Quickguide

KRL-Syntax

KUKA Roboter

KSS Relase 8.x

Contents – KRL Syntax

Variables and declarations

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

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

Declaration and initialization of variables:
- Variables (simple and complex) must be declared in the SRC file before the INI line and initialized after the INI line.
- Variables can optionally also be declared and initialized in a local or global data list.
- In order to place syntax before the INI line, the DEF line must be activated:
  Open file > **Edit** > **View** > **DEF line**

> The declaration and initialization in local and global data lists must be located in one line.

Syntax:

```
DECL Data_Type Variable_Name
```

Example:

```
DECL INT Counter ; Declaration
INI                              ; in the *.src file
Counter = 5
```

or

```
DECL INT Counter = 5         ; in the *.dat file
```

Simple data types – INT, REAL, BOOL, CHAR

Data type	Integer	Real	Boolean	Character
Keyword	*INT*	*REAL*	*BOOL*	*CHAR*
Meaning	Integer	Floating-point number	Logic state	1 character
Range of values	$-2^{31} \ldots 2^{31}-1$	±1.1E-38 … ±3.4E+38	TRUE, FALSE	ASCII characters
Example	32	1.43	TRUE	“A”

Arrays

Arrays in KRL:

- Arrays are used to group objects of the same data type to form a data object.
- An array may be of any data type.
- The index always starts at 1.
- Only the data type Integer is allowed for the index.
- Besides constants and variables, arithmetic expressions are also allowed for assigning values to array elements.

Syntax:

```
DECL  Data_Type Variable_Name[No_of_Array_Elements]
```

Value assignment:

```
Variable_Name[Array_Index] = Value
```

Example:

```
DECL REAL Measurement[5];Declaration of a REAL array
                  ;"Measurement" with five array elements
Measurement[3] = 7.23   ;Value assignment for the third
                  ;array element in the Measurement array
```

In the case of multidimensional arrays, the number of array elements is separated off by commas.
A maximum of 3 dimensions are possible.

Example:
```
DECL BOOL Matrix[2,4];Declaration of the 2-dimensional
                     ;Boolean array "Matrix"
Matrix[1,2] = TRUE   ;Value assignment for the
                     ;2-dimensional array element [1,2]
```

Structures – STRUC

Structures in KRL:

- A structure is a combination of different data types.
- A structure is initialized by means of an aggregate (not all the parameters have to be specified).
- A structure element can be initialized with a point separator or an aggregate.
- The order of the parameters is insignificant.

Syntax:
```
STRUC Structure name Data_Type1 A, B, Data_Type2 C, D …
```

Example of a predefined structure:

```
STRUC E6POS REAL X, Y, Z, A, B, C, E1, E2, E3,
                 E4, E5, E6, INT S, T
```

Example of value assignments with point separator and aggregate:

```
DECL POS Position   ;Declaration of the variable
                    ;"Position" of type POS
Position.X = 34.4   ;Value assignment for the X
                    ;component with the point separator
Position.Y = value  ;Value assignment for the Y
                    ;component with the point separator
Position = {X 34.4, Y -23.2} ;Value assignment with
                             ;variables not possible
```

Enumeration type – ENUM

Enumeration type in KRL:

- Each constant may only occur once in the definition of the enumeration type.
- The constants are freely definable names.
- An **ENUM** variable can only take on the values of its constants.
- In the value assignment, a **#** sign must always be placed before the constant.

Syntax:

```
ENUM Enumeration_Type_Name Constant_1, Constant_n
```

Example of a predefined enumeration type:

```
ENUM MODE_OP T1, T2, AUT, EX, INVALID
```

Example:

```
ENUM Form Straight, Angle, T_Piece, Star  ;Declaration
                                 ;of the enumeration type "Form"
DECL Form Part      ;Declaration of the variable PART of
                    ;type Form
Part = #Straight     ;Value assignment for the
                     ;enumeration type "Form"
```

Data manipulation

Arithmetic operators

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

Operands	INT	REAL
INT	INT	REAL
REAL	REAL	REAL

Geometric operator

The geometric operator “:” performs frame linkage.

Left operand (reference CS)	Operator	Right operand (target CS)	Result
POS	:	POS	POS
POS	:	FRAME	FRAME
FRAME	:	POS	POS
FRAME	:	FRAME	FRAME

Relational operators

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

Logic operators

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

Bit operators

Operator	Operand number	Description
B_NOT	1	Inversion (bit-by-bit)
B_AND	2	AND (bit-by-bit)
B_OR	2	OR (bit-by-bit)
B_EXOR	2	Exclusive OR (bit-by-bit)

Priority of operators

Priority	Operator					
1 high	NOT	B_NOT				
2	*	/				
3	+	-				
4	AND	B_AND				
5	EXOR	B_EXOR				
6	OR	B_OR				
7 low	==	<>	<	>	>=	<=

The following applies to all operators in KRL:

- Bracketed expressions are processed first.
- Non-bracketed expressions are evaluated in the order of their priority.
- Logic operations with operators of the same priority are executed from left to right.

Motion programming

Coordinate systems

Coordinate system	System variable	Status
World coordinate system	$WORLD	Write-protected
Robot coordinate system	$ROBROOT	Write-protected (can be changed in $MACHINE.DAT)
Tool coordinate system	$TOOL	Writable
Base coordinate system	$BASE	Writable

Working with coordinate systems:

Tool, base and load data selection

```
$TOOL = TOOL_DATA[n]   ;n tool number 1..16
$BASE = BASE_DATA[n]   ;n base number 1..32
$LOAD = LOAD_DATA[n]   ;n load data number 1..16
```

If $TOOL is changed during program execution, $LOAD must also be adapted accordingly. Otherwise the robot will be moved with the wrong load data, which might result in loss of warranty.

On delivery, the base coordinate system corresponds to the world coordinate system:
$BASE = $WORLD
The tool coordinate system corresponds to the flange coordinate system.

- Robot guides the **tool**:
  $IPO_MODE = #BASE
- Robot guides the **workpiece**:
  $IPO_MODE = #TCP

Status and Turn

- The entries “S” and “T” in a `POS/E6POS` structure serve to select a specific, unambiguously defined robot position where several different axis positions are possible for the same point in space.
- The specification of Status and Turn is saved for every position, but is **only** evaluated for `PTP` motions. To enable the robot to start from an unambiguous position, the first motion in a program must always be a `PTP` motion.

Point-to-point motions (PTP)

Syntax:

```
PTP Point_Name           ; PTP motion to point name
PTP {Point}              ; PTP motion to aggregate
specification (absolute position), e.g. PTP {A1 -45}
PTP_REL {Point}          ; PTP motion to aggregate
specification (motion relative to the previous point)
PTP Point_Name C_PTP     ; PTP motion to point name with
                           approximate positioning
```

Unspecified components in the aggregate are adopted from the preceding position.

The approximation distance, which is normally set in the inline form, must be entered during expert programming in KRL. It is also possible to set the max. axis velocities and accelerations.

System variables	Unit	Function
$VEL_AXIS[Axis_Number]	%	Axis velocity
$ACC_AXIS[Axis_Number]	%	Axis acceleration
$APO.CPTP	*1	Approximation range
*1 depends on the registry entry (see Page 44)		

CP motions (LIN and CIRC)

Syntax:

```
LIN Point_Name       ;LIN motion to point name
LIN_REL {Point}      ;LIN motion to aggregate specif. (absolute
                     ;position), e.g. LIN_REL {x 300, Z 1000}
CIRC Auxiliary_Point, End_Point, CA Angle
;CIRC motion to end point via auxiliary point, with
specification of the angle
CIRC {Auxiliary_Point}, {End_Point}, CA Angle
;CIRC motion to end point via auxiliary point, with absolute
position specifications
 in aggregates, and specification of the angle
CIRC_REL {Auxiliary_Point}, {End_Ppoint}, CA Angle
;CIRC motion to end point via auxiliary point, with relative
position specifications in aggregates (relative to the
preceding point), and specification of the angle
```

Approximate positioning with CP motions:

System variable	Unit	Description	Keyword
$APO.CDIS	mm	Distance criterion	**C_DIS**
$APO.CORI	°	Orientation criterion	**C_ORI**
$APO.CVEL	%	Velocity criterion	**C_VEL**

The keyword for approximate positioning is appended at the end of the normal **LIN** or **CIRC** command.
e.g. **LIN *Point_Name C_DIS***

The path, swivel and rotational velocities and accelerations must be initialized before a CP motion can be executed. Otherwise the values saved in $config.dat are used.

System variables	Unit	Max. value	Function
$VEL.CP	m/s	3	CP velocity
$VEL.ORI1 *1	°/s	400	Swivel velocity
$VEL.ORI2 *1	°/s	400	Rotational velocity
$VEL_AXIS[4]-[6] *2	%	100	Wrist axis velocity
$ACC.CP	m/s^2	10	CP acceleration
$ACC.ORI1 *1	$°/s^2$	1000	Swivel acceleration
$ACC.ORI2 *1	$°/s^2$	1000	Rotational acceleration
$ACC_AXIS[4]-[6] *2	%	100	Wrist axis acceleration

*1 Required information for $ORI_TYPE = #CONSTANT or #VAR
*2 Required information for $ORI_TYPE = #JOINT

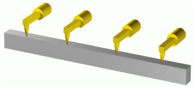
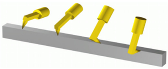
Computer advance run

- $ADVANCE = 0, approximation not possible, every point is positioned exactly.
- To make approximate positioning possible, a computer advance run of at least 1 must be set:
  $ADVANCE = 1 (default value is 3, maximum value is 5)
- Instructions and data that influence the periphery trigger an advance run stop. (e.g. HALT, WAIT, PULSE, ANIN ON/OFF, ANOUT ON/OFF, $IN[x], $OUT[x], $ANIN[x], $ANOUT[x])
- In applications where the advance run stop should be prevented, the CONTINUE command must be programmed immediately before the relevant instruction. However, this command **only** affects the next program line (even if this line is empty).
- The advance run stop can be forced without changing the $ADVANCE variable, using WAIT SEC 0.

Default settings of $ADVANCE:

	in the **system**	by **BAS (#INITMOV,0)**
$ADVANCE	0	3

Orientation control with linear motions

Variable	Value	Description
$ORI_TYPE	#CONSTANT	The orientation remains constant during the path motion.  The programmed orientation is disregarded for the end point.
	#VAR	During the path motion, the orientation changes continuously from the start point to the end point. 
	#JOINT	During the path motion, the orientation of the tool changes continuously from the start position to the end position. The wrist axis singularity ($\alpha5$) is avoided.

Orientation control with circular motions

Variable	Value	Description
$ORI_TYPE	#CONSTANT	The orientation remains constant during the circular motion. The programmed orientation is disregarded for the end point.
	#VAR	During the circular motion, the orientation changes continuously from the start orientation to the orientation of the end point.
	#JOINT	During the circular motion, the orientation of the tool changes continuously from the start position to the end position ($\alpha5$).
$CIRC_TYPE [*1]	#BASE	Space-related orientation control during the circular motion
	#PATH	Path-related orientation control during the circular motion
[*1] $CIRC_TYPE is meaningless if $ORI_TYPE = #JOINT.		

Program execution control

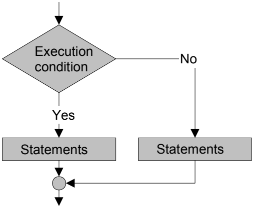
General information regarding loops

- Loops are required for repeating program sections.
- A distinction is made between counting loops and conditional loops.
- A jump into the loop from outside is not allowed and is refused by the controller.
- The nesting of loops is an advanced programming technique.

Conditional branch IF/THEN/ELSE

Syntax:

```
IF Execution_Condition THEN
   Statements
ELSE ; optional
   Statements
ENDIF
```



Example:

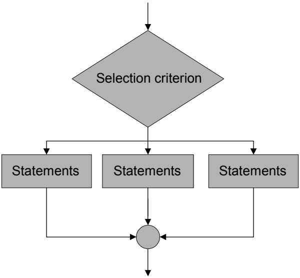
```
IF $IN[10] == FALSE THEN
   PTP HOME
ENDIF
```

The execution condition can consist of several components. If it is composed of several variables, brackets must be used. This applies both to IF branches and to **WHILE** and **REPEAT** loops.

Example:

```
IF (Counter1 == 50) AND (Counter2 == 100) THEN
   PTP HOME
Else
   PTP P1
ENDIF
```

Switch statements SWITCH/CASE



Syntax:

```
SWITCH Selection_Criterion
   CASE 1
      Statements
   CASE n
      Statements
   DEFAULT
      Statements
ENDSWITCH
```

Example:

```
SWITCH $MODE_OP
    CASE #T1
         $OUT[1] = TRUE
    CASE #T2
         $OUT[2] = TRUE
    CASE #AUT, #EXT
         $OUT[3] = TRUE
ENDSWITCH
```

Jump statement GOTO

Syntax:

```
GOTO Marker
…
Marker:
```

Example:

```
GOTO Calculation
…
Calculation:
```

Since GOTO statements very quickly lead to a loss of structure and clarity within a program, they should be avoided if at all possible. Every GOTO statement can be replaced by a different loop instruction.

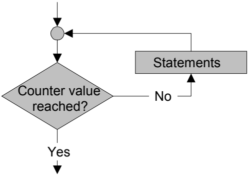
Counting loop FOR

Syntax:

```
FOR Counter = Start TO End STEP Increment
   Statements
ENDFOR
```

Example:

```
INT Counter
...
FOR Counter = 6 TO 1 STEP -1
   $VEL_AXIS[i] = 100
ENDFOR
```



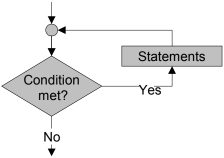
Rejecting loop WHILE

Syntax:

```
WHILE Condition
   Statements
ENDWHILE
```

Example:

```
WHILE $IN[4] == TRUE
   PULSE($OUT[2], TRUE, 1.2)
   CIRC AUX, END, CA 360
ENDWHILE
```



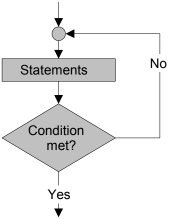
Non-rejecting loop REPEAT

Syntax:

```
REPEAT
   Statements
UNTIL Condition
```

Example:

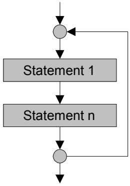
```
REPEAT
   PULSE($OUT[2], TRUE, 1.2)
   CIRC AUX, END, CA 360
UNTIL $IN[4] == TRUE
```



Endless loop LOOP

Syntax:

```
LOOP
   Statements
ENDLOOP
```



Example:

```
LOOP
   PTP Pos_1
   PTP Pos_2
   PTP Pos_3
ENDLOOP
```

Premature termination of loop execution

Syntax:

```
EXIT
```

Example:

```
DEF EXIT_PRO()
PTP HOME
LOOP                 ;Start of endless loop
   PTP Pos_1
   PTP Pos_2
   IF $IN[1] == TRUE THEN
      EXIT           ;Terminate when input TRUE and
   ENDIF             ;move to HOME
   PTP Pos_3
ENDLOOP              ;End of endless loop
PTP HOME
END
```

`EXIT` can be used for **all** loop types. Only the current loop is terminated.
If loops are nested, multiple `EXIT` commands are required.

Waiting for an event WAIT FOR

Syntax:
```
WAIT FOR Condition
```

Example:
```
WAIT FOR $IN[5] ; Wait until input 5 is TRUE
```

```
CONTINUE
WAIT FOR $OUT[5] == FALSE ;Wait until input 5
                          ;is FALSE
```

- If the logic expression (*condition*) is already **TRUE** when **WAIT FOR** is called, the advance run stop is triggered, thereby triggering exact positioning nevertheless. If approximate positioning is required, the **CONTINUE** command must be used.

> i The advance run pointer is responsible for the evaluation here; subsequent changes are therefore no longer detected.

- If the expression is **FALSE**, exact positioning is carried out and the robot stops until the condition takes the value **TRUE**.

Wait times WAIT SEC

Syntax:
```
WAIT SEC Time
```

Example:
```
PTP Pos_4
        WAIT SEC 7          ;Wait for 7 seconds
                            ;to elapse
PTP Pos_5
```

> i *Time* is an arithmetic **REAL** expression specifying the program interruption in seconds.
> If the value is zero or negative, the program does not wait.

Stopping the program

The **HALT** statement is used to stop programs.

The last motion instruction to be executed will be completed. The program can be resumed by pressing the START key. This function is used only for testing.

Syntax:

```
HALT
```

Example:

```
PTP Pos_4
HALT      ;Stop until the Start key is pressed again
PTP Pos_5
```

Special case:
In an interrupt routine, program execution is only stopped after the advance run has been completely executed.

Inputs/outputs

Binary inputs/outputs

Setting outputs at the end point:

Syntax:
```
$OUT[No] = Value          or          $IN[No]
```

Setting outputs at the approximated end point:

Syntax:
```
$OUT_C[No] = Value
```

Argument	Type	Function
No	INT	Input/output number [1 … 1024, 2048 or 4096, depending on $SET_IO_SIZE]
Value	BOOL	TRUE: Input/output is set FALSE: Input/output is reset

Digital inputs/outputs – signal declaration

- Inputs/outputs can be assigned names.
  The signal declaration must be located in the declaration section or in a data list.
- Signal declarations must be specified without gaps and in ascending sequence.
- A maximum of 32 inputs or outputs can be combined.
- An output may appear in more than one signal declaration.

Syntax:
```
SIGNAL Variable $OUT[No] TO $OUT[No]        or

SIGNAL Variable $IN[No] TO $IN[No]
```

Argument	Type	Function
No	INT	Input/output number [1 ... 4096]
Variable	Name	Name of the declared signal variable

Pulse outputs

Syntax:

```
PULSE ( $OUT[No], Value, Time )
```

Argument	Type	Function
No	INT	Output number [1 ... 4096]
Value	BOOL	**TRUE**: Output is set **FALSE**: Output is reset
Time	REAL	Range of values 0.012s ... 2^{31}s (increment: 0.1 s, the controller rounds values to the nearest tenth of a second)

Analog inputs/outputs

Syntax:

```
$ANOUT[No] = Value          or          $ANIN[No]
```

Argument	Type	Function
No	INT	Analog input/output [1 ... 32]
Value	REAL	Analog output voltage [-1.0 … +1.0], corresponds to ±10 V

Starting cyclical analog output:

Syntax:

```
ANOUT ON Signal_Name = Factor * Control_Element <±Offset>
<DELAY = Time> <MINIMUM = u1> <MAXIMUM = u2>
```

Ending cyclical analog output:

Syntax:

```
ANOUT OFF Signal_Name
```

A maximum of 4 cyclical analog outputs may be active simultaneously.

Argument	Type	Function
Signal_Name	Name	Analog output declared with **SIGNAL**
Factor	REAL	Any factor, as variable, signal or constant
Control_Element	REAL	Influence analog voltage by means of variable or signal
Offset	REAL	Optional offset as constant
Time	REAL	Positive or negative delay (in seconds)
u1	REAL	Minimum voltage (perm. values -1.0 ... 1.0)
u2	REAL	Maximum voltage (perm. values -1.0 ... 1.0)

Starting cyclical reading of an analog input:

Syntax:
```
ANIN ON Value = Factor * Signal_Name <±Offset>
```

Ending cyclical reading of an analog input:

Syntax:
```
ANIN OFF Signal_Name
```

Argument	Type	Function
Value	REAL	Save the result of the reading to a variable or signal
Factor	REAL	Any factor, as variable, signal or constant
Signal_Name	REAL	Analog input declared with **SIGNAL**
Offset	REAL	Optional offset as constant, variable or signal

A maximum of 3 analog inputs can be read cyclically at the same time.

Subprograms and functions

Local subprograms:

- An SRC file may contain a maximum of 255 local subprograms.
- The maximum nesting depth for subprograms is 20.
- Local subprograms are positioned after the dividing line in the main program; they start with **DEF** ***Program_Name ()*** and end with **END**.
- Local subprograms can be called repeatedly.
- Point coordinates are saved in the corresponding DAT list and are available in the entire program.

Global subprograms:

- Global subprograms have their own SRC and DAT files.
- Variables declared in global subprograms are not recognized in main programs, and vice versa, because each has its own DAT file.

Functions:

- Functions, like subprograms, are programs which are accessed by means of branches from the main program.
- A function returns a certain function value to the main program. For this reason, a function must always end with Return(x).
- The function also has a data type. This must always correspond to the function value.

Syntax:

```
DEFFCT Data_Type Function_Name()
      …
RETURN (Function_Value)
ENDFCT
```

Example:

```
Y = Square(x) ;Call the function in the main program
. . .
DEFFCT INT Square(Number:IN)
   INT Number                  ;Declaration of Number
   Number = Number * Number ;Square of Number
   RETURN(Number)              ;Return the content of the
ENDFCT            ;variable Number to the main program
```

Interrupt programming

- The interrupt declaration is an executable statement and must therefore not be situated in the declaration section.
- An existing declaration may be overwritten by another at any time.
- A maximum of 32 interrupts may be declared simultaneously.
- A maximum of 16 interrupts may be enabled simultaneously.
- Priorities 1 … 39 and 81 ... 128 are available.
  The values 40 ... 80 are reserved by the system.
- Priority level 1 is the highest possible priority.
- This event is detected by means of an edge when it occurs.

Declaration of an interrupt:

Syntax:

```
GLOBAL INTERRUPT DECL Priority WHEN Event DO Subprogram
```

Argument	Type	Function
Global		If an interrupt is declared in a subprogram, it will only be recognized in the main program if it is prefixed with the keyword **GLOBAL** in the declaration.
Priority	INT	If several interrupts occur at the same time, the interrupt with the highest priority is processed first, then those of lower priority. 1 ≙ highest priority.
Event	BOOL	Event that is to trigger the interrupt. Example: ▪ Boolean variable ▪ Signal ▪ Comparison
Subprogram	Name	The name of the interrupt program to be executed.

Switching interrupts on and off:

Syntax:
```
INTERRUPT ON Number
INTERRUPT OFF Number
```

Disabling and enabling interrupts:

Syntax:
```
INTERRUPT DISABLE Number
INTERRUPT ENABLE Number
```

If no number is specified, all declared interrupts are switched on or off.

Stopping active motions

The **BRAKE** statement brakes the robot motion. The interrupt program is not continued until the robot has come to a stop.

Syntax:
```
BRAKE        ;brakes the robot motion (ramp-down braking)
BRAKE F      ;brakes with maximum values
             ;(path-maintaining braking)
```

The **BRAKE** statement may only be used in interrupt programs. In other programs it leads to an error-induced stop.

After returning to the interrupted program, a motion stopped by means of **BRAKE** or **BRAKE F** in the interrupt program is resumed!

Canceling interrupt routines

The **RESUME** statement cancels all running interrupt programs and subprograms up to the level at which the current interrupt was declared.

Syntax:

```
RESUME
```

Example:

```
DEF MY_PROG( )
INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
SEARCH()
END
```

```
DEF SEARCH()
INTERRUPT ON 25
…
WAIT SEC 0 ; Stop advance run pointer; *1
INTERRUPT OFF 25
END
```

```
DEF ERROR()
BRAKE
PTP $POS_INT
RESUME
END
```

The **RESUME** statement may only be used in interrupt programs. In other programs it leads to an error-induced stop.

*1 When the **RESUME** statement is activated, the advance run pointer must not be at the level where the interrupt was declared.
It must be situated at least one level (subprogram) lower.

Trigger – path-related switching actions

Switching functions at the start or end point of a path

Syntax:

```
TRIGGER WHEN DISTANCE = Switching_Point  DELAY = Time
DO Statement <PRIO = Priority>
```

Argument	Type	Function
Switching_Point	INT	With **exact positioning points**: DISTANCE = 0 Reference to the start point of the motion. DISTANCE = 1 Reference to the end point of the motion. With **approximate positioning points**: DISTANCE = 0 Reference to the end of the preceding approximate positioning arc. DISTANCE = 1 Reference to the middle of the following approximate positioning arc.
Time	INT	DELAY delays the switching point by the specified time. The unit is **milliseconds**.
Statement		Subprogram call, value assignment to a variable or OUT statement.
Priority	INT	Every TRIGGER statement with a subprogram call must be assigned a priority. Values from 1 ... 39 and 81 ... 128 are permissible. The values 40 ... 80 are reserved for automatic priority allocation by the system. To use these, program PRIO = -1.

A maximum of 8 trigger statements can be used simultaneously.

Switching function at any point on the path

Syntax:

```
TRIGGER WHEN PATH = Distance DELAY = Time DO Statement
<PRIO = Priority>
```

This statement always refers to the following point.

Argument	Type	Function
Distance	INT	With **exact positioning points**: Distance from programmed end point. **End point is an approx. positioning point**: Distance of the switching action from the peak of the approximate positioning parabola. The switching point can be shifted back as far as the start point by entering a negative value for Distance. **Start point is an approx. positioning point**: The switching point can be brought forward, at most, as far as the start of the approximate positioning range. By entering a positive value for Distance, a shift as far as the next exact positioning point programmed after the trigger is possible. The unit is **millimeters**.
Time	INT	Using **DELAY**, it is possible to delay or advance the switching point by a certain amount of time relative to **PATH**. The switching point can only be moved within the switching range specified above. The unit is **milliseconds**.
Statement		Subprogram call, value assignment to a variable or **OUT** statement
Priority	INT	Every **TRIGGER** statement with a subprogram call must be assigned a priority. Values from 1 ... 39 and 81 ... 128 are permissible. The values 40 ... 80 are reserved for automatic priority allocation by the system. To use these, program **PRIO = -1**.

Message programming

Message properties

Defining the originator, number and message text:

Syntax:
```
Message = {MODUL[]"USER", NR 0815,MSG_TXT[]"Text"}
```

Element	Type	Function/elements
Message	KrlMsg_T	User-defined variable name of the message
MODUL[]	CHAR[24]	Originator of the message
NR	INT	Message number
MSG_TXT[]	CHAR[80]	Message text or message key. The placeholders %1, %2 and %3 can be used.

Assigning parameters to placeholders (e.g. %1):

Syntax:
```
Parameter[n]={PAR_TYPE #VALUE, PAR_TXT[]"Text"}
```

Element	Type	Function/elements
Parameter[n]	KrlMsgPar_T	User-defined variable name of the parameter Array index[n]=1…3
PAR_TYPE	KrlMsgParType_T	Type of parameter: #VALUE – parameter is inserted in the message text as text, INT, REAL or BOOL. #KEY – message from database #EMPTY – parameter is empty
PAR_TXT[]	CHAR[26]	Placeholder filled with text.
PAR_INT	INT	Placeholder filled with INT values.
PAR_REAL	REAL	Placeholder filled with REAL values.
PAR_BOOL	BOOL	Filled with BOOL values.

Defining the response to a message:

Syntax:

```
Option ={VL_STOP TRUE, CLEAR_P_RESET TRUE,
        CLEAR_P_SAW TRUE, LOG_TO_DB FALSE}
```

Element	Type	Function/elements
Option	KrlMsgOpt_T	User-defined variable for the message response
VL_STOP	BOOL	TRUE: Set_KrlMsg/Set_KrlDlg triggers an advance run stop FALSE: Set_KrlMsg/Set_KrlDlg does not trigger an advance run stop
CLEAR_P_RESET	BOOL	Delete message when the program is reset or deselected? TRUE: All status, acknowledgement and wait messages generated by Set_KrlMsg() with the corresponding message options are deleted. FALSE: The messages are not deleted. => Notification messages can only be deleted using the acknowledge keys.
CLEAR_P_SAW	BOOL	Delete message when a block selection is carried out using the softkey **Block selection**. TRUE: All status, acknowledgement and wait messages generated by Set_KrlMsg() with the corresponding message options are deleted. FALSE: The messages are not deleted.
LOG_TO_DB	BOOL	TRUE: Message is logged FALSE: Message is not logged

Labeling softkeys for dialog messages:

Syntax:
```
Softkey[n]={SK_TYPE #VALUE, SK_TXT[]"Name"}
```

Element	Type	Function/elements
Softkey[]	KrlMsgDlgSK_T	User-defined variable name for softkeys 1 to 7
SK_TYPE	KrlMsgParType_T	Type of softkey: #VALUE – SK_TXT[] corresponds to the softkey label #KEY – SK_TXT[] is the database key containing language-specific labels of the softkey #EMPTY – softkey is not assigned
SK_TXT	CHAR[10]	Softkey text

Generating a message

Syntax:
```
Ticket = Set_KrlMsg(#TYPE, Message, Parameter[], Option)
```

Element	Type	Function/elements
Ticket	INT	User-defined variable for the return value. This is used to delete defined messages. -1: Message not output >0: Message output successfully
TYPE	EKrlMsgType	Message type: #NOTIFY - Notification message #STATE - Status message #QUIT - Acknowledgement message #WAITING - Wait message
Message	KrlMsg_T	User-defined variable
Parameter[]	KrlMsgPar_T	User-defined variable
Option	KrlMsgOpt_T	User-defined variable

Generating dialogs

Syntax:

```
Ticket = Set_KrlDlg(Message, Parameter[], Softkey[],
                    Option)
```

Element	Type	Function/elements
Ticket	INT	User-defined variable for the return value. This is used to delete defined messages. -1: Dialog not output >0: Dialog output successfully
Message	KrlMsg_T	User-defined variable
Parameter[]	KrlMsgPar_T	User-defined variable
Softkey[]	KrlMsgDlgSK_T	User-defined variable
Option	KrlMsgOpt_T	User-defined variable

Checking a message:

Syntax:

```
Buffer = Exists_KrlMsg(Ticket)
```

Element	Type	Function/elements
Buffer	BOOL	User-defined variable for the return value of the Exists_KrlMsg() function. TRUE: Message still exists in the message buffer FALSE: Message has been deleted from the message buffer
Ticket	INT	The handle provided for the message by the Set_KrlMsg() function.

Checking a dialog:

Syntax:

```
Buffer = Exists_KrlDlg(Ticket, Answer)
```

Element	Type	Function/elements
Buffer	BOOL	User-defined variable for the return value of the Exists_KrlDlg() function. TRUE: Dialog still exists in the message buffer FALSE: Dialog has been deleted from the message buffer
Ticket	INT	The handle provided for the dialog by the Set_KrlDlg() function.
Answer	INT	Number of the softkey used to answer the dialog. The variable does not have to be initialized. It is written by the system. 1 to 7: respective key answer 0: If the dialog has not been answered, but cleared by other means.

Deleting a message:

Syntax:

```
Deleted = Clear_KrlMsg(Ticket)
```

Element	Type	Function/elements
Deleted	BOOL	User-defined variable for the return value of the Clear_KrlMsg() function. TRUE: Message has been deleted. FALSE: Message could not be deleted.
Ticket	INT	The handle provided for the message by the Set_KrlMsg() function is deleted.- 1: All messages initiated by this process are deleted. -99: All user-defined messages are deleted.

Reading the message buffer:

Syntax:

```
Quantity = Get_MsgBuffer(MsgBuf[])
```

Element	Type	Function/elements
Quantity	INT	User-defined variable for the number of messages in the message buffer.
MsgBuf[]	MsgBuf_T	Array containing all the messages in the buffer.

Programming examples

Notification message

```
;Required declarations
DECL KrlMsg_T Message
DECL KrlMsgPar_T Parameter[3]
DECL KrlMsgOpt_T Opt
DECL INT Ticket
```

```
;Compile message
Message = {Modul[]“USER“,Nr 2810,Msg_txt[]“This is a
          notification!”}
Parameter[1]= {Par_type #empty}
Parameter[2]= {Par_type #empty}
Parameter[3]= {Par_type #empty}
```

```
;Set message options
Opt = {VL_Stop TRUE, Clear_P_Reset TRUE,
       Clear_P_SAW FALSE, Log_To_DB FALSE}
```

```
;Generate message
Ticket = Set_KrlMsg (#Notify,Message,Parameter[],Opt)
```

Output:

Date	Time	USER	2810
This is a notification!			

Notification message with variables

```
;Required declarations and
DECL KrlMsg_T Message
DECL KrlMsgPar_T Parameter[3]
DECL KrlMsgOpt_T Opt
DECL INT Ticket, PartCount
;initialization
PartCount = 5
```

```
;Compile message
Message ={Modul[]"USER",Nr 2810,Msg_txt[] "%1 %2 %3."}
Parameter[1] = {Par_type #value,Par_txt[] "Count result:"}
Parameter[2] = {Par_type #value, Par_int 0}
Parameter[2].Par_int = PartCount
Parameter[3] = {Par_type #value, Par_txt[] "parts
                found!"}
```

```
;Set message options
Opt = {VL_Stop TRUE, Clear_P_Reset TRUE,
       Clear_P_SAW FALSE, Log_To_DB FALSE}
```

```
;Generate message
Ticket = Set_KrlMsg(#Notify,Message,Parameter[], Opt)
```

Output:

Date	Time	USER	2810
Count result: 5 parts found!			

Acknowledgement message

```
;Required declarations
DECL KrlMsg_T Message
DECL KrlMsgPar_T Parameter[3]
DECL KrlMsgOpt_T Opt
DECL INT Ticket

;Compile message
Message = {Modul[]"USER",Nr 1408,Msg_txt[]"No %1"}
Parameter[1] = {Par_type #value,Par_txt[]"cooling water!"}
Parameter[2]= {Par_type #empty}
Parameter[3]= {Par_type #empty}

;Set message options
Opt = {VL_Stop TRUE,Clear_P_Reset TRUE,
      Clear_P_SAW FALSE,Log_To_DB FALSE}

;Generate message
Ticket = Set_KrlMsg(#Quit, Message, Parameter[], Opt)

;Wait for acknowledgement
WHILE(Exists_KrlMsg(Ticket))
   WAIT SEC 0.1
ENDWHILE
```

Output:

Date	Time	USER	1408
No cooling water!			

	Status message

```
;Required declarations
DECL KrlMsg_T Message
DECL KrlMsgPar_T Parameter[3]
DECL KrlMsgOpt_T Opt
DECL INT Ticket
DECL BOOL Deleted

;Compile message
Message = {Modul[]"USER", Nr 1801, Msg_txt[]"No %1"}
Parameter[1] = {Par_type #value,Par_txt[]"cooling water"}
Parameter[2]= {Par_type #empty}
Parameter[3]= {Par_type #empty}

;Set message options
Opt = {VL_Stop TRUE,Clear_P_Reset TRUE,
      Clear_P_SAW TRUE, Log_To_DB FALSE}

;Generate message
Ticket = Set_KrlMsg(#State, Message, Parameter[], Opt)

;Delete message once cooling water ($IN[18]) is refilled
Repeat
  IF $IN[18] AND (Ticket > 0) THEN
    Deleted = Clear_KrlMsg(Ticket)
  ENDIF
Until Deleted
```

Output:

Date	Time	USER	1801
No cooling water!			

Wait message

```
;Required declarations
DECL KrlMsg_T Message
DECL KrlMsgPar_T Parameter[3]
DECL KrlMsgOpt_T Opt
DECL INT Ticket
DECL BOOL Deleted

;Compile message
Message = {Modul[]"USER",Nr 1801,
          Msg_txt[]"%1 Flag[%2]!"}
Parameter[1] = {Par_type #value, Par_txt[]"Wait for"}
Parameter[2] = {Par_type #value, Par_int 79}
Parameter[3]= {Par_type #empty}

;Set message options
Opt = {VL_Stop TRUE, Clear_P_Reset TRUE,
       Clear_P_SAW FALSE, Log_To_DB FALSE}

;Generate message
Ticket = Set_KrlMsg(#Waiting, Message,
                    Parameter[], Opt)

;Wait for flag[79]

REPEAT
UNTIL $FLAG[79] == TRUE

;Delete message once FLAG[79] is set
Deleted = Clear_KrlMsg (Ticket)
```

Output:

Date	Time	USER	1801
Wait for flag[79]			

	Dialog

```
;Required declarations
DECL KrlMsg_T Message
DECL KrlMsgPar_T Parameter[3]
DECL KrlMsgOpt_T Opt
DECL KrlMsgDlgSK_T Key[7]
DECL INT Ticket, Answer
```

```
;Compile message
Message = {Modul[]"USER", Nr 1508, Msg_txt[]"Select
           form"}
Key[1]= {SK_type #value, SK_txt[]"Circle"}
Key[2]= {SK_type #value, SK_txt[]"Triangle"}
Key[3]= {SK_type #value, SK_txt[]"Square"}
Key[4]= {SK_type #empty}
. . .
```

```
;Set message options
Opt = {VL_Stop TRUE, Clear_P_Reset TRUE,
      Clear_P_SAW TRUE, Log_To_DB FALSE}
```

```
;Generate message
Ticket = Set_KrlDlg(Message, Parameter[],Key[], Opt)
```

```
;Wait for answer
IF (Ticket > 0) THEN
   WHILE (Exists_KrlDlg (Ticket, Answer))
      WAIT SEC 0
   ENDWHILE

   SWITCH Answer
      CASE 1
      …
      CASE 2
      …
      CASE 3
      …
   ENDSWITCH
ENDIF
```

Output:

Date	Time	USER	1508
Select form!			
Circle	Triangle	Square	

Important system variables

Timers

The system variables `$TIMER[1]` ... `$TIMER[32]` serve the purpose of measuring time sequences. A timing process is started and stopped by means of the system variables `$TIMER_STOP[1]` ... `$TIMER_STOP[32]`.

Syntax:

```
$TIMER_STOP[No] = Value
```

Argument	Type	Function
No	INT	Timer number, range of values: 1 ... 32
Value	Bool	FALSE: start, TRUE: stop

Flags and cyclical flags

Static and cyclical flags are 1-bit memories and are used as global markers. $CYCFLAG is an array of Boolean information that is cyclically interpreted and updated by the system.

Syntax:

```
$FLAG[No] = Value

$CYCFLAG[No] = Value
```

Argument	Type	Function
No	INT	Flag number, range of values: 1 ... 1024 Cyclical flag number, range of values: 1 ... 256
Value	BOOL	FALSE: reset, TRUE: set

Overview

$AXIS_ACT

Meaning/function:	Current axis-specific robot position
Data type:	E6AXIS structure
Range of values/unit:	[mm, °]

$AXIS_INT

Meaning/function:	Axis-specific position at interrupt trigger
Data type:	E6AXIS structure
Range of values/unit:	[mm, °]

$MODE_OP

Meaning/function:	Current operating mode
Data type:	ENUM mode_op
Range of values/unit:	#T1, #T2, #AUT, #EX, #INVALID

$OV_PRO

Meaning/function:	Program override
Data type:	INTEGER
Range of values/unit:	0 ... 100 [%]

$POS_ACT

Meaning/function:	Current Cartesian robot position
Data type:	E6POS structure
Range of values/unit:	[mm, °]

$POS_INT

Meaning/function:	Cartesian position at interrupt trigger
Data type:	E6POS structure
Range of values:	[mm, °]

$POS_RET

Meaning/function:	Cartesian position at which the robot left the path
Data type:	E6POS structure
Range of values:	[mm, °]

$VEL_ACT

Meaning/function:	Current CP velocity
Data type:	REAL
Range of values/unit:	>0 ... $VEL_MA.CP [m/s]

Registry entries

Path:
C:\KRC\Util\Regutil

Name	Description
AddZuli_On/Off	The long texts in inline forms can be activated and deactivated.
AutoRobNameArchiveOn/Off	Archives are automatically labeled with the robot name.
DisableUserDel	The menu item “Edit → Delete” is deactivated.
EnableUserDel	The menu item “Edit → Delete” is activated.
DistCriterionPTPOn/Off	The approximation distance for PTP motions can be set to mm or %.
Hibernate	When switched off, the controller is set to Hibernate mode.
Lettering_StatuskeyOn/Off	Status keys are labeled with the configured names.
ToolDirectionZOn/Off	The tool direction is set to Z direction. The tool direction can alternatively be set by means of the $TOOL_Direction variable. Options: #X, #Y or #Z