Quickguide

KRL-Syntax

KUKA Roboter
KSS Release 8.x


Technische Änderungen ohne Beeinflussung der Funktion vorbehalten.

Die KUKA Roboter GmbH übernimmt keinerlei Haftung für etwaige Fehler in technischen Informationen, die in den Schulungen mündlich oder schriftlich übermittelt werden oder in den Unterlagen enthalten sind. Ebenso wird keine Haftung für daraus resultierende Schäden und Folgeschäden übernommen.

Verantwortlich für diese Schulungsunterlage: College Development (WSC-IC)

This documentation or excerpts therefrom may not be reproduced or disclosed to third parties without the express permission of the publishers. Other functions not described in this documentation may be operable in the controller. The user has no claims to these functions, however, in the case of a replacement or service work. We have checked the content of this documentation for conformity with the hardware and software described. Nevertheless, discrepancies cannot be precluded, for which reason we are not able to guarantee total conformity. The information in this documentation is checked on a regular basis, however, and necessary corrections will be incorporated in subsequent editions. Subject to technical alterations without an effect on the function.

KUKA Roboter GmbH accepts no liability whatsoever for any errors in technical information communicated orally or in writing in the training courses or contained in the documentation. Nor will liability be accepted for any resultant damage or consequential damage.

Responsible for this training documentation: College Development
## Contents – KRL Syntax

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents – KRL Syntax</td>
<td>1</td>
</tr>
<tr>
<td>Variables and declarations</td>
<td>3</td>
</tr>
<tr>
<td>Declaration and initialization of variables:</td>
<td>3</td>
</tr>
<tr>
<td>Simple data types – INT, REAL, BOOL, CHAR</td>
<td>4</td>
</tr>
<tr>
<td>Arrays</td>
<td>4</td>
</tr>
<tr>
<td>Structures – STRUC</td>
<td>5</td>
</tr>
<tr>
<td>Enumeration type – ENUM</td>
<td>6</td>
</tr>
<tr>
<td>Data manipulation</td>
<td>7</td>
</tr>
<tr>
<td>Arithmetic operators</td>
<td>7</td>
</tr>
<tr>
<td>Geometric operator</td>
<td>7</td>
</tr>
<tr>
<td>Relational operators</td>
<td>7</td>
</tr>
<tr>
<td>Logic operators</td>
<td>8</td>
</tr>
<tr>
<td>Bit operators</td>
<td>8</td>
</tr>
<tr>
<td>Priority of operators</td>
<td>8</td>
</tr>
<tr>
<td>Motion programming</td>
<td>9</td>
</tr>
<tr>
<td>Coordinate systems</td>
<td>9</td>
</tr>
<tr>
<td>Status and Turn</td>
<td>10</td>
</tr>
<tr>
<td>Point-to-point motions (PTP)</td>
<td>10</td>
</tr>
<tr>
<td>CP motions (LIN and CIRC)</td>
<td>11</td>
</tr>
<tr>
<td>Computer advance run</td>
<td>12</td>
</tr>
<tr>
<td>Orientation control with linear motions</td>
<td>13</td>
</tr>
<tr>
<td>Orientation control with circular motions</td>
<td>13</td>
</tr>
<tr>
<td>Program execution control</td>
<td>14</td>
</tr>
<tr>
<td>General information regarding loops</td>
<td>14</td>
</tr>
<tr>
<td>Conditional branch IF/THEN/ELSE</td>
<td>14</td>
</tr>
<tr>
<td>Switch statements SWITCH/CASE</td>
<td>15</td>
</tr>
<tr>
<td>Jump statement GOTO</td>
<td>15</td>
</tr>
<tr>
<td>Counting loop FOR</td>
<td>16</td>
</tr>
<tr>
<td>Rejecting loop WHILE</td>
<td>16</td>
</tr>
<tr>
<td>Non-rejecting loop REPEAT</td>
<td>16</td>
</tr>
<tr>
<td>Endless loop LOOP</td>
<td>17</td>
</tr>
<tr>
<td>Waiting for an event WAIT FOR</td>
<td>18</td>
</tr>
<tr>
<td>Wait times WAIT SEC</td>
<td>18</td>
</tr>
<tr>
<td>Stopping the program</td>
<td>19</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Data type</th>
<th>Integer</th>
<th>Real</th>
<th>Boolean</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>INT</td>
<td>REAL</td>
<td>BOOL</td>
<td>CHAR</td>
</tr>
<tr>
<td>Meaning</td>
<td>Integer</td>
<td>Floating-point number</td>
<td>Logic state</td>
<td>1 character</td>
</tr>
<tr>
<td>Range of values</td>
<td>(-2^{31} \ldots 2^{31}-1)</td>
<td>(\pm1.1\text{E}-38 \ldots \pm3.4\text{E}+38)</td>
<td>TRUE, FALSE</td>
<td>ASCII characters</td>
</tr>
<tr>
<td>Example</td>
<td>32</td>
<td>1.43</td>
<td>TRUE</td>
<td>“A”</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition or positive sign</td>
</tr>
<tr>
<td>–</td>
<td>Subtraction or negative sign</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operands</th>
<th>INT</th>
<th>REAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>INT</td>
<td>REAL</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
<td></td>
</tr>
</tbody>
</table>

## Geometric operator

The geometric operator “:” performs frame linkage.

<table>
<thead>
<tr>
<th>Left operand (reference CS)</th>
<th>Operator</th>
<th>Right operand (target CS)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>POS</td>
<td>:</td>
<td>POS</td>
<td>POS</td>
</tr>
<tr>
<td>POS</td>
<td>:</td>
<td>FRAME</td>
<td>FRAME</td>
</tr>
<tr>
<td>FRAME</td>
<td>:</td>
<td>POS</td>
<td>POS</td>
</tr>
<tr>
<td>FRAME</td>
<td>:</td>
<td>FRAME</td>
<td>FRAME</td>
</tr>
</tbody>
</table>

## Relational operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Permissible data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
<td>INT, REAL, CHAR, ENUM, BOOL</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal to</td>
<td>INT, REAL, CHAR, ENUM, BOOL</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>INT, REAL, CHAR, ENUM</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>INT, REAL, CHAR, ENUM</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>INT, REAL, CHAR, ENUM</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>INT, REAL, CHAR, ENUM</td>
</tr>
</tbody>
</table>
## Data manipulation

### Logic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>1</td>
<td>Inversion</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
<td>Logic AND</td>
</tr>
<tr>
<td>OR</td>
<td>2</td>
<td>Logic OR</td>
</tr>
<tr>
<td>EXOR</td>
<td>2</td>
<td>Exclusive OR</td>
</tr>
</tbody>
</table>

### Bit operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B_NOT</td>
<td>1</td>
<td>Inversion (bit-by-bit)</td>
</tr>
<tr>
<td>B_AND</td>
<td>2</td>
<td>AND (bit-by-bit)</td>
</tr>
<tr>
<td>B_OR</td>
<td>2</td>
<td>OR (bit-by-bit)</td>
</tr>
<tr>
<td>B_EXOR</td>
<td>2</td>
<td>Exclusive OR (bit-by-bit)</td>
</tr>
</tbody>
</table>

### Priority of operators

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 high</td>
<td>NOT, B_NOT</td>
</tr>
<tr>
<td>2</td>
<td>* /</td>
</tr>
<tr>
<td>3</td>
<td>+ -</td>
</tr>
<tr>
<td>4</td>
<td>AND, B_AND</td>
</tr>
<tr>
<td>5</td>
<td>EXOR, B_EXOR</td>
</tr>
<tr>
<td>6</td>
<td>OR, B_OR</td>
</tr>
<tr>
<td>7 low</td>
<td>== &lt; &gt; &gt;= &lt;=</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>Coordinate system</th>
<th>System variable</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>World coordinate system</td>
<td>$WORLD</td>
<td>Write-protected</td>
</tr>
<tr>
<td>Robot coordinate system</td>
<td>$ROBROOT</td>
<td>Write-protected (can be changed in $MACHINE.DAT)</td>
</tr>
<tr>
<td>Tool coordinate system</td>
<td>$TOOL</td>
<td>Writable</td>
</tr>
<tr>
<td>Base coordinate system</td>
<td>$BASE</td>
<td>Writable</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>System variables</th>
<th>Unit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VEL_AXIS[Axis_Number]</td>
<td>%</td>
<td>Axis velocity</td>
</tr>
<tr>
<td>$ACC_AXIS[Axis_Number]</td>
<td>%</td>
<td>Axis acceleration</td>
</tr>
<tr>
<td>$APO.CPTP</td>
<td>*1</td>
<td>Approximation range</td>
</tr>
</tbody>
</table>

*1 depends on the registry entry (see Page 44)
CP motions (LIN and CIRC)

Syntax:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIN Point_Name</td>
<td>LIN motion to point name</td>
</tr>
<tr>
<td>LIN_REL {Point}</td>
<td>LIN motion to aggregate specif. (absolute position), e.g. LIN_REL {x 300, Z 1000}</td>
</tr>
<tr>
<td>CIRC Auxiliary_Point, End_Point, CA Angle</td>
<td>CIRC motion to end point via auxiliary point, with specification of the angle</td>
</tr>
<tr>
<td>CIRC {Auxiliary_Point}, {End_Point}, CA Angle</td>
<td>CIRC motion to end point via auxiliary point, with absolute position specifications in aggregates, and specification of the angle</td>
</tr>
<tr>
<td>CIRC_REL {Auxiliary_Point}, {End_Point}, CA Angle</td>
<td>CIRC motion to end point via auxiliary point, with relative position specifications in aggregates (relative to the preceding point), and specification of the angle</td>
</tr>
</tbody>
</table>

Approximate positioning with CP motions:

<table>
<thead>
<tr>
<th>System variable</th>
<th>Unit</th>
<th>Description</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>$APO.CDIS</td>
<td>mm</td>
<td>Distance criterion</td>
<td>C_DIS</td>
</tr>
<tr>
<td>$APO.CORI</td>
<td>°</td>
<td>Orientation criterion</td>
<td>C_ORI</td>
</tr>
<tr>
<td>$APO.CVEL</td>
<td>%</td>
<td>Velocity criterion</td>
<td>C_VEL</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Component</th>
<th>Unit</th>
<th>Max. Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VEL_CP</td>
<td>m/s</td>
<td>3</td>
<td>CP velocity</td>
</tr>
<tr>
<td>$VEL_ORI1 *1</td>
<td>°/s</td>
<td>400</td>
<td>Swivel velocity</td>
</tr>
<tr>
<td>$VEL_ORI2 *1</td>
<td>°/s</td>
<td>400</td>
<td>Rotational velocity</td>
</tr>
<tr>
<td>$VEL_AXIS[4]-[6] *2</td>
<td>%</td>
<td>100</td>
<td>Wrist axis velocity</td>
</tr>
<tr>
<td>$ACC_CP</td>
<td>m/s²</td>
<td>10</td>
<td>CP acceleration</td>
</tr>
<tr>
<td>$ACC_ORI1 *1</td>
<td>°/s²</td>
<td>1000</td>
<td>Swivel acceleration</td>
</tr>
<tr>
<td>$ACC_ORI2 *1</td>
<td>°/s²</td>
<td>1000</td>
<td>Rotational acceleration</td>
</tr>
<tr>
<td>$ACC_AXIS[4]-[6] *2</td>
<td>%</td>
<td>100</td>
<td>Wrist axis acceleration</td>
</tr>
</tbody>
</table>

*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:

<table>
<thead>
<tr>
<th>System Component</th>
<th>in the system</th>
<th>by BAS (#INITMOV,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ADVANCE</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

12/44 07.12.00 en
## Orientation control with linear motions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ORI_TYPE</td>
<td>#CONSTANT</td>
<td>The orientation remains constant during the path motion. The programmed orientation is disregarded for the end point.</td>
</tr>
<tr>
<td></td>
<td>#VAR</td>
<td>During the path motion, the orientation changes continuously from the start point to the end point.</td>
</tr>
<tr>
<td></td>
<td>#JOINT</td>
<td>During the path motion, the orientation of the tool changes continuously from the start position to the end position. The wrist axis singularity ($\alpha_5$) is avoided.</td>
</tr>
</tbody>
</table>

## Orientation control with circular motions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ORI_TYPE</td>
<td>#CONSTANT</td>
<td>The orientation remains constant during the circular motion. The programmed orientation is disregarded for the end point.</td>
</tr>
<tr>
<td></td>
<td>#VAR</td>
<td>During the circular motion, the orientation changes continuously from the start orientation to the orientation of the end point.</td>
</tr>
<tr>
<td></td>
<td>#JOINT</td>
<td>During the circular motion, the orientation of the tool changes continuously from the start position to the end position ($\alpha_5$).</td>
</tr>
<tr>
<td>$CIRC_TYPE$</td>
<td>#BASE</td>
<td>Space-related orientation control during the circular motion</td>
</tr>
<tr>
<td></td>
<td>#PATH</td>
<td>Path-related orientation control during the circular motion</td>
</tr>
</tbody>
</table>

* $CIRC_TYPE$ is meaningless if $ORI_TYPE = #JOINT$. 

---

* Image and diagram related to orientation control are not translated.
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:

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

Example:

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

Rejecting loop WHILE

Syntax:

```plaintext
WHILE Condition
  Statements
ENDWHILE
```

Example:

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

Non-rejecting loop REPEAT

Syntax:

```plaintext
REPEAT
  Statements
UNTIL Condition
```

Example:

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

Syntax:

```plaintext
LOOP
Statements
ENDLOOP
```

Example:

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

Premature termination of loop execution

Syntax:

```plaintext
EXIT
```

Example:

```plaintext
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: \texttt{WAIT FOR \textit{Condition}}

Example: \texttt{WAIT FOR $\text{IN}[5]$ ; Wait until input 5 is TRUE}

\begin{itemize}
  \item If the logic expression (\textit{condition}) is already \textbf{TRUE} when \texttt{WAIT FOR} is called, the advance run stop is triggered, thereby triggering exact positioning nevertheless. If approximate positioning is required, the \texttt{CONTINUE} command must be used.
  \item If the expression is \textbf{FALSE}, exact positioning is carried out and the robot stops until the condition takes the value \textbf{TRUE}.
\end{itemize}

Wait times WAIT SEC

Syntax: \texttt{WAIT SEC \textit{Time}}

Example: \texttt{PTP Pos_4}
\begin{verbatim}
    WAIT SEC 7 ;Wait for 7 seconds
    ;to elapse
\end{verbatim}

\texttt{PTP Pos_5}

\begin{itemize}
  \item \textit{Time} is an arithmetic \textbf{REAL} expression specifying the program interruption in seconds. If the value is zero or negative, the program does not wait.
\end{itemize}
**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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>INT</td>
<td>Input/output number [1 … 1024, 2048 or 4096, depending on $SET_IO_SIZE]</td>
</tr>
</tbody>
</table>
| 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]  
SIGNAL Variable $IN[No] TO $IN[No]
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>INT</td>
<td>Input/output number [1 ... 4096]</td>
</tr>
<tr>
<td>Variable</td>
<td>Name</td>
<td>Name of the declared signal variable</td>
</tr>
</tbody>
</table>
Pulse outputs
Syntax: \[ \text{PULSE ( $OUT[No], Value, Time )} \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{No}</td>
<td>INT</td>
<td>Output number [1 ... 4096]</td>
</tr>
<tr>
<td>\textit{Value}</td>
<td>BOOL</td>
<td>\textit{TRUE}: Output is set \textit{FALSE}: Output is reset</td>
</tr>
<tr>
<td>\textit{Time}</td>
<td>REAL</td>
<td>Range of values 0.012s ... $2^{31}$s (increment: 0.1 s, the controller rounds values to the nearest tenth of a second)</td>
</tr>
</tbody>
</table>

Analog inputs/outputs
Syntax: $\text{ANOUT[No]} = \text{Value} \quad \text{or} \quad \text{ANIN[No]}$

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{No}</td>
<td>INT</td>
<td>Analog input/output [1 ... 32]</td>
</tr>
<tr>
<td>\textit{Value}</td>
<td>REAL</td>
<td>Analog output voltage [-1.0 ... +1.0], corresponds to ±10 V</td>
</tr>
</tbody>
</table>

Starting cyclical analog output:
Syntax: \[ \text{ANOUT ON Signal\_Name = Factor} \times \text{Control\_Element} <\pm \text{Offset}> <\text{DELAY} = \text{Time}> <\text{MINIMUM} = u1> <\text{MAXIMUM} = u2> \]

Ending cyclical analog output:
Syntax: \[ \text{ANOUT OFF Signal\_Name} \]

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

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

### 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
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>REAL</td>
<td>Save the result of the reading to a variable or signal</td>
</tr>
<tr>
<td>Factor</td>
<td>REAL</td>
<td>Any factor, as variable, signal or constant</td>
</tr>
<tr>
<td>Signal_Name</td>
<td>REAL</td>
<td>Analog input declared with SIGNAL</td>
</tr>
<tr>
<td>Offset</td>
<td>REAL</td>
<td>Optional offset as constant, variable or signal</td>
</tr>
</tbody>
</table>

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:
```plaintext
DEFFCT Data_Type Function_Name()
...
RETURN (Function_Value)
ENDFCT
```

Example:
```plaintext
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 variable Number to the main program
ENDFCT
```
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`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td></td>
<td>If an interrupt is declared in a subprogram, it will only be recognized in the main program if it is prefixed with the keyword <code>GLOBAL</code> in the declaration.</td>
</tr>
<tr>
<td>Priority</td>
<td>INT</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 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>
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
</table>
| **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.
Trigger – path-related switching actions

Switching function at any point on the path

Syntax:  
\[
\text{TRIGGER WHEN PATH = Distance DELAY = Time DO Statement} \\
\langle \text{PRIO = Priority} \rangle
\]

This statement always refers to the following point.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>INT</td>
<td>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.</td>
</tr>
<tr>
<td>Time</td>
<td>INT</td>
<td>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.</td>
</tr>
<tr>
<td>Statement</td>
<td></td>
<td>Subprogram call, value assignment to a variable or OUT statement</td>
</tr>
<tr>
<td>Priority</td>
<td>INT</td>
<td>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.</td>
</tr>
</tbody>
</table>
Message programming

Message properties

Defining the originator, number and message text:

Syntax: 

\[
Message = \{MODUL[], \text{"USER"}, \ NR \ 0815, MSG.TXT[] \text{"Text"} \}
\]

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

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

Syntax: 

\[
Parameter[n]=\{PAR_TYPE \ #VALUE, \ PAR.TXT[] \text{"Text"} \}
\]

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Function/elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter[n]</td>
<td>KrlMsgPar_T</td>
<td>User-defined variable name of the parameter Array index[n]=1…3</td>
</tr>
<tr>
<td>PAR_TYPE</td>
<td>KrlMsgParType_T</td>
<td>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</td>
</tr>
<tr>
<td>PAR_TXT[]</td>
<td>CHAR[26]</td>
<td>Placeholder filled with text.</td>
</tr>
<tr>
<td>PAR_INT</td>
<td>INT</td>
<td>Placeholder filled with INT values.</td>
</tr>
<tr>
<td>PAR_REAL</td>
<td>REAL</td>
<td>Placeholder filled with REAL values.</td>
</tr>
<tr>
<td>PAR_BOOL</td>
<td>BOOL</td>
<td>Filled with BOOL values.</td>
</tr>
</tbody>
</table>
Defining the response to a message:

Syntax:

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

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Function/elements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option</strong></td>
<td>KrImsgOpt_T</td>
<td>User-defined variable for the message response</td>
</tr>
<tr>
<td><strong>VL_STOP</strong></td>
<td>BOOL</td>
<td>TRUE: Set_KrlMsg/Set_KrlDlg triggers an advance run stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE: Set_KrlMsg/Set_KrlDlg does not trigger an advance run stop</td>
</tr>
<tr>
<td><strong>CLEAR_P_RESET</strong></td>
<td>BOOL</td>
<td>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. =&gt; Notification messages can only be deleted using the acknowledge keys.</td>
</tr>
<tr>
<td><strong>CLEAR_P_SAW</strong></td>
<td>BOOL</td>
<td>Delete message when a block selection is carried out using the softkey <strong>Block selection.</strong> TRUE: All status, acknowledgement and wait messages generated by Set_KrlMsg() with the corresponding message options are deleted. FALSE: The messages are not deleted.</td>
</tr>
<tr>
<td><strong>LOG_TO_DB</strong></td>
<td>BOOL</td>
<td>TRUE: Message is logged</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FALSE: Message is not logged</td>
</tr>
</tbody>
</table>
Labeling softkeys for dialog messages:

**Syntax:**

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

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

Generating a message

**Syntax:**

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

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

Syntax:  
\[
\text{Ticket} = \text{Set}_\text{KrlDlg} (\text{Message}, \text{Parameter}[], \text{Softkey}[], \text{Option})
\]

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Function/elements</th>
</tr>
</thead>
</table>
| **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:  
\[
\text{Buffer} = \text{Exists}_\text{KrlMsg} (\text{Ticket})
\]

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Function/elements</th>
</tr>
</thead>
</table>
| **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: \[ Buffer = \text{Exists\_KrlDlg}(Ticket, Answer) \]

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Function/elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>BOOL</td>
<td>User-defined variable for the return value of the \text{Exists_KrlDlg()} function. TRUE: Dialog still exists in the message buffer. FALSE: Dialog has been deleted from the message buffer.</td>
</tr>
<tr>
<td>Ticket</td>
<td>INT</td>
<td>The handle provided for the dialog by the \text{Set_KrlDlg()} function.</td>
</tr>
<tr>
<td>Answer</td>
<td>INT</td>
<td>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.</td>
</tr>
</tbody>
</table>
Deleting a message:

Syntax:  

\[
\text{Deleted} = \text{Clear_KrlMsg}(\text{Ticket})
\]

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Function/elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deleted</td>
<td>BOOL</td>
<td>User-defined variable for the return value of the Clear_KrlMsg() function. TRUE: Message has been deleted. FALSE: Message could not be deleted.</td>
</tr>
<tr>
<td>Ticket</td>
<td>INT</td>
<td>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.</td>
</tr>
</tbody>
</table>

Reading the message buffer:

Syntax:  

\[
\text{Quantity} = \text{Get_MsgBuffer}(\text{MsgBuf[]})
\]

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Function/elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantity</td>
<td>INT</td>
<td>User-defined variable for the number of messages in the message buffer.</td>
</tr>
<tr>
<td>MsgBuf[]</td>
<td>MsgBuf_T</td>
<td>Array containing all the messages in the buffer.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>USER</th>
<th>2810</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Count result: 5 parts found!</td>
<td></td>
</tr>
</tbody>
</table>
Message programming

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:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>USER</th>
<th>1408</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>No cooling water!</td>
</tr>
</tbody>
</table>
Message programming

### 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:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>USER</th>
<th>1801</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 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:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>USER</th>
<th>1801</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wait for flag[79]</td>
</tr>
</tbody>
</table>
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
    ENDFWHILE

    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 $\text{T}IME\text{R}[1] \ldots \text{T}IME\text{R}[32]$ serve the purpose of measuring time sequences. A timing process is started and stopped by means of the system variables $\text{T}IME\_\text{STOP}[1] \ldots \text{T}IME\_\text{STOP}[32]$.

Syntax: $\text{T}IME\_\text{STOP}[No] = \text{Value}$

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>INT</td>
<td>Timer number, range of values: 1 ... 32</td>
</tr>
<tr>
<td>Value</td>
<td>Bool</td>
<td>FALSE: start, TRUE: stop</td>
</tr>
</tbody>
</table>

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

Syntax: $\text{FLAG}[No] = \text{Value}$  
$\text{CYCFLAG}[No] = \text{Value}$

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
</table>
| 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, °]
**Important system variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning/function</th>
<th>Data type</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$POS_{INT}$</td>
<td>Cartesian position at interrupt trigger</td>
<td>E6POS structure</td>
<td>[mm, °]</td>
</tr>
<tr>
<td>$POS_{RET}$</td>
<td>Cartesian position at which the robot left the path</td>
<td>E6POS structure</td>
<td>[mm, °]</td>
</tr>
<tr>
<td>$VEL_{ACT}$</td>
<td>Current CP velocity</td>
<td>REAL</td>
<td>&gt;0 ... $VEL_{MA.CP}$ [m/s]</td>
</tr>
</tbody>
</table>
# Registry entries

**Path:**
C:\KRC\Util\Regutil

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddZuli_On/Off</td>
<td>The long texts in inline forms can be activated and deactivated.</td>
</tr>
<tr>
<td>AutoRobNameArchiveOn/Off</td>
<td>Archives are automatically labeled with the robot name.</td>
</tr>
<tr>
<td>DisableUserDel</td>
<td>The menu item “Edit → Delete” is deactivated.</td>
</tr>
<tr>
<td>EnableUserDel</td>
<td>The menu item “Edit → Delete” is activated.</td>
</tr>
<tr>
<td>DistCriterionPTPOn/Off</td>
<td>The approximation distance for PTP motions can be set to mm or %.</td>
</tr>
<tr>
<td>Hibernate</td>
<td>When switched off, the controller is set to Hibernate mode.</td>
</tr>
<tr>
<td>Lettering_StatuskeyOn/Off</td>
<td>Status keys are labeled with the configured names.</td>
</tr>
<tr>
<td>ToolDirectionZOn/Off</td>
<td>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</td>
</tr>
</tbody>
</table>