dans des espaces confinés (difficilement accessibles). Son temps de cycle minimal, son grand rendement et sa petite taille font qu'il s'avère particulièrement adapté pour des charges utiles ne dépassant pas 3 kg dans le cas, par exemple, de manipulation d'objets, de vissage, de collage, d'emballage, de test.

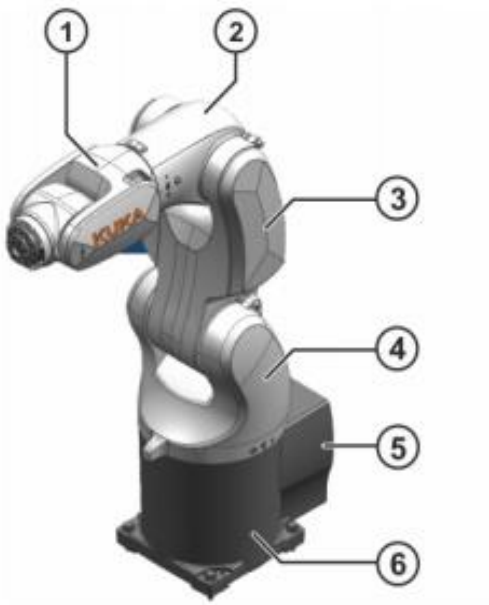
Quelques caractéristiques du robot :

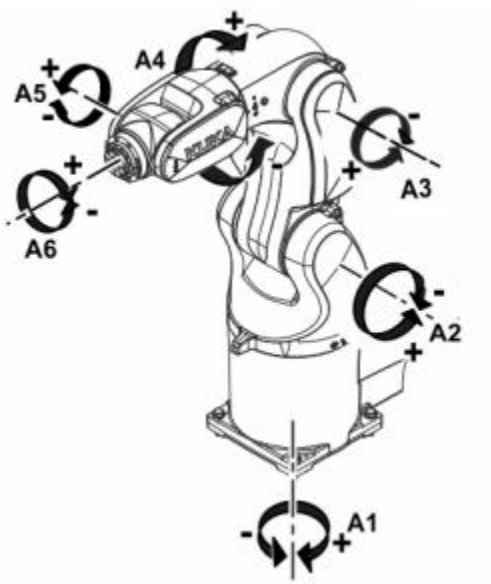
Axes	6
Charge utile	3 Kg
Rayon d'action	540 mm (20+260+260)
Répétabilité	+/- 0,02 mm
Poids du robot	26 Kg
Montage	Plancher, plafond, mur

Le système est composé du KR 3 Agilus ①, de câbles connecteurs (③, ⑤, ⑥, ⑦ pour l'alimentation), du contrôleur KRC 4 compact ④ et du smartPAD ②. Le robot peut être connecté au réseau *via* un câble Ethernet branché sur le contrôleur (prise X66), par exemple, pour le connecter au logiciel WorkVisual afin de configurer, programmer et diagnostiquer un robot KUKA.



Comme illustré dans la figure suivante située à gauche, le bras est constitué d'un poignet (In-Line wrist) ①, d'un bras (Arm) ②, d'un bras de liaison (Link arm) ③, d'une colonne de rotation (Rotating column) ④, d'un socle (Base frame) ⑥ avec un accès aux différents connecteurs électriques ⑤.





Les valeurs angulaires et de vitesses admissibles au niveau des articulations *A1, ..., A6* (voir figure précédente située à droite) sont les suivantes :

Motion range	
A1	±170 °
A2	-170 ° / 50 °
A3	-110 ° / 155 °
A4	±175 °
A5	±120 °
A6	±350 °
Speed with rated payload	
A1	530 °/s
A2	529 °/s
A3	538 °/s
A4	600 °/s
A5	600 °/s
A6	800 °/s

Attention : pour des raisons de sécurité, le fait que le robot soit installé sur un chariot impose de brider la vitesse angulaire des articulations à une valeur de 20% (voir *2.2.i*, touches ⑥ et ⑦ du smartPAD pour effectuer le réglage).

1.2 Mise en route du système

Règles de sécurité :

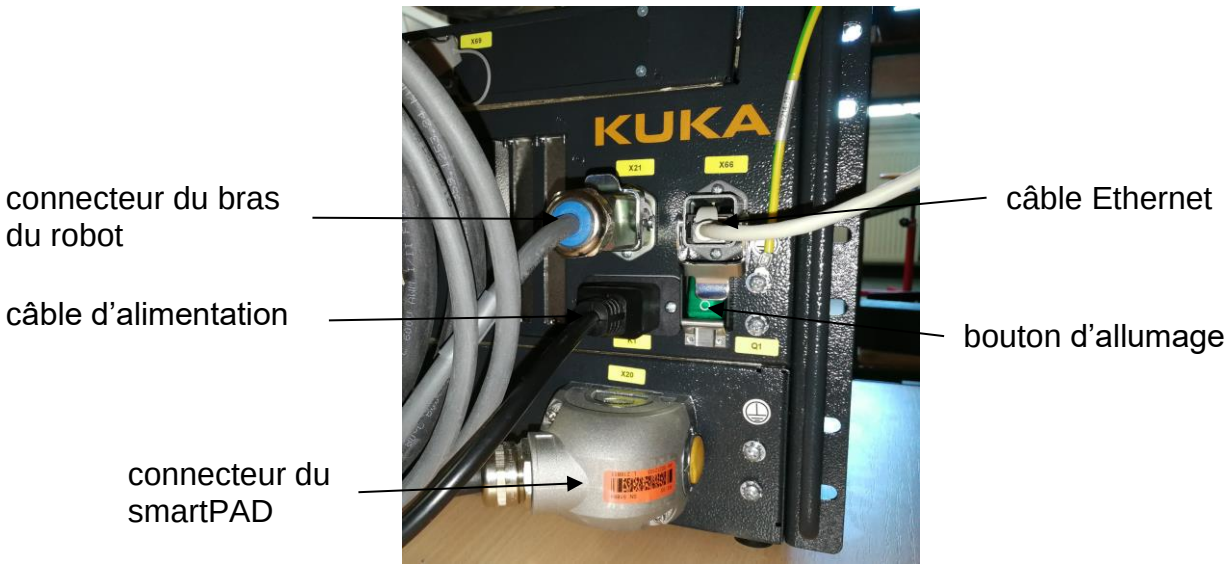
- **Personne ne doit être dans l'espace de travail du bras du robot, soit 1m (le robot a une portée de 54 cm, sans compter l'outil).**
- **La phase de test du robot doit s'effectuer en vitesse réduite : via le smartPAD (voir *2.2.i*), se mettre en mode T1 (touche ②) avec une vitesse manuelle <= 20% (touche ⑦).**
- **Etre toujours prêt à appuyer sur le bouton d'arrêt d'urgence lors de la mise en**

mouvement du bras (bouton ③ du smartPad).

Mise en route :

- Brancher la prise électrique du robot sur une prise murale 230V.
- Allumer la multiprise présente sur le chariot. Après un temps d'allumage, le robot est prêt pour être utilisé.
  Si le smartPAD et le contrôleur ne s'allument pas, vérifier que :
  - le bouton d'allumage du contrôleur, situé sous la prise X66, est sur ON,
  - les câbles d'alimentation du robot et du contrôleur sont bien branchés (voir figure suivante).
- Se situer dans le groupe d'utilisateurs « Expert » : Pour cela, sélectionner la touche du menu principal, située en haut à gauche, dans la barre d'état de l'interface smartHMI (voir *2.2.iv*, ①), puis : **configuration > Groupe d'utilisateurs**, mot de passe : **kuka**.

N.B. : Au bout d'un certain temps, le groupe d'utilisateurs, par défaut, « Opérateur » est réactivé, ce qui nécessite de s'authentifier de nouveau en tant qu'utilisateur « Expert ».



1.3 Mise en position initiale HOME du bras du robot

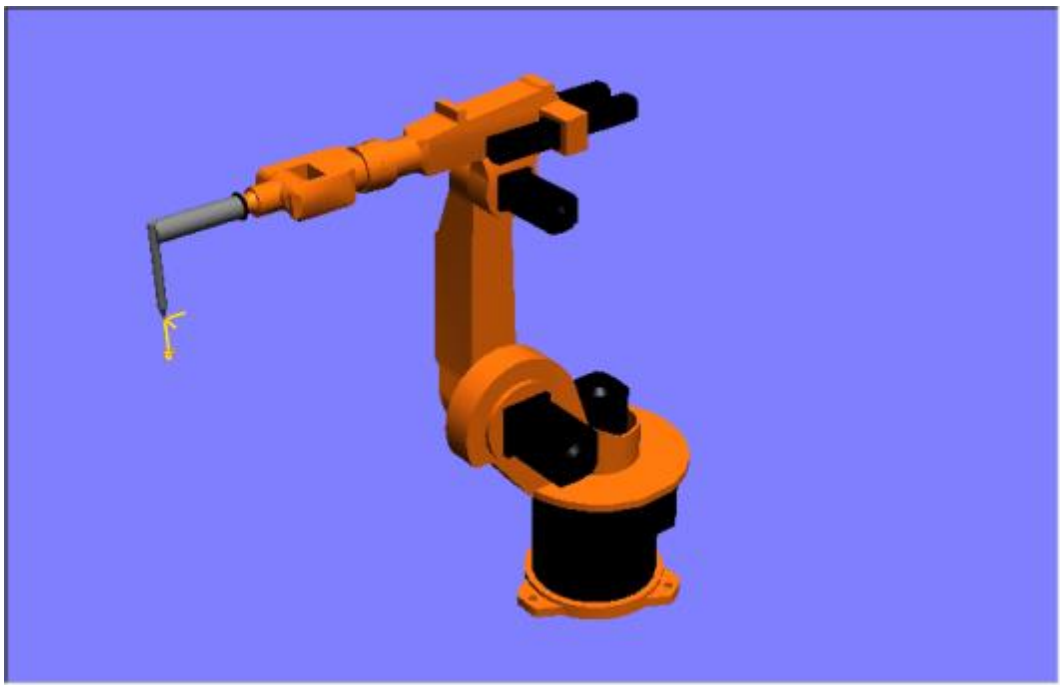
La première instruction de déplacement dans un programme doit définir une situation initiale sans équivoque, c'est le rôle de la position HOME. Cette position est définie par défaut comme suit :

Axe	A1	A2	A3	A4	A5	A6
Position (en degré)	0°	-90°	+90°	0°	0°	0°

En général, la position HOME est également utilisée comme dernière position du robot.

➢ **La position HOME est commune à tous les programmes du robot. Aussi, une modification de cette position concerne tous les programmes dans lesquels elle est utilisée.**

La position HOME correspond à la posture décrite dans la figure suivante.



1.4 Arrêt du système

Condition préalable :

- Être dans le groupe d'utilisateurs « Expert » : Pour cela, sélectionner la touche du menu principal, située en haut à gauche, dans la barre d'état de l'interface smartHMI (voir *2.2.iv*, ①), puis : **configuration > Groupe d'utilisateurs**, mot de passe : **kuka**.

Procédure :

- Dans le menu principal, sélectionner « **Arrêter** »,
- Appuyer sur « **Arrêter le PC de commande** »,
- Confirmer la question de sécurité par « **Oui** »,
- **Attendre que le contrôleur s'arrête**, éteindre la multiprise présente sur le chariot afin que le contrôleur ne redémarre pas immédiatement lors de la prochaine mise sous tension.

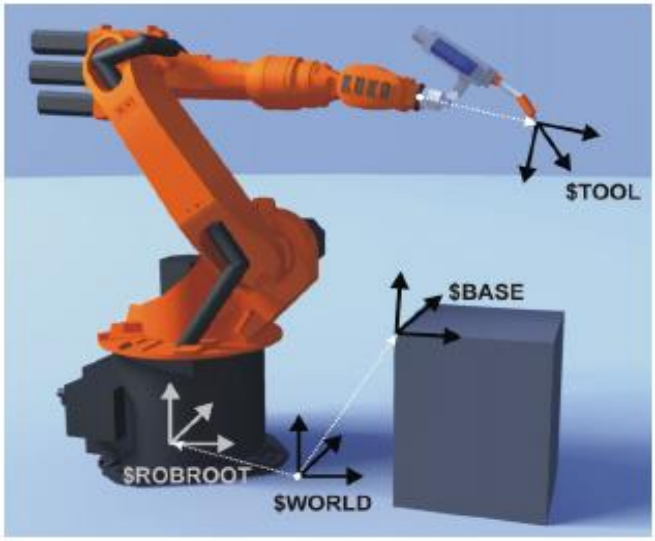
2 MISE EN MOUVEMENT DU BRAS DU ROBOT KUKA KR3

2.1 Les modes de déplacement

Une fois la mise en route générale effectuée, 2 moyens sont disponibles pour mettre le bras en mouvement :

- manuellement *via* le pendant (mode T1) auquel cas la vitesse du Centre de l'Outil (CDO) n'excède pas 250 *mm/s*,
- automatiquement *via* l'exécution d'un programme (mode AUTO).

Il existe 4 repères cartésiens différents pour déplacer le robot, comme illustré dans la figure qui suit.



➢ ***WORLD***

Le système de coordonnées **WORLD** est situé, de manière standard, au pied du robot. Il est le système de référence pour les systèmes de coordonnées BASE et ROBROOT.

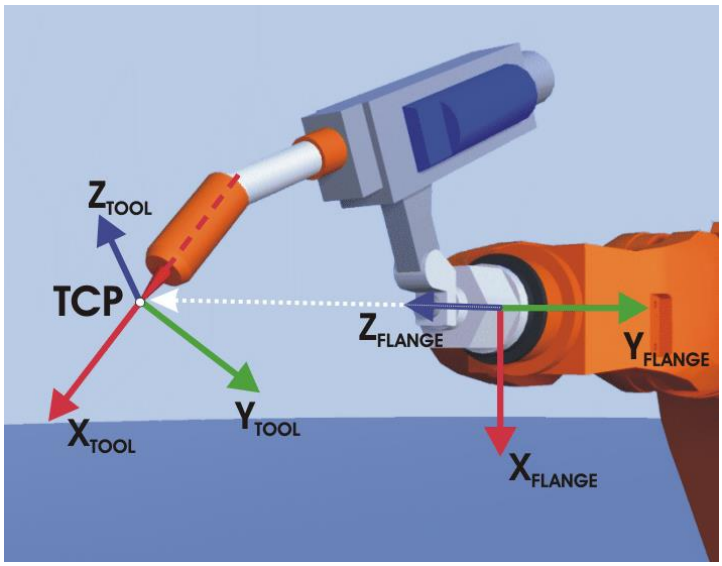
ROBROOT est un système de coordonnées dont l'origine se trouve toujours au pied du robot. Il décrit la position du robot par rapport au système WORLD. Par défaut, le système de coordonnées ROBROOT est identique au système de coordonnées WORLD.

➢ ***BASE***

BASE est un système de coordonnées définissant la position de la pièce ou de la surface de travail. Il se réfère au système de coordonnées WORLD. Par défaut, le système de coordonnées BASE est identique au système de coordonnées WORLD.

➢ ***TOOL***

TOOL est un système de coordonnées se trouvant au point de travail de l'outil (*Tool Center Point*, TCP), voir figure suivante. Par défaut, le système de coordonnées TOOL est identique au système de coordonnées FLANGE dont l'origine est située au centre de la flasque/bride du robot.



2.2 Teach pendant SmartPAD et interface utilisateur smardHMI

Le **smartPAD** est le boîtier de programmation portatif (‘*teach pendant*’) du robot. Il possède toutes les fonctions de commande et d'affichage indispensables à la commande et à la programmation du robot. Il dispose d’un écran tactile qui donne accès à l'interface smartHMI.

Le **smartHMI** (‘*Human-Machine Interface*’) est l’interface utilisateur du KUKA System Software. Elle permet notamment la programmation, la gestion des utilisateurs, l’édition de programme, l’affichage de messages.

i) Face avant du smartPAD

Pos.	Description
1	Bouton pour déconnecter le smartPAD.
2	Interrupteur à clé pour changer de mode (manuel **T1** ou automatique **AUTO**).
3	Bouton d'**ARRET D'URGENCE** pour stopper le robot, il est verrouillé lorsqu'il est actionné.
4	Space Mouse pour le déplacement manuel du robot.
5	Touches de déplacement pour le déplacement manuel du robot .
6	Touche pour le réglage (en %) de la vitesse du programme (en mode **AUTO**).
7	Touche pour le réglage (en %) de la vitesse manuelle (en mode **T1**).
8	Touche de menu principal pour afficher les options de menu de l’interface smartHMI.
9	Touches de fonction : les touches d'état servent principalement à régler les paramètres des progiciels technologiques (leur fonction précise dépend des progiciels).

10	Touche **Start** pour exécuter le programme en cours.
11	Touche **Start en arrière** pour exécuter en arrière le programme en cours. Le programme est traité pas à pas.
12	Touche **STOP** pour arrêter le programme en cours.
13	Touche **clavier** pour afficher explicitement le clavier (sachant qu'en règle générale, l'interface smartHMI détecte lorsque des entrées avec le clavier sont nécessaires et affiche celui-ci automatiquement).

Le clavier n'indique que les caractères nécessaires, par exemple que des chiffres dans l'exemple décrit dans la figure qui suit.

ii) Face arrière du smartPAD

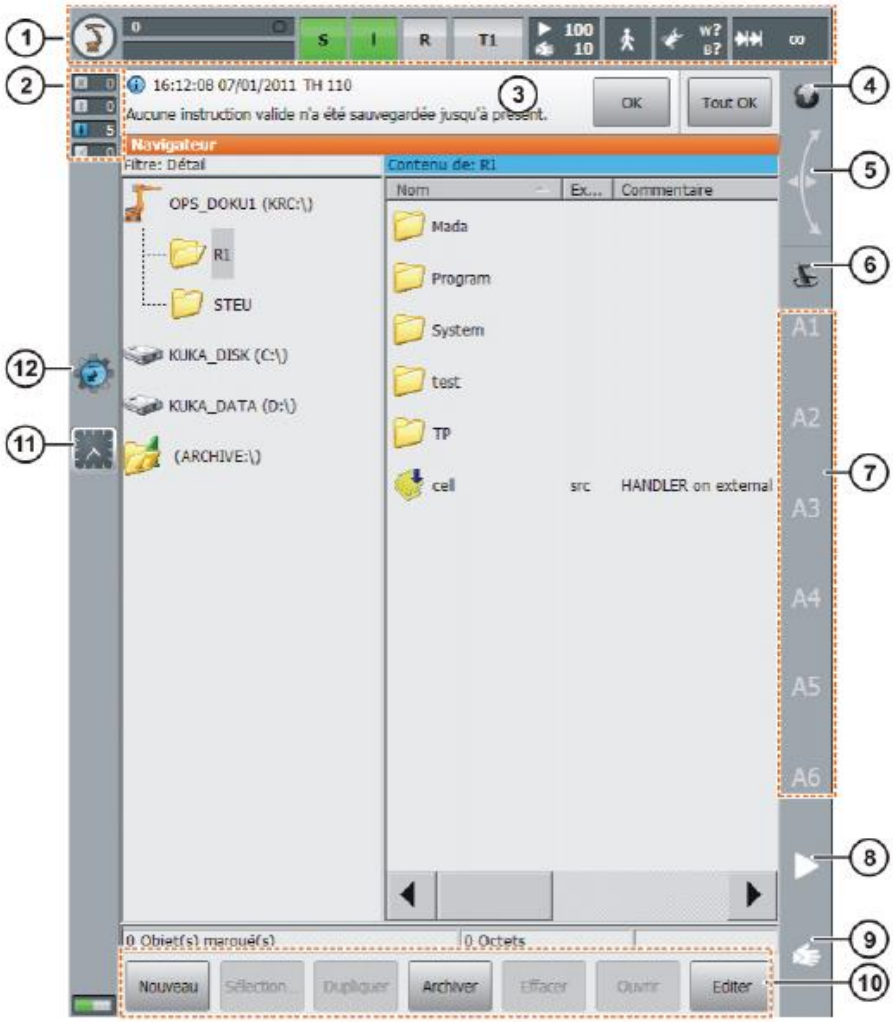
Pos.	Description
1, 3, 5	Interrupteur d'homme mort à 3 positions (non enfoncé, position moyenne, enfoncé). Dans le mode **T1**, l'interrupteur d'homme mort doit être maintenu en position moyenne pour permettre la mise en mouvement du robot.

	Cet interrupteur reste sans effet dans le mode Automatique.
2	Cette touche permet d'exécuter le programme en cours.
4	Connexion pour clé USB (au format FAT 32) utilisée par exemple pour l'archivage, la restauration de données.
6	Plaque signalétique.

iii) Interface utilisateur smartHMI

La fenêtre centrale, représentée dans la figure qui suit, est le 'navigateur'. Il permet d'explorer les fichiers présents dans la baie de commande.

N.B. : Les fichiers que vous aurez à concevoir devront se trouver dans le répertoire : **R1 > Program**.



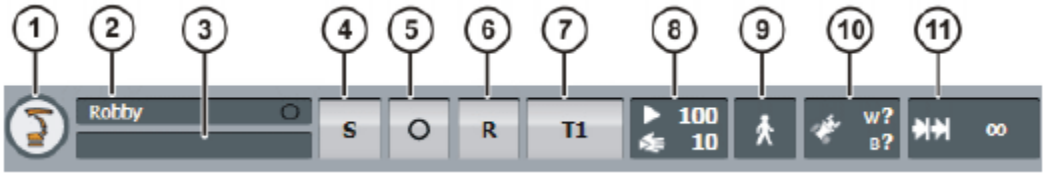
Interface utilisateur smartHMI

No.	Description
1	**Barre d'état** (voir 2.2.iv).

2	**Compteur de messages** : Il indique le nombre de messages présents pour chaque type de message d'erreur. L'affichage est agrandi en touchant le compteur de messages.
3	**Fenêtre de messages** : Par défaut, seul le dernier message est affiché. En touchant la fenêtre de messages, celle-ci s'agrandit et indique tous les messages présents. Un message acquittable peut être acquitté avec la touche **OK**, tous les messages acquittables peuvent être acquittés avec la touche **Tout OK**.
4	**Affichage d'état Space-Mouse** : Cet affichage indique le système de coordonnées actuel pour le déplacement manuel avec la Space Mouse. En touchant l'affichage, tous les systèmes de coordonnées sont affichés avec la possibilité d'en sélectionner un autre.
5	**Affichage Orientation Space-Mouse** : En touchant l'affichage, l'orientation actuelle de la Space Mouse est affichée et peut être modifiée.
6	**Affichage d'état Touches de déplacement** : Cet affichage indique le système de coordonnées actuel pour le déplacement manuel avec les touches de déplacement (voir ⑦). En touchant l'affichage, tous les systèmes de coordonnées (*Axes, World, Base, Outil*) sont affichés avec la possibilité d'en sélectionner un autre.
7	Affichage des 6 touches de déplacement, à savoir : - les axes *A1, ..., A6* si l'espace sélectionné (voir ⑥) est articulaire, - la position (*X, Y, Z*) et l'orientation (*A, B, C*), selon la convention *ZYX*, du repère *World, Base* ou *Tool* si l'espace sélectionné est opérationnel. Le fait d'appuyer sur l'une de ces 6 touches provoque l'affichage du type de déplacement (articulaire, opérationnel).
8	'Override' programme : Cette touche permet de réduire la vitesse des déplacements (en %) dans le cas d'une exécution automatique du programme (mode AUTO).
9	'Override' manuel : Cette touche permet de réduire la vitesse des déplacements (en %) dans le cas d'une exécution manuelle du programme (mode T1).
10	**Barre de boutons** : Les boutons changent de façon dynamique en fonction de la fenêtre en cours d'affichage sur l'interface utilisateur smartHMI. Tout à droite se trouve le bouton **Editer** lequel permet d'appeler un grand nombre d'instructions se référant au navigateur (explorateur de fichiers).
11	**Horloge** : Elle affiche le temps système. En touchant l'horloge, le temps système est affiché sous forme numérique, ainsi que la date actuelle.
12	Symbole WorkVisual : Si aucun projet ne peut être ouvert, le symbole a un petit X rouge en bas à droite (c'est par exemple le cas lorsque des fichiers correspondant au projet manquent).

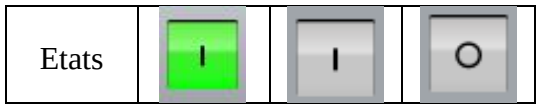
iv) Barre d'état

La barre d'état indique l'état de certains réglages centraux du robot. Pour la plupart des affichages, un contact tactile ouvre une fenêtre permettant de modifier les réglages.



No.	Description
1	Touche de **menu principal** pour afficher les options de menu sur l'interface utilisateur smartHMI.
2	Affichage du nom du robot.
3	Affichage du programme sélectionné.
4	Affichage de l'état Interpréteur Submit[1].
5	Affichage de l'état **Entraînements**, voir (*) ci-dessous pour les détails.
6	Affichage de l'état **Interpréteur robot**, voir (**) ci-dessous pour les détails. Un programme peut être sélectionné ou abandonné.
7	Affichage du mode actuel (**T1** ou **AUTO**).
8	Affichage en % des états actuels des 'overrides' (vitesses de déplacement) programme et manuel.
9	Affichage de l'état du **Mode de traitement du programme** en cours, voir (***) ci-dessous pour les détails.
10	Affichage de l'état Outil/Base. Affiche l'outil et la base actuels.
11	Affichage de l'état Déplacement manuel incrémental.

(*) Affichage de l'état Entraînements
L'affichage **Entraînements** peut afficher les états suivants :



Signification des symboles et des couleurs :

Symbole I	Les entraînements sont prêts.
Symbole O	Les entraînements sont à l'arrêt.
Couleur verte	L'interrupteur d'homme mort est actionné (position moyenne) ou n'est pas nécessaire et il n'y a pas de messages empêchant le déplacement du robot.

[1] Un fichier *Submit* contient des instructions et peut être utilisé, par exemple, pour effectuer une surveillance cyclique d'un capteur en interaction avec le robot. Un fichier *Submit* s'exécute en parallèle au programme robot et est traité par l'« Interpréteur Submit ».

Couleur grise	L'interrupteur d'homme mort n'est pas actionné, ou est enfoncé, et/ou il y a des messages empêchant le déplacement du robot.

Le fait de toucher l'affichage **Entraînements** provoque l'ouverture de la fenêtre « Conditions de déplacement », ce qui permet d'activer, ou de désactiver, les entraînements.

(**) Affichage de l'état de l'interpréteur robot

Symbole	Couleur	Description
R	Gris	Aucun programme n'est sélectionné.
R	Jaune	L'indicateur de bloc se trouve sur la première ligne du programme sélectionné.
R	Vert	Le programme est sélectionné et est en cours d'exécution.
R	Rouge	Le programme sélectionné qui était en cours d'exécution a été arrêté.
R	Noir	L'indicateur de bloc se trouve à la fin du programme sélectionné.

(***) Sélectionner le mode de traitement du programme

Le fait de toucher l'affichage de l'état **Mode de traitement de programme** fait apparaître une fenêtre qui permet de sélectionner le mode de traitement de programme souhaité, voir le tableau ci-dessous.

Désignation	Affichage de l'état	Description
Continu #GO		Le programme est traité sans arrêt jusqu'à la fin.
Point par Point #MSTEP		Le programme est traité avec un stop à chaque point, même aux points auxiliaires et aux points de segments Spline. La touche **Start** doit être actionnée à nouveau pour chaque point. Le programme est traité sans avance.
Ligne par Ligne #ISTEP		Le programme est traité avec un stop après chaque ligne du programme. La touche **Start** doit être actionnée pour lancer chaque nouvelle ligne. Le programme est traité sans avance.
En arrière #BSTEP		Ce mode est choisi automatiquement si la touche **Start en arrière** (voir 2.2.i, ⑪) est actionnée. Il ne peut pas être choisi d'une autre façon. Le comportement est le même que pour **Point par Point** excepté que les déplacements CIRC (voir 2.7.ii) sont parcourus en arrière de la même façon que lors du dernier déplacement en avant. Cela signifie que s'il n'y a pas eu d'arrêt au point auxiliaire lors du déplacement en avant, il n'y en aura pas non plus lors du déplacement en arrière. Cette exception ne s'applique pas aux déplacements SCIRC. Ici, il y a toujours un arrêt au point auxiliaire en déplacement en arrière.

2.3 Lancement d'un programme

Avant l'exécution d'un programme, une coïncidence de bloc (notée BCO) doit être réalisée afin que la position du robot corresponde aux coordonnées du point de programme en cours (indiqué par le pointeur de bloc (Block pointer)). Sachant que l'exécution du mouvement permettant cette correspondance ne représente pas un mouvement testé et programmé, le mouvement est toujours réalisé à vitesse réduite en appuyant sur le bouton **Start** (voir 2.2.i, ⑩) et en activant l'homme mort afin de pouvoir arrêter le robot en cas de besoin.

i) Mode manuel

Condition préalable :

- Un programme est sélectionné.
- Mode **T1** (voir 2.2.i, ②, 2.2.iv, ⑦).

Procédure :

1. Sélectionner le mode de traitement du programme. Par défaut, le programme est traité sans stop  (voir 2.2.iv, ⑨, 2.2.iv (***)).
2. Maintenir l'interrupteur d'homme mort enfoncé et attendre jusqu'à ce que la barre d'état « Entraînements » (2.2.iv, ⑤) affiche « Entraînements prêts », c-à-d :

3. Maintenir la touche **Start** enfoncée (afin d'exécuter une coïncidence de bloc) jusqu'à ce que la « Fenêtre des messages » (voir 2.2.iii, ③) affiche « Coïncidence de blocs atteinte ». Le robot s'arrête alors et le programme est prêt à être exécuté. Pour exécuter le programme, maintenir la touche **Start** enfoncée (inutile de maintenir l'interrupteur d'homme mort enfoncé).

Pour arrêter un programme lancé manuellement, cesser d'appuyer sur la touche **Start**. Appuyer de nouveau sur **Start** (en gardant la touche enfoncée) pour reprendre son déroulement.

ii) Mode automatique

Condition préalable :

- Un programme est sélectionné.
- Mode **AUTO** (voir 2.2.i, ②, 2.2.iv, ⑦).

Procédure :

1. Sélectionner le mode de traitement de programme (par défaut, sans stop  , voir 2.2.iv, ⑨, 2.2.iv (***)).
2. Activer les entraînements, c-à-d :

3. Maintenir la touche **Start** enfoncée jusqu'à ce que la « Fenêtre des messages » (voir 2.2.iii, ③) affiche « Coïncidence de blocs atteinte ». Le robot s'arrête et le programme est prêt à être exécuté.
4. Appuyer sur la touche **Start** pour lancer l'exécution du programme.

Pour arrêter un programme lancé en mode automatique, appuyer sur la touche **STOP**.

2.4 Affichage de la position actuelle

Procédure :

1. Dans le menu principal , sélectionner **Affichage** > **Position réelle**. Sont affichées :
   - la position actuelle (X, Y, Z) et l'orientation (A, B, C) du Centre de l'Outil (CDO),
   - la posture du robot à travers les entrées S (pour Status (état)) et T(pour Turn (rotation)), voir 2.6.ii.
2. Appuyer sur **Spécifique aux axes** pour afficher la position du bras du robot dans l'espace articulaire, c-à-d, les axes A1, …, A6.
3. Appuyer sur **Cartésien** pour afficher à nouveau la situation du robot dans l'espace opérationnel.

2.5 Structures, créations et éditions de programmes

i) Structure de programmes

Un programme en KRL (Kuka Robot Language) peut être constitué d'un ou plusieurs fichiers, selon la complexité des tâches à effectuer.

Un programme est souvent composé :
- d'un fichier « .SRC » qui contient le code du programme,
- d'un fichier « .DAT » qui contient une *Data List* (liste de données) permettant la déclaration et l'initialisation de données utilisées par le programme.

Le fichier de codes et la liste de données sont identifiés par un même nom, 'PROG1' dans l'exemple qui suit, mais avec des extensions différentes :

Fichier de codes : PROG1.**SRC**
Liste de données : PROG1.**DAT**

Un programme simple contient un seul fichier. Des tâches plus complexes peuvent être mieux résolues à l'aide d'un programme composé de plusieurs fichiers.

Fichier SRC

Chaque fichier SRC commence par la déclaration « DEF » suivie par le nom du fichier (composé d'au plus 24 caractères et ne doit pas être un mot-clé) et se termine par « END ».

```
DEF NAME( )
    Declaration section
    Initialization section
    Instruction section
END
```

- Déclarations

Une déclaration consiste à assigner un type de données à un nom, voir 2.6.i et 2.6.ii. Elle est évaluée pendant la compilation du programme, avant son exécution. Aucune instruction ne peut être située dans la section « Declaration », la première instruction (qui débute la section « Instruction ») met fin à la section « Declaration ».

- Initialisations

Une initialisation consiste à assigner une valeur à une déclaration, voir 2.6.ii. Notons que l'initialisation d'une variable peut se faire également dans la section « Instruction ».

- Instructions

Contrairement aux déclarations, les instructions sont de nature dynamique au sens où elles sont exécutées lors du traitement du programme.

N.B. : Une ligne commençant par le caractère ';' est un commentaire (donc non traité par le compilateur).

Fichier DAT

Une *data list* (liste de données) contient **seulement** des déclarations et des initialisations. Il est possible de simultanément déclarer et initialiser une variable (seules les assignations de valeur avec le signe « = » sont autorisées). Lorsque la liste de données et le fichier de codes ont le même nom, les variables déclarées dans la liste de données peuvent être utilisées comme si ces variables avaient été déclarées dans le fichier SRC.

Un fichier DAT commence par la déclaration « DEFDAT » et se termine par « ENDDAT ». Dans l'exemple qui suit, la variable entière OTTO est initialement égale à 0 puis à 25 une fois le programme exécuté.

PROG_1.SRC

```
DEF PROG_1()
…
HALT
…
OTTO = 25
…
HALT
END
```

PROG_1.DAT

```
DEFDAT PROG_1
INT OTTO = 0
ENDDAT
```

```
DEFDAT PROG_1
INT OTTO = 25
ENDDAT
```

N.B. : Les variables « System » (repérables par un premier caractère correspondant à un '$', voir 2.6.iv) ne sont pas acceptées.

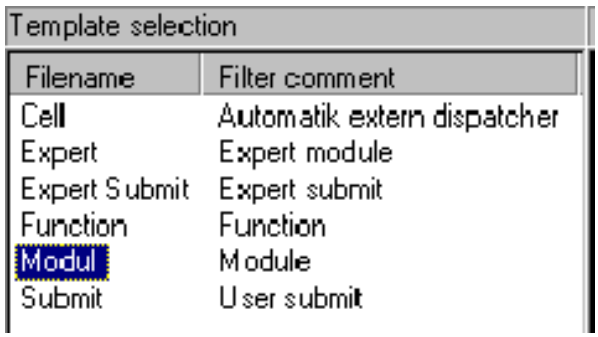
Voir détails au §2.4 du document KR C2 / KR C3 – Expert Programming.

ii) Création et édition de programmes

Création d'un programme

La fenêtre 'Navigateur', qui apparaît suite à la mise en route du système, permet de créer un nouveau programme (voir 2.2.iii). Pour cela, sélectionner le répertoire **R1 > Program** (dans lequel le fichier va être créé) dans la fenêtre située à gauche (intitulée 'Filtre :') du Navigateur. La fenêtre située à droite (intitulée 'Contenu de Program') du navigateur liste les différents fichiers présents dans le répertoire Program. Après avoir sélectionné un fichier quelconque contenu dans la fenêtre 'Contenu de Program', appuyer sur la touche « **Nouveau** », située dans la barre de Boutons (en bas à gauche)[2]. La fenêtre suivante apparaît :

[2] Le fait de sélectionner la fenêtre 'Filtre' (plutôt que 'Contenu de Program') avant d'appuyer sur la touche 'Nouveau' permet de créer un sous répertoire (plutôt qu'un programme).



En utilisant les touches fléchées, sélectionner le ‘template’ (modèle) **Modul** pour créer un fichier SRC et un fichier DAT contenant chacun un programme standard prédéfini. Confirmer en appuyant sur la touche « **OK** » située dans la barre de Boutons (voir 2.2.iii, ⑩). Entrer le nom (sans accent) des fichiers à créer à l’aide du pavé alphanumérique qui apparaît. Confirmer en appuyant sur la touche « **OK** » ou sur la touche « **Entrée** » (située sur le pavé).

Si vous souhaitez ajouter un commentaire, déplacer le curseur dans la case correspondante à l'aide de la touche fléchée droite.

Edition, compilation et linkage d’un programme

Une fois un fichier de programme (SRC), ou de liste de données (DAT), sélectionné, il est possible de l’éditer à l’aide :

- de la touche « **Ouvrir** » située dans la barre de Boutons (voir 2.2.iii, ⑩),
- ou de l’éditeur en utilisant la touche « **Editer** » (située dans la barre de Boutons), puis « **Ouvrir > Fichier/Répertoire** » pour ouvrir le fichier SRC ou « **Ouvrir > Liste de données** » pour ouvrir le fichier DAT.

Il est possible, pour améliorer la lisibilité du programme, d’indenter (avec plusieurs niveaux possibles) les lignes de codes. Utiliser, pour cela, la barre d'espacement.

À la fermeture de l’éditeur, le code de programme complet est compilé/traduit dans un langage machine compréhensible par le contrôleur du robot.

- Compilation

Le compilateur vérifie que le code de programme est correct du point de vue de la syntaxe et de la sémantique. Si une erreur est détectée, un message correspondant est généré et un fichier d'erreur créé avec l'extension de fichier « .ERR ».

Seuls les programmes sans erreur peuvent être sélectionnés et exécutés.

- Editeur de liens/linkage editor

Lors du chargement d’un programme (*via* la touche « **Sélectionner** » située dans la barre de Boutons) en vue d’être exécuter, tous les fichiers et listes de données nécessaires sont *linkés* pour créer un programme exécutable. Lors du *linkage*, il est vérifié si tous les fichiers nécessaires sont présents, compilés et exempts d’erreurs. Lors d’un transfert de paramètres, l’éditeur de liens vérifie également la compatibilité des types de paramètres de transfert. Si une erreur se produit pendant le *linkage*, un fichier d’erreur (.ERR) est créé, comme lors de la compilation.

N.B. : Un fichier qui comporte une erreur apparait barré d'une croix rouge. Le bouton '**Liste de défauts**', situé dans la barre de Boutons, aide dans la compréhension de l'erreur ou des erreurs commises.

Voici un premier exemple de programme où sont notamment définies les vitesses et les accélérations des axes du robots (en pourcentage par rapport aux valeurs maximales admissibles) :

```
DEF PROG1()

;--- Declaration section ---
INT J

;--- Instruction section ---
$VEL_AXIS[1]=100  ; definition of the axis velocities
$VEL_AXIS[2]=100
$VEL_AXIS[3]=100
$VEL_AXIS[4]=100
$VEL_AXIS[5]=100
$VEL_AXIS[6]=100
$ACC_AXIS[1]=100  ; definition of the axis accelerations
$ACC_AXIS[2]=100
$ACC_AXIS[3]=100
$ACC_AXIS[4]=100
$ACC_AXIS[5]=100
$ACC_AXIS[6]=100

PTP {A1 0, A2 -90, A3 90, A4 0, A5 0, A6 0}  ; goto HOME position
FOR J=1 TO 5
    PTP {A1 4}
    PTP {A2 -7, A3 5}
    PTP {A1 0, A2 -9, A3 9}
ENDFOR
PTP {A1 0, A2 -90, A3 90, A4 0, A5 0, A6 0}

END
```

iii) Ecriture d'un programme

Une fois le programme sélectionné (bouton « **Selectionner** » dans la barre de Boutons (situé en bas)), deux moyens sont disponibles pour écrire des instructions dans le programme : le clavier, accessible uniquement en mode Expert et/ou des menus déroulants disponibles à partir du bouton « **Instructions** » situé dans la barre de Boutons. *Via* ces menus, il est possible :

- d'introduire un **commentaire** *via* **Instructions > Commentaire**,
- de programmer des instructions de déplacement :
  - un **déplacement PTP**, pour cela :
    a) Amener le CDO à la position à apprendre comme destination,
    b) Positionner le curseur dans la ligne après celle où l'on souhaite insérer l'instruction de déplacement,
    c) Sélectionner successivement les options **Instructions > Déplacement > PTP**,
    d) Procéder au réglage des paramètres dans le formulaire en ligne,
    e) Sauvegarder l'instruction avec **OK**.
  - un **déplacement LIN**, pour cela :
    a) Amener le CDO à la position à apprendre comme destination,
    b) Positionner le curseur dans la ligne après celle où l'on souhaite insérer l'instruction de déplacement,
    c) Sélectionner successivement les options **Instructions > Déplacement > LIN**,
    d) Procéder au réglage des paramètres dans le formulaire en ligne,

e) Sauvegarder l'instruction avec **OK**.

- un **déplacement CIRC**, pour cela :
  a) Amener le CDO à la position à apprendre comme destination,
  b) Positionner le curseur dans la ligne après celle où l'on souhaite insérer l'instruction de déplacement,
  c) Sélectionner successivement les options **Instructions > Déplacement > CIRC**,
  d) Procéder au réglage des paramètres dans le formulaire en ligne,
  e) Actionner **Modif PA**,
  f) Amener le CDO à la position à apprendre comme destination,
  g) Sauvegarder l'instruction avec **OK**.

- de programmer des **instructions logiques**. Soient :
  - une sortie numérique – **OUT**, pour cela :
    a) Positionner le curseur dans la ligne après laquelle on souhaite insérer l'instruction logique,
    b) Sélectionner successivement les options **Instructions > Logique > OUT > OUT**,
    c) Définir les paramètres dans le formulaire en ligne,
    d) Sauvegarder l'instruction avec **OK**.
  - une sortie d'impulsion – **PULSE**, pour cela :
    a) Positionner le curseur dans la ligne après laquelle on souhaite insérer l'instruction logique,
    b) Sélectionner successivement les options **Instructions > Logique > OUT > PULSE**,
    c) Définir les paramètres dans le formulaire en ligne,
    d) Sauvegarder l'instruction avec **OK**.
  - une sortie analogique – **ANOUT**, pour cela :
    a) Positionner le curseur dans la ligne après laquelle on souhaite insérer l'instruction logique,
    b) Sélectionner successivement les options **Instructions > Sortie analogique > Statique** ou **Dynamique**,
    c) Définir les paramètres dans le formulaire en ligne,
    d) Sauvegarder l'instruction avec **OK**.

- de programmer un temps d'attente - **WAIT**, pour cela :
  a) Positionner le curseur dans la ligne après laquelle on souhaite insérer l'instruction logique,
  b) Sélectionner successivement les options **Instructions > Logique > WAIT**,
  c) Définir les paramètres dans le formulaire en ligne,
  d) Sauvegarder l'instruction avec **OK**.

- de programmer un temps d'attente – **WAIT FOR**, pour cela :
  a) Positionner le curseur dans la ligne après laquelle on souhaite insérer l'instruction logique,
  b) Sélectionner successivement les options **Instructions > Logique > WAIT FOR**,
  c) Définir les paramètres dans le formulaire en ligne,
  d) Sauvegarder l'instruction avec **OK**.

iv) Modifications de programmes

Il existe essentiellement deux méthodes pour modifier un programme (en mode Expert).

- Correction de programme,
- Éditeur.

▪ Correction de programme

La correction de programme est la méthode standard. Ce mode est automatiquement actif lorsqu'un programme est sélectionné ou qu'un programme en cours est arrêté. Vous pouvez entrer ou éditer des commandes qui n'affectent qu'une seule ligne de programme (c-à-d, sans structure de contrôle (boucles, etc.), ni déclaration de variable) en utilisant les menus déroulants ou le clavier (voir 2.5.iii).

- **Editeur**

Vous devez ouvrir le programme dont vous souhaitez modifier le contenu à l'aide de fonctions de blocs *via* la touche « **Ouvrir** » accessible dans la barre de Boutons. Si vous souhaitez éditer ou insérer certaines commandes ou structures de programme KRL, vous devez utiliser l'éditeur. Étant donné que le code complet est compilé lorsque l'éditeur est fermé, sont détectables des erreurs qui ne se produisent que dans l'interaction de plusieurs lignes (par exemple, des variables déclarées incorrectement).

Fonctions de bloc

Positionnez d'abord le curseur d'édition clignotant au début ou à la fin de la section du programme à déplacer. Maintenez ensuite la touche « **Shift** » du clavier enfoncée tout en déplaçant le curseur vers le haut ou le bas. Vous sélectionnez ainsi une section de programme qui peut ensuite être modifiée à l'aide des fonctions de blocs décrits ci-dessous. La section sélectionnée peut être reconnue par la couleur en surbrillance.

Appuyez sur la touche de menu « **Programme** » et sélectionnez la fonction souhaitée : **Copier, Coller, Couper, Supprimer, Rechercher, Remplacer** dans le menu qui est ouvert, voir détails au §1.3.2 du document KR C2 / KR C3 – Expert Programming.

2.6 Variables and declarations

i) Variables and names

A variable is represented by a name in the program knowing that names of variables can:

- have a maximum length of 24 characters,
- consist of letters (A--Z), numbers (0--9) and the signs '_' and '$',
- must not begin with a number,
- must not be a keyword.

N.B.: As all system variables begin with the '$' sign, this sign should not be used as the first character in self-defined names.

ii) Data objects

Declaration and initialization of data objects

- **DECL**

Assignment of a variable name to a data type and reservation of the memory space are accomplished with the aid of the **DECL** declaration. You can declare for example two variables QUANTITY and NUMBER of the data type 'INTEGER' by using:

```
DECL INT QUANTITY, NUMBER
```

- Initialization

The first value assignment to a variable is called initialization. A value assignment to a variable is an instruction and must therefore never be located in the Declaration section. All declared variables are initialized in an Initialization section directly after the Declaration section, as described in the following :

```
DEF NAME()
;Declaration section
…
;Initialization section
…
;Instruction section
…
END
```

Simple data types
These data types are available in most programming languages. They contain just one single value as shown in the following table.

Data type	Integer	Real	Boolean	Character
Keyword	INT	REAL	BOOL	CHAR
Meaning	Integer	Floating point number	Logic state	1 character
Range of values	-2^{31} … $2^{31}-1$	+/-1.1E-38 … +/-3.4E+38	TRUE, FALSE	ASCII character

Arrays
The term « arrays » refers to the combination of objects of the same data type to form a data object; the individual components of an array can be addressed *via* indices. For example, by means of the declaration:

```
DECL INT OTTO[7]
```

7 different integers are stored in the array OTTO[].

Character strings
Using the data type **CHAR**, you can only store individual characters, as described. For the purpose of using entire strings of characters, e.g. words, you simply define a one-dimensional array of type CHAR:

```
DECL CHAR NAME[8]
```

Structures

- **STRUC**

If different data types are to be combined, the array is not suitable and the more general form of linkage must be used. Using the declaration statement **STRUC**, different data types which have been previously defined or are predefined data types are combined to form a new composite data type. In particular, other composites and arrays can also form part of a composite.

A typical example of the use of composites is the standard data type **POS**. It consists of 6 REAL values and 2 INT values (defined in the file $OPERATE.SRC) as follows:

```
STRUC POS REAL X, Y, Z, A, B, C, INT S, T
```

The following assignments are permissible for **POS** variables, for instance:

```
POSITION = {X 34.4, Y -23.2, Z 100.0, A 90, B 29.5, C 3.5, S 2, T 6}
POSITION = {B 100.0, X 29.5, T 6}
POSITION = {A 54.6, B -125.64, C 245.6}
POSITION = {POS: X 230, Y 0.0, Z 342.5}
```

N.B.: In the case of **POS**, **AXIS**, and **FRAME** structures, missing components are not altered.

The following structures are predefined:

```
STRUC AXIS REAL A1,A2,A3,A4,A5,A6
STRUC FRAME REAL X,Y,Z,A,B,C
STRUC POS REAL X,Y,Z,A,B,C, INT S,T
```

The components A1, …, A6 of the structure **AXIS** are angle values (rotational axes) of robot axes 1, …, 6.

In the structure **FRAME** you can define 3 position values in space (X, Y, Z) and 3 orientations in space (A, B, C) with the convention **ZYX**. A point in space is thus unambiguously defined in terms of position and orientation.

As robot can address one and the same point in space with several axis positions, the integer variables **S** and **T** in the structure **POS** are used to define an unambiguous axis position, see details at §3.2.3, p. 62-65 of document KR C2 / KR C3 – Expert Programming.

iii) Data Manipulation

Operators

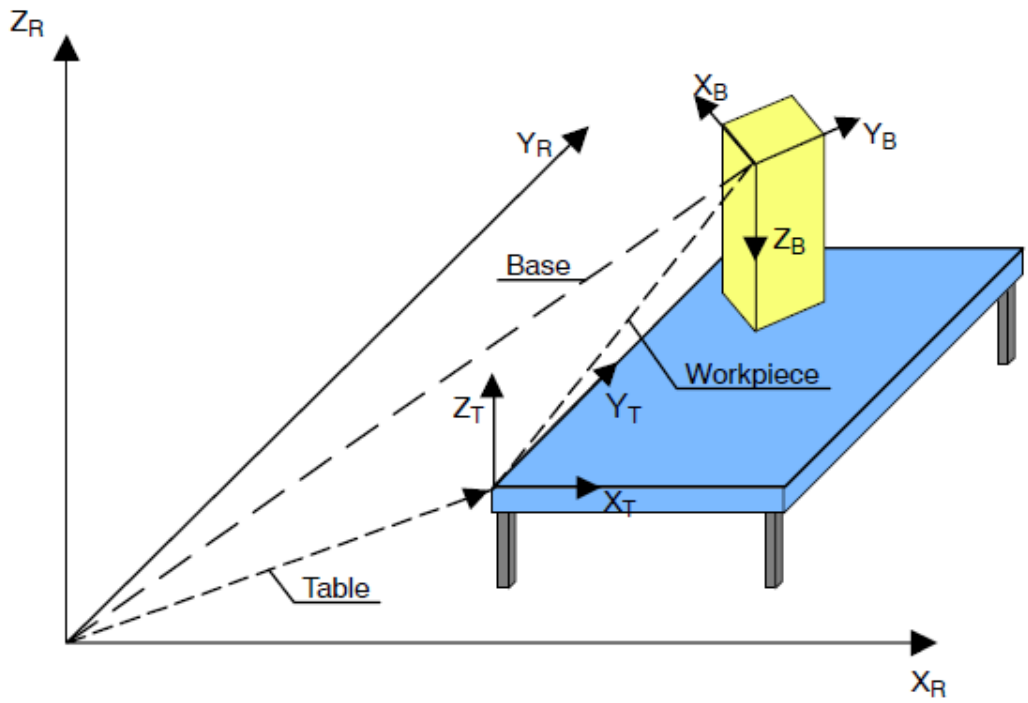
- An **arithmetic operators** concern the data types **INTEGER** and **REAL**. All 4 basic arithmetic operations are allowed as shown in the following Table.

Operator	Description
+	Addition or positive sign
-	Subtraction or negative sign
*	Multiplication
/	Division

- A **geometric operator** is symbolized by a colon ':', it performs a frame linkage (logic operation) on operands of the data types **FRAME** and **POS**.

The linkage of two frames is the usual transformation of coordinate systems. The linkage of a **FRAME** structure and a **POS** structure therefore only affects the frame within the **POS** structure. The components **S** and **T** remain unaffected by the transformation and therefore do not have to be assigned a value. The values X, Y, Z, A, B, C must, however, always be assigned a value in both **POS** operands and **FRAME** operands.

A simple example will be used in order to explain the mode of functioning of the geometric operator (see the following figure):



Let us consider a table in a room. The **ROOM** coordinate system is defined as a fixed coordinate system with its origin at the front left corner of the room.

The table is located parallel to the walls of the room. The front left corner of the table is located exactly 600 mm from the front wall and 450 mm from the lefthand wall of the room. The table is 800 mm high.

On the table is a cuboidal workpiece. The **WORKPIECE** coordinate system has its origin at one corner of the workpiece, as shown in the figure. To allow the part to be optimally handled in later operation, the Z-axis of the **WORKPIECE** coordinate system points downwards. The workpiece is rotated by 40° in relation to the Z-axis of the **TABLE** coordinate system. The position of the **WORKPIECE** coordinate system with reference to the **TABLE** coordinate system is: X = 80 mm, Y = 110 mm, Z = 55 mm.

The task is now to define the **WORKPIECE** coordinate system in relation to the **ROOM** coordinate system. For this purpose, the following frame variables must first be defined:

```
FRAME TABLE, WORKPIECE, BASE
```

The **ROOM** coordinate system is already defined specifically to the system. The **TABLE** and **WORKPIECE** coordinate systems are now initialized in accordance with the given constraints knowing that **TABLE** is defined in relation to the **ROOM** and **WORKPIECE** is defined in relation to **TABLE**.

```
TABLE = {X 450, Y 600, Z 800, A 0, B 0, C 0}
WORKPIECE = {X 80, Y 110, Z 55, A -40, B 180, C 0}
```

The **TABLE** axes correspond to the World axes due to $A = 0, B = 0, C = 0$ (with the convention ZYX).

The **WORKPIECE** coordinate system in relation to the **ROOM** coordinate system is now obtained with the aid of the geometric operator as:

```
BASE = TABLE:WORKPIECE
```

In our case, **BASE** is then defined as follows:

```
BASE = {X 530, Y 710, Z 855, A 140, B 0, C -180}
```

Another possibility would be:

BASE = {X 530, Y 710, Z 855, A -40, B 180, C 0}

N.B.: Only in this specific case do the components of **BASE** result from the addition of the components of **TABLE** and **WORKPIECE**. This is due to the fact that the **TABLE** coordinate system is not rotated in relation to the **ROOM** coordinate system ($A = 0, B = 0, C = 0$). In general, though, simple addition of the components is not possible! Frame linking is not commutative either, meaning that if the reference frame and the target frame are interchanged, the result too will normally change!

As example, let us consider the program GEOM_OP in which are defined various coordinate systems and linkages of coordinate systems. In order to illustrate changes in orientation, the tool center point (TCP) is moved in each coordinate system: first a short distance in the *X* direction, then in the *Y* direction and finally in the *Z* direction.

```
DEF GEOM_OP ( );

;------- Declaration section -------
EXT BAS (BAS_COMMAND :IN,REAL :IN)
; Variable HOME of type AXIS
DECL AXIS HOME
; Array of type FRAME
DECL FRAME MYBASE[2]
; Variable of type POS
DECL POS REF_POS_X, REF_POS_Y, REF_POS_Z

;---------- Initialization ---------
; Initialization of velocities, accelerations, $BASE, $TOOL, etc.
BAS (#INITMOV,0)
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 30,A6 0}
; Set base coordinate system
$BASE={X 1000, Y 0, Z 1000, A 0, B 0, C 0}
; Reference position
REF_POS_X={X 100,Y 0,Z 0,A 0,B 0,C 0}
REF_POS_Y={X 100,Y 100,Z 0,A 0,B 0,C 0}
REF_POS_Z={X 100,Y 100,Z 100,A 0,B 0,C 0}
; Define own coordinate systems
MYBASE[1]={X 200,Y 100,Z 0,A 0,B 0,C 180}
MYBASE[2]={X 0,Y 200,Z 250,A 0,B 90,C 0}

------------- Main section ------------
PTP HOME       ; BCO run; Motion in relation to $BASE coordinate system
; Direct positioning to origin of $BASE coordinate system
PTP $BASE
WAIT SEC 2                 ; Wait 2 seconds
PTP REF_POS_X              ; Move 100 mm in x direction
PTP REF_POS_Y              ; Move 100 mm in y direction
PTP REF_POS_Z              ; Move 100 mm in z direction

; Motion in relation to $BASE coordinate system offset by MYBASE[1]
PTP MYBASE[1]
WAIT SEC 2
PTP MYBASE[1]:REF_POS_X
PTP MYBASE[1]:REF_POS_Y
PTP MYBASE[1]:REF_POS_Z

; Motion in relation to $BASE coordinate system offset by MYBASE[2]
PTP MYBASE[2]
WAIT SEC 2
PTP MYBASE[2]:REF_POS_X
```

```
PTP MYBASE[2]:REF_POS_Y
PTP MYBASE[2]:REF_POS_Z

; Motion in relation to $BASE coordinate system offset by
; MYBASE[1]:MYBASE[2]
PTP MYBASE[1]:MYBASE[2]
WAIT SEC 2
PTP MYBASE[1]:MYBASE[2]:REF_POS_X
PTP MYBASE[1]:MYBASE[2]:REF_POS_Y
PTP MYBASE[1]:MYBASE[2]:REF_POS_Z

; Motion in relation to $BASE coordinate system offset by
; MYBASE[2]:MYBASE[1]
PTP MYBASE[2]:MYBASE[1]
WAIT SEC 2
PTP MYBASE[2]:MYBASE[1]:REF_POS_X
PTP MYBASE[2]:MYBASE[1]:REF_POS_Y
PTP MYBASE[2]:MYBASE[1]:REF_POS_Z
PTP HOME

END
```

- **Relational operators**

It is possible to form logic expressions by using the relational operators listed in the following Table. The result of a comparison is therefore always of the data type **BOOL**, since a comparison can only ever be (TRUE) or (FALSE).

Operator	Description	Permissible data types
==	equal to	INT, REAL, CHAR, ENUM, BOOL
<>	not equal to	INT, REAL, CHAR, ENUM, BOOL
>	greater than	INT, REAL, CHAR, ENUM
<	less than	INT, REAL, CHAR, ENUM
>=	greater than or equal to	INT, REAL, CHAR, ENUM
<=	less than or equal to	INT, REAL, CHAR, ENUM

Comparisons can be used in program execution instructions, and the result of a comparison can be assigned to a Boolean variable.

- **Logic operators**

These operators are used for performing logic operations on Boolean variables, constants and simple logic expressions, as are formed with the aid of relational operators.

Operator	Operand number	Description
NOT	1	Inversion
AND	2	Logic AND
OR	2	Logic OR
EXOR	2	Exclusive OR

- **Priority of operators**

Priority	Operator
1	NOT
2	*
3	+
4	AND

5	EXOR
6	OR
7	== <> < > >= <=

Standard functions

For calculating certain mathematical problems, a number of standard functions are predefined, see the following Table. They can be used directly without further declaration.

Description	Function	Data type of argument	Range of values of argument	Data type of function	Range of values of result
Absolute value	ABS(X)	REAL	-∞ … +∞	REAL	0 … +∞
Square root	SQRT(X)	REAL	0 … +∞	REAL	0 … +∞
Sine	SINE(X)	REAL	-∞ … +∞	REAL	-1 … +1
Cosine	COS(X)	REAL	-∞ … +∞	REAL	-1 … +1
Tangent	TAN(X)	REAL	-∞ … +∞	REAL	-∞ … +∞
Arc cosine	ACOS(X)	REAL	-1 … +1	REAL	0° … 180°
Arc tangent	ATAN2(Y,X)	REAL	-∞ … +∞	REAL	-90° … +90°

iv) System variables and system files

The full functionality of a robot controller can only be utilized if the control parameters can be completely and yet easily integrated into a robot program. This is optimally solved in the KRC by means of the concept of predefined system variables and files.
System variables possess a corresponding data type and can be read and written like any other variable. For example the current robot position cannot be written but only read (restrictions of this nature are checked by the controller).

Examples of predefined variables are: **$POS_ACT** (current robot position), **$BASE** (base coordinate system) or **$VEL.CP** (Continuous Path velocity), **$TIMER[]** (measuring time sequence). A list of all the predefined variables can be found in the separate documentation ‘System variables’.

2.7 Motion programing

One of the most important tasks of the robot controller is moving its arm. The programmer controls the movements by means of special motion commands. These are also the main features which distinguish robot languages from conventional computer programming languages such as C.

- Motion instructions

Depending on the type of controller, these motion instructions can be subdivided into commands for simple **Point-To-Point (PTP) motions** and commands for **path movements**. Whereas, with continuous path (CP) movements, the end effector (e.g. gripper or tool) describes a precise, geometrically defined path in space (straight line or arc), the motion path in PTP movements is dependent on the robot’s kinematic system and cannot, therefore, be accurately predicted. Common to both these types of motion is that **programming takes place from the current position to a new position.** For this reason, **a motion instruction generally only requires the specification of the end position** (excepted for circular motions).

Position coordinates can be specified either as text, by entering numeric values, or by moving the robot to them and saving the actual values (by using the teaching). The possibility exists, in each case, of relating the entries to various coordinate systems.

Further motion properties, such as velocity and acceleration, and orientation control, can be set using system variables. The approximation of auxiliary points is initiated with the aid of optional parameters in the motion instruction. In order to carry out approximation, a computer advance run must be set.

i) Point-to-Point (PTP) motions

The point-to-point (PTP) motion is the quickest way of moving the tip of the tool (Tool Center Point: TCP) from the current position to a programmed end position. To do this, the controller calculates the necessary angle differences for each axis.

The following system variables are used:

$VEL_AXIS[axis number] to program maximum axis specific velocities,

and

$ACC_AXIS[axis number] to program maximum axis specific acceleration rates.

The motions of the axes are **synchronized** in such a way (synchronous PTP) that all of the axes start and stop moving at the same time. This means that only the axis with the longest trajectory, the so-called leading axis, is actually moved with the programmed acceleration and velocity limits. All other axes move only with the velocity and acceleration rates necessary for them to reach the end point of the motion at the same moment, irrespective of the values programmed in **$VEL_AXIS[ ]** and **$ACC_AXIS[ ]**.

If individual components are omitted when the axis coordinates are entered, the robot only moves the axes that have been specified; the others do not change position. For example with

```
PTP {A3 45}
```

only axis 3 is moved to 45°. Note that the angle specifications in the PTP instruction are absolute values (the robot does not, therefore, rotate the axis 45° further, but to the absolute axis position of 45°).

It is also possible to omit individual components of the geometrical specification when entering the end point using Cartesian coordinates. The instruction

```
PTP {Z 1300, B 180}
```

moves the TCP in the direction of the Z axis to the absolute position 1300 mm and « tilts » the TCP by 180°.

For relative motion, the instruction **PTP_REL** is used.

ii) Continuous Path (CP) motions

Velocity and acceleration

Unlike with PTP motions, it is not just start and end positions that are predefined in the case of continuous-path motions. Additionally, movement of the TCP along **a linear or circular path** between these points is also required.

The velocities and rates of acceleration to be entered do not relate any longer, therefore, to the individual axes, but to the motion of the TCP. The TCP is thereby moved at a precisely defined velocity and acceleration, see system variables:

$VEL.CP and $ACC.CP

(see details at §3.3.1, p. 68-69 of document KR C2 / KR C3 – Expert Programming).

Orientation control
If the orientation in space of the tool is to change during the path motion, the orientation control mode can be set using the system variable $ORI_TYPE. The value of this variable remains valid until a new value is assigned or the program is reset. See details at §3.3.2, p. 70 of document KR C2 / KR C3 – Expert Programming.

Linear motions
In the case of a linear motion, a straight line is realized from the current position (the last point programmed in the program) to the position specified in the motion command.

A linear motion is programmed using **LIN** or **LIN_REL** keywords in connection with the specification of the end point, i.e., analogous to PTP programming. The end position for linear motions is entered with Cartesian coordinates. Only the data types **FRAME** or **POS** are thus permissible.

N.B.: In the case of linear motions, the status of the end point is the same as that of the start point. Specification of Status (S) and Turn (T) for an end point of the data type POS will thus be ignored. A PTP motion (e.g. HOME run) with complete position specification (coordinate specification including status) must therefore be programmed before the first LIN instruction.

```
DEF LIN_BEW ()
;------- Declaration section -------
EXT BAS (BAS_COMMAND: IN, REAL: IN)
DECL AXIS HOME  ; Variable HOME of type AXIS

;----------- Initialization ----------
BAS (#INITMOV, 0) ; Initialization of velocities,
; accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----------- Main section ----------
PTP HOME  ; BCO run
PTP {A5 30}

; Linear motion to the specified position, the orientation
; is continuously changed to the end orientation
LIN {X 327,Y 271,Z 369,A -112,B 0,C -176}

; Linear motion in the Y-Z plane, S and T are ignored
LIN {POS: Y 50,Z 400,A 0,S 2,T 35}

; Linear motion to the specified position, the orientation
; is not changed
$ORI_TYPE=#CONSTANT
LIN {FRAME: X 200,Y -200,Z 100,A 23,B 230,C -90}

; The orientation is still not changed
LIN {FRAME: Z 300,A 90,B 0,C 0}

; Relative motion along the X axis
LIN_REL {FRAME: X 170}

PTP HOME
END
```

Circular motions
To define a circle or arc in space unambiguously, three points are needed which are different from one another and do not lie on a straight line.

The **start point** of a circular motion is again formed, as with PTP or LIN, by the current position.

In order to program a circular motion with the instructions **CIRC** or **CIRC_REL**, therefore, an **auxiliary point** must be defined in addition to the **end point**. When the controller calculates the motion path, only the translational components (X, Y, Z) of the auxiliary point are evaluated. Depending on the orientation control mode, the orientation of the TCP either changes continuously from the start point to the end point or remains constant.

```
DEF CIRC_BEW ( )
;------- Declaration section -------
EXT BAS (BAS_COMMAND :IN,REAL :IN )
DECL AXIS HOME

;----------- Initialization ----------
BAS (#INITMOV,0 )  ; Initialization of velocities,
; accelerations, $BASE, $TOOL, etc.
HOME={AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}

;----------- Main section ----------
PTP HOME ;BCO run
PTP {POS: X 173,Y -239,Z 487,S 'B010',T 'B001010'}

; Space-related variable orientation control (default setting)
CIRC {X 278,Y -304,Z 425},{X 372,Y -131,Z 491,A -164,B 40,C -148}

; Space-related constant orientation control
; End point defined by angle specification
$ORI_TYPE=#CONSTANT
CIRC {X 384,Y 108,Z 493,A -127,B 40,C -148},{X 244,Y 306,Z 334,A -100,B
15,C -165}, CA 90.0

; Path-related constant orientation control
; End point defined by angle specification (backwards)
$CIRC_TYPE=#PATH
CIRC {X 380,Y 151,Z 484},{X 418,Y -221,Z 490},CA 80

$ORI_TYPE=#VAR

; Path-related variable orientation control
CIRC {X 368,Y 146,Z 364},{X -31,Y 414,Z 288,A -160,B 8,C 146}

; Relative circular motion
CIRC_REL {X -50,Y 50},{X 0,Y 100}

PTP HOME
END
```

2.8 Program execution control

As in most programming languages, it is possible to carry out connection instructions, loops or standby instructions.

i) Program branches

GOTO

The simplest form of program branch is the unconditional **GOTO** command. This is executed in every case without having to fulfill any conditions. By means of the statement

GOTO MERKER

the program pointer moves to the position MERKER. However, this position must also be defined using the format

MERKER:

somewhere in the program.

IF

The structured **IF** statement allows instructions to be formulated conditionally with a choice of two alternatives. The general form for these instructions is:

IF Execution condition THEN
  Instructions
ELSE
  Instructions
ENDIF

The execution condition is a Boolean expression. If the execution condition is fulfilled, the **THEN** block is executed. If it is not fulfilled, the **ELSE** block can be either executed or dispensed with. If it is dispensed with, the branch is left immediately.

SWITCH

If more than 2 alternatives are available, this can either be programmed using a nested IF construction or, much more conveniently, using the **SWITCH** multi way branch.

The SWITCH statement is a selection instruction for various program branches. A selection criterion is assigned a certain value ahead of the SWITCH statement. If this value agrees with a block identifier, the corresponding branch is executed and the program jumps straight to the ENDSWITCH statement without taking subsequent block identifiers into consideration. If no block identifier agrees with the selection criterion, the DEFAULT statement block is executed, if there is one. Otherwise, the program resumes at the instruction after ENDSWITCH statement.

```
;INT variable PROG_NO by the PLC

SWITCH PROG_NO
CASE 1              ;if PROG_NO=1
     PART_1()
CASE 2              ;if PROG_NO=2
     PART_2()
     PART_2A()
CASE 3,4,5          ;if PROG_NO=3, 4 or 5
     $OUT[3]=TRUE
     PART_345()
DEFAULT             ;if PROG_NO<>1,2,3,4,5
     ERROR_UP()
ENDSWITCH
```

ii) Loops

FOR

Counting loops are executed until a counting variable either exceeds or falls below a certain end value by counting up or down. The **FOR** statement is available for this. Using

FOR Counter = Start TO End STEP Increment
  Instructions
ENDFOR

a specified number of runs can be very clearly programmed.

Enter integer type expressions as **Start** and **End** values for the counter. The expressions are evaluated once at the start of the loop. The INT variable **Counter** (which must be declared in advance) is preset with the start value and then increased or decreased by the programmed increment after each loop execution.

The **increment** must be neither a variable nor zero. If no increment is specified, it has the default value 1. Negative values can also be used for the increment.

There must be an ENDFOR statement for every FOR statement. After completion of the last loop execution, the program is resumed with the first instruction after ENDFOR.

```
DEF FOR_PROG()
…
INT I,J
INT ARRAY[10,6]
…
FOR I=1 TO 6
    $VEL_AXIS[I] = 100 ;all axis velocities to 100%
ENDFOR
…
FOR I=1 TO 9 STEP 2
    FOR J=6 TO 1 STEP -1
        ARRAY[I,J] = I*2 + J*J
        ARRAY[I+1,J] = I*2 + I*J
    ENDFOR
ENDFOR
;I now has the value 11, J the value 0
…
END
```

WHILE

The **WHILE** loop requests an execution condition at the start of the repetition. It is a rejecting loop, because it will not run a single time unless the execution condition is satisfied from the outset. The WHILE loop has the following syntax:

```
WHILE Execution condition
        Instructions
ENDWHILE
```

The **Execution condition** is a logic expression which can be: a Boolean variable, a Boolean function call, or a logic operation with a Boolean result.

The instruction block is executed if the logic condition has the value TRUE, i.e., the execution condition is fulfilled. If the logic condition has the value FALSE, the program is resumed with the next instruction after ENDWHILE. Each WHILE statement must therefore be ended with an ENDWHILE statement.

REPEAT

The counterpart of the WHILE loop is the **REPEAT** loop. With REPEAT, the termination condition is not checked until the end of the loop. For this reason, REPEAT loops always run once, even if the termination condition is already fulfilled before the loop begins.

```
REPEAT
    Instructions
UNTIL Termination condition
```

The **Termination condition**, similarly to the execution condition of a WHILE loop, is a logic expression which can be: a Boolean variable, a Boolean function call, or a logic operation with a Boolean result.

LOOP
You can program endless loops using the **LOOP** statement:

```
LOOP
    Instructions
ENDLOOP
```

The repeated execution of the instruction block can only be terminated using the EXIT statement.

Any loop can be terminated prematurely by using the EXIT statement. By calling EXIT within the instruction block of a loop, the loop run is immediately terminated and the program resumed after the ENDLOOP statement.

```
DEF EXIT_PRO()
PTP HOME
LOOP                        ; Start of endless loop
   PTP POS_1
   LIN POS_2
   IF $IN[1] == TRUE THEN
      EXIT                  ; Terminate when input 1 set
   ENDIF
   CIRC HELP_1,POS_3
   PTP POS_4
ENDLOOP                     ; End of endless loop
PTP HOME
END
```

iii) WAIT instructions

Using the **WAIT** statement, you can cause the program to stop until a certain situation arises. A distinction is made between waiting for the occurrence of a certain event and waiting for a wait time to elapse.

Waiting for an event
Using the instruction:

WAIT FOR condition

you can stop program execution until the event specified under condition arises.

- If the logic expression condition is already TRUE when WAIT is called, program execution is not stopped (an advance run stop is triggered, however),
- If condition FALSE, program execution is stopped until the expression takes the value TRUE.

Example: The condition WAIT FOR can be applied to the variables $IN[ ], $OUT[ ], $CYCFLAG[ ], $TIMER_FLAG[ ] and $FLAG[ ].

Wait times
The WAIT SEC statement allows wait times to be programmed in seconds:

WAIT SEC time

Time is an arithmetic REAL expression which can be used to specify the number of seconds for which program execution is to be interrupted. If the value is negative, the program does not wait.

Examples:

```
WAIT SEC 17.542
WAIT SEC TIME*4+1
```