Kemro

KeMotion
KAIRO Language Reference
Programming manual V 3.16

Translation of the original manual

KEBA®
Automation by innovation.

Document: V 3.16 / Document No.: 1008721
Filename: KeMotion3_KAIRO_LanguageReference_en.pdf
Pages: 112

KEBA AG Headquarters: Gewerbepark Urfahr, 4041 Linz, Austria, Phone: +43 732 7090-0,
Fax: +43 732 7309-10, keba@keba.com

For information about our subsidiaries please look at www.keba.com.

Record of Revision

Version	Date	Change in chapter	Description	changed by
2.20	01.05.2010	-	adopted forKeMotion3 2.20	nmr
2.30	21.10.2010		added chapter 'Extending KAIROinstruction set'	nmr
2.40	22.09.2011		Kemro_Automation 2.40	nmr
2.50	10.04.2012		Kemro_Automation 2.50	pend
2.55	23.01.2013		KeMotion3 2.55	nmr
2.60	06.06.2013		KeMotion3 2.60	nmr
2.62	14.04.2014		KeMotion3 2.62	nmr
2.64	12.11.2014		KeMotion3 2.64	nmr
2.66	21.05.2015		KeMotion3 2.66	nmr
3.04	12.08.2015		KeMotion3 3.04	nmr
3.06	28.11.2016		KeMotion3 3.06	nmr
3.10	12.05.2017		KeMotion3 3.10	nmr
3.10a	07.06.2017		KeMotion3 3.10a	nmr
3.12	13.11.2017		KeMotion3 3.12	nmr
3.14	28.05.2018		KeMotion3 3.14	nmr
3.16	13.11.2018		KeMotion3 3.16	nmr

Table of contents

1 Introduction

1.1 Purpose of the document

This document describes how robot programs for KeMotion3 r-series can be written by the end-user. We assume an up-and-running controller (see: „User Manual Robot Controller“) as well as some knowledge of the basics of robotics (see: „Basics of Robotics“).

1.2 Preconditions

This document contains information for persons with the following skills:

Target group	Knowledge and skills pre-requirement
Project engineer	Basic technical training (University of Applied Science/University level, engineering degree or corresponding professional experience). Knowledge in: ● working mode of a PLC, ● current valid safety regulations, ● the application.
Programmer	Technical training (University of Applied Science, engineering degree or corresponding professional experience). Knowledge about: ● method of operation of a PLC, ● current valid safety regulations, ● programming of PLC (IEC61131).
Machine operator	In-plant training Knowledge about: ● current valid safety regulations, ● production process ● filling out display formulas with touch screen
Start-up technician	Basic technical training (technical college, engineer training or corresponding professional experience). Knowledge about: ● current valid safety regulations, ● method of operation of the machine or system, ● fundamental functions of the application, ● system analysis and troubleshooting, ● setting options on the operating devices.

Target group	Knowledge and skills pre-requirement
Service technician	Basic technical training (technical college, engineer training or corresponding professional experience). Knowledge about: ● method of operation of a PLC, ● current valid safety regulations, ● method of operation of the machine or system, ● diagnostic options, ● systematic fault analysis and remedial action.

1.3 Intended use

This software has been developed to control industrial robots and electrical servo drives. It must not be used for applications other than those described in this document, and it must exclusively be used for recommended or approved target systems.

WARNING!

The KeMotion3-system has not been designed for safety-relevant control applications such as stopping in the case of emergency or safely reduced speed.

The KeMotion3-system complies only with safety category B PL a according to EN ISO 13849-1. It is therefore inadequate for implementation of safety functions to protect humans.

For safety-relevant control applications or personnel protection additional external protection measures must be taken, ensuring a safe operation condition also in case of failures.

1.4 Notes on this document

This manual is an integral part of the product. It is to be retained over the entire life cycle of the product and should be forwarded to any subsequent owners or users of the product. For end user necessary safety information and information must be integrated in the instruction manual for end users in the specific national language by the engine builder or the system provider.

This documentation must be legible and available to the specified persons and must be read and understood from them.

1.4.1 Contents of document

- KAIRO Language structure
- KAIRO-Reference of functions, variables, types, and units

1.4.2 Not contained in this document

- Configuration and commissioning of the KeMotion3-System
- Hardware components of the KeMotion3-System
- Programming of the integrated PLC
- Interface definitions to extend the KAIRO- instruction set
- Special technology units
- System diagnosis

2 Safety notes

2.1 Representation

At various points in this manual, you will see notes and precautionary warnings regarding possible hazards. The symbols used have the following meaning:

DANGER!

indicates an imminently hazardous situation, which will result in death or serious bodily injury if the corresponding precautions are not taken.

WARNING!

indicates a potentially hazardous situation, which can result in death or serious bodily injury if the corresponding precautions are not taken.

CAUTION!

means that if the corresponding safety measures are not taken, a potentially hazardous situation can occur that may result in slight bodily injury.

Caution

means that damage to property can occur if the corresponding safety measures are not taken.

ESD

This symbol reminds you of the possible consequences of touching electrostatically sensitive components.

Safety information

Describes important safety-related requirements or informs about essential safety-related coherences.

Information

Identifies practical tips and useful information. No information that warns about potentially dangerous or harmful functions is contained.

3 Language definition

3.1 Introduction

KAIRO is a programming language designed for machine operators to implement user programs. The language has been kept simple deliberately to make programming machine sequences easy - without the necessity of a fundamental software engineering education.

The programmed code is being stored in files with the pre-defined file extensions "tip" and "tid". KAIRO programs ("tip"-files) contain flows, whereas KAIRO data files ("tid"-files) contain the corresponding data, e.g. position data.

3.2 File structure

3.2.1 Projects

A project is a directory with the file extension "tt", e.g. "project1.tt". It contains KAIRO files which may be organized in further subdirectories. These subdirectories only serve for organizational purposes - they have no logical meaning. All KAIRO files of a project and all subdirectories of a project are elements of the project.

If a program shall be executed the whole project the program belongs to must be loaded as a whole.

There are two particular projects named "_system.tt" and "_global.tt". These projects are automatically activated during start-up of the control software. Objects of _system are visible throughout all projects, objects of _global are visible in all projects except for _system. Objects of any other project are visible only in the own project.

3.2.2 End-user programs

End-user programs are identified by their "tip" extension. An end-user program only contains instructions. When a program is executed its instructions are subsequently processed.

A program can be called by another program via its file name.

3.2.3 End-user data files

End-user data files are identified by their "tid" extension. An end-user data file contains declarations of static variables belonging to the end-user program of the same name. Inside the program a variable is accessed via its name, otherwise via its program name, e.g. "program1.x" for variable "x" of program "program1".

Every project may contain a unique file named "_globalvars.tid". Any variable declared in this file is global, i.e. accessible via its name throughout the whole project

3.3 Table of instructions

This table summarizes all pre-defined names and symbols of KAIRO. For detailed information please follow the corresponding links.

Movement

	PTP	Chapter 5.1
	Lin	
	Circ	
	PTPRel	
	LinRel	
	MoveRobotAxis	
	StopRobot	
	PTPSearch	
	LinSearch	
	WaitIsFinished	
	WaitJustInTime	
	WaitOnPath	
Homing		
	RefRobotAxis	Chapter 5.1.14
	RefRobotAxisAsync	
	WaitRefFinished	

Settings

	Dyn	Chapter 5.2
	DynOvr	
	Ovl	
	Ramp	
	RefSys	
	ExternalTCP	
	Tool	
	OriMode	
	Workpiece	
	SetJointMonitoringLimits	see User's Manual WorkspaceMonitoring
	SetGuardMonitoringLimits	

System Functions

	… := … (Assignment))	Chapter 5.3
	// … (Comment))	
	WaitTime	
	Stop	
	Info	
	Warning	
	Error	
	Random	
Time Measurement		
	CLOCK.Reset	Chapter 5.3
	CLOCK.Start	
	CLOCK.Stop	
	CLOCK.Read	
	CLOCK.ToString	
	TIMER.Start	
	TIMER.Stop	
	SysTime	
	SysTimeToString	
Mathematical Functions		
	SIN, COS, TAN, COT	Chapter 5.3
	ASIN, ACOS, ATAN, ACOT	
	ATAN2	
	LN	
	EXP	
	ABS	
	SQRT	
Bitwise Operations & Conversions		
	SHR	Chapter 5.3
	SHL	
	ROR	
	ROL	
	SetBit	
	ResetBit	
	CheckBit	
	STR	
IEC Data Interface		see User's Manual Teach-Control Data Interface

Flow Control

	CALL …	Chapter 5.4
	WAIT …	
	SYNC.Sync	
	IF … THEN … END_IF	
	ELSIF … THEN	
	ELSE	
	WHILE … DO … END_WHILE	
	LOOP … DO … END_LOOP	
	RUN …	
	KILL …	
	RETURN	
	LABEL …	
	GOTO …	
	IF … GOTO …	

Signals

	WaitBool	Chapter 5.5
	WaitBit	
	WaitBitMask	
	WaitLess	
	WaitGreater	
	WaitInside	
	WaitOutside	
	BOOLSIGOUT.Set	
	BOOLSIGOUT.Pulse	
	BOOLSIGOUT.Connect	

Technology Options

Triggers		
	OnDistance	Chapter 5.6
	OnParameter	
	OnPlane	
	OnPosition	
Workspace Monitoring		see User's Manual WorkspaceMonitoring
Tracking		see User's Manual Tracking
Palletizing		see User's Manual Palletizing

Data Types

Base Types		
	BOOL	Chapter 6.1
	SINT, INT, DINT, LINT	
	USINT, UINT, UDINT, ULINT	
	BYTE, WORD, DWORD, LWORD	
	REAL, LREAL	
	STRING	
	ANY	
Positions		
	POSITION_	Chapter 6.2
	AXISPOS_	
	AXISPOS	
	AXISPOSREF	
	ROBAXISPOS	
	AUXAXISPOS	
	CARTPOS_	
	CARTPOS	
	CARTPOSTURN0	
	CARTPOSREF	
	ROBCARTPOS	
	POLARPOS	
	DISTANCE_	
	AXISDIST	
	CARTDIST	
	CARTFRAME	
Reference Systems and Tools		

	REFSYS_	Chapter 6.3
	WORLDREFSYS	
	CARTREFSYS_	
	CARTREFSYS	
	CARTREFSYSEXT	
	CARTREFSYSAXIS	
	CARTREFSYSVAR	see User's Manual Tracking
	EXTERNALTCP	Chapter 6.3
	TOOL_	
	TOOL	
	TOOLSTATIC	
	TOOLVAR	see User's Manual Tracking
	WORKPIECE	Chapter 6.3
Dynamics and Overlapping		
	OVERLAP_	Chapter 6.4
	OVLREL	
	OVLABS	
	OVLSUPPOS	
	DYNAMIC_	
	DYNAMIC	
	PERCENT	
	PERC200	
	GUARDMONITORINGLIM-ITS	see User's Manual WorkspaceMonitoring
	JOINTMONITORINGLIMITS	
Signals		
	BOOLSIGIN	Chapter 6.5
	BOOLSIGOUT	
System and Technology		

	ROBOTDATA	Chapter 6.6
	CLOCK	
	TIMER	
	SYNC	
	AREA	see User's Manual WorkspaceMonitoring
	GUARDMASK	
	PALLET	see User's Manual Palletizing
	CONVEYOR	see User's Manual Tracking
	TRACKOBJECT	
	WAITRESULT	
	EUROMAP	see User's Manual EUROMAP

3.4 Language structure

3.4.1 File format

End-user files are human-readable text files. Instructions and declarations are separated by line breaks. In general the language is case-sensitive.

3.4.2 Designators

Designators serve to identify projects, programs, blocks, routines, variables, constants, and types. A designator is a sequence of letters, numbers, and the symbol "_". It must start with a letter or the symbol "_". The length of a designator is limited to 32 characters.

Elements of KAIRO whose name refer to the file system (i.e. projects and programs) are generally treated in lower case. When accessing such elements case sensitivity is neglected, e.g. a file "test.tip" may be referred to as "TEST" or "Test".

Pre-defined names of KAIRO instructions or data types must not be used as designators for projects, programs, or variables (see chapters 5 and 6). Generally, a new object must not be given the name of another object visible at this point. Consequently, a variable must not bear the name of a global variable, and a program must not bear the name of the corresponding project.

Examples of valid designators:

```
i
index
name
nameLen
MAX_VALUE
_rootHdl
```

Also see about this

- Instructions [} 32]
- Data types [} 85]

3.4.3 Constant numbers

KAIRO distinguishes between integer numbers and floating point numbers. Integers may be expressed in decimal, dual, or hexadecimal representation.

Examples of valid integer numbers:

```
100     -100     integer numbers (decimal)
2#1010     -2#1010     integer numbers (dual)
16#1ABF     -16#1ABF     integer numbers (hexadecimal)
```

Floating point numbers consist of an integer part, a decimal point, and at least one trailing digit. Floating point numbers in exponential representation consist of an integer part, an optional decimal point with at least one trailing digit, and the exponent.

Examples of valid floating point numbers:

```
1.01     1.99E4 1.99e+8     1e-8
```

3.4.4 Strings

Strings are delimited by the character ". They may contain all printable characters. The length of a string is limited to 255.

Operators and delimiters

```
"Hello World"
" "
```

3.4.5 Operators and Delimiters

Operators are being used in arithmetic expressions and define in what way variables and constants have to be interrelated. Operators are predominantly represented by special characters and key words.

Arithmetic operators

```
+     Addition
-     Subtraction
*     Multiplication
 /     Division
MOD     MOD
    (Definition: if f := x MOD y, then x = i * y + f, where i is an integer,
f has the same sign as x, and the magnitude of f is less than the magnitude
of y)
```

Logical operators

These operators may be applied to integer numbers, bit fields, and logical values. They work bitwise in integer numbers and bit fields.

```
AND     AND operation
OR     OR operation
```

```
XOR     Exclusive-Or operation
NOT     negation
```

Relational operators

```
<     less
<=     less or equal
=     equal
<>     unequal
>=     greater or equal
>     greater
```

Other operators

```
.     dot operator, e.g. vector.x
[]     index, e.g. list[index]
()     parentheses for enumerations, e.g. for parameter and
     initialization lists; parentheses for arithmetical expressions
```

Delimiters

```
:=     assignment operator
 : D    colon
 ,     comma
```

3.4.6 Comments

Line comments are preluded by *//*.

Example of a valid comment:

```
// Comment to the end of line
```

3.5 Declaration of variables

To declare a variables means to define its name and data type. The variable declaration is contained in data files. The colon separates the name from the data type.

Array and reference declarations (ARRAY and MAPTO) are supported.

Examples of valid variable declarations:

Declaration (in *.tid):

```
k : REAL
mk99 : BOOL
name : STRING
P11 : AXISPOS
// array declaration:
// array of three REAL elements, indices 0, 1, and 2:
v : ARRAY [3] OF REAL
// array of three STRING elements, indices 1, 2, and 3:
as : ARRAY [1..3] OF STRING
// 3 by 3 matrix with 9 DINT elements:
Matrix1 : ARRAY [3] OF ARRAY [3] OF DINT
// or:
Matrix2 : ARRAY [3, 3] OF DINT
```

```
// reference declarations:
m1 : MAPTO BOOL
m2 : MAPTO AXISPOS
```

Usage (in *.tip):

```
 // KAIROVersion 2.20
P11.a1 := v[0]
Matrix1[2][2] := P11.a2
```

Information

All KAIRO variables are initialized with 0. Enumeration variables (e.g. of type RAMPTYPE) are initialized with the first value of the enumeration. STRING variables are initialized with the empty string (""). Boolean variables are initialized with FALSE. Reference variables are initialized with "not mapped ".

3.5.1 Explicit initialization

Variables may explicitly be initialized upon declaration (contrary to the implicit initialization with 0).

Initializing basic types

The initial value must be stated in the tid-file where the variable is declared. REAL values may be initialized using integer constants.

Examples of valid initializations:

Declaration (in *.tid):

```
i : DINT := 1 // i = 1
pi : REAL := 3.1415 // pi = 3.1415
r : REAL := 10 // r = 10.0
mk99 : BOOL := TRUE // mk99 = TRUE
text : STRING := "HELLO" // text = "HELLO"
```

Initializing composed types

Variables of composed types may also be initialized. However, such complicated initializations will normally not be left to the end-user, but rather automatically be generated by the runtime system upon a request of the dialog system TeachView (based on current values). For the sake of completeness we will explain the syntax of such an initialization.

For composed types (arrays, structs, or blocks) an initialization list must be provided, consisting of comma-separated values delimited by parentheses. These values are subsequently assigned to the individual elements of the array or struct.

In case there are fewer initial values than elements all remaining elements maintain their current initialization

A list may also contain voids, also in this case the corresponding elements retain their current initialization. References may be mapped upon initialization (using MAP or MAPX).

When an instance of an inherited block is initialized the elements of the base block are placed prior to those of the inherited block.

When a struct is initialized a list of assignments may be provided instead of a list of values (e.g. "x := 1"). In this case the left part of each assignment must directly refer to an element of the struct, and such a list must not contain voids or direct values.

Examples of struct initialization:

Declaration (in *.tid):

```
a : ARRAY [5] OF DINT := (1, 2, , 4) // -> (1,2,0,4,0)
p : AXISPOS := (a1 := 1.1, a3 := 2.2) // ->
(1.1,0,2.2,0,0,0,0,0,0)
```

3.6 Arithmetic expressions

An arithmetic expression describes a value which is of a certain data type. Such an expression may contain literal constants, constants, variables, built-in functions, and user-defined functions. The members of an arithmetic expressions are interrelated by operators.

Literal constants are numbers or strings. Constants, variables, and functions have names. Elements of a struct variable are referred to by their element name separated from the variable name by a dot. Array elements are indexed using brackets (number of array element, enclosed by brackets).

Example:

```
// in test.tid
a : REAL
f : REAL
 // in test.tip
 ...
// Assigning an expression to the variable d:
d := 2 * SQRT (a) * f
```

3.6.1 Evaluation order

Operators in parentheses are evaluated prior to those outside. Apart from this the following evaluation order is used:

1.	. [](element access in structs and arrays)
2.	NOT
3.	* / MOD AND
4.	+ - OR XOR
5.	< <= = <> >= >

Operators of equal priority are evaluated from left to right.

Operands of the Boolean operators AND and OR are evaluated from left to right. If the result is already defined by the left operand evaluation of the right operand will be omitted (short-circuit evaluation).

3.7 Mapping

The mapping mechanism is used to define a variable as a reference to a certain data type and then connect it to an object of this data type. The reference variable is then said to be "mapped to the object". If the reference variable is part of an operation in fact the corresponding mapped object is being used. If a reference variable is used without an existing mapped object, an error is issued. .

Mapped objects may be variables or system variables. System variables have a unique name throughout the whole system.

A reference variable allocated as much memory as is required to store a reference to the mapped object. This memory size is independent from the type of the mapped object.

If a reference variable r shall later be mapped to a variable of type 'tType' it must be declared as `'r : MAPTO tType'`

If the reference variable shall now be mapped to the KAIRO variable x, the mapping is done by `'r := MAP(x)'`. MAP is a built-in function used to map KAIRO variables containing as argument the object to which it shall be mapped.

A system variables is mapped using the mapping `'r := MAPX(s)'`. MAPX is a built-in function used to map system variables containing as argument the name string of the system object to which it shall be mapped. In case no system object with the specified name exists MAPX tries to interpret the name string as internal static variable. The specified string may be the complete path of the structure tree (e.g. "_system.Robot1") or may refer to an object of the _system-project (e.g. "Robot1").

If all these trials fail the reference variable remains unmapped or an already existing mapping is cancelled. This implies that the mapping `'r := MAPX("")'` may be used to cancel an already existing mapping.

If a reference variable is read in fact the mapped variable is read. Likewise, if a reference variable is written to in fact the mapped variable is written to.

Example of mapping reference variables:

```
// in test.tid
x : MAPTO REAL
y : MAPTO REAL
z : MAPTO REAL
r : REAL
s : REAL
```

```
// in test.tip
x := MAP(r) // x is mapped to r
y := MAP(x) // equal to y := MAP(r);
x := MAP(s) // x is re-mapped to s
y := MAP(z) // runtime error, z has not yet been mapped!
```

Example of MAP and MAPX with string argument:

```
// in test.tid
s : STRING
r1 : MAPTO STRING
r2 : MAPTO BOOL
 // in test.tip
s := "SYSVAR_BOOL";
r1 := MAP(s); // r1 : string containing "SYSVAR_BOOL"
r2 := MAPX(s); // r2 : Boolean value of system variable
     // named "SYSVAR_BOOL"
r2 := TRUE; // system variable is set to TRUE
r1 := ""; // string s is deleted
```

Example of MAPX stating the name of an internal static variable:

```
// Program _globalvars.tid of project project.tt:
gX : REAL
// in test.tid of project project.tt
r1 : MAPTO REAL
/// in test.tip of project project.tt
r1 := MAPX("project.gX") // r1 is mapped to gX
```

3.8 Blocks

Blocks are real objects cast into software. A block summarizes properties (member variables) and functions (methods). The representation of a certain object by a variable is called a block instance.

Blocks are being used where several objects with identical properties and functions exist.

KAIRO defines a series of general blocks. Blocks cannot be used in programs, but instances of blocks. Before a block can be used it is necessary to create an instance of it which must then be initialized.

Example:

The hardware offers digital inputs. Each of these inputs has the same functions and differs only by its field bus address. A digital input can be set or not set.

The control software models digital inputs as a block. Each individual input is a block instance.

The members of this block are a reference to the field bus address (so that the input may be uniquely identified) and its current value (set or not set).

Furthermore, this block has a method used to wait for a desired value (set or not set). Before a digital input can be used in a program it must be initialized. In this case the reference to the hardware must be established.

Accessing member variables and member functions

Member variables and methods are accessed like elements of a struct. The name of the block instance is followed by the name of the member variable or method, separated by a dot.

Example:

```
DO49.Set(TRUE)     // set digital signal DO49
DO50.Pulse(500, TRUE)    // set digital signal DO50 for 500ms
IF DI51.val AND DI52.posEdge THEN ...
```

Set .. method to set the value of a digital signal

Pulse ..method to set the value of a digital signal for a certain time span

val .. member variable containing the current value

posEdge .. member variable which saves a positive edge in the signal

DO49, DO50 .. names of block instances of digital output signals (BOOL-SIGOUT)

DI51, DI52 .. names of block instances of digital input signals (BOOLSIGIN)

3.9 Attributes

3.9.1 CONST

When a variable is declared in a data file the end-user may use the attribute CONST to modify a type. CONST-variables are write-protected and cannot be directly modified be a KAIRO program. However, they may obtain a value through initialization.

Example:

```
// in tid-File
i: DINT CONST := 15

 // in tip-File
i := 1 // error: no assignment-target
```

3.9.2 READONLY

Using the READONLY attribute member variables of blocks or programs can be protected against modification from the outside. Reading access from outside, however, is permitted.

3.10 Conventions and limits

- case-sensitive
- maximum designator length: 32 characters

- maximum string length: 255 characters
- range of a string character: 8 Bit
- maximum number of routine parameters: 16
- maximum number of variables in a waiting condition: 32

4 Program structure

KAIRO programs consist of a series of instructions separated by line breaks. KAIRO supports the following instruction types:

- assignments
- branching: IF, IF .. GOTO, GOTO .. LABEL, RETURN
- loops: WHILE, LOOP
- routine execution: CALL, macro-call, RUN, KILL
- synchronization: WAIT
- deactivation of instructions: ##

4.1 Macro call

Routines that can be called in end-user programs are termed "macros". Macros may be called as subroutines, stating the routine name followed by the actual parameter list. The parameter list is delimited by parentheses, the parameters are separated by commas.

Actual parameters may be simple expressions such as literal constants, constants, variables, or variables combined with element accessing (struct or array elements), but no function calls or arithmetic expressions.

In general the types of formal and actual parameter must coincide, but there are exceptions to the rule:

- If the formal parameter has been declared as value parameter it is sufficient that the actual parameter type can be assigned to the formal parameter type.
- If the formal parameter has been declared as reference parameter to a block type, the type of the actual parameter may be a type inherited from the formal parameter type.
- If the formal parameter has been declared as non-modifiable reference parameter (CONST), it is sufficient that the actual parameter type is a subset of the formal parameter type (enumeration or sub-range type versus integer type).
- If the formal parameter is of type REAL integer constants are accepted as actual parameters.

Suited constants may be used for reference parameters; in this case a copy will be created.

Optional parameters may be omitted inside the actual parameter list and at its end.

Example of a macro call:

Declaration (in *.tid):

```
Pos1 : AXISPOS
dyn1 : DYNAMIC
```

Usage (in *.tip):

```
// KAIROVersion 2.20
PTP(Pos1, dyn1)    // optional parameter ovl is omitted
                    // DYNAMIC is inherited from DYNAMIC_
```

4.2 Disable-comment

In end-user programs instructions may be deactivated using the preluding 'disable'-comment '##'. The affected instructions will remain to be part of the execution structure, but will be missed out during the execution. If the instructions contain expressions composed of variables these variables will be referenced – despite the 'disable'-comment. Referenced variables are not deleted when a program is cleaned. Therefore, all data of such variables are retained.

Instructions spanning more than one line can only be deactivated as a whole. In case of an IF-instruction also all ELSIF-parts, ELSE and END_IF must be deactivated.

Example of disable-comment:

```
// in tip-File
PTP(P1)    //PTP is a macro of the robotics instruction set
PTP(P2)     // P1, P2, P3 are position variables
## PTP(P3)     // => disable comment, PTP(P3) will not be called,
     // but P3 remains referenced!
```

5 Instructions

Optional parameters

Some commands feature optional parameters. These are parameters that may be omitted. In this case the command will show a defined default behavior.

Optional parameters are enclosed by brackets in the parameter description and marked by the leading word „OPTIONAL“ in the command definition.

The default behavior (= behavior if no parameter is supplied) is also documented in the command and parameter description.

Example:

```
MacroX (
param1 : DINT
OPTIONAL param2 : REAL
 )
```

Parameter:

param1	must-parameter
[param2]	optional parameter (default: 2)

5.1 Robot movements

The commands of this section dictate the movements of the robot.

Overview

- PTP - Point-to-point motion command
- Lin - Linear motion command
- Circ - Circular motion command
- PTPRel - Relative point-to-point motion command
- LinRel - Relative linear motion command
- LinRelTCP - Relative linear motion command in TCP direction
- MoveRobotAxis - Movement of one single robot axis
- StopRobot - Stops the movement and discards commanded path
- PTPSearch - Point-to-point search movement until a digital input signal is set
- LinSearch - Linear search movement until a digital input signal is set
- WaitIsFinished - Waits until main-run has reached a certain position
- WaitJustInTime - Delays program execution as long as possible
- WaitOnPath - Waiting on path without delaying program execution

- Referencing (Homing)
  - RefRobotAxis - Referencing/Homing of an axis
  - RefRobotAxisAsync - Referencing/Homing of an axis (asynchronous)
  - WaitRefFinished - Wait until asynchronous referencing/homing is completed

5.1.1 General notes

Zero segments

Zero segments are motion commands whose path length is 0.

Therefore, a zero segment will occur if two successive motion commands with the same target position are sent to the robot. If zero segments are programmed the robot stops at the target position (overlapping is impossible as the overlapping radius goes to 0).

Information

It is impossible to overlap on zero segments, therefore the robot stops.

Optional parameters of motion commands

Dynamics and overlapping (dyn and ovl)

For all motion commands optional dynamics and overlap data may be specified. This data is valid only for the corresponding command.

If these parameters are not supplied pre-set values in the program will be used (see Settings).

5.1.2 PTP

Motion command using point-to-point interpolation.

```
PTP (
pos : POSITION_
OPTIONAL dyn : DYNAMIC_
OPTIONAL ovl : OVERLAP_
)
```

The PTP-motion command is a synchronous point-to-point command, specifying a target position and optional dynamics and overlapping data. All axes will start to move and arrive at their target position simultaneously. The TCP-movement results from the combination of single axis movements.

Parameters:

pos	target position
[dyn]	dynamics data
[ovl]	overlapping data

5.1.3 Lin

Motion command using linear interpolation.

```
Lin (
pos : POSITION_
OPTIONAL dyn : DYNAMIC_
OPTIONAL ovl : OVERLAP_
)
```

The Lin-motion command is a Cartesian command, specifying a target position and optional dynamics and overlapping data. The robot's TCP will move in a straight line from its initial position to the target position. If, in addition, a change of orientation is required, the robot's TCP will be moved from its initial orientation to the target orientation, according to the orientation interpolation type.

Information

Linear interpolation does not support changing the robot mode.

Parameters:

pos	target position
[dyn]	dynamics data
[ovl]	overlapping data

5.1.4 Circ

Motion command using circular interpolation.

```
Circ (
circPos : POSITION_
pos : POSITION_
OPTIONAL dyn : DYNAMIC_
OPTIONAL ovl : OVERLAP_
)
```

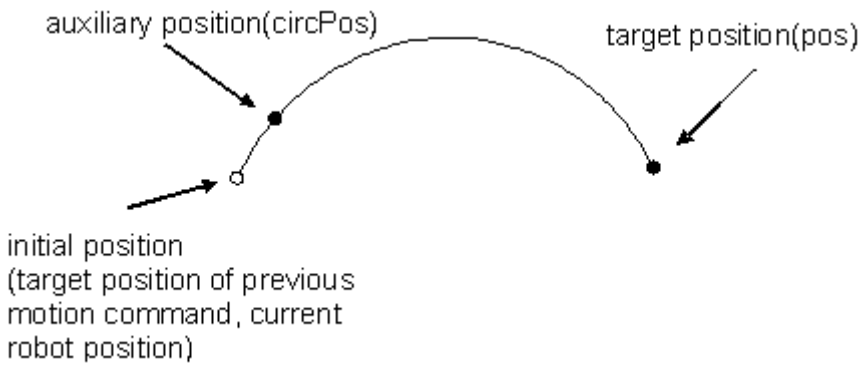


Fig. 5-1: Circular motion command

The circle is defined by the initial position, the auxiliary position `circPos`, and the target position pos. First the robot will move to the auxiliary position, then to the target position.

Please note the following conditions:

- For a full circle two Circ-commands must be executed.
- Initial, auxiliary, and target position must differ from another.
- The used type of orientation interpolation (command OriMode) is crucial. It must be set according to the application to avoid an unexpected robot movement.

Information

Circular interpolation does not support changing the robot mode.

The circle changes if the initial position is altered!

Parameters:

circPos	auxiliary position on circle
pos	target position
[dyn]	dynamics data
[ovl]	overlapping data

5.1.5 PTPRel

Relative motion command using point-to-point interpolation.

```
PTPRel (
dist : DISTANCE_
OPTIONAL dyn : DYNAMIC_
OPTIONAL ovl : OVERLAP_
)
```

The PTPRel-motion command is a relative point-to-point command, specifying a distance and optional dynamics and overlapping data. The distance is specified relative to the robot's initial position which may also depend on the previous motion command.

Parameters:

dist	distance of movement
[dyn]	dynamics data
[ovl]	overlapping data

5.1.6 LinRel

Relative motion command using linear interpolation.

```
LinRel (
dist : DISTANCE_
OPTIONAL dyn : DYNAMIC_
```

```
OPTIONAL ovl : OVERLAP_
)
```

The LinRel-motion command is a relative linear command, specifying a distance and optional dynamics and overlapping data. The distance is specified relative to the robot's initial position which may also depend on the previous motion command.

Parameters:

dist	distance of movement
[dyn]	dynamics data
[ovl]	overlapping data

5.1.7 LinRelTCP

Relative motion command using linear interpolation.

```
LinRelTCP (
dist : CARTDIST
OPTIONAL dyn : DYNAMIC_
OPTIONAL ovl : OVERLAP_
)
```

The LinRelTCP-motion command is a relative linear command, specifying a distance and optional dynamics and overlapping data. The distance is specified relative to the robot's **TCP**, which distinguishes this macro from the conventional "LinRel" macro.

Parameters:

dist	distance of movement
[dyn]	dynamics data
[ovl]	overlapping data

5.1.8 MoveRobotAxis

Motion command for one single robot axis.

```
MoveRobotAxis (
CONST axis : ROBOTAXIS
CONST pos : REAL
OPTIONAL dyn : DYNAMIC_
OPTIONAL ovl : OVERLAP_
)
```

Only the given axis moves, all other robot axes remain where they are.

Parameters:

axis	robot axis to be moved (see 5.1.15 Referencing (Homing)).
pos	target position
[dyn]	dynamics data
[ovl]	overlapping data

5.1.9 StopRobot

Stops the robot and discards the programmed path.

```
StopRobot( )
```

This command stops the robot on the path. All motion commands currently being executed are aborted.

The stop is executed using the pre-set maximum axis deceleration, whereby the stopping path is synchronized to the slowest axis to make sure the robot stays on the path. The override value is not considered.

Information

The pre-set axis deceleration may be reduced by the manufacturer to enable a softer stopping.

An example for using this command are sensor-controlled search movements (e.g. move towards a stack until a sensor reports contact).

StopRobot stops the movement, but not the program, hence successive motion commands start where the robot has come to halt after StopRobot.

5.1.10 PTPSearch

Search motion command using point-to-point interpolation (see 5.1.2 PTP). As soon as a trigger signal is set, the movement can be stopped and the trigger position can be stored.

```
PTPSearch (
CONST targetPos : POSITION_
CONST triggerSignal : ANY
OPTIONAL CONST dyn : DYNAMIC
OPTIONAL CONST trigger : EDGETYPE
OPTIONAL triggeredPos : POSITION_
OPTIONAL CONST stopRobot : BOOL
OPTIONAL CONST stopMode : STOPMODE_
) : BOOL
```

Function returns true if digital signal has been received.

Parameters:

targetPos	target position for search movement. If trigger signal is not received until reaching this position, the macro returns false.
triggerSignal	digital signal to wait for. If the signal is already set at the beginning, an error is set.
[dyn]	dynamics data
[trigger]	wait for a rising or a falling edge on the trigger signal (default: RISINGEDGE)
[triggeredPos]	robot position at time of trigger signal
[stopRobot]	stop robot after receiving trigger signal (default: TRUE)
[stopMode]	method to stop robot movement (default: CONTINUETRACKING) (see 5.3.4 Stop)

Parameter `triggerSignal` is used to provide a digital signal (type BOOL). To maintain compatibility with existing applications, it is currently implemented as `ANY`-parameter. Thus it is possible to use instances of the outdated block `DIN` with this macro.

Information

The movement has to be stopped or the trigger position has to be stored, or both. It is not possible to omit the parameter ► **triggeredPos** and set ► **stopRobot=FALSE** at the same time, doing so would result in an error.

EDGETYPE

Enumeration to define the target signal state transition.

Elements:

RISINGEDGE	Rising edge (transition from FALSE to TRUE)
FALLINGEDGE	Falling edge (transition from TRUE to FALSE)

5.1.11 LinSearch

Search motion command using linear interpolation (see 5.1.3 Lin). As soon as a trigger signal is set, the movement can be stopped and the trigger position can be stored.

```
LinSearch (
CONST targetPos : POSITION_
CONST triggerSignal : ANY
OPTIONAL CONST dyn : DYNAMIC
OPTIONAL CONST trigger : EDGETYPE
OPTIONAL triggeredPos : POSITION_
OPTIONAL CONST stopRobot : BOOL
OPTIONAL CONST stopMode : STOPMODE_
) : BOOL
```

Function returns true if digital signal has been received.

Parameters:

targetPos	target position for search movement. If trigger signal is not received until reaching this position, the macro returns false.
triggerSignal	digital signal to wait for. If the signal is already set at the beginning, an error is set.
[dyn]	dynamics data
[trigger]	wait for a rising or a falling edge on the trigger signal (default: RISINGEDGE)
[triggeredPos]	robot position at time of trigger signal
[stopRobot]	stop robot after receiving trigger signal (default: TRUE)
[stopMode]	method to stop robot movement (default: CONTINUETRACKING) (see 5.3.4 Stop).

Parameter `triggerSignal` is used to provide a digital signal (type BOOL). To maintain compatibility with existing applications, it is currently implemented as `ANY`-parameter. Thus it is possible to use instances of the outdated block `DIN` with this macro.

Information

The movement has to be stopped or the trigger position has to be stored, or both. It is not possible to omit the parameter ► **triggeredPos** and set ► **stopRobot=FALSE** at the same time, doing so would result in an error.

5.1.12 WaitIsFinished

This command synchronizes the robot's movement and the program execution.

```
WaitIsFinished (
OPTIONAL param : PERCENT
)
```

The program execution is interrupted until the robot has reached the programmed position. Optionally a parameter may be specified indicating a position on the last segment (segment parameter). In this case the program execution is interrupted until the robot has reached the specified parameter.

Parameter:

[pos]	Synchronization point on last segment, specified as segment parameter [%] (see 6.4.7 PERCENT)

5.1.13 WaitJustInTime

Command for approximate synchronization of the robot's movement and the program execution without negative influence on the path dynamics.

```
WaitJustInTime(
OPTIONAL CONST condition : WAITCONDITION
)
```

Program execution is interrupted until a new segment must be appended to the path in order to avoid braking to standstill. This command enables a last-minute evaluation of peripheral signals while maintaining the full path velocity (SPEEDDROP) or assuring overlapping (ASSUREOVL).

Parameter:

condition	Condition for the synchronization (default: SPEEDDROP)

WAITCONDITION

This enumeration defines which conditions should be met by the synchronization (unchanged path geometry or unchanged path dynamics).

Elements:

SPEEDDROP	Synchronization without negative influence on the path dynamics
ASSUREOVL	Synchronization without negative influence on the path geometry (overlapping is assured)

5.1.14 WaitOnPath

Waits for a certain period of time on the programmed path.

```
WaitOnPath (
timeMs : DINT
)
```

This macro halts the robot for a certain time. Although the program execution is not delayed.

Parameter:

timeMs	Time in [ms]

Example:

```
Lin(pos1) // move to pos1
WaitOnPath(100) // wait for 100ms without delaying program execution
Lin(pos2) // move to pos2
i := i + 1 // i is incremented while robot is still approaching
 // pos1 or it is waiting on the path
```

5.1.15 Referencing (Homing)

RefRobotAxis

Referencing/homing of a robot axis.

```
RefRobotAxis (
CONST axis : ROBOTAXIS
OPTIONAL CONST addMoveTarget : REAL
OPTIONAL CONST dyn : DYNAMIC_
) : BOOL
```

This command is used to reference/home a robot's axis.

The homing movement must be parameterized in the PLC. This macro starts homing according to the parameterization.

All homing methods specified by CANopen DS 402 are available.

Information

Not all homing modes are supported by all drives. Please find more information in the Programming Manual PLC/Robotics.

Parameters:

axis	robot axis to be homed

[addMoveTarget]	target position for movement after homing has been completed
[dyn]	dynamic used for optional movement (-> addMoveTarget) after homing has completed

If drive operation is not enabled when homing starts, homing cannot be executed and an error will be issued. After homing has been successfully executed motion commands may be used without any additional user interaction.

During homing both the start and the stop button maintain their function, hence homing may be interrupted by pressing the stop button, and may be continued by pressing the start button.

If the program counter is dislocated during homing or if homing is interrupted and the program unloaded homing will be finished and the operation mode of the drive will be reset to normal operation.

RefRobotAxisAsync

Asynchronous homing of a robot axis.

```
RefRobotAxisAsync (
CONST axis : ROBOTAXIS
OPTIONAL CONST addMoveTarget : REAL
OPTIONAL CONST dyn : DYNAMIC_
)
```

This macro allows a simultaneous homing of several robot axes. With this commands it is possible to define the reference position of an axis out of a motion program. This command does not wait until homing is completed, but returns immediately after the homing command has been sent. To find out whether homing has been completed the macro `WaitRefFinished` is used.

Information

Not all homing modes are supported by all drives. Please find more information in the Programming Manual PLC/Robotics.

Parameters:

axis	robot axis to be homed
[addMoveTarget]	target position for movement after homing has been completed
[dyn]	dynamic used for optional movement (-> addMoveTarget) after homing has completed

WaitRefFinished

Wait until all asynchronously started homing movements have been completed.

```
WaitRefFinished (
) : BOOL
```

This macro waits until all asynchronous homing movements have been finished or an error has occurred during a homing procedure.

If homing has been completed successfully, TRUE will be returned; otherwise FALSE will be returned.

ROBOTAXIS

Enumeration used to specify an axis.

Elements:

A1...6	robot axes
AUX1...3	auxiliary axes

5.1.16 Example program

The following example uses a good portion of the commands discussed in this chapter. It refers to a Cartesian handling with one rotational wrist axis.

Procedure:

Before processing starts it is checked if all axes are homed. If not, homing is initiated.

After homing the actual processing starts. The process consists of grabbing a part from a stack of variable height and depositing it on a certain deposit position:

- Move to home position
- Move along a straight line with a subsequent semi-circle
- Slow searching movement; as soon as a digital sensor fires, the searching movement is stopped
- Set an output to grab a part
- Starting from the point where the sensor stopped the robot a Cartesian relative movement shall be executed, followed by a relative axis movement to re-orient the wrist axis
- Then the movement is optimally synchronized with the process with respect to dynamics (WaitJustInTime), and the robot is moved to a certain position, depending on the state of a digital output
- Output is reset to deposit the part
- Move back to home position

Data (*.tid):

```
1: apos0 : AXISPOS := ()
2: dyn0 : DYNAMIC := (velAxis := 100, accAxis := 100, de-
cAxis := 100, jerkAxis := 100, vel := 50, acc := 250,
3: dec := 400, jerk := 35000, velOri := 90, accOri := 180,
decOri := 180, jerkOri := 1e+06)
4: dyn1 : DYNAMIC := (velAxis := 10, accAxis := 20, decAxis :=
```

```
20, jerkAxis := 20, vel := 10, acc := 100,
5: dec := 100, jerk := 5000, velOri := 90, accOri := 180,
decOri := 180, jerkOri := 1e+06)
6: ovl0 : OVLREL := ()
7: cpos0 : CARTPOS := (x := 100, c := -90)
8: cpos1 : CARTPOS := (x := 100, y := 100, c := -90)
9: cpos2 : CARTPOS := (x := 200, y := -500, c := -90)
10: circHelp : CARTPOS := (x := 150, y := 150, c := -90)
11: circEnd : CARTPOS := (x := 200, y := 100, c := -90)
12: adist0 : AXISDIST := (da4 := -180)
13: cdist0 : CARTDIST := (dz := 100)
14: cpos3 : CARTPOS := (x := 100, y := -100, z := 100, c :=
90)
15: cpos4 : CARTPOS := (x := 200, y := 200, z := 200, c := 90)
```

Program (*.tid):

```
1: // KAIROVersion 2.20
2: IF NOT RobotData.isReferenced THEN
3:     RefRobotAxis(A1)
4:     RefRobotAxis(A4)
5:     RefRobotAxisAsync(A2)
6:     RefRobotAxisAsync(A3)
7:     WaitRefFinished() // wait until A2 and A3 have been
8:     // homed
9: END_IF
10: PTP(apos0, dyn0, ovl0) // PTP to axis position with op-
tional
11: // parameters
12: PTP(cpos0) // PTP to Cartesian position
13: Lin(cpos1) // line Cart0 - Cart1
14: Circ(circHelp, circEnd) // circle to circEnd via circHelp
15: WaitIsFinished(80) // wait until 80% of circle is
16: // finished
17: LinSearch(cpos2, touchSensor, dyn1)
18: // searching movement circEnd - cpos2 with
19: // reduced dynamics, wait for sensor signal,f
20: // stop robot and discard rest of path
21: IO.gripperClose := TRUE // sets a digital output
22: LinRel(cdist0, dyn0) // line, leading 100mm in z-direc-
tion,
23: // starting from current position
24: // and specifying dynamics
25: PTPRel(adist0) // relative axis movement to re-orient
26: // wrist axis
27: WaitJustInTime() // dynamics-optimal synchronization
28: IF IO.workPosReady THEN // due to WaitJustInTime
29:     // the condition will be evaluated not before program
30:     // and movement are almost synchronous
31:     Lin(cpos3)
32: ELSE
33:     Lin(cpos4)
34: END_IF
35: WaitIsFinished() // wait until movement is finished
36: IO.gripperClose := FALSE // reset the output
37: MoveRobotAxis(A4, 0) // move wrist-axis to 0-position
38: PTP(apos0)
```

5.2 Settings

This group contains commands which modify the settings for all subsequent motion commands.

Dynamics commands are used to program velocity, acceleration, acceleration ramp, and jerk of the robot's axes and the TCP.

The overlap command influences the path behavior in the vicinity of the programmed positions.

RefSys and Tool have an effect on the path geometry, OriMode serves to set the kind of orientation interpolation.

Overview:

- Dyn = Velocity, acceleration and jerk settings for TCP and axes
- DynOvr = Dynamic-override settings
- Ovl = Overlap absolute settings in %
- Ramp = Acceleration ramp settings
- RefSys = Reference system settings
- ExternalTCP = External TCP settings
- Tool = Robot tool settings
- OriMode = Orientation-interpolation settings
- Workpiece = Work piece settings

5.2.1 Programming dynamics and overlapping

Dynamics and overlapping may be programmed in two ways: by pre-setting those using special commands or by specifying parameters when a motion command is sent, where passed parameters overrule pre-set values. When a program is loaded the pre-settings are taken from the robot configuration.

Example (using pre-set values):

```
Ovl (ovl1)      // set overlapping according to ovl1
Dyn (d0)      // set global dynamics
PTP (p1)      // move to p1 and p2 using dynamics d0 and overlap ovl1
PTP (p2)
Ovl (ovl2)      // set overlapping according to ovl2
Dyn (d1)      // change global dynamics
PTP (p3)      //move to p3 using dynamics d1 and overlap ovl2
```

Example (using parameters along with the motion command):

```
Dyn (d0)      // set global dynamics
PTP (p1, dyn1, ovl1)      // move to p1 using dyn1 and ovl1
Lin (p2, dyn2, ovl2)      // move to p2 using dyn2 and ovl2
PTP (p3)      // move to p3 using global dynamics d0
```

5.2.2 Relative and absolute dynamics

The dynamics of a PTP-movement is specified in percent of the axis limits. 100% means, that the velocity-dominant axis moves at 100% of its velocity limit.

The dynamics of the Cartesian movement is programmed in absolute values. Whenever axis limits would be violated the robot control automatically reduces the programmed dynamics such that the dynamics stays within the configured axis limits.

5.2.3 Effect of dynamics commands

Motion commands without dynamics parameters

Motion commands without dynamics parameterization obtain their dynamics by multiplication of steps 1, 2, and 3.

1	User-override (on handheld terminal)
2	DynOvr (%)
3	Dyn (abs)
	PTP () Lin, Circ ()

Step 1: Override

The override is equal to the override set on the handheld terminal (keys V+, V-) or the values set by an external source. The override cannot be modified by programmed commands. It is specified in percent.

Information

The override does not affect the path geometry!

Step 2: Dynamics override

Using the dynamics override **DynOvr** the programmed dynamics may be lowered globally in the program. It is specified in percent.

Step 3: Dynamics

Using the command **Dyn** the dynamics for PTP, for Cartesian position, and Cartesian orientation may be separately programmed. For PTP the dynamics is specified in percent, whereas Cartesian dynamics is specified in absolute values (mm/s, mm/s², mm/s³ or °/s, °/s², °/s³).

Motion commands with dynamics parameters

For motion commands with own dynamics parameters the programmed dynamics is multiplied by the override and by the currently set dynamics override.

1	User override (on handheld terminal)
2	DynOvr (%)
	PTP () Lin, Circ ()

Example (Handling application):

The absolute value of the path velocity is unessential. The system integrator configures the default dynamics in terms of maximum values. In the program the dynamics is programmed exclusively in percent.

```
DynOvr (50%)     // pre-set to 50% globally
PTP (p1)      // 50% max. axis velocity
Lin (p3)      // 50% of Cartesian limits
DynOvr (100%)     //pre-set to 100% globally
Lin (p4)      // 50% of Cartesian limits
PTP (p5)      // 50% max. axis velocity
```

Example (Processing task):

The absolute value of the path velocity is essential. The dynamics is entirely specified in the program and passed as parameter along with the corresponding motion command.

```
Dyn (dyn)     //global pre-set for all motion commands
PTP (p1)      // PTP using pre-set dyn
dyn Lin (p3)     // Lin using pre-set dyn
dyn PTP (p5, dyn1)     // dyn1 only used for this motion command
Lin (p4, dyn1)     // sdyn1 only used for this motion command
```

5.2.4 Dyn

Specifies the full dynamics data

```
Dyn (
dyn : DYNAMIC_
)
```

Velocity, acceleration, deceleration, and jerk for all subsequent movements are simultaneously set. The values for PTP must be specified as a percentage of the axis limits, Cartesian values must be provided in absolute values.

Parameter:

dyn	Dynamics

5.2.5 DynOvr

Specifies the dynamics override.

```
DynOvr (
ovr : PERCENT
)
```

This command scales the dynamics for all subsequent motion commands and dynamics macros. It is subject to the user-override set at the handheld terminal.

Information

The override does not affect the path geometry!

Parameter:

DynOvr	Override in % (see 6.4.7 PERCENT)

5.2.6 Ovl

The overlap parameterization defines the movement in the vicinity of the programmed positions.

```
Ovl (
ovl : OVERLAP_
)
```

Depending on the specified data relative or absolute parameterization is used.

Relative overlap parameterization (parameter ovl of type OVLREL) defines an overlap degree using a percentage.

Absolute overlap parameterization (parameter ovl of type OVLABS) specifies the maximum permitted deviations from the programmed target position.

Parameter:

ovl	Overlapping data

Relative and absolute programming

Relative overlap parameterization:

A degree of overlapping is defined by a percentage value. 100% means time-optimized overlapping that utilizes the acceleration reserves of the axes at lowest-possible deviation from the programmed target position.

Values less than 100% will lead to smaller deviations, but prolong the cycle time. Values greater than 100% cause larger deviations and smaller axis accelerations at nearly a constant cycle time.

Absolute overlap parameterization

The maximum permitted deviations from the programmed target position are specified. This requires several parameters:

- Distance of TCP-position in length units = overlap radius
- Distance of TCP-orientation in angle units
- Distance for linear and rotational auxiliary axes

The effect of the parameters on the different motion commands is described in the section on data types OVLREL and OVLABS.

5.2.7 Ramp

Sets the ramp type for acceleration.

```
Ramp (
type : RAMPTYPE
OPTIONAL param : REAL
)
```

The desired ramp type for acceleration can be set here. For type TRAPEZOID the parameter is used to define the time ratio to build up the acceleration (0 < param <= 0.5). The following figure depicts the different ramp shapes.

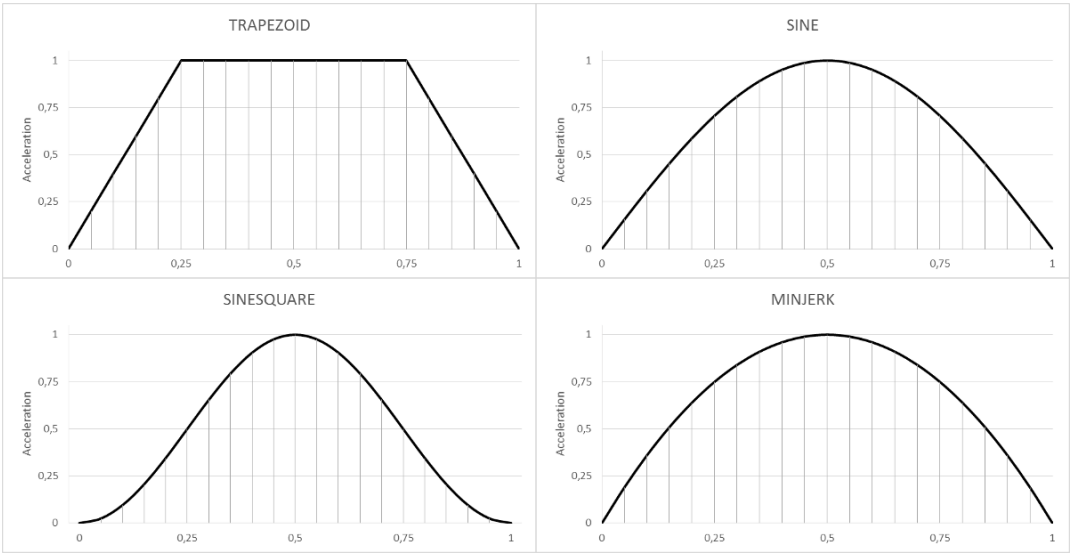


Fig. 5-2: Acceleration ramp shapes

Parameter:

type	Type of ramp
[param]	Ramp parameter to define the ramp shape (required only for TRAPEZOID)

Behavior upon missing optional parameter:

If no ramp parameter is specified along with ramp type TRAPEZOID, a default value of 0.5 will be used. This results in a triangular shape.

RAMPTYPE

This enumeration indicates the ramp type used for acceleration.

Elements:

TRAPEZOID	trapezoid ramp
SINE	sinusoidal ramp

SINESQUARE	sine-square ramp
MINJERK	ramp with minimum integral jerk
TIMEOPTIMAL	Trapezoid ramp for which the ramp factor is set in a way that the acceleration is built up with the maximum configured jerk.

5.2.8 RefSys

CAUTION!

Changing the reference system will result in an abrupt jump of the Cartesian TCP-path. If a reference system command is skipped using SetPC you may encounter unwanted results!

Sets a reference system.

```
RefSys (
refSys : REFSYS_
)
```

The RefSys command sets a new reference system for the following positioning commands. This reference system remains active until a new RefSys command or a new ExternalTCP command is set.

Cartesian positions refer to the currently set reference system. As long as no reference system has been set the valid reference system is WORLD.

Information

If a reference system with all values equal to 0 is referenced to itself then the WORLD system is used. Values other than zero trigger an error.

Information

If an external TCP is given as reference system, a runtime error is set. To set an external TCP the command ExternalTCP has to be used.

Parameter:

refSys	Reference system

5.2.9 ExternalTCP

CAUTION!

Changing the external TCP will result in an abrupt jump of the Cartesian TCP-path. If an external TCP command is skipped using SetPC you may encounter unwanted results!

Sets an external TCP. An external TCP is used like a Cartesian reference system. The difference is that usually the TCP is at the end of the robot tool. But if an external TCP is set, the TCP is at a position independent of the robot.

An external TCP is typically used, if the robot holds a work piece and moves it relative to a static tool (e.g.: Robot holds a cast iron part and moves it relative to a grinding machine which is mounted on the floor). The programming of the path can proceed as normal, using positions in the work piece coordinate system.

```
ExternalTCP (
extTCP : EXTERNALTCP
)
```

The ExternalTCP command sets a new external TCP for the following positioning commands. This external TCP remains active until a new RefSys command or a new ExternalTCP command is set.

Cartesian positions refer to the currently set reference system. As long as no external TCP has been set, the TCP of the robot and the reference system set via RefSys command remain active. A RefSys command deactivates the external TCP, the TCP of the robot is reactivated.

Parameter:

extTCP	External TCP

5.2.10 Tool

CAUTION!

Changing the tool will result in an abrupt jump of the Cartesian TCP-path. If a tool command is skipped using SetPC you may encounter unwanted results!

Sets a tool.

```
Tool (
tool : TOOL_
)
```

The tool command sets a new tool for the robot. It allows changing the robot's point of operation. If an external TCP is set, then the Tool command has the same effect as otherwise moving the reference system (the external TCP is not influenced by the robot tool

Parameter:

tool	Tool to be set

5.2.11 OriMode

Sets the type of orientation interpolation.

```
OriMode (
ori : ORIMODE
)
```

The OriMode command sets the type of orientation interpolation for all successive motion commands. As long as no OriMode is specified the type of orientation interpolation from the robot configuration is used.

Parameter:

ori	Type of orientation interpolation

ORIMODE

Type of orientation interpolation.

Elements:

CART	The z-axes of initial and target orientation define a plane in which the z-axis of the TCP is tilted. Simultaneously it will be rotated around the current z-axis (see Figure 3). Advantages: intuitive orientation movement : Disadvantage: Singularities may cause problems Usage: for processing paths
CARTCURVE	No difference to CART interpolation for linear motion commands. For circles the TCP will be rotated in the circular plane; initial and target orientation are transited using CART interpolation (see Figure 4). Advantages: intuitive orientation movement Disadvantage: Singularities may cause problems Usage: at present only for circular processing paths

WRISTJOINT	Wrist axes positions are calculated for initial and target orientation. Transiting is done using PTP-interpolation in axis space. The Cartesian position, however, is linear interpolated in Cartesian space. Advantages: enables quick re-orienting no problems concerning wrist-axes singularities Disadvantages: Orientation path is hard to predict and depends on the wrist type of the robot (danger of collision!) Usage: for paths whose orientation behavior between the programmed positions is irrelevant.

Orientation interpolation type CART

Fig. 5-3: Orientation interpolation type CART

Orientation interpolation type CARTCURVE

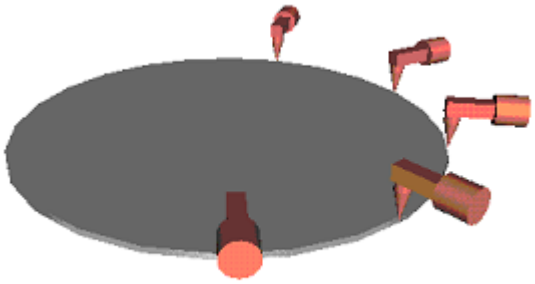


Fig. 5-4: Orientation interpolation type CARTCURVE

5.2.12 Workpiece

CAUTION!

Changing the work piece will result in an abrupt jump of the Cartesian TCP-path. If a tool command is skipped using SetPC you may encounter unwanted results!

Sets a work piece

```
Workpiece (
     workpiece : WORKPIECE
)
```

The workpiece command sets a new work piece for the robot. It allows changing the robot's point of operation.

Parameter:

workpiece	Work piece to be set

5.2.13 Example program

The following example uses a good portion of the commands described in this chapter. It refers to an articulated arm robot which is used to deburr a part.

Procedure: Before the processing is started a series of parameters are set: overlapping, type of orientation interpolation, acceleration ramp type, start override, reference system, and tool.

After that processing may begin:

- move to home position
- change tool

- move along a straight line with a subsequent semi-circle using the given path dynamics; the type of orientation interpolation ensures a constant angle between tool and part; furthermore, constant path velocity is required.
- move to home position
- change tool for second processing step
- reduce processing dynamics
- re-process part using the same path again
- move to home position

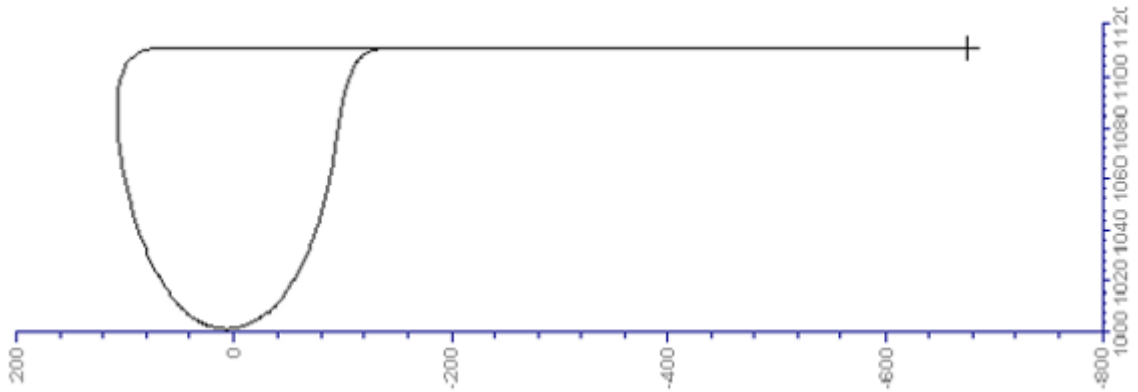


Fig. 5-5: Resulting path

Data (*.tid):

```
1: apos0 : AXISPOS := (a1 := -35, a2 := -20, a3 := 25, a4 :=
-100, a5 := 35, a6 := 100)
2: cpos0 : CARTPOS := (x := 1111, y := -98, z := 1335, a :=
20, b := 92, c := 180, mode := 0)
3: cpos1 : CARTPOS := (x := 1111, y := 110, z := 1335, a :=
-20, b := 92, c := -180, mode := 0)
4: cp0 : CARTPOS := (x := 1000, y := 21, z := 1335, a := -37,
b := 92, c := -180, mode := 0)
5: crs0 : CARTREFSYS := (baseRefSys := MAP(World), z := 30)
6: t0 : TOOL := (x := 5, y := 5, z := 15, a := 90)
7: cpos2 : CARTPOS := (x := 1125, y := -67, z := 1310, b :=
92, c := -90, mode := 0)
8: dyn0 : DYNAMIC := (velAxis := 100, accAxis := 100, de-
cAxis := 100, jerkAxis := 100, vel := 300,
9: acc := 2000, dec := 2000, jerk := 10000, velOri := 200,
10: accOri := 2000, decOri := 2000, jerkOri := 3600)
11: or0 : OVLREL := ()
12: oa0 : OVLABS := (posDist := 50, oriDist := 50,
linAxDist := 50, rotAxDist := 50, vConst := TRUE)
12: t1 : TOOL := (x := 5, y := 5, z := 5)
```

Program (*.tid):

```
1: // KAIROVersion 2.20
2: Ovl(oa0) // overlapping distance
3: OriMode(CART) // type of orientation interpolation
```

```
4: Ramp(MINJERK) // set ramp type
5: DynOvr(100) // dynamics override
6: RefSys(crs0) // reference system of part
7: Tool(Flange) // set no tool for moving to home position
8: PTP(cpos2) // move to home position
9: Dyn(dyn0) // specify processing dynamics (300mm/s)
10: Tool(t0) // set tool for first processing step
11: Lin(cpos0) // start of processing
12: Circ(cp0, cpos1) // processing
13: Lin(cpos2) // move to home position
14: Tool(t1) // set tool for next processing step
15: DynOvr(50) // reduce dynamics to 50%
16: Lin(cpos0) // restart processing
17: Circ(cp0, cpos1) // processing
18: DynOvr(100) // reset reduced dynamics
19: RefSys(World) // new reference system = World
20: PTP(cpos2) // move to home position
```

5.3 System Functions

This group comprises assignments, comments as well as commands to communicate with the world outside, and functions for measuring time.

Overview:

- ... := … assignment
- // … comment
- WaitTime Waiting time
- Stop Stops all active programs
- Info Issues an information
- Warning Issues a warning
- Error Issues an error
- Random - Creates a random number
- Time Measurement
  - SysTime Reads the system time
  - SysTimeToString Converts the system time to a string
- Mathematical Functions
  - SIN, COS, TAN, COT

    ASIN, ACOS, ATAN, ACOT
  - ATAN2
  - LN
  - EXP
  - ABS
  - SQRT
- Bitwise Operations & Conversions
  - SHL

- SHR
- ROL
- ROR
- SetBit
- ResetBit
- CheckBit
- STR

5.3.1 … := … (Assignment)

Assigns a value to a variable. An assignment consists of a variable (left hand side), the assignment operator :=, and an expression (right hand side). The type of expression must be assignable to the data type of the variable (see Basic data types).

When assigning structs (the type is a struct, an array, or a unit) the associated memory is copied from one variable to the other. For variables not containing any MAPTO-elements this is equal to assigning all elements, where the elements are subdivided until basic data types are reached.

For variables containing MAPTO-elements this is different, as references are copied rather than the referenced contents.

Examples for valid assignments:

```
i := 1
x := (a + b) * 2
```

5.3.2 // … (Comment)

Line comments are preluded using the character sequence //.

Example of a valid comment:

```
// Comment to the end of line
```

5.3.3 WaitTime

Waits for a certain period of time.

```
WaitTime (
timeMs : DINT
)
```

This command halts the robot for the specified time in milliseconds.

Information

At first WaitTime synchronizes movement and program, afterwards the waiting time starts. Hence the robot will be halted in any case.

Parameter:

timeMs	Time in [ms]

Example:

```
WaitTime(100) // wait for 100ms
WaitTime(0) // wait for main-run
```

5.3.4 Stop

Stops all running programs.

```
Stop (
OPTIONAL mode : STOPMODE
)
```

This command stops the execution of all active programs. Without the optional parameter the behavior is equal to pressing the stop button on the handheld terminal, i.e. the macro will be executed at once, without waiting for the main-run. If the macro shall be executed only after the previous movements have been completed a program synchronization must be programmed prior to the stop macro (e.g. using the macro WaitOnPos()).

Parameter:

[mode]	Type of stop

Without ‚mode' or if ALLAXIS is specified, the movement will be completely stopped. In tracking applications also the movement due to the moving reference system will be stopped. If CONTINUETRACKING is chosen the movement due to the programmed path will be stopped, whereas the movement due to tracking will be maintained.

```
RUN prog2 // starts parallel program prog2
PTP (home)
PTP (apos0)
IF inVal > 12 THEN
Stop() // global program stop; also prog2 is stopped
END_IF
PTP (apos2)
```

STOPMODE

This enumeration specifies how a program shall be stopped. Particularly in tracking applications it may be desirable to stop only the programmed movement while maintaining the movement that originates from a moving reference system.

Elements:

ALLAXIS	Complete stop of the robot's movement. In tracking applications also the movement originating from a moving reference system is stopped. To do so, first the programmed movement on the path is stopped and afterwards the tracking movement is stopped.
CONTINUETRACKING	Stops the programmed movement while maintaining tracking movements.

HARDSTOP	In contrast to ALLAXIS the movement is stopped immediately without stopping the path movement first.

5.3.5 Info

Issues an info message.

Information

Whenever a system message is issued the program pre-processing is stopped, leading to robot halt. If the program pre-processing should not be affected, DO instructions synchronized with the program execution must be used (see Triggers).

```
Info (
text : STRING
OPTIONAL param1 : ANY
OPTIONAL param2 : ANY
)
```

Info messages are displayed in the message protocol and in the report protocol on the masks ► **Alarm** and ► **Report**.

Moreover, it is possible to display a maximum of two values of any type using message functions. As placeholders „%1“ for the first value and „%2“ for the second value are used.

If the parameters are of a basic data type their value will be issued in the message, otherwise their name (and possibly type).

Parameter:

text	message text
param1	variables of any type that a part of the message text
param2	

Example:

```
av := analogInput7;
cn := cn+1
Info ("Cycle %1 completed! Actual value: %2", cn, av)
```

5.3.6 Warning

Issues a warning message.

Information

Whenever a system message is issued the program pre-processing is stopped, leading to robot halt. If the program pre-processing should not be affected, DO instructions synchronized with the program execution must be used (see Triggers).

```
Warning (
text : STRING
OPTIONAL param1 : ANY
OPTIONAL param2 : ANY
)
```

Warning messages are displayed in the message protocol and in the report protocol on the masks **Alarm** and **Report**.

For description of the optional parameters see Info.

Parameter:

text	message text
param1 param2	variables of any type that a part of the message text

Example

```
cn := cn+1
Warning ("Cycle %1 - pressure increased!", cn)
```

5.3.7 Error

Issues an error message.

Information

Whenever a system message is issued the program pre-processing is stopped, leading to robot halt. If the program pre-processing should not be affected, DO instructions synchronized with the program execution must be used (see Triggers).

```
Error (
text : STRING
OPTIONAL param1 : ANY
OPTIONAL param2 : ANY
)
```

Error messages are displayed in the message protocol and in the report protocol on the masks **Alarm** and **Report**.

Error messages lead to a program interruption. An error must be acknowledged to be able to continue with the program execution.

For description of the optional parameters see Info.

Information

When an error message is issued the robot movement will be interrupted in any case. The robot can continue only after the error message has been acknowledged.

Parameter:

text	message text
param1 param2	variables of any type that a part of the message text

Example:

```
PTP(startPos)
     WHILE input1 DO // assert vacuum for process
     Error ("vacuum error - has part been lost?")
END_WHILE
Lin(cPos1)
PTP(cPos2)
```

Information

Here it looks as if a cyclic message was issued, but in reality it is ensured that after the message has been acknowledged the condition is re-checked, before the program execution resumes.

5.3.8 Random

Creates a random number within the given range.

```
Random (
CONST minVal : REAL
CONST maxVal : REAL
) : REAL
```

Parameter:

minVal	Minimal value of the resulting random number
minVal	Maximal value of the resulting random number

Example:

```
randomVal := Random(0.0, 10.0); // A random value between 0 and 10 is created
```

5.3.9 Time Measurement

CLOCK.Reset

See CLOCK

CLOCK.Start

See CLOCK

CLOCK.Stop

See CLOCK

CLOCK.Read

See CLOCK

CLOCK.ToString

See CLOCK

TIMER.Start

See TIMER

TIMER.Stop

See TIMER

SysTime

Reads the system time in seconds since January 1st, 1970.

```
SysTime (
): DINT
```

This command reads the current system time from the control system and returns it as DINT-value.

Example:

Value := SysTime() // read current system time

SysTimeToString

Converts time to a text string.

```
SysTimeToString (
OPTIONAL time : DINT
): STRING
```

This command converts the system time to a formatted text string of the format „DDD mon dd hh:mm:ss yyyy“. This is helpful for the use in log files. When called without parameter sysTime returns the formatted current system time.

Parameter:

[time]	Time in [s]

Example:

```
actTime := SysTimeToString() // read current system time
Info(actTime) // and display it as an info
```

5.3.10 Mathematical Functions

SIN, COS, TAN, COT

Trigonometric functions. The argument unit is degrees.

```
Function (
deg : REAL
): REAL
```

Parameter:

deg	angle [°] at which the trigonometric function shall be evaluated.

ASIN, ACOS, ATAN, ACOT

Inverse trigonometric functions. The result unit is degrees.

```
Function (
val : REAL
 ): REAL
```

Parameter:

val	Value for which the corresponding angle shall be determined.

ATAN2

Arc tangent with two arguments.

```
ATAN2 (
valY: REAL,
valX: REAL
): REAL
```

Example:

```
r := SIN(x) // calculate sine of x
Info("%1", r) // display result as info
r := ATAN2(y, x) // calculate arc tangent of y, x
Info("%1", r) // display result as info
```

LN

Natural logarithm.

```
LN (
val: REAL
): REAL
```

Parameter:

val	Value whose natural logarithm shall be determined

EXP

Exponential function.

```
EXP (
val: REAL
): REAL
```

Parameter:

val	Value whose exponential value shall be determined

EXPT

Power function.

```
EXPT (
base: REAL,
exp: DINT
): REAL
```

Parameter:

base	Base of power function
exp	Exponent of power function

ABS

Absolute value.

```
ABS (
val: REAL
): REAL
```

Parameter:

val	Value whose absolute value shall be determined

SQRT

Square root.

```
SQRT (
val: REAL
): REAL
```

Parameter:

val	Value whose square root shall be determined.

5.3.11 Bitwise Operations & Conversions

SHR

Bitwise shifted to the right.

```
SHR (
expr : DWORD
digits : DINT
) : DWORD
```

Parameter:

expr	Value to be shifted
digits	Number of bits to be shifted

Example:

```
out := SHR(in, n)
```

SHL

Bitwise shifting to the left.

```
SHL (
expr : DWORD
digits : DINT
) : DWORD
```

Parameter:

expr	Value to be shifted
digits	Number of bits to be shifted

Example:

```
out := SHL(in, n)
```

ROR

Bitwise rotation to the right.

```
ROR (
expr : DWORD
digits : DINT
) : DWORD
```

Parameter:

expr	Value to be rotated
digits	Number of bits to be rotated

Example:

```
out := ROR(in, n)
```

ROL

Bitwise rotation to the left.

```
ROL (
expr : DWORD
digits : DINT
) : DWORD
```

Parameter:

expr	Value to be shifted
digits	Number of bits to be shifted

Example

```
out := ROL(in, n)
```

SetBit

Sets a bit in a specified word.

```
SetBit (
val : LWORD
bitNr : DINT
) : DWORD
```

Implementation: SetBit(value, bitNr) := value OR 2^bitNr

Parameter:

val	Value in which a bit shall be set
bitNr	Bit to be set (least significant bit = bit 0)

Example:

```
d := 16 // 16#0010
d := SetBit(d, 5) // set bit 5
Info("%1", d) // display result as info (16#0030)
```

ResetBit

Resets a bit in a specified word.

```
ResetBit (
val : LWORD
bitNr : DINT
 ) : DWORD
```

Implementation: ResetBit(value, bitNr) := value AND NOT 2^bitNr

Parameter:

valBit	Value in which a bit shall be set
bitNr	Bit to be set (least significant bit = bit 0)

Example:

```
d := 17 // 16#0011
d := ResetBit(d, 4) // reset bit 4
Info ("%1", d) // display result as info (16#0001)
```

CheckBit

Checks if a specific bit is set in the specified word.

```
CheckBit (
val : LWORD
bitNr : DINT
) : BOOL
```

Parameter:

valBit	Value which shall be checked for the given bit.
bitNr	Bit to be checked (least significant bit = bit 0)

Example:

```
d :=17 // 16#0011
IF CheckBit(d, 4) THEN      // check bit 4
Info("Bit 4 is set")    // display result as info
END_IF
```

STR

Supplies a formatted textual representation of a value.

```
STR (
expression : ANY
) : STRING
```

Parameter:

expression	Value to be converted to a string

Example:

```
s := STR(65) // s : "65"
```

Also see about this

WaitBit [} 72]

5.4 Flow Control

This group contains instructions for flow control.

Overview:

- CALL … calls a subroutine
- WAIT … waiting condition
- SYNC.Sync … Synchronization of programs running parallel
- IF … THEN … END_IF conditional branching
- ELSIF … THEN additional alternative branch
- ELSE alternative branch
- WHILE … DO … END_WHILE conditional loop
- LOOP … DO … END_LOOP loop
- RUN … start parallel program
- KILL … stop parallel program
- RETURN returns to caller
- LABEL … label for goto-jumps
- GOTO … goto-jump
- IF … GOTO … conditional jump

5.4.1 CALL ...

An end-user program may call another end-user program as a subroutine. This call is introduced using the key word "CALL". An end-user program must not call itself (not even indirectly via another program).

Example CALL-instruction:

```
// in test.tip
CALL sub // call end-user program sub.tip
```

5.4.2 WAIT ...

The WAIT-instruction enables the synchronization of the processing sequence. If the value of the WAIT-expression is TRUE the next instruction will be executed. Otherwise, the routine waits until the expression evaluates to TRUE. A WAIT-expression must not contain:

- Relational expressions of structs
- Function calls

Example:

```
// in test.tid
signal : BOOL
digin21 : BOOL
digin22 : BOOL
r : REAL

// in test.tip
WAIT signal
WAIT digin21 and digin22
WAIT r < 0
WAIT r + 1 > 0
```

5.4.3 SYNC.Sync

Command to synchronize programs running parallel.

Using this command it is possible to synchronize the execution of two running programs. Thus it is, e.g., possible to synchronize the movement of two robots. See 6.6 System and Extensions.

5.4.4 IF ... THEN ... END_IF

IF-instructions are used to introduce conditional branches into the program. The result of the IF-condition must be of type BOOL. The number of instructions after THEN, ELSIF and ELSE is unlimited. Every IF-instruction must be terminated using the key word END_IF.

Example:

```
// in test.tip
IF     x < 100 THEN
    y := 10
ELSIF     x < 400 THEN
     y := 20
```

```
ELSIF     x < 900 THEN
    y := 30
ELSE     y := 40
END_IF
```

5.4.5 ELSIF ... THEN

The ELSIF-instruction enables to insert an ELSIF-branch into an existing IF-instruction.

Example:

```
// in test.tip
IF     x < 100 THEN
     y := 10
ELSIF     x < 400 THEN
    y := 20
ELSIF     x < 900 THEN
     y := 30
ELSE     y := 40
END_IF
```

5.4.6 ELSE

The ELSE-instruction is used to insert an ELSE-branch into an existing IF-instruction.

Example:

```
// in test.tip
IF     x < 100 THEN
     y := 10
ELSIF     x < 400 THEN
    y := 20
ELSIF     x < 900 THEN
    y := 30
ELSE     y := 40
END_IF
```

5.4.7 WHILE ... DO ... END_WHILE

The WHILE-instruction serves to repeat an instruction sequence as long as a condition is fulfilled. The result of the loop condition must be of type BOOL. The loop may contain an unlimited number of instructions. The WHILE-instruction must be terminated using the key word END_WHILE.

Information

If a frequently executed loop contains no WAIT-instruction it may obstruct the execution of other programs. In this case the control system issues a warning and interrupts the uncooperative program from time to time.

Example for a WHILE-instruction:

```
WHILE ch <> " " DO
     i := i + l
    read (ch)
END_WHILE
```

5.4.8 LOOP ... DO ... END_LOOP

The LOOP-instruction serves to repeat an instruction sequence for a given number of times.

For each LOOP-instruction an internal loop variable is automatically created, which is initialized with 1 when the loop is entered. The loop is repeatedly executed until the loop variable excesses the programmed end value. If the end value is less than 1 the loop will be skipped altogether. After each loop pass the loop variable is increased by 1 and the end value is recalculated. The loop may contain an infinite number of instructions. By omitting the end value an infinite loop can be programmed.

Information

If a frequently executed loop contains no WAIT-instruction it may obstruct the execution of other programs (see WHILE)!

Example for a LOOP-instruction:

```
LOOP 10 DO
     read(ch) // read 10 times
END_LOOP

j := 0
 ... LOOP j DO
    read(ch) // will be skipped
END_LOOP

LOOP DO //infinite loop
     WAIT chg
    chg := FALSE
     ...
END_LOOP
```

5.4.9 RUN ...

The instruction `RUN` program starts an end-user program. This program will be executed in parallel, as opposed to CALL which is used to call a program and then continue when the called program has returned.

```
// in test.tip
RUN Prog189 // starts the end-user program Prog189.tip
 ...
KILL Prog189
...
```

5.4.10 KILL ...

The instruction `KILL` program serves to abort an end-user program started with RUN.

```
// in test.tip
RUN Prog189
 ...
KILL Prog189 // aborts the end-user program Prog189.tip
...
```

5.4.11 RETURN

The RETURN-instruction is used to end the execution of a routine.

Example:

```
// in test.tip
 ...
IF nok THEN
RETURN
END_IF
...
```

5.4.12 LABEL ...

The LABEL-instruction defines jump targets. Please note that a label contained in a GOTO- or IF-GOTO-instruction must be defined exactly once in the corresponding program.

Example:

```
LABEL label99
```

5.4.13 GOTO ...

The GOTO-instruction is used to jump to different parts of the program. The jump target must be defined by a LABEL-instruction which has to be part of the same program as the GOTO-instruction. Jumps into an instruction block from outside are prohibited. An instruction block may be a WHILE loop or the different cases of an IF-instruction.

Example:

```
GOTO label99
...
LABEL label99
```

5.4.14 IF ... GOTO ...

The IF-GOTO-instruction is a shortened IF-instruction, which simplifies conditional jumps inside a program. The result of the IF-condition must be of type BOOL. If the condition is fulfilled, the program execution branches to the specified jump target. The jump target must be defined by a LABEL-instruction.

Example for an IF-GOTO-instruction:

```
IF x < 100 GOTO label99 // IF-GOTO-instruction
...
LABEL label99
```

5.5 Signals

The commands in this group provide functionalities for arbitrary signals (e.g. IO end-points which are released for KAIRO by the IEC, or just common KAIRO variables).

Methods:

- WaitBool - Wait until a signal is set (or until it is not set)
- WaitBit - Wait until a specified bit of the signal is set (or until it is not set)
- WaitBitMask - Wait for a certain value of a bit-pattern signal
- WaitLess - Wait until a signal is lower than a given limit
- WaitGreater - Wait until a signal is greater than a given limit
- WaitInside - Wait until a signal is within a given interval
- WaitOutside - Wait until a signal is outside a given interval
- BOOLSIGOUT.Set - Set a signal
- BOOLSIGOUT.Pulse - Set a signal for a certain period of time
- BOOLSIGOUT.Connect - Establish a connection between a signal and a state variable

5.5.1 Runtime behavior of signal commands

All commands are evaluated during program execution. If synchronization with the robot movement is desired, triggers or waiting commands (WaitIsFinished, WaitTime) may be used.

5.5.2 Optional parameter timeoutMs

All commands used to wait for a certain value of an input may be limited by a timeout. In this case the function returns when the timeout has expired. The return value will be FALSE. If the timeout is not specified, the function waits infinitely long for the desired value.

5.5.3 WaitBool

Wait until the signal is set or reset.

```
WaitBool (
CONST variable : BOOL
OPTIONAL VAR_IN boolVal : BOOL
OPTIONAL VAR_IN timeoutMs : DINT
) : BOOL
```

This function waits until the signal is set or reset, or until the optional timeout expires.

Parameters:

variable	Boolean signal
[boolVal]	Target value, function waits until the signal takes this value (default: TRUE)
[timeoutMs]	When this period [ms] has passed, waiting is aborted (default: wait without time limit)

Return value:

FALSE if timeout expires, TRUE if signal takes target value.

Example:

Wait for signal "pieceAvail". If signal is already set when Pos0 is approached, the robot may overlap to Pos1.

```
PTP(Pos0)
WaitBool(pieceAvail)
PTP(Pos1)
```

Wait for signal "pieceAvail". Overlapping to Pos1 is impossible, as WaitBool is evaluated only after Pos0 has been reached.

```
PTP(Pos0)
WaitIsFinished()
WaitBool(pieceAvail)
PTP(Pos1)
```

5.5.4 WaitBit

Wait until a specified bit of a signal is set or reset.

```
WaitBit (
CONST variable : ANY
VAR_IN bitNr : DINT
OPTIONAL VAR_IN bitVal : BOOL
OPTIONAL VAR_IN timeoutMs : DINT
) : BOOL
```

This function waits until the specified bit is set or reset, or until the optional timeout expires. It can be used with all integer value signals and all bit-pattern value signals.

Parameters:

variable	Signal (variable of an integer or bit-pattern type)
bitNr	Bit number in the signal (0..63)
[bitVal]	Target value, function waits until specified bit takes this value (default: TRUE)
[timeoutMs]	When this period [ms] has passed, waiting is aborted (default: wait without time limit)

Return value:

FALSE if timeout expires, TRUE if input bit takes target value.

Example:

Wait for bit 7 of variable "inData". If the bit is already set when Pos0 is approached, the robot may overlap to Pos1.

```
PTP(Pos0)
WaitBit(inData, 7)
PTP(pos1)
```

Wait for bit 7 in "inData". Overlapping to Pos1 is impossible, as WaitBit is evaluated only after Pos0 has been reached.

```
PTP(Pos0)
WaitIsFinished()
WaitBit(inData, 7)
PTP(Pos1)
```

5.5.5 WaitBitMask

Wait for a certain value of a bit-pattern signal.

```
WaitBitMask (
CONST variable : ANY
VAR_IN mask : LWORD
OPTIONAL VAR_IN maskedVal : LWORD
OPTIONAL VAR_IN timeoutMs : DINT
) : BOOL
```

This function waits until the signal takes the specified value, or until the optional timeout expires. The signal may be masked. In this case the function waits until (actVal AND mask) = (maskedVal AND mask). This function can be used with all integer value signals and all bit-pattern value signals.

Parameters:

variable	Signal (variable of an integer or bit-pattern type)
mask	Mask for masking the signal
[maskedVal]	Target value, function waits until (masked) signal matches this value (default: maskedVal = mask)
[timeoutMs]	When this period [ms] has passed, waiting is aborted (default: wait without time limit)

Return value:

FALSE if timeout expires, TRUE if target value is reached.

Example:

Wait for value 33 of the least significant 8 bits (mask = 255) of signal "inData". If this value is already present when Pos0 is approached, the robot may overlap to Pos1.

```
PTP(Pos0)
WaitBitMask(inData, 255, 33, 1000)
PTP(Pos1)
```

5.5.6 WaitLess

Wait until signal value is less than a given limit.

```
WaitLess (
CONST variable : ANY
VAR_IN limit : LREAL
OPTIONAL VAR_IN timeoutMs : DINT
) : BOOL
```

This function waits until the signal value is less than the specified limit, or until the optional timeout expires. It can be used with all integer value signals and all floating point value signals.

Parameters:

variable	Signal (variable of an integer or floating-point type)
limit	Limit. Signal should fall below this value.
[timeoutMs]	When this period [ms] has passed, waiting is aborted (default: wait without time limit)

Return value:

FALSE if timeout expires, TRUE if signal value is lower than limit value.

Example:

Wait until signal "aiTemp1" is below 40. If this value is already below the limit when Pos0 is approached, the robot may overlap to Pos1. After a maximum waiting time of 1200 ms the program continues even if "aiTemp1" is greater than 40.

```
PTP(Pos0)
IF WaitLess(aiTemp1, 40.0, 1200) THEN
     PTP(Pos1)
ELSE
    PTP(PosHome)
END_IF
```

5.5.7 WaitGreater

Wait until signal value is greater than a given limit.

```
WaitGreater (
CONST variable : ANY
VAR_IN limit : LREAL
OPTIONAL VAR_IN timeoutMs : DINT
) : BOOL
```

This function waits until the signal value is greater than the specified limit, or until the optional timeout expires. It can be used with all integer value signals and all floating point value signals.

Parameters:

variable	Signal (variable of an integer or floating-point type)
limit	Limit. Signal should surpass this value.
[timeoutMs]	When this period [ms] has passed, waiting is aborted (default: wait without time limit)

Return value:

FALSE if timeout expires, TRUE if signal value is greater than the limit value.

Example:

Wait until signal "aiTemp1" is above 10. WaitGreater() is evaluated not before the robot has passed the middle of the previous segment.

```
PTP(Pos0)
WaitIsFinished(50)
WaitGreater(aiTemp1, 10.0)
PTP(Pos1)
```

5.5.8 WaitInside

Wait until a signal is within a given interval.

```
WaitInside (
CONST variable : ANY
VAR_IN minVal : LREAL
VAR_IN maxVal : LREAL
OPTIONAL VAR_IN timeoutMs : DINT
) : BOOL
```

This function waits until the signal value is within the specified interval, or until the optional timeout expires. The limits of the interval are valid signal values (i.e. waiting condition: (value >= minVal) AND (value <= maxVal)). This function can be used with all integer value signals and all floating point value signals.

Parameters:

variable	Signal (variable of an integer or floating-point type)
minVal	Minimum value (lower value of target range)
maxVal	Maximum value (upper value of target range)
[timeoutMs]	When this period [ms] has passed, waiting is aborted (default: wait without time limit)

Return value:

FALSE if timeout expires, TRUE if signal value is within the given interval.

Example:

Wait until signal "aiTemp1" is within specified limits with the additional condition, that the observation of the signal should be stopped after at most 2s. In case of this timeout, a warning is set.

```
b := WaitInside(aiTemp1, 5.0, 15.0, 2000)
IF NOT b THEN
    Warning("Timeout waiting on temperature inside target-window")
END_IF
```

5.5.9 WaitOutside

Wait until a signal is outside a given interval.

```
WaitOutside (
CONST variable : ANY
VAR_IN minVal : LREAL
VAR_IN maxVal : LREAL
OPTIONAL VAR_IN timeoutMs : DINT
) : BOOL
```

This function waits until the signal value is outside the specified interval, or until the optional timeout expires. The limits of the interval are no valid signal values (i.e. waiting condition: (value < minVal) OR (value > maxVal)). This function can be used with all integer value signals and all floating point value signals.

Parameters:

variable	Signal (variable of an integer or floating-point type)
minVal	Minimum value (lower value of target range)
maxVal	Maximum value (upper value of target range)
[timeoutMs]	When this period [ms] has passed, waiting is aborted (default: wait without time limit)

Return value:

FALSE if timeout expires, TRUE if signal value is outside the given interval.

Example:

Wait until signal "aiPressure1" is outside specified limits.

```
realMaxVal := 4.0
WaitOutside(aiPressure1, 0.5, realMaxVal)
```

5.5.10 BOOLSIGOUT.Set

Set a signal

See 6.5.4 Digital output signal BOOLSIGOUT

5.5.11 BOOLSIGOUT.Pulse

Set a signal for a certain period of time

See 6.5.4 Digital output signal BOOLSIGOUT

5.5.12 BOOLSIGOUT.Connect

Establish a connection between a signal and a state variable

See 6.5.4 Digital output signal BOOLSIGOUT

5.6 Technology Options

This group contains advanced functions.

5.6.1 Triggers

In many cases it is required to perform certain actions depending on the geometry of the movement. This can be achieved using programmable triggers that trigger actions without influencing the dynamics of the movement.

This kind of synchronization between program and movement is done using the DO-instruction.

Overview:

- OnParameter = Trigger on segment parameter
- OnPlane = Trigger defined by a plane
- OnDistance = Trigger defined by a distance to the beginning/end of a segment
- OnPosition = Trigger on segment end

General information

General runtime behavior and DO-instruction

- Macros accessing the periphery (I/O) stop the program pre-processing, and hence stop the robot's movement. If this is not desired, DO-instructions synchronized with the program execution must be employed which have no influence on the program pre-processing. All other macros and instructions are executed by the program preprocessing.
- The DO-instruction is a general instruction which may be used after any other macro. Every DO contains precisely one instruction. Every line of program may contain at most one DO.
- The DO-instruction is activated synchronously to the main-run, i.e. upon termination of the previous instruction. The program pre-processing and hence the movement will not be influenced by the DO-instruction.

Programming options

Using the DO-instruction action may be performed synchronously with the program execution. For this a particular trigger macro is not necessarily required. For example, a DO-instruction may be programmed after a Lin macro. In this case the DO-part will be executed when the motion command has been completed.

Example:

Data (*.tid):

```
1: Pos1 : CARTPOS := (x := 952, y := 196, z := 1228, mode :=
1)
2: Pos2 : CARTPOS := (x := 490.7, y := 150.2, z := 1557.1,
mode := 1)
```

Program (*.tip):

```
1: // KAIROVersion 2.20
2: Lin(Pos1) DO Info ("Pos1 reached")
3: Lin(Pos2)
```

If an action shall be triggered e.g. right in the middle of a motion command special trigger macros must be used.

Example

Data (*.tid):

```
1: Pos1 : CARTPOS := (x := 952, y := 196, z := 1228, mode :=
1)
2: Pos2 : CARTPOS := (x := 490.7, y := 150.2, z := 1557.1,
mode := 1)
```

Program (*.tip):

```
1: // KAIROVersion 2.20
2: Lin(Pos1) OnParameter(50) DO Info("Half way from Pos1 to
Pos2 completed")
3: Lin(Pos2)
```

Furthermore, the actual point of time of trigger execution may be shifted using the optional parameter time. This feature may be used to compensate valve switching times etc. The time-parameter is added to the corresponding parameter specified in the robot configuration.

Boundary conditions

- Allowed trigger actions (instruction preluded by DO) are assignments, macros, and subroutine calls.
- Prohibited trigger actions are macros with path influence (motion commands, dynamics settings, tools or reference systems, etc.). If such a macro is used an error will be issued. The same goes for macros which are not directly used as a trigger action, but are part of the called subroutine. After a DO-instruction TeachView offers only suited macros.
- If the program counter is set to OnParameter, OnPlane, or OnDistance the corresponding trigger action will not be executed. The trigger would be undefined because the resulting movement depends on the starting position. Consequently, a trigger in the first line of a program has no effect. However, trigger macros of type OnPosition will be executed.
- In single-stepping mode trigger actions based on OnParameter, OnPlane, and OnDistance will not be executed. Trigger actions based on OnPosition and actions preluded by DO (e.g. Lin(P1) DO x := 5) will be executed even in single-stepping mode.
- Per motion command a maximum number of eight triggers may be programmed.

OnParameter

Trigger on a segment parameter of the next motion segment. Optionally, a time-shifting parameter may be specified. If it is negative, the assigned action will be triggered before the trigger is reached, if it is positive the action

will be triggered after the trigger has been reached. If no time is specified the default time-shifting set by the system integrator is being used, otherwise the specified time is added.

The maximum negative time-shifting is limited by the time-shifting set by the system integrator (typical -200 ms). The maximum positive time-shifting is 1 s. These limits refer to the sum of configured and programmed time-shifting. In case an invalid time-shifting is specified a warning will be issued and the trigger action will not be executed.

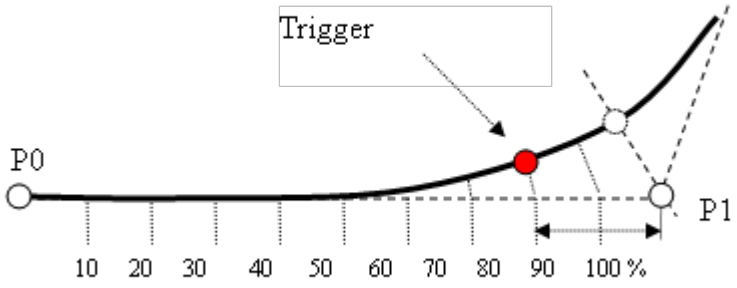
```
OnParameter (
pos : PERCENT
OPTIONAL timeMs : DINT
)
```

Parameter:

pos	Segment parameter on which the trigger is placed [%] (see 6.4.7 PERCENT)
[timeMs]	Time-shifting of trigger action vs. trigger [ms] Limits: -200 ms (or alternative value specified by system integrator) .. 1000 ms

In an overlapping area the segment parameter is projected to the overlapping path. This leads to small deviations between programmed and executed distances.

```
1: Lin(P0)
2: OnParameter(80) DO ...
3: Lin(P1)
```



Example:

Measuring the time in ms the robot takes to move half-way from Pos1 to Pos2 along a straight line.

Data (*.tid):

```
1: Pos1 : CARTPOS := (x := 300, y := 196, z := 1220, mode :=
1)
2: Pos2 : CARTPOS := (y := 150, z := 1550, mode := 1)
3: Clock1 : CLOCK
4: duration : DINT
```

Program (*.tip):

```
1: // KAIROVersion 2.20
2: Lin(Pos1) DO Clock1.Start()
3: OnParameter(50) DO duration := Clock1.Read()
4: Lin(Pos2)
```

OnPlane

Trigger defined on a plane in Cartesian space. The plane must be parallel to one of the principal planes of the current reference system and is specified by a distance offset. (E.g. OnPlane(YZPLANE, 100)… plane parallel to the YZ-plane at a distance of x=100 mm).

Optionally, a time-shifting parameter may be specified (see OnParameter).

The offset may also be taught. In this case, after the macro has been inserted into the program, the desired principal plane and the corresponding coordinate axis must be selected before the current position value is taught.

Example:

Robot is moving along the z-axis, from z=100 to z=0. A trigger shall be placed to a certain position along this line. As the movement takes place normal to the XY-plane, it makes sense to use a plane trigger.

So, OnPlane is inserted into the program and XYPLANE is selected. Then the robot is moved to the desired trigger position. Pressing the "TEACH" key finishes the procedure.

```
OnPlane (
plane : PLANETYPE
pos : REAL
OPTIONAL timeMs : DINT
)
```

Parameter:

plane	Plane on which the trigger is placed (XYPLANE, XZPLANE, YZ-PLANE)
pos	Position offset of plane with respect to principal plane
[timeMs]	Time-shifting of trigger action vs. trigger [ms] (see OnParameter).

Example:

Set a digital output 100 ms before the robot hits the plane at x=100 (in the current reference system Rs1).

Data (*.tid):

```
1: Pos1 : CARTPOS := (x := 300, y := 196, z := 1220, mode :=
1)
2: Pos2 : CARTPOS := (y := 150, z := 1550, mode := 1)
3: DigOut1 : BOOLSIGOUT := (signal := MAP(IO.digOut1))
4: Rs1 : CARTREFSYS := (baseRefSys := MAP(World), z := 70,
c := 45)
```

Program (*.tip):

```
1: // KAIROVersion 2.20
2: RefSys(Rs1)
3: Lin(Pos1)
4: OnPlane(YZPLANE, 100.0, -100) DO DigOut1.Set(TRUE)
5: Lin(Pos2)
```

PLANETYPE

This enumeration specifies the principal Cartesian plane.

Elements:

XYPLANE	Plane spanned by X- and Y-axis
XZPLANE	Plane spanned by X- and Z-axis
YZPLANE	Plane spanned by Y- and Z-axis

OnDistance

Trigger at a specified distance to the beginning or end of a segment. Optionally, a time-shifting parameter may be specified (see OnParameter).

```
OnDistance (
type : DISTANCETYPE
dist : REAL
OPTIONAL timeMs : DINT
)
```

Parameter:

type	Specifies whether the distance refers to the beginning or end of the segment (FROMBEGIN, FROMEND)
dist	Distance
[timeMs]	Time-shifting of trigger action vs. trigger [ms] (see OnParameter).

Information

If the specified distance is too large (longer than the segment) a warning will be issued and the trigger will be pulled at the end (type = FROMBEGIN) or the beginning (type = FROMEND) of the segment.

Example:

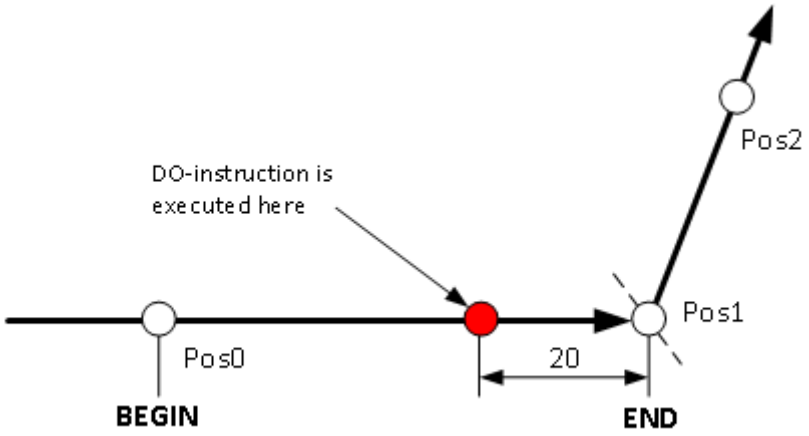
Set a digital output 20 mm before position Pos1 is reached

Data (*.tid):

```
1: Pos0 : CARTPOS := (y := 150, z := 1000, mode := 1)
2: Pos1 : CARTPOS := (x := 50, y := 150, z := 1000, mode := 1)
3: DigOut1 : BOOLSIGOUT := (signal := MAP(IO.digOut1))
```

Program (*.tip):

```
1: // KAIROVersion 2.20
2: Lin(Pos0)
3: OnDistance (FROMEND, 20) DO DigOut1.Set(TRUE)
4: Lin(Pos1)
```



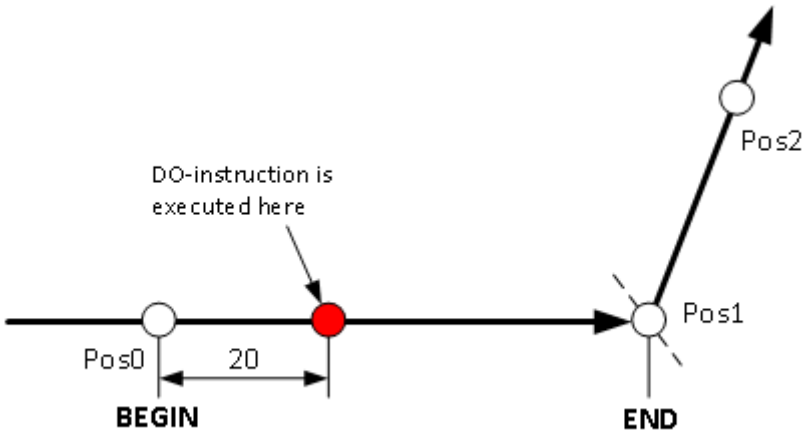
Set a digital output 20 mm after position Pos0 has been reached:

Data (*.tid):

```
1: Pos0 : CARTPOS := (y := 150, z := 1000, mode := 1)
2: Pos1 : CARTPOS := (x := 50, y := 150, z := 1000, mode := 1)
3: DigOut1 : BOOLSIGOUT := (signal := MAP(IO.digOut1))
```

Program (*.tip):

```
1: // KAIROVersion 2.20
2: Lin(Pos0)
3: OnDistance (FROMBEGIN, 20) DO DigOut1.Set(TRUE)
4: Lin(Pos1)
```



DISTANCETYPE

This enumeration specifies whether the distance refers to the beginning or end of a segment.

Elements:

FROMBEGIN	Distance refers to the beginning of the segment
FROMEND	Distance refers to the end of the segment

OnPosition

Trigger for synchronization between program execution and robot movement. The trigger action will be executed when the robot passes the specified position. This trigger is used to execute peripheral macros synchronously with the robot movement, but without stopping the pre-processing. Time-shifting is not supported.

```
OnPosition(
 )
```

Example:

Sets a digital output synchronously with the movement, but without influencing the movement.

Data (*.tid):

```
1: Pos1 : CARTPOS := (x := 952, y := 196, z := 1228)
2: Pos2 : CARTPOS := (x := 490.7, y := 150.2, z := 1557.1)
3: Pos3 : CARTPOS := (x := 490.7, y := 100, z := 1557.1)
4: Pos4 : CARTPOS := (x := 490.7, y := 200, z := 1557.1)
5: DigOut1 : BOOLSIGOUT := (signal := MAP(IO.digOut1))
```

Program (*.tip):

```
1: // KAIROVersion 2.20
2: Lin(Pos1)
3: LOOP 3 DO
4:     Lin(Pos3)
5:     Lin(Pos4)
6: END_LOOP
7: OnPosition() DO DigOut1.Set(TRUE)
8: Lin(Pos2)
```

Program pre-processing vs. program execution

1) Set an output during program pre-processing (output is set after the segment Lin(P1) has been appended to the path, i.e. shortly after the program has been started, without any correlation to the robot position):

   ```
   Lin(P1)

   DigOut1.Set(TRUE)
   ```

2) Set an output when position P1 is reached (during program execution):

   ```
   Lin(P1) DO DigOut1.Set(TRUE)
   ```

is equivalent to

3)
```
Lin(P1)
OnPosition() DO DigOut1.Set(TRUE)
```

Runtime behavior of a DO-action

Information

The following example demonstrates that trigger actions are always executed in a parallel program, hence the program linearity is lost.

```
Lin(P1)
DigOut1.Set(TRUE) DO boolVal := TRUE
IF boolVal THEN // condition possibly not fulfilled
    // as boolVal in parallel DO-program
    // may be assigned only after If-condition
    // has been evaluated
```

6 Data types

The structuring of these types is the same as offered by TeachView when a new variable is created.

6.1 Basic types

KAIRO defines 16 basic data types, which require no declaration. Each data type is assigned a certain range and a series of operators.

Meaning and range:

Title	Description	Range
BOOL	Boolean value	8 Bit TRUE or FALSE
SINT	Integer number	8 Bit, signed
INT	Integer number	16 Bit, signed
DINT	Integer number	32 Bit, signed
LINT	Integer number	64 Bit, signed
USINT	Integer number	8 Bit, unsigned
UINT	Integer number	16 Bit, unsigned
UDINT	Integer number	32 Bit, unsigned
ULINT	Integer number	64 Bit, unsigned
BYTE	Bit pattern	8 Bit
WORD	Bit pattern	16 Bit
DWORD	Bit pattern	32 Bit
LWORD	Bit pattern	64 Bit
REAL	Floating point number	IEEE 32 Bit
LREAL	Floating point number	IEEE 64 Bit
STRING	Strings	max. 255 8-Bit characters

Operators:

Title	Operators
BOOL	AND, OR, XOR, NOT (logical)
SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT (Integer types)	Arithmetic and comparison operators AND, OR, XOR, NOT (bitwise) SHL, SHR, ROL, ROR
BYTE, WORD, DWORD, LWORD (Bit pattern types)	AND, OR, XOR, NOT (bitweise) SHL, SHR, ROL, ROR, =, <>
REAL, LREAL (Floating point types)	Arithmetic and comparison operators
STRING	Comparison operators, +

Assignments

From / To	BOOL	Integer types	Bit pattern types	REAL	STRING
BOOL	y	x	x	x	STR
Integer types	x	y	y	y	STR
Bit pattern types	x	y	y	x	STR, CHR
Floating point types	x	y	x	y	STR, CHR
STRING	x	ORD	ORD	x	y

y ... allowed (possibly automatic type conversion)

x ... not allowed

CHR, ORD, STR ... Built-In functions to convert a basic type to a string

Furthermore, the generic data type **ANY** is available which may exclusively be used as parameter in routines and as referenced type of reference variables. This type may not be instantiated.

Title	Description
ANY	Any type

6.1.1 BOOL

The BOOL-type may hold the values TRUE or FALSE. The result of a relation is of type BOOL.

6.1.2 Integer-types, Floating-point-types, Bit-pattern-types

Integer-types are all integer numbers. Floating-point-types (Real-types) are all fractional numbers. The assignment from Integer to Real and from Real to Integer involves implicit type conversions. When converting large values from Integer to Real a loss in precision has to be expected. When converting from Real to Integer the positions after the decimal point are rounded off and attention must be given to the fact that the value range for the Integer type could be exceeded.

When converting from a short to a long Integer-type a Sign-Extend will occur and the represented value is preserved.

As soon as one bit-sample type is involved the most significant Bits are set to 0 at the conversion from a short Bit-sample or Integer-type to a long Bit-sample or Integer-type. At conversion from a long Bit-sample or Integer-type to a short (or equally long) Bit-sample or Integer-type the least-significant Bits are retained.

The result type of a permitted operation is determined according to the following rules:

- If an operand is of type LREAL, the result is of type LREAL.
- Otherwise: If an operand is of the type REAL, the result will have the type REAL.
- Otherwise: If one type is longer than the other, the result will follow the longer type.
- Otherwise: If a type is an integer-type, the result will have the integer type.
- Otherwise: The result will have the same type as the operands.

Example for conversion of integer- and bit-sample types:

```
// in test.tid
s : SINT
i : DINT
w : DWORD
txt : STRING

 // in test.tip
s := -1
i := s //Conversion SINT->DINT
txt := STR(i) // txt contains "-1", sign-extend
i := 510
s := i //Conversion DINT->SINT
txt := STR(s) //txt contains "-2", bits retained
s := -1
w := s //Conversion SINT->DWORD
txt := STR(w) // txt enthält "16#0000_00FF", filled with 0
s := -1
i := BYTE(s) // first explicit conversion SINT->BYTE, then
    //implicit conversion BYTE->DINT because of assignment to i
txt := STR(i) // txt contains "255", filled with 0
```

6.1.3 STRING

Strings are strings of characters with a maximum length of 255 characters. Strings can be added on to one another with the help of the + operator. The Built-in function STR converts the content of variables of another type into a formatted string of characters. If the Built-in function CHR is applied to a DINTnumber, it will return the corresponding ASCII-string.

Example (string assignment):

```
// in test.tid
s : STRING
b : BOOL
i : DINT
r : REAL

// in test.tip
//...
b := TRUE
i := 65
r := 1.234
s := "Bool = " + STR(b) + ", Int = " + STR(i) + ", Real = " + STR(r)
// value of s: "Bool = TRUE, Int = 65, Real = 1.234"
s := CHR(i)
// value of s: "A"
```

6.1.4 ANY

This data type is generic. It may only be used as parameter in routines and as referenced type of reference variables. It may not be instantiated. Constants may be passed as ANY-parameter.

Example (usage of ANY):

```
// in test.tid
i : DINT
r : REAL

// in test.tip
//...
i := 65
r := 1.23
//...
Info("Int = %1, Real = %2", i, r) // Info: Int = 65, Real = 1.23
Info("Int = %1, Real = %2", 12, 9.99) // Info: Int = 12, Real = 9.99
```

6.2 Positions

6.2.1 POSITION_

A position defines the position and orientation of a robot's TCP. It may either be specified in axis coordinates or in Cartesian coordinates.

Hierarchy:

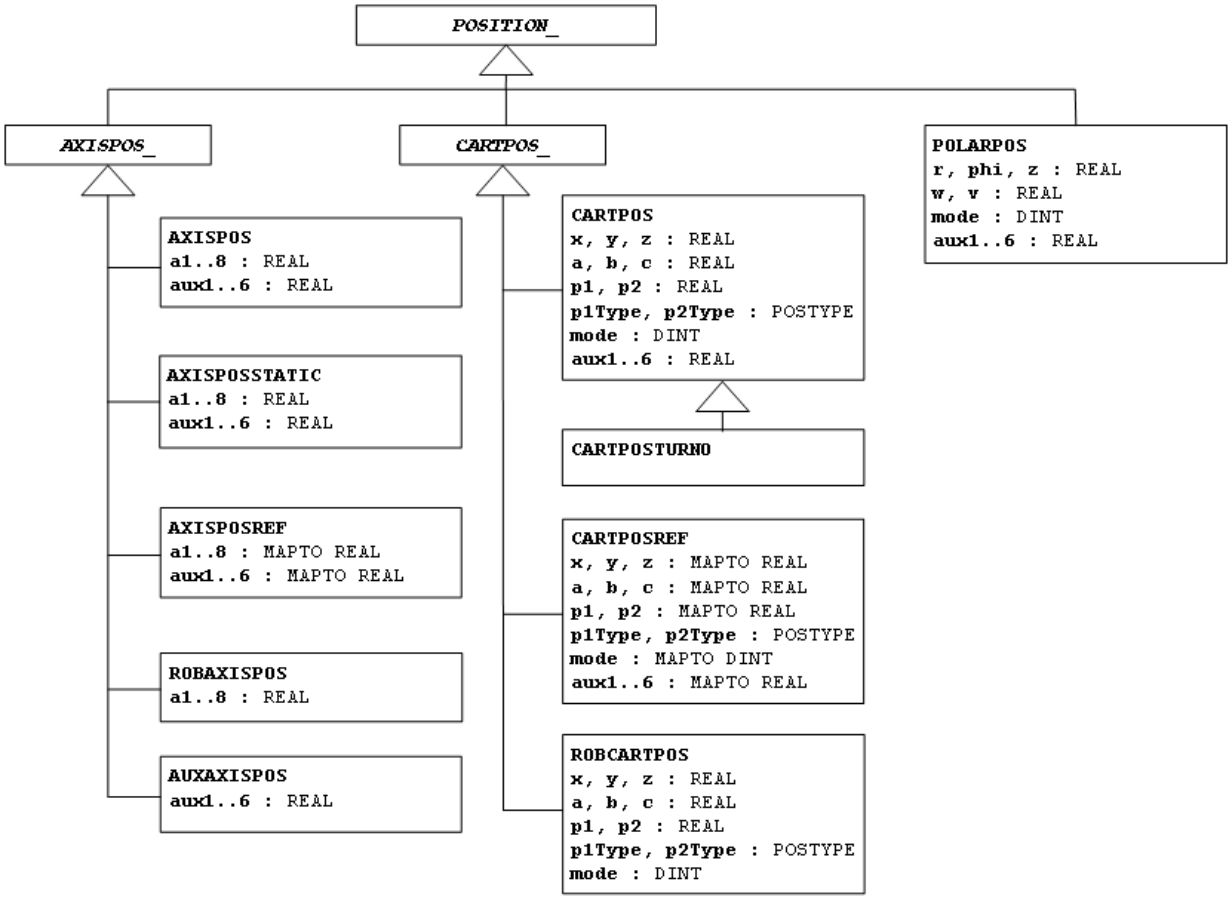


Fig. 6-6: Hierarchy of position data types

6.2.2 AXISPOS_

Position specified by axis positions.

6.2.3 AXISPOS

Position specified by axis positions.

The position of all robot axes and auxiliary axes is specified.

Variables

a1..8	REAL	Position of robot axes
aux1..6	REAL	Position of auxiliary axes

6.2.4 AXISPOSSTATIC

Position specified by axis positions.

This type has the same members as AXISPOS, but it cannot be created or modified by the application. It is used for internal positions which shall be readable for the application.

6.2.5 AXISPOSREF

Position specified by axis positions. Like AXISPOS, but the variables are references.

Variables

a1..8	REAL MAPTO	Position of robot axes
aux1..6	REAL MAPTO	Position of auxiliary axes

6.2.6 ROBAXISPOS

Position specified by axis positions.

The position of all robot axes is specified.

Positions of auxiliary axes are not specified. This data type is only available in systems with external chained auxiliary axes. It is used to move the robot independently from position of auxiliary axes and without moving these auxiliary axes.

Variables

a1..8	REAL	Position of robot axes

6.2.7 AUXAXISPOS

Position specified by axis positions.

The position of all auxiliary axes is specified.

Positions of robot axes are not specified. This data type is only available in systems with external chained auxiliary axes. It is used to move the auxiliary axes independently from robot axes and without moving these robot axes.

Variables

aux1..6	REAL	Position of auxiliary axes

6.2.8 CARTPOS_

Position specified by Cartesian components.

6.2.9 CARTPOS

Position specified by Cartesian components.

CARTPOS contains a position in Cartesian coordinates referring to the currently set reference system (set by macro 'RefSys'). Some positions may be reached with different axis positions (several possibilities). To specify the desired position of the axes every possibility is assigned a robot mode. The axes position in which the robot achieves this position depends on currently set tool (set by macro 'Tool').

For robots with more than 6 degrees of freedom, the position of the additional Cartesian components must also be specified. The meaning of these additional components depends on the respective robot type; for individual robot types, you can choose between different interpretations (`p1Type`, `p2Type`) of these values. For robots without additional degrees of freedom, the values specified for the additional components are ignored.

Variables

x, y, z	REAL	Position of TCP
a, b, c	REAL	Orientation of TCP
p1, p2	REAL	Position of the additional Cartesian components for robots with more than 6 degrees of freedom
p1Type, p2Type	POSTYPE	Type of the additional Cartesian components for robots with more than 6 degrees of freedom (see 6.2.9.1 POSTYPE)
mode	DINT	Information on axes ambiguity, meaning depends on type of robot
aux1..6	REAL	Position of auxiliary axes

6.2.9.1 POSTYPE

This enumeration specifies the type of an additional Cartesian component (only for robots with additional Cartesian degrees of freedom). This determines how the position specification for the additional component is to be interpreted. Depending on the robot type, different options are available. A type specification is only possible for robots that support more than one type, otherwise the type is selected automatically.

Elements:

POS_DEFAULT	Standard selection (actual type used depends on robot type)
ELBOW_ANGLE	Type available for 7-axis articulated robots
ELBOW_JOINT0	Type available for 7-axis articulated robots
CARPAL_PLUNGE	Type of additional Cartesian component for Delta Carpal robots
WRIST1	Type available for Scara robots with parallel wrist

WRIST2	Type available for Scara robots with parallel wrist

6.2.10 CARTPOSTURN0

Position specified by Cartesian components and Turn 0.

This type has the same members as CARTPOS, but the robot approaches such positions in turn 0 (if this is possible, i.e. if orientation interpolation WRISTJOINT is set for a Cartesian movement and for PTP movements). Using CARTPOSTURN0 should be considered especially to avoid overturning (and thus reaching of the limit switch) for wrist joints.

6.2.11 CARTPOSREF

Position specified by Cartesian components. Like CARTPOS, but the variables are references.

Variables

x, y, z	REAL	Position of TCP
a, b, c	REAL	Orientation of TCP
p1, p2	REAL	Position of the additional Cartesian components for robots with more than 6 degrees of freedom
p1Type, p2Type	POSTYPE	Type of the additional Cartesian components for robots with more than 6 degrees of freedom (see 6.2.9.1 POSTYPE)
mode	DINT	Information on axes ambiguity, meaning depends on type of robot
aux1..6	REAL	Position of auxiliary axes

6.2.12 ROBCARTPOS

Position specified by Cartesian components.

Like CARTPOS but does not contain positions of auxiliary axes. This data type is only available in systems with external chained auxiliary axes. It is used to move the robot independently from position of auxiliary axes and without moving these auxiliary axes.

Variables

x, y, z	REAL	Position of TCP
a, b, c	REAL	Orientation of TCP

p1, p2	REAL	Position of the additional Cartesian components for robots with more than 6 degrees of freedom
p1Type, p2Type	POSTYPE	Type of the additional Cartesian components for robots with more than 6 degrees of freedom (see 6.2.9.1 POSTYPE)
mode	DINT	Information on axes ambiguity, meaning depends on type of robot

6.2.13 POLARPOS

Position display in polar resp. cylindrical coordinates.

POLARPOS contains a position in polar resp. cylindrical coordinates. Some positions may be reached with different axis positions (several possibilities). To specify the desired position of the axes every possibility is assigned a robot mode. The axes position in which the robot achieves this position depends on currently set tool (set by macro ‘Tool’).

The position type POLARPOS is only available for selected robots.

Variables

r, phi, z	REAL	Position of TCP
w, v	REAL	Position of robot wrist axes
mode	DINT	Information on axes ambiguity, meaning depends on type of robot
aux1..6	REAL	Position of auxiliary axes

6.2.14 DISTANCE_

Distance given in axis or Cartesian coordinates.

For the type of Cartesian representation see Positions.

Hierarchy:

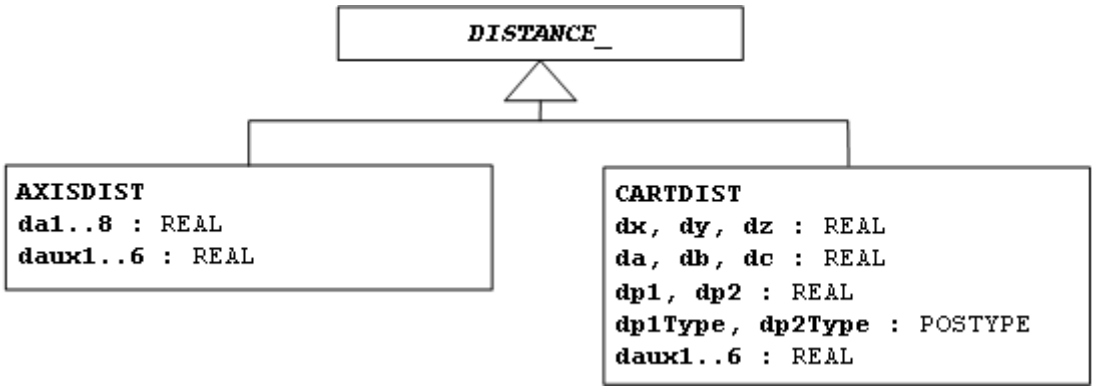


Fig. 6-7: Hierarchy of distance data types

6.2.15 AXISDIST

Distance specified in axis displacements. The displacement of all robot and auxiliary axes is specified.

Variables

da1..8	REAL	Displacement of robot axes
daux1..6	REAL	Displacement of auxiliary axes

6.2.16 CARTDIST

Distance specified in Cartesian coordinates. The distance is specified as displacement in the currently set reference system.

The currently set robot mode of the axes is kept.

Strukturvariablen

dx, dy, dz	REAL	Displacement of position
da, db, dc	REAL	Displacement of orientation
dp1, dp2	REAL	Displacement of the additional Cartesian components for robots with more than 6 degrees of freedom
dp1Type, dp2Type	POSTYPE	Type of the additional Cartesian components for robots with more than 6 degrees of freedom (see 6.2.9.1 POSTYPE)
daux1..6	REAL	Displacement of auxiliary axes

6.2.17 CARTFRAME

General Cartesian frame.

Variables

x, y, z	REAL	Position
a, b, c	REAL	Orientation

6.3 Reference system and tool

6.3.1 REFSYS_

Reference system for positions

Reference systems are used in combination with Cartesian positions to relocate a programmed geometry. Every position of a program refers to a previously set reference system. The default system may be changed using the command REFSYS.

Hierarchy

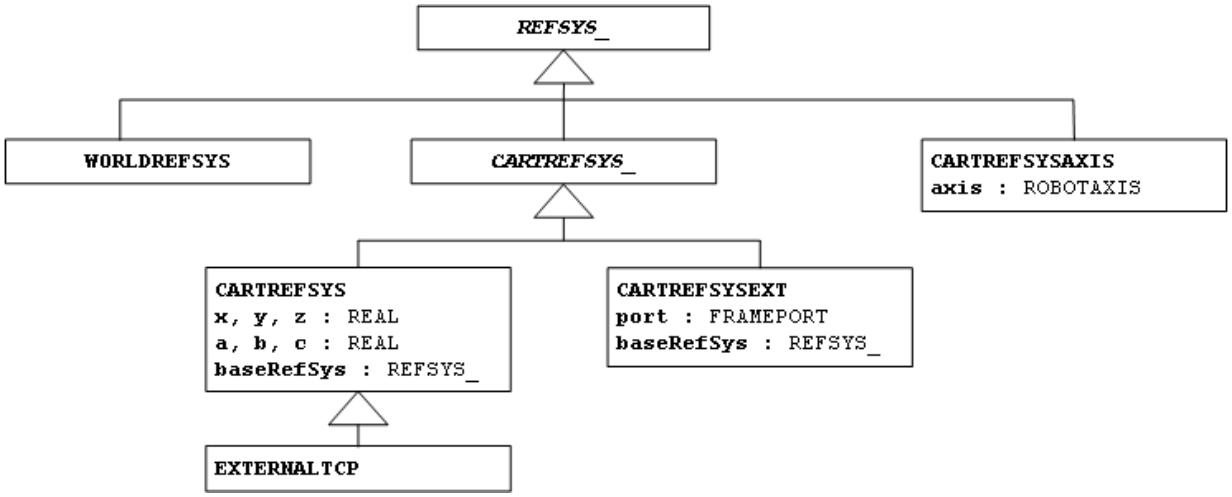


Fig. 6-8: Hierarchy of reference systems

6.3.2 WORLDREFSYS

All Cartesian reference systems are based upon the WORLD-system (possibly via a number of intermediate systems). The WORLD-system is always present, and no further instances may be created.

6.3.3 CARTREFSYS_

Reference system for Cartesian positions.

Cartesian positions refer to a pre-set Cartesian reference system. When a program is started the WORLD-system is active. The WORLD-system is a pre-defined coordinate system.

6.3.4 CARTREFSYS

Cartesian reference system.

Variables

x, y, z	REAL	Position displacement w.r.t. reference system
a, b, c	REAL	Orientation displacement w.r.t. reference system
baseRefSys	MAPTO REFSYS_	Referred reference system

6.3.5 CARTREFSYSEXT

Cartesian reference system where the offset values are external given from PLC. This reference system cannot be taught. To use this reference system, it has to be initially written once by the PLC. See also “RCE_SetFrame” in the PLC robotic manual. With this PLC function block it is possible to set the offset data of this reference system, which is used in the ‘port’ variable.

Variables

port	FRAMEPORT	Offset data of this reference system from PLC
baseRefSys	MAPTO REFSYS_	Referred reference system

6.3.6 CARTREFSYSAXIS

Cartesian reference system which is based upon an external auxiliary axis. This type of reference system is only available in systems containing an external auxiliary axis.

Positions defined on such a reference system refer to the position of the corresponding auxiliary axis. This may e.g. be used to write programs for processing objects placed on a turntable, independently from the current turntable position (the turntable must be defined as an external auxiliary axis).

Variables

axis	ROBOTAXIS	Externally coupled auxiliary axis upon which this reference system is based

6.3.7 EXTERNALTCP

External TCP.

Variables

x, y, z	REAL	Position displacement w.r.t. reference system
a, b, c	REAL	Orientation displacement w.r.t. reference system
baseRefSys	MAPTO REFSYS_	Referred reference system

Information

An external TCP cannot be used as reference system for the RefSys command.

6.3.8 TOOL_

The struct TOOL_ specifies a tool which is mounted to the robot flange. The flange is located at the end of the last robot axis. The tool tip is called Tool Center Point (TCP).

Hierarchy:

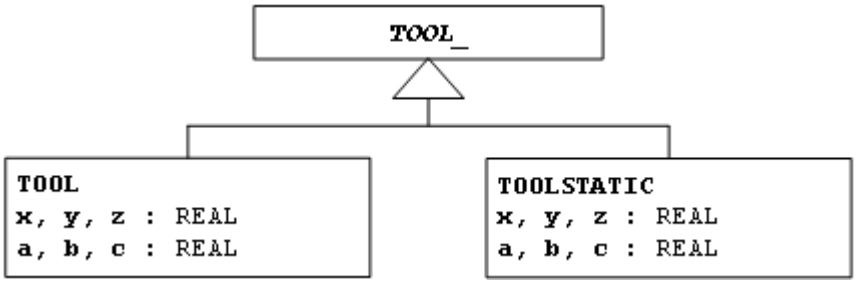


Fig. 6-9: Hierarchy of tools

6.3.9 TOOL

TOOL specifies position and orientation of the tool tip relative to the robot flange.

Variables

x, y, z	REAL	TCP-position w.r.t. flange
a, b, c	REAL	TCP-orientation w.r.t. flange

6.3.10 TOOLSTATIC

TOOLSTATIC is equivalent to TOOL. However, instances of TOOLSTATIC may not be created or modified. This data type is used for tools created by the system (flange and default tool, if available).

Variables

x, y, z	REAL	TCP-position w.r.t. flange
a, b, c	REAL	TCP-orientation w.r.t. flange

6.3.11 TOOLAXIS

Tool, which is based upon an external auxiliary axis. This type of tool is only available in systems containing an external auxiliary axis.

With this type it is possible to define an auxiliary axis as a tool. As a result, the TCP will be influenced by the position of the configured auxiliary axis.

Variables

axis	ROBOTAXIS	External auxiliary axis
guard	GUARD	Guard points on the tool

6.3.12 WORKPIECE

WORKPIECE specifies position and orientation of the work piece relative to the tool tip (i.e. the TCP).

Variables

x, y, z	REAL	Work piece position w.r.t. the TCP
a, b, c	REAL	Work piece orientation w.r.t. the TCP

6.4 Dynamics and overlapping

6.4.1 OVERLAP_

OVERLAP_ is used to specify the overlapping movement between two segments.

Hierarchy:

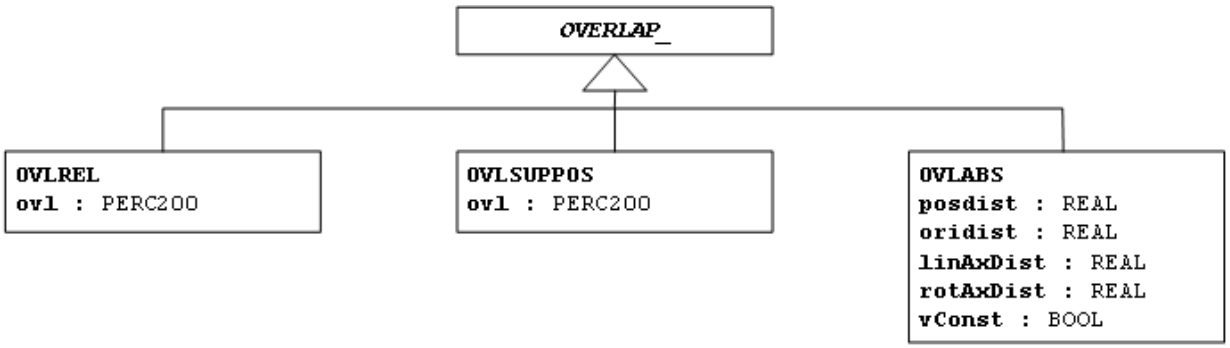


Fig. 6-10: Hierarchy of OVERLAP

6.4.2 OVLREL

Relative overlapping parameterization.

Relative parameterization defines an overlap degree using a percentage. 100 % means time-optimal overlapping, utilizing the acceleration reserves of the axes at lowest-possible deviation from the programmed target position. Values less than 100% will lead to smaller deviations, but prolong the cycle time. Values greater than 100% cause larger deviations and smaller axis accelerations at nearly a constant cycle time.

Variables

ovl	PERC200	Percentage in the range of 0..200 % (see 6.4.8 PERC200)

6.4.3 OVLSUPPOS

Super-position overlapping

Super-position overlapping defines a time-based overlap degree of two single start-stop movements using a percentage. 100 % means that the deceleration ramp of the first segment or the acceleration ramp of the second segment are completely overlapped. Values less than 100% will lead to smaller deviations, but prolong the cycle time. Values greater than 100% cause larger deviations and smaller axis accelerations. 200% means that the two segments are overlapped to the middle of the shorter of those segments.

Variables

ovl	PERC200	Percentage in the range of 0..200 % (see 6.4.8 PERC200)

Information

Due to the time-base overlapping it is possible that, depending on the segment geometry, the acceleration exceeds its limit (e.g. at acute-angled movements). To avoid this, super-position overlapping is disabled in such situations.

6.4.4 OVLABS

Absolute overlapping parameterization.

Absolute parameterization specifies the maximum permitted deviations from the programmed target position. This requires several parameters:

- Distance of TCP-position in length units = overlap radius
- Distance of TCP-orientation in angle units
- Distance for linear and rotational axes

Furthermore, when Cartesian movements are overlapped, constant path velocity may be demanded in the overlapping area.

Variables

posDist	REAL	Overlapping position distance of TCP
oriDist	REAL	Overlapping orientation distance of TCP
linAxDist	REAL	Distance for linear axes
rotAxDist	REAL	Distance for rotational axes
vConst	BOOL	Flag: keep path velocity constant during over-lapping

How the parameters are used:

The beginning of the overlap is determined by the smallest parameter. Overlapping radius and TCP-orientation distance can only be evaluated for Cartesian movements. During Point-to-point movements the parameters for the axes are used.

Parameter ->	Lin, Circ	oriDist	linAxDist	rotAxDist
PTP	-	-	Linear robot and auxiliary axes	Rotational robot and auxiliary axes
Lin, Circ	mm	grad	Linear wrist and auxiliary axes	Rotational wrist and auxiliary axes

6.4.5 DYNAMIC_

The data type DYNAMIC allows to program the dynamics of a movement.

Hierarchy:

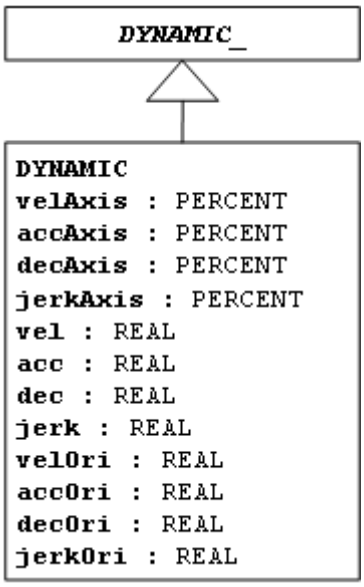


Fig. 6-11: Hierarchy of DYNAMIC

6.4.6 DYNAMIC

Velocity, acceleration, deceleration, and jerk as a percentage of the maximum axis limits of robot and auxiliary axes.

Velocity, acceleration, deceleration, and jerk of the Cartesian TCP position movement.

Velocity, acceleration, deceleration, and jerk of the Cartesian TCP orientation movement.

velAxis	PERCENT	% of maximum axis velocity (see 6.4.7 PERCENT)
accAxis	PERCENT	% of maximum axis acceleration (see 6.4.7 PERCENT)
decAxis	PERCENT	% of maximum axis deceleration (see 6.4.7 PERCENT)
jerkAxis	PERCENT	% of maximum axis jerk (see 6.4.7 PERCENT)
vel	REAL	Path velocity in mm/s
acc	REAL	Path acceleration in mm/s²
dec	REAL	Path deceleration in mm/s²
jerk	REAL	Path jerk in mm/s³

velOri	REAL	Path orientation velocity in deg/s
accOri	REAL	Path orientation acceleration in deg/s²
decOri	REAL	Path orientation deceleration in deg/s²
jerkOri	REAL	Path orientation jerk in deg/s³

It is permitted not to specify a full data block (all axis values = 0, all path values = 0, or all orientation values = 0). In this case the currently set data are retained.

6.4.7 PERCENT

To specify percentages.

Integer value in the range of 0..100%

6.4.8 PERC200

To specify percentages in overlap commands.

Integer value in the range of 0..200%

6.5 Signals

Signal blocks are used to provide functions for Boolean signals (e.g. IO endpoints which are released for KAIRO by the IEC, or just common KAIRO variables) which require a retentive behavior. Using these blocks it is very simple, e.g., to connect a Boolean variable with a system status flag or to realize an edge detection.

6.5.1 Runtime behavior of Signal-blocks

All methods are evaluated during program execution. If synchronization with the robot movement is desired, triggers or waiting commands (WaitIsFinished, WaitTime) may be used.

6.5.2 Optional parameter timeoutMs

All macros used to wait for a certain value of an input may be limited by a timeout. In this case the function returns when the timeout has expired. The return value will be FALSE. If the timeout is not specified, the function waits infinitely long for the desired value.

6.5.3 Digital input signal BOOLSIGIN

Block for digital input signals.

Member variables

signal	MAPTO BOOL	Reference to connected variable
val	MAPTO BOOL READONLY	Current signal value
posEdge	BOOL	Current value of rising edge detection
negEdge	BOOL	Current value of falling edge detection

The signal may be directly accessed, so there is no need for access functions.

Edge detection:

`posEdge` detects and stores the transition FALSE-TRUE, `negEdge` the transition TRUE-FALSE. These variables stay set until they are reset by the user.

6.5.4 Digital output signal BOOLSIGOUT

Block for reading and writing a digital output signal.

Member variables

signal	MAPTO BOOL	Reference to connected variable
val	MAPTO BOOL	Current signal value

Methods:

- Set - Set signal
- Pulse - Set signal for a certain period of time
- Connect - Establish a connection with a state variable

BOOLSIGOUT.Set

Setting a signal and (optionally) observe a feedback signal.

```
BOOLSIGOUT.Set (
OPTIONAL VAR_IN value : BOOL
OPTIONAL VAR_IN fbSignal: BOOL
OPTIONAL VAR_IN fbTimeoutMs : DINT
OPTIONAL VAR_IN waitOnFeedback : BOOL
)
```

Alternatively, the signal may be directly written to.

Parameters:

[value]	Value to be written to the signal (default: TRUE)
[fbSignal]	Feedback signal. If a feedback signal is used, it is observed until the output is reset. Whenever the feedback signal differs from the output signal, an error is set (default: no feedback signal).

[fbTimeoutMs]	Max. time [ms] until the feedback signal must show the desired value (default: 0, i.e. feedback signal has to be set immediately).
[waitOnFeedback]	TRUE: Program waits until feedback is set. If no `fbTimeoutMs` is given (or a timeout <= 0ms), the program waits without time limit. Otherwise an error is set, if the signal is still not set after the given waiting time has expired. FALSE: Program continues without waiting on the feedback signal (default). I the feedback signal does not show the desired value and no `fbTimeoutMs` is given (or a timeout <= 0ms), the observation sets an error immediately. default: FALSE

Example 1:

Setting an output signal

```
Out5.Set(TRUE) // set signal to TRUE
```

Alternative possibility

```
Out5.val := TRUE // set signal to TRUE
```

Example 2:

Observing a feedback signal:

A gripper should be closed using a digital output signal. A digital input signal gives the feedback if a part has been taken. As soon as the feedback signal vanishes (i.e. the gripper has lost the part), an error is set and the program is interrupted.

```
GripperOut.Set(TRUE, GripperFeedback, 100, TRUE) // close gripper
    // and wait on feedback signal
Lin(p2)
    // if the feedback signal is reset during
    // movement, an error is set and the program is interrupted
GripperOut.Set(FALSE)    // open gripper, therefore the feedback
    // observation is stopped
```

BOOLSIGOUT.Pulse

Set an output signal to a certain value for a certain period of time.

Command will be triggered synchronously with the main-run.

```
BOOLSIGOUT.Pulse (
VAR_IN pulseLengthMs : DINT
OPTIONAL VAR_IN pulseValue : BOOL;
OPTIONAL VAR_IN pauseAtInterrupt : BOOL
)
```

Pulses a digital output signal with a signal of defined duration. The pulse duration is specified in milliseconds.

At the beginning of the pulse the signal is set to `pulseValue`, at the end of the pulse the signal is set to the inverted value of `pulseValue`. If the output is already set to `pulseValue` when the pulse is executed, it will only be reset at the end of the pulse.

The beginning of the pulse is synchronous with the main-run. Hence, overlapping to next motion command is possible and the robot will not be stopped during the pulse.

Using the optional parameter `pauseAtInterrupt` it is possible to indicate whether the pulse shall be held until its end, or whether a program interruption shall cause also a pulse interruption. If no value or FALSE is specified, the pulse will be executed to its end, a program interruption will not reset the signal. If `pauseAtInterrupt` is set TRUE, the time will be put on hold upon program interruption and the signal is reset. When the program is continued the signal is set again for the remaining time. If the program is terminated (unloaded) the signal will not be reset.

Parameters:

pulseLengthMs	Pulse duration [ms]
[pulseValue]	TRUE = positive pulse, FALSE = negative pulse (default: TRUE)
[pauseAtInterrupt]]	Indicates if the pulse shall be interrupted upon program interruption (default: FALSE).

Example:

```
Out5.Pulse(500)     // sets signal to TRUE for half a second
Out6.Pulse(2000, TRUE, TRUE)     // sets signal to TRUE for two seconds
    // when interrupted, the time is put on hold and signal is set
    // to FALSE
    // time. When the program is continued the signal is set again for
    // the remaining time.
```

BOOLSIGOUT.Connect

Establish a connection of a signal with a state variable.

```
BOOLSIGOUT.Connect (
VAR_IN status : STATECONNECTION
)
```

The value of the signal always reflects the value of the connected status flag. The connection can be released by execution another BOOLSIGOUT-command (`Set` or `Pulse` or `Connect` with another status flag).

Parameters:

status	State variable which should be connected with the digital signal.

```
Out5.Connect(ROBOT_MOVING)    // Connect with status ROBOT_MOVING
 // Signal gets TRUE whenever robot is moving on a programmed path
WaitBool(In3, FALSE)     // Wait until input signal is set
Out5.Set(FALSE)     // Disconnect from status
// ROBOT_MOVING and reset the signal
```

STATECONNECTION

Enumeration of the available status flags which can be connected with a digital output signal.

Elements:

CURRENT_PROGRAM_RUNNIG	TRUE whenever the program in which the status connection has been established is running. Useful, e.g., to react on a program interruption.
ROBOT_PROGRAM_RUNNING	TRUE whenever a robot program is running (this can be the program in which the status connection has been established, or another program).
ROBOT_MOVING	TRUE whenever robot is moving on a programmed path. I.e. ROBOT_ MOVING is not set, e.g., during jogging.

6.6 System and Extensions

6.6.1 Unit ROBOTDATA

This block offers access to a number of state variables of a robot. For each robot an instance of this type is available.

Member variables

name	STRING READONLY	name of the robot
index	DINT READONLY	Index of robot in the global robot array
axisSetPos	AXISPOSREF READONLY	Current axis position
axisPosMin	AXISPOSSTATIC READONLY	Position structure containing the lower position limit for each axis
axisPosMax	AXISPOSSTATIC READONLY	Position structure containing the upper position limit for each axis
nrOfMainJoints	DINT READONLY	Number of main axes
nrOfAuxJoints	DINT READONLY	Number of auxiliary axes
cartSetPos	CARTPOSREF READONLY	Current TCP-position
isOnPathEnd	BOOL READONLY	Indicates that the robot has reached the end of the actual path. Useful e.g. to synchronize a robot movement with a peripheral IO via the KAIRO program.
isReferenced	BOOL READONLY	Referenced-state of robot (Homing state)
axisIsReferenced	ARRAY[1..9] OF BOOL READONLY	Referenced-state of all drives

6.6.2 Unit CLOCK

The clock may be used for simple time measurement tasks in the program. The time is measured in milliseconds.

Member variables

timeMs	DINT	Time when clock was stopped [ms]

Overview of methods

- Reset - Reset the clock
- Start - Start the clock
- Stop - Stop the clock
- Read - Read time
- ToString - Convert to a string of format tt hh:mm:ss.ms

CLOCK.Reset

Resets the clock.

The clock is reset even if the clock is currently running.

```
CLOCK.Reset ()
```

Example:

```
Clk1.Reset() //clock is reset
```

CLOCK.Start

Starts the clock.

If the clock is started more often than once the previously measured time is lost.

```
CLOCK.Start ()
```

The starting time is stored in the clock.

Example:

```
Clk1.Start() // clock is started
```

CLOCK.Stop

Stops the clock.

The clock can only be stopped if it has been started previously.

```
CLOCK.Stop ()
```

The stopping time is stored in clock and the time difference to the starting time is calculated.

Example:

```
Clk1.Stop() // clock is stopped
```

CLOCK.Read

Read the measured time. If measurement is pending, the current intermediate time is read, otherwise the stopped time.

```
CLOCK.Read (
) : DINT
```

Return value:

	Measured time or current intermediate time [ms]

Example:

```
Value := Clk1.Read() // clock is read
```

CLOCK.ToString

Convert time to text of the format tt hh:mm:ss.ms

When measurement is pending the current intermediate time is read and formatted, otherwise the stopped time.

```
CLOCK.ToString (
) : STRING
```

Return value:

	Measured time or current intermediate time [ms], converted to format tt hh:mm:ss.ms

Example:

```
value := Clk1.ToString() /// clock is read; time is returned as
     // formatted text
```

6.6.3 Unit TIMER

A timer can be set to a certain time. When this time has passed a flag is signaled. Such a flag may be used within a waiting condition.

If the program is interrupted while the timer is running the remaining time to signal is stored. Upon program continuation the timer is reset with the remaining time.

Member variables

q	BOOL	Flag signalizing a timer event

Methods:

- Start - Start the timer
- Stop - Stop the timer

TIMER.Start

Starts the timer.

If a timer is started more often than once, previous starts are discarded.

The timer flag is automatically reset upon timer start.

```
TIMER.Start (
timeMs : DINT
)
```

Example:

```
Tm1.Start(2000)     // start a 2-seconds-timer
WAIT Tm1.q     // wait until timer is signaled
```

TIMER.Stop

Stops the timer. .

The timer can only be stopped if it has been started previously.

```
TIMER.Stop (
) : DINT
```

The timer flag stays unmodified when Stop() is called.

Return value:

	Remaining time to timer signal [ms]

Example:

```
r := Tm1.Stop()     // timer is stopped
Info("Remaining time: %1 ms", r)     // display remaining time
```

6.6.4 Synchronization point SYNC

Unit to synchronize the execution of programs running parallel.

Information

It is not reasonable to create SYNC units in the program scope. Typically SYNC units are defined in the MULTI ROBOT scope.

Methods:

- Sync Synchronization of programs running parallel

Command to synchronize the execution of two running programs. Thus it is, e.g., possible to synchronize the movement of two robots.

```
SYNC.Sync(
syncNr : DINT
) : BOOL
```

Parameter:

syncNr	Number of the synchronization point

The first program which reaches a sync-macro sets the given syncNr and waits until the synchronization point is cleared. I.e. until another program resets the syncNr.

If a program reaches an already set synchronization point, this synchronization point is reset. Afterwards both programs continue.

Whenever a synchronization point with a set syncNr differing from the given syncNr is reached, an error is set and the program execution is stopped.

Example:

Robot 1

```
Lin(p1)
SyncRobs.Sync(3)     // Waiting for the second robot
Lin(p2)    // Movement is executed after second robot has reached synchro-
nization point
```

Robot 2

```
Lin(p1)
SyncRobs.Sync(3)     // Waiting for the first robot
Lin(p2)    // Movement is executed after second robot has reached synchro-
nization point
```

7 Global variables

Some variables are automatically created by the system and may be used by end-user programs.

World: WORLDREFSYS

WORLD-coordinate system. May be set as a reference system. All other Cartesian systems refer to this system, directly or indirectly.

Flange: TOOLSTATIC

The robot flange may be used as a tool and is always available. Flange represents a tool of length zero with zero orientation offset (all six values of the frame are zero).

DefaultTool : TOOLSTATIC

A default tool may be configured. Variable will be instantiated for each robot in the corresponding GLOBAL-project.

RobotData : ROBOTDATA

Contains robot state information. Variable will be instantiated for each robot in the corresponding GLOBAL-project.

RobotBase : REFSYSSTATIC

RobotBase can be used as reference system. Variable will be instantiated for each robot in the corresponding GLOBAL-project.

8 Extending KAIRO instruction set

Extending KAIRO instruction set

The main parts of the system project are delivered as encrypted Teachtalk-archives (`RcCore.tta` and `RcInternal.tta`) and cannot be modified. The definition of the KAIRO instructions for end-users is not encrypted (subdirectory 'userapi') and can be adopted or extended for customer needs. The instructions defined in the open part of the system project access the encrypted core over an application interface defined in an open Teachtalk archive (`RcInterface.tta`).

This application interface is held compatible over several versions and allows the customer to program its own extensions and modification in the KAIRO instruction set. If the customer uses parts of the encrypted core which are not part of the defined application interface compatibility with future versions is not guaranteed.