RPL

Robot Programming language

Robox S.P.A.
Castelletto S.Ticino, Italy – www.robox.it

Summary

Data Types

BOOL

This type of data is used for logical values.
BOOL value can be **true** or **false**.
Default value after initialization is **false**.

Example:

```
BOOL firstRun
…
IF firstRun = false THEN
    firstRun:= true ;
    …
END_IF ;
```

This example shows how to test the first execution of a portion of code.

DINT

This type of data is used for integer value that can be positive or negative.
The integer value is represented with a 32-bit value.
DINT value can be between -2147483648 and 2147483647.
Default value after initialization is **0**.

Example:

```
DINT counter
DINT diff
…
counter := counter + 1 ;
…
```

UDINT

This type of data is used for integer value that are only positive.
The integer value is represented with a 32-bit value.
UDINT value can be between 0 and 4294967295.
Default value after initialization is **0**.

Example:

```
DINT counter
…
counter := counter + 1 ;
```

LREAL

This type of data is used for decimal number with double precision.
The value is a 64-bit signed value.
LREAL value can be between -1.7976931348623158e+308 and 1.7976931348623158e+308.
Default value after initialization is **0.0**.

Example:

```
LREAL width
...
width := 88.506 ;
...
```

VECT3

This type of data is used to represent a three-dimension vector.
Data is made with 3 **LREAL** values.
Data has **x**, **y**, **z** component of type **LREAL**.
To compose this type of data with 3 **LREAL** values it is possible to use function **VECT3**.
Default value after initialization is a vector with all component **x**,**y**,**z** set to **0.0**.

Example:

```
VECT3 pose
...
pose := VECT3( 100, 200, 300) ;
```

POINTC

This type of data is used to represent a cartesian point.
Data has **x**, **y**, **z, a, b, c** component of type **LREAL**. Components **x**,**y**,**z** refer to position and **a**,**b**,**c** refer to orientation with Euler conventions. Component **a** refers to orientation on z axis, **b** to orientation on y axis and **c** to x axis. These orientation are referred to the rotated frame convention.
To compose this type of data with 6 **LREAL** values it is possible to use function **POINTC**.
POINTC can be used for cartesian and joint movements.
Default value after initialization is a not valid point (so that issue an error if used in movement instructions) with all component **x**,**y**,**z**,**a**,**b**,**c** set to **0.0**.

Example:

```
POINTC target
...
target := POINTC( 100, 200, 300, 10, 20, 30) ;
MLIN (target, v250, fine, tool0, wobj0) ;
```

ROBOT

This type of data is used to interact with axes group defined in the system.
For example is possible to set reference system of an axis group.
To set this type of data must be used function **ROBOT** that requires as parameter a string with the name of axes group.
Default value after initialization is a not valid **ROBOT**.

Example:

```
ROBOT conveyor
REFSYS cvywobj
POINTC cvyOrigin
…
conveyor := ROBOT("conveyor") ;
cvywobj:= SETROBOTWOBJ(conveyor, "", cvyOrigin) ;
…
```

In this example it is set the reference system of **conveyor**. This setting is done to specify reference system of **conveyor** in the program that can be used for example in tracking applications.

REFSYS

This type of data is used to define a coordinate reference system for Cartesian movements.
If used in movement instructions the position will be based on a specific coordinate system.
REFSYS can be based on a hierarchical chain and can be fixed or movable.
To set this type of data is possible to use several functions. For example is possible to use function **REFSYS** that require a parent **REFSYS** and six **LREAL** values.
Default value after initialization is a **REFSYS** with all data set to **0.0** that is identical to world reference system.

Example:

```
REFSYS wobj
POINTC p0
…
wobj:= REFSYS( wobj0, 100, 200, 50, 0, 0, 0) ;
p0 := POINTC(0, 0, 0, 0, 0, 0) ;
MLIN (p0, v500, fine, tool0, wobj) ;
```

In this example robot will move to origin of reference system **wobj**.

POINTJ

This type of data is used to define position referred to robot joints.
Data has **j1**, **j2**, **j3, j4, j5, j6** components of type **LREAL**.
To set this type of data is possible to use function **POINTJ** that require 6 **LREAL** values.
POINTJ is used in **MJOINT** instruction to move robot in a particular position defined in joint positions.
POINTJ cannot be used as target in cartesian movement.
Default value after initialization is a not valid point (so that issue an error if used in movement instructions) with all component **j1**, **j2**, **j3, j4, j5, j6** set to **0.0**.

Example:

```
POINTJ startpos
…
startpos:= POINTJ( 0, 0, 0, 0, -90, 0) ;
MJOINT(startpos, v500, fine, tool0) ;
```

TRIGGER

TRIGGER datatype is used to store data involved in event related with movement instructions.
With instruction **TRIGSET** is possible to store data inside a **TRIGGER** variable, with instruction **TRIGON** is possible to activate trigger before execution of a movement instructions.
Default value after initialization is a not valid trigger.

Example:

```
TRIGGER trig
…
trig := TRIGSET when Distance is 20 do (io.output[5] := true) ;
…
TRIGON (trig) ;
MLIN (target, v500, fine, tool1, wobj1) ;
```

In this example **io.output[5]** is set to **true** when the robot is 20 mm before target.

STRING

This type of data is used to store character strings.
STRING can contain up to 128 character.
Default value after initialization is an empty string.

Example:

```
STRING str
…
str := "Hello world" ;
MESSAGE ("%1", str) ;
```

The string **str** that contain "Hello world" is reported in user log.

SPEED

SPEED datatype is used to specify the target velocity in movement instructions.
It contains tangential speed expressed in mm/s and orientation speed in degree/s.
To store data in a variable of **SPEED** type is possible to use function **SSPEED**.
Default value after initialization is a **SPEED** with all parameters set to **0.0**.

Example:

```
SPEED vel
…
vel := SSPEED( 250, 5) ;
MLIN (target, vel, fine, tool0) ;
```

In this example robot will move with a target velocity of 250 mm/s and 5 degree/s.

Example:

```
SPEED vel
…
vel := v250 ;
…
MLIN (target1, vel, fine, tool0) ;
MLIN (target2, vel, fine, tool0) ;
MLIN (target3, vel, fine, tool0) ;
```

This example shows how to define a target velocity for a group of movements using a predefined velocity settings defined in variable **v250**.

ZONE

ZONE datatype is used to specify the blending parameters between two successive movement instructions.
It contain linear distance in mm and reorientation angular distance in degree.
To set data of this type is possible to use function **SZONE**.
Default value after initialization is a **ZONE** with all parameters set to **0.0**.

Example:

```
ZONE blend
…
blend := SZONE(100, 5) ;
…
MLIN (target1, v250, blend, tool0) ;
MLIN (target2, v250, fine, tool0) ;
```

In this example the movement to **target2** starts at 100 mm before robot reach **target1**.

TOOL

This type of data is used to describe the tool's parameters.
TOOL datatype is used in movement instructions.
To set this type of data is possible to use **STOOL** function.
Default value after initialization is a **TOOL** with all parameters set to **0.0**. that is identical to use flange of robot.

Example:

```
TOOL gun
…
gun := STOOL(0, 0, 100, 0, 0, 0) ;
…
MLIN (target, v250, fine, tool0) ;
MLIN (target, v250, fine, gun) ;
```

This example can be used to check how robot position is different using same target and two different tool.

INTR

This type of data is used for interrupt handling.
To bind a variable of this type with a specific interrupt routine must be used **INTRSET** instruction.
Condition that cause interrupt can be specified using instruction **INTRCOND** or **INTRERRNO**.
Note that this type of data must be set only once in program execution otherwise an error will be issued.
For other information see instruction beginning with **INTR** and guide "How to use interrupt".
Default value after initialization is an invalid **INTR**.

Example:

```
INTR intr1
BOOL firstRun
…
(* set interrupt only once *)
IF NOT firstRun THEN
    firstRun := true ;
    INTRSET (intr1, intHnd()) ;
    INTRCOND (intr1, io.input[8]) ;
END_IF ;
…
```

This example shows how to bind an interrupt and how to set the condition that trigger that interrupt.

CLOCK

CLOCK data type is used for time measurement.
Time measurement is expressed in seconds.
This type of data cannot be set or read directly.
Is possible to reset time measurement with instruction **CLOCKRESET**.
To start measurement must be used instruction **CLOCKSTART** and **CLOCKSTOP** to stop measurement.
For reading must be used **CLOCKREAD** function that return an **LREAL** value.
Default value after initialization is a measurement of **0.0** seconds.

Example:

```
CLOCK clk1
…
CLOCKRESET (clk1) ;
CLOCKSTART (clk1) ;
…
CLOCKSTOP (clk1) ;
MESSAGE ("Time elapsed %1", CLOCKREAD(clk1)) ;
```

This example shows how to measure time elapsed during execution of instructions.

TRACKING

TRACKING datatype is used to store data for tracking applications.
To set this type of data must be used function **TRACKING**.
For an explanation of parameters see tracking application note document.
Default value after initialization is an invalid **TRACKING**.

Example:

```
TRACKING cvy
REFSYS wobj
REFSYS rsPhoto
…
cvy := TRACKING(1000, 700, 100, 1000, Linear, 0.1) ;
…
wobj := GETTRKWOBJ(rsPhoto, cvy) ;
…
MLIN (target, v500, fine, tool1, wobj) ;
…
```

This example shows how to set tracking parameters for a tracking application. For further detail see tracking application note document.

EPOINTC

This type of data is used to represent a cartesian point and auxiliary axes joint positions.
It contain same data as in **POINTC** plus component **ea1**, **ea2**, **ea3**, **ea4**, **ea5**, **ea6** related to auxiliary axes joint positions.
EPOINTC can be used for cartesian and joint movements.
There are several functions to set this type of data, one of this is function **EPOINTC**.
Default value after initialization is a not valid point (so that issue an error if used in movement instructions) with all component set to **0.0**.

Example:

```
EPOINTC ep0
...
ep0 := EPOINTC( 100, 200, 300, 10, 20, 30, 100, 0, 0, 0, 0, 0) ;
...
MLIN (ep0, v500, fine, tool0) ;
...
```

EPOINTJ

This type of data is used to define position referred to robot joints with auxiliary axes joint position.
It contains same data as in **POINTJ** plus component **ea1**, **ea2**, **ea3**, **ea4**, **ea5**, **ea6** related to auxiliary axes joint positions.
EPOINTJ is used in **MJOINT** instruction to move robot in a specific position defined in joint positions.
EPOINTJ cannot be used as target in cartesian movement.
There are several functions to set this type of data, one of this is function **EPOINTJ**.
Default value after initialization is a not valid point (so that issue an error if used in movement instructions) with all component set to **0.0**.

Example:

```
EPOINTJ ep0
...
ep0 := EPOINTJ( 0, 0, 0, 0, -90, 0, 10, 0, 0, 0, 0, 0) ;
...
MJOINT (ep0, v500, fine, tool0) ;
```

Types of data for extern variables derived from the machine environment

POINT_L

This type of data is used to access to a library point loaded in the system.
Data can be used in movement instruction.

Example:

```
POINT_L p0
…
MLIN (p0, v500, fine, tool0) ;
```

In this example robot will move to point **p0** defined in a library loaded in the system.

PATH_L

This type of data is used to access to a library path loaded in the system.
Data can be used with instruction **EPATH**.

Example:

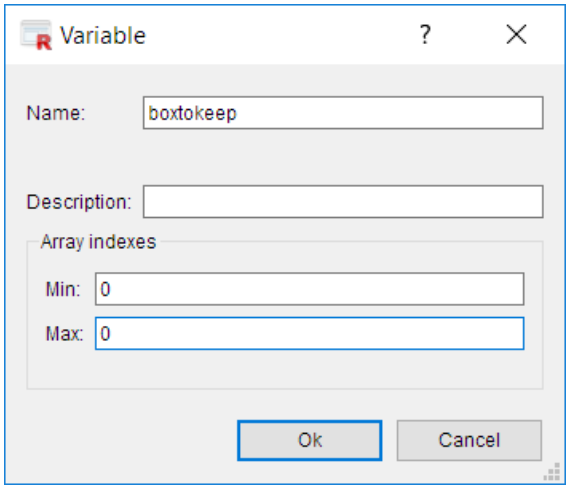
```
PATH_L path0
…
EPATH (path0, v500, fine, tool0) ;
```

In this example robot will execute trajectory **path0** defined in a library loaded in the system.

Variables

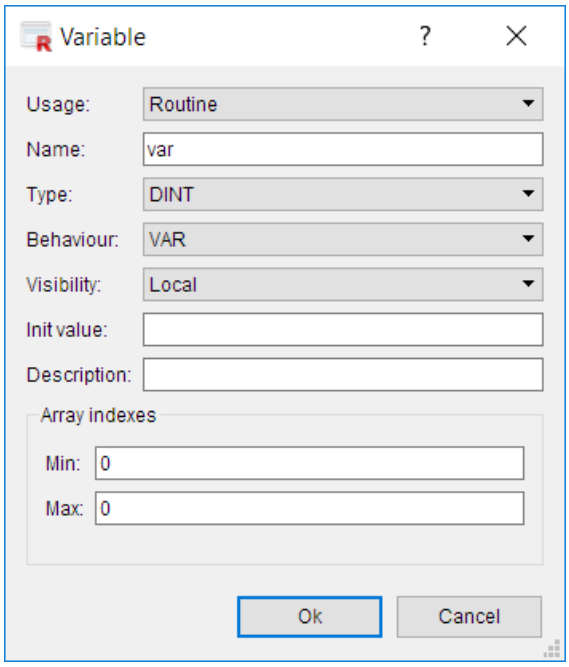
Variables can be defined inside an RPL program or subroutine (in the Vars tab). The type of the variable can be of any of the data types defined in RPL. A variable can be defined as an array by setting a maximum index greater than the minimum index.

A variable can also be defined as 'External'. This type of variable is defined inside the machine program and can be used in all programs and subroutines (it is a 'Global' variable). The type and name of the variable is defined inside the machine program. To add this type of variable you must know the name of the external variable. The type of the variable cannot be selected and depends on the machine program definition. This type of variable is used to exchange data between machine program and RPL robotic program.



A variable is defined with:

- **Type**
- **Behavior**
- **Visibility**



The behavior of a variable can be:

- **VAR**

  It's the default behavior of variable. Variable can be set in program and its value is lost when the program is restarted.

- **CONST**

  A variable that have this behavior cannot be changed in program and must be set with an Init value.

- **RETAIN**

  The value of variable is preserved when program is restarted and stored when program will be unloaded from memory. For **LOCAL** and **TASK** variable the value is saved inside program. This behavior can be selected only when **Usage** field is **Module**.

The visibility of a variable can be:

- **Routine**'s Local

  This kind of variable can be seen and used only in the program or subroutine where it is defined. It cannot be seen from outside the program or subroutine in which it is defined. You can define a local variable named ***var*** in two or more subroutines. Each subroutine uses its own ***var*** variable.

- Routine's **Input**

  This kind of variable is used to define the input parameters of a subroutine. It is used like a local variable, but its initial value comes from the calling program. The order of definition of the input variables is the same of the arguments passed from the CALL instruction. When the selected subroutine is **Main** it is not possible to select this type of visibility.

- Routine's **Output**

  This kind of variable is used to define the output parameters of a subroutine. It is used like a local variable, but its final value will be set in a variable in the calling program. The order of definition of the output variables is the same of the variables that will be set by the CALL instruction. When the selected subroutine is **Main** it is not possible to select this type of visibility.

- **Module**'s Local

  This kind of variable can be seen and used in all programs or subroutines. With a module's local variable, you can set a value in a subroutine and later you can read that variable from another subroutine. You cannot define more than one module's local variable with same name. These variables cannot be seen from other modules.

- Module's **Public**

  It's like Module's local but this variable can be seen from other modules. In other modules this kind of variable can be used by preceding it with module name (e.g. moduleName.variableName)

  ```
  localVar := Utility.moduleVar ;
  ```

  localVar is set with value of variable **moduleVar** of module **Utility**.

- **Task**

  It's like Module's public. In other modules this kind of variable can be used without preceding it with module name (e.g. variableName)

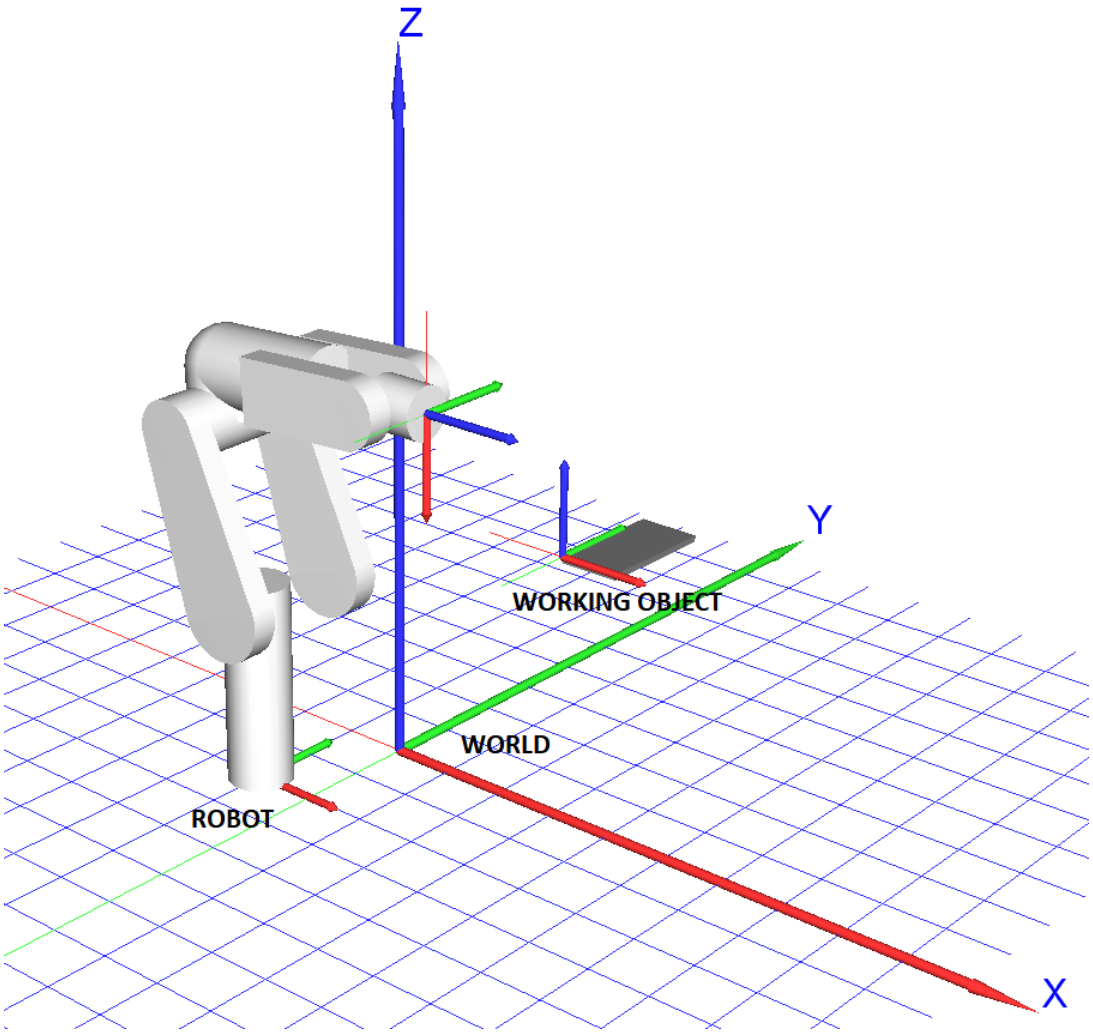
  ```
  localVar := moduleVar ;
  ```

  localVar is set with value **moduleVar**. Module of **moduleVar** is not specified.

- **Global**

  This kind of variable is common to all TASK of system. It is useful to share data between different TASK. If there is different definition of same GLOBAL an error is reported. A GLOBAL variable is not preceded with module name when it is used.

Reference systems



Reference systems are used to simplify program displacement. If position of robot and/or position of workpiece are changed the program can be reused changing only value of reference system.

World reference

The world reference is a fixed coordinate system that is used to reference all other reference used in the system.

Robot reference

The robot reference identifies the position of robot in the system. Normally robot reference is set identical to world reference.

Working object reference

The working reference identify the position of the workpiece.

Instructions

(* *)

Represented as a comment between '(*' and '*)' in the teachgun display. With this instruction you can enter a comment in the code.

Example:

```
(* This is a comment *)
started := true ;
…
(* wait door's input *)
WAIT (door_opened) ;
```

:=

Represented as **:=** in the teachgun display. With this instruction you can assign a value to a variable

variable := ***expression ;***

Example:

```
i := 123 ;
temp := SIN(0.392) ;
```

In this example variable ***i*** is set with value 123 and ***temp*** with the result of the expression ***SIN(0.392)***.

Note: It is not possible to assign directly one array to another with this instruction. If you want to copy values from an array to another you must iterate this instruction for each element.

ALIAS

Bind a variable with another variable.

ALIAS (**variable**, **reference variable**) ;

Example:

```
BOOL grimper
BOOL out_1
…
ALIAS (grimper, out_1) ;
…
grimper := true ;
```

After the instruction **ALIAS** is possible to use ***grimper*** as ***out_1***. In the example both **out_1** and **grimper** will be **true**.

CALL

Used for executing a user defined subroutine

CALL [***assigned variables list***] *:=* ***subroutine name*** (***input parameters***) ;

Example:

```
CALL ExecuteSquare(A,B,C,D) ;
```

Calls the subroutine ***ExecuteSquare*** with 4 parameters and using no result. This type of call is useful to simplify the program by grouping some instructions inside a subroutine.

Subroutines tab

```
ExecuteSquare (POINTC A, POINTC B, POINTC C, POINTC D)
```

Vars tab

```
Input
  POINTC A
  POINTC B
  POINTC C
  POINTC D
```

Code tab

```
MJOINT (A, v100, fine, tool1) ;
MLIN (B, v100, fine, tool1) ;
MLIN (C, v100, fine, tool1) ;
MLIN (D, v100, fine, tool1) ;
MLIN (A, v100, fine, tool1) ;
```

Example:

```
CALL val,inRange := acquireValue(val, min, max) ;
```

Calls the subroutine ***acquireValue*** passing three arguments and setting two variables after the execution with the results of the subroutine called. In this example an input value is tested.

Subroutines tab

```
acquireValue (LREAL val, LREAL min, LREAL max) => (LREAL
outval, BOOL inRange)
```

Vars tab

```
Input
  LREAL val
  POINTC min
  POINTC max
Output
  LREAL outval
  BOOL inRange
```

Code Tab

```
outval := LIMIT(val, min, max) ;
IF outval <> val THEN
  inRange := false ;
ELSE
  inRange := true ;
END_IF
```

Note: It is not possible to neither use array as parameters nor as result.

CLEARMOVE

With this instruction all pending movements will be aborted. Before calling this instruction, the ROBOT must be in hold or standstill state. If ROBOT is in motion while executing **CLEARMOVE** an error occurs. If this instruction is executed in a subroutine called from interrupt or trigger and program returns to an uncompleted movement instruction, the program will stop with an error.

CLEARMOVE ;

Example:

```
(* Hold pending movements *)
STOPMOVE () ;
(* Abort pending movements *)
CLEARMOVE ;
```

Hold actual movement then abort all pending movements.

CLOCKRESET

Reset a variable of datatype **CLOCK** used for time measurement.

CLOCKRESET (**variable**) ;

Example:

```
CLOCK clk1
…
CLOCKSTART (clk1) ;
…
CLOCKSTOP (clk1) ;
MESSAGE ("Time elapsed %1 for phase 1", CLOCKREAD(clk1)) ;
…
CLOCKRESET (clk1) ;
CLOCKSTART (clk1) ;
…
CLOCKSTOP (clk1) ;
MESSAGE ("Time elapsed %1 for phase 2", CLOCKREAD(clk1)) ;
…
```

This example shows how to make two measurement using only one variable.

CLOCKSTART

Start time measurement based on a variable of datatype **CLOCK**.

CLOCKSTART (**variable**) ;

Example:

```
LREAL partial
LREAL total
…
CLOCKRESET (clk1) ;
CLOCKSTART (clk1) ;
…
CLOCKSTOP (clk1) ;
partial := CLOCKREAD(clk1) ;
CLOCKSTART (clk1) ;
…
CLOCKSTOP (clk1) ;
total := CLOCKREAD(clk1) ;
```

This example shows how to measure partial time and total time.

CLOCKSTOP

Stop time measurement based on a variable of datatype **CLOCK**.

CLOCKSTOP (**variable**) ;

Example:

```
LREAL time
…
CLOCKRESET (clk1) ;
CLOCKSTART (clk1) ;
…
CLOCKSTOP (clk1) ;
time := CLOCKREAD(clk1) ;
```

This example shows how to measure an execution time.

CONTINUE

Breaks the current execution of a **WHILE** or **FOR** cycle and forces the re-evaluation of the cycle condition.

Example:

```
WHILE (i <= 10) DO
     i := i + 1 ;
     IF (MOD(i, 2) = 1) THEN
          CONTINUE ;
     END_IF ;
     MLIN (POINTC(200, -500, 200, 0, 180, 0), v100, fine, tool1) ;
     MLIN (POINTC(200, -900, 200, 0, 180, 0), v100, fine, tool1) ;
END_WHILE ;
```

In the above example if the value of variable ***i*** is odd the subsequent **MLIN** instructions will be skipped and the condition of the loop will be re-evalutated.

DWELL

Waits (stops execution) for a specified time in seconds before executing subsequent instruction.

DWELL (***time in seconds***) ;

Example:

```
MJOINT (POINTC(500, 500, 0, 0, 0, 0), v100, fine, tool1) ;
DWELL (1.5) ;
MJOINT (POINTC(800, 800, 50, 0, 0, 0), v100, fine, tool1) ;
```

In this example, after the first move, the robot waits 1.5 seconds before executing the next move.

ENDPROG

Terminate execution of current program. Pending movement will be finished before terminating current program. Program execution can be resumed only with a restart request or a request of setting program pointer.

ENDPROG ;

Example:

```
IF unexpected=TRUE
     ENDPROG ;
END_IF
```

If variable **unexpected** is **true** the program execution is terminated.

EPATH

Launches the execution of a path.

EPATH (***PATH_L variable, speed, zone, tool, [refsys]***) ;

Before you enter this instruction you must add an external variable to access path that you want to execute. If ***refsys*** parameter is omitted world coordinate system is used for movement.
Before executing the selected path robot will move to first point of trajectory with a joint movement.

Example:

In the ***Vars*** tab you add this variable:

```
External
  PATH_L trajectory
```

In the ***Code*** tab you insert this instruction:

```
EPATH (trajectory, v100, fine, tool1) ;
```

In this example the path named ***trajectory*** is executed.

ERROR

Set the ***expr*** value in system variable ***_errno_***. Then a message in system and user log is printed and execution of program is breaked and error handling code is executed. Default error handling is stop execution of current program.
Message in log is composed with a string “ERROR CODE” plus the number of error code followed by the two optional parameters **expr1** and **expr2**.

ERROR (***expr***, [***expr1***], [***expr2***]) ;

Example:

```
DINT Value
…
IF Value < 10 THEN
    ERROR (1, “Value is less than”, 10) ;
END_IF ;
```

In this example the program is stopped if variable **Value** is less than 10 and the string “ERROR CODE 1: Value is less than 10” is printed to system and user log.

EXEC

Execute a **subroutine** defined in a module loaded with instruction **LOAD**. Subroutine is evalutated only runtime. During compilation subroutine name is not checked because reside in a runtime loaded module.

EXEC [***result*** =] ***subroutine*** ([***parameters***]) ;

Example:

```
LOAD "parprog.xpl" ;
…
EXEC "parprog.work" ;
…
EXEC value = "parprog.calc" (a, b) ;
```

In this example subroutine **work** and subroutine **calc** will be executed.

Example:

```
LOAD "sub_p01.xpl" ;
…
EXEC "sub_p01.Main" ;
```

This example shows how to run a complete program from another program. To run a program from another program it is needed to run the **Main** subroutine.

EXIT

Terminate the execution of a **WHILE** or **FOR** cycle.

Example:

```
WHILE i <= 10 DO
     i := i + 1 ;
     IF (y = 4) THEN
          EXIT ;
     END_IF ;
     MLIN (POINTC(200, -500, 200, 0, 180, 0), v100, fine, tool1) ;
     MLIN (POINTC(200, -900, 200, 0, 180, 0), v100, fine, tool1) ;
END_WHILE ;
```

In this example the execution of while is interrupted if the value of variable ***y*** is equal to 4.

FOR

Executes a block of code several times. The first execution sets the ***variable*** with value of ***init expression***. After the completion of the block of code, this variable is updated adding the value of ***increment expression*** and if the new value of ***variable*** is inside the specified range (***init expression*** … ***end expression*** …) the block of code is executed again. The default increment is ***1,*** if you do not specify any ***increment expression***. It is possible to use **EXIT** and **CONTINUE** instructions.

FOR ***variable*** := ***init expression*** TO ***end expression*** BY [***increment expression***] DO … END_FOR ;

Example:

```
FOR i := 1 TO 4 BY 1 DO
     MLIN (POINTC(200, -500, 50, 0, 180, 0), v100, fine, tool1) ;
     MLIN (POINTC(200, -900, 50, 0, 180, 0), v100, fine, tool1) ;
END_FOR ;
```

Executes a loop four times using the variable ***i***. The first execution sets variable ***i*** with value 1 , after each iteration the variable ***i*** is incremented by adding value 1. The last iteration is when value of variable ***i*** becomes 4. At the end of the execution ***i*** is equal to 5.

When you enter a **FOR** instruction a line with **END_FOR** ; is automatically added.

GOTO

Jumps execution to a specified label.

GOTO ***name of label*** ;

Example:

```
LABEL Start:
MJOINT (POINTC(500, 500, 0, 0, 0, 0), v100, fine, tool1) ;
MJOINT (POINTC(800, 800, 50, 0, 0, 0), v100, fine, tool1) ;
GOTO Start ;
```

In this example after ***GOTO Start*** the execution continues with the instruction ***LABEL Start***.

IF THEN ELSE

Execute a block of code if the condition is true (section IF) and another if it is false (ELSE).

When you enter an **IF** istruction a line with **END_IF** ; is automatically added. If you want to insert the **ELSE** statement you must select the **END_IF** ; line and click on **ELSE** button displayed in teachgun near editing line.

IF ***condition*** THEN
…
ELSE
…
END_IF ;

Example:

```
IF j := 1 THEN
    sum := sum + 1 ;
ELSE
    sum := sum + 2 ;
END_IF ;
```

In the example if variable ***j*** is equal to 1 the variable ***sum*** is incremented by 1, otherwise it is incremented by 2.

INTRCOND

Instruction for set the condition that trigger the interrupt connected with an **INTR** variable. Expression must be Boolean. After setting the condition the related interrupt is also enabled. Interrupt are triggered in the transition from false to true of condition.

INTRCOND (***INTR variable***, ***Boolean expression***) ;

Before use this instruction **INTR** variable must be set with **INTRSET** instruction, if not an error is reported.

Example:

```
INTR trap
…
INTRSET (trap, trapRoutine()) ;
INTRCOND (trap, io.input[8]<>false) ;
…
```

In this example when **io.input[8]** will be **true** the interrupt routine **trapRoutine** will be executed.

INTRALLOW

Instruction for enable triggering of a specific interrupt that was previously disabled with instruction **INTRDENY**.

INTRALLOW (***INTR variable***) ;

Before this instruction the **INTR** variable must be connected to an interrupt, if not an error is reported.

Example:

```
INTR trap
…
INTRSET (trap, trapRoutine()) ;
INTRCOND (trap, io.input[8]<>false) ;
…
INTRDENY (trap) ;
(* Interrupt trap must be avoid in following code *)
…
(* Now is possible to enable interrupt trap *)
INTRALLOW (trap) ;
…
```

This example shows how to avoid a single interrupt in a specific portion of code.

INTRDENY

Instruction for disable triggering of a specific interrupt. After this instruction any trigger condition of specified interrupt will be discarded.

INTRDENY (***INTR variable***) ;

Before this instruction the **INTR** variable must be connected to an interrupt, if not an error is reported.

Example:

```
INTR trap
INTR trap2
…
INTRSET (trap, trapRoutine()) ;
INTRSET (trap2, trap2Routine()) ;
INTRCOND (trap, io.input[8]<>false) ;
INTRCOND (trap2, io.input[9]<>false) ;
…
INTRDENY (trap) ;
(* Interrupt trap will be avoid in following code *)
…
```

This example shows how to avoid interrupt **trap** meanwhile interrupt **trap2** is still active.

INTRDIS

Instruction for disable all interrupt or a single interrupt. If optional parameter is omitted all interrupt are disabled. When interrupts are disabled, interrupt condition are anyway checked. If interrupt condition is met interrupt will be called as soon as interrupt will be reenabled.

INTRDIS ([***INTR variable***]) ;

If optional parameter is not omitted the INTR variable must be connected to an interrupt, if not an error is reported.

Example:

```
INTR trap
…
INTRSET (trap, trapRoutine()) ;
INTRCOND (trap, io.input[8]<>false) ;
…
INTRDIS () ;
(* Interrupt will be disabled in following code *)
…
INTRENA () ;
(* Interrupt will be reenabled in following code *)
…
```

This example shows how to disable interrupt and then reenable interrupts.

INTRENA

Instruction for enable all interrupt or a single interrupt. If parameter variable is omitted all interrupt are enabled.

INTRENA ([***INTR variable***]) ;

If optional parameter is not omitted the INTR variable must be connected to an interrupt, if not an error is reported.

Example:

```
INTR trap
…
INTRSET (trap, trapRoutine()) ;
INTRCOND (trap, io.input[8]<>false) ;
…
INTRDIS () ;
(* Interrupts will be disabled in following code *)
…
INTRENA () ;
(* Interrupts will be reenabled in following code *)
…
```

This example shows how to disable interrupt and then reenable interrupts.

INTRERRNO

Instruction for set the condition that trigger an interrupt connected with an **INTR** variable when variable ***_errno_*** is set with a specific value (if optional integer value is not omitted) or is set with a value different from zero. Interrupt is triggered when variable ***_errno_*** is set.
Variable **_errno_** can be set by instruction **ERROR** or when an execution error occurs.

INTRERRNO (***INTR variable***, [integer value]) ;

Before use this instruction **INTR** variable must be set with **INTRSET** instruction, if not an error is reported.

Example:

```
INTR trapErr
…
INTRSET (trapErr, errnoHnd()) ;
INTRERRNO (trapErr) ;
…
```

In this example when variable **_errno_** is set the subroutine **errnoHnd** will be called.

INTRSET

Instruction for connect an **INTR** variable with a subroutine.

INTRSET (***INTR variable***, ***subroutine***) ;

It is not possible to set an **INTR** variable that was already set by a previous **INTRSET** instruction, in such case an error is reported.

Example:

```
INTR trapInput
INTR trapErr
…
INTRSET (trapInput, inputHnd()) ;
INTRSET (trapErr, errnoHnd()) ;
…
INTRCOND (trapInput, io.input[5]) ;
INTRERRNO (trapErr) ;
…
```

This example shows how to set two interrupts.

JOIN

Wait end of execution of a subroutine started with instruction **THREAD**.
If optional parameter **timeout** is present waiting can be interrupted by end of thread or by timeout.
The optional parameter **result** will be set to ***0*** if thread is terminated or ***not equal to zero*** if thread is not ended before **timeout**. The ***timeout*** and ***result*** parameters are optional.

JOIN (***pid_thread***, [***timeout***], [***result***]) ;

Example:

```
UDINT pid
…
(* start activation of glue *)
THREAD pid, activateGlue() ;
(* meanwhile go to starting point *)
MLIN (pstart, v500, fine, tool1, wobj1) ;
…
(* wait end activation of glue *)
JOIN (pid) ;
(* start job *)
MLIN (p1, v500, z50, tool1, wobj1) ;
…
```

This example shows a possible use of **JOIN**. Activation of glue system is started with instruction **THREAD** calling subroutine **activateGlue** then some other instructions are executed before activation of glue is done.

KMAXT

Used to set the main axes maximum working parameters for all cartesian move. This is a modal setting.

KMAXT([***speed***], [***acceleration***], [***deceleration***], [***jerk***]) ;

The unit of each parameter depend on machine configuration, normally speed is in mm/s. All parameters are optional.
Speed parameter is used as a limit for all cartesian movement.
If in a movement is set a speed value bigger than value set with KMAXT the used value is reduced to the value set in KMAXT.

Note that each parameter cannot exceed the parameters configured for the axes group in the machine. If you set a bigger value, such value is set to the maximum configured in machine.
These parameters (TAN_SPE, TAN_ACC, TAN_JER) by default are computed by RPE as the minimum between maximum cartesian parameter of main axes (MAX_SPE_C[1…3], MAX_ACC_C[1…3], MAX_JER_C[1…3]).

Example:

```
KMAXT (50, 15) ;
MLIN (POINTC(600, 600, 0, 0, 0, 0), v100, fine, tool1) ;
```

In this example ***speed*** is changed to 50 mm/s and ***acceleration*** is changed to 15 mm/s^2 for all subsequent cartesian move; ***deceleration*** and ***jerk*** remain as set in the machine configuration.

KMAXJ

Used to set the maximum working parameters for all joint move. This is a modal setting.

KMAXJ ([***speed***], [***acceleration***], [***deceleration***], [***jerk***]) ;

Parameters are expressed in percentage and are referred to the MAX_SPE_J, MAX_ACC_J and MAX_JER_J configured for the axes group in the machine. All parameters are optional.

Examples:

```
KMAXJ (10) ;
MJOINT (POINTC(-400, 0, 0, 0, 180, 0), v1000, fine, tool1) ;
```

In this example ***maximum speed*** is changed to 10% of the maximum speed for the all subsequent joint move. If TAN_SPE is 2000 mm/s joint speed would be 10% that is the minimum between 10% and 50%.

```
KMAXJ (10) ;
MJOINT (POINTC(-400, 0, 0, 0, 180, 0), v100, fine, tool1) ;
```

In this example ***maximum speed*** is changed to 10% of the maximum speed for the all subsequent joint move. If TAN_SPE is 2000 mm/s joint speed would be 5% that is the minimum between 10% and 5%.

KMAXW

Used to set the orientation maximum working parameters for all cartesian move. This is a modal setting.

KMAXW([***speed***], [***acceleration***], [***deceleration***], [***jerk***]) ;

The unit of each parameter depend on machine configuration, normally speed is in degree/s. All parameters are optional.
Speed parameter is used as a limit for all cartesian movement.
If in a movement is set a speed value bigger than value set with KMAXW the used value is reduced to the value set in KMAXW.

Note that each parameter cannot exceed the parameters configured for the axes group in the machine. If you set a bigger value, such value is set to the maximum configured in machine.
These parameters (W_SPE, W_ACC, W_JER) by default are computed by RPE as the minimum between maximum cartesian parameter of wrist axes (MAX_SPE_C[4…6], MAX_ACC_C[4…6], MAX_JER_C[4…6]).

Example:

```
KMAXW (50, 15) ;
MLIN (POINTC(600, 600, 0, 0, 0, 0), v100, fine, tool1) ;
```

In this example ***speed*** is changed to 50 degree/s and ***acceleration*** is changed to 15 degree/s^2 for all subsequent cartesian move; ***deceleration*** and ***jerk*** remain as set in the machine configuration.
If in v100 reorientation speed is lower than 50 the resulting speed is the value inside v100.

LABEL

Enters a name for a position in the code, that can be used to jump to a specified section of code with GOTO instruction.

LABEL (***name of label***) :

Example:

```
LABEL loop:
MLIN (pstart, v500, fine, tool1, wobj1) ;
…
GOTO loop ;
```

This is an example for doing an infinite loop.

LOAD

Load a program from file into memory as module of current program. After loading will be possible to execute subroutine and access data defined in the module.

LOAD (***filename***);

Example:

```
LOAD ("parprog.xpl") ;
EXEC "parprog.work" ;
```

In this example file "**parprog.xpl**" is loaded as module. After that subroutine **work** defined in the loaded module is executed.

MCIRC

Starts the execution of a circular (cartesian) movement of the TCP from the last position to the target position, passing throught an intermediate point.

Note that if the final point is coincident with the initial one the rotation sense is **impredictable**!

If a change of the orientations is required, the orientation axes (a, b and c) will be interpolated (will start to move and arrive at their target position simultaneously) to the main axes (x, y and z).

If ***refsys*** parameter is omitted world coordinate system is used for movement.

MCIRC (***intermediate point***, ***target point, speed, zone, tool, [refsys]***)

Example:

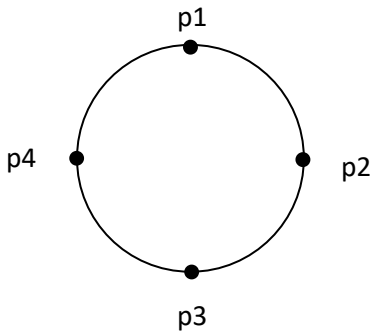
```
MLIN (a, v100, fine, tool1) ;
MCIRC (b, c, v100, fine, tool1) ;
```

In this example a circle arc is executed starting from point ***a,*** passing from point ***b,*** to the final point ***c***.

Example:

```
MLIN (p1, v100, fine, tool1) ;
MCIRC (p2, p3, v100, fine, tool1) ;
MCIRC (p4, p1, v100, fine, tool1) ;
```

This is an example to make a full circle starting from p1 passing throw p2, p3, p4.



MESSAGE

Print a message in system and user log. In ***text*** expression is possible to use placeholder %1 and %2 that are substituted with value converted as text of ***expr1*** and ***expr2*** respectively.

MESSAGE (***text***, ***expr1***, ***expr2***) ;

Example:

```
MESSAGE ("First value %1 second value %2", 10, 20) ;
```

In this example the string “First value 10 second value 20” is printed in user and system log.

MJOINT

Starts the execution of a joint (point to point) movement from the last position to the ***target*** position. Target position can be of type **POINTC** or **POINTJ**. All axes will start to move and arrive at their target position simultaneously. The TCP movement will be the combination of single axis movements. If ***refsys*** parameter is omitted world coordinate system is used for movement.

MJOINT (***target, speed, zone, tool, [refsys]***) ;

Example:

```
MJOINT (HomePos, v100, fine, tool1) ;
```

The **speed** parameter is expressed in **cartesian** unit relative to TCP, but in this type of movement it is used to calculate a percentage for joint's speed.
The percentage is calculate like:

[joint's speed percentage] = [SPEED tangential speed] / [TAN_SPE]

For example:

SPEED tangential speed = 100 mm/s
TAN_SPE = 2000 mm/s
joint's speed percentage=5%

Parameters TAN_SPE by default is computed by RPE as the minimum between maximum cartesian speed parameter of main axes (MAX_SPE_C[1…3]).

MLIN

Starts the execution of a linear (cartesian) movement of the TCP from the last position to a ***target*** position. If a change of the orientations is required, the orientation axes (a, b and c) will be interpolated (will start to move and arrive at their target position simultaneously) to the main axes (x, y and z). If ***refsys*** parameter is omitted world coordinate system is used for movement.

MLIN (***point, speed, zone, tool, [refsys]***) ;

Example:

```
MLIN (target1, v100, fine, tool1) ;
```

PULSE

Set a Boolean variable with a specific value for a predefined amount of time. Time is expressed in second. After time is elapsed the Boolean variable is set with opposite value. Optionally is possible to advance the execution of successive step by setting optional parameter ***advance*** to ***true***.

PULSE (**Boolean var**, **Value to set**, **time**, **advance**) ;

Example:

```
BOOL gun
…
PULSE (gun, true, 2, true) ;
MLIN (p1, v500, z20, tool1) ;
```

Variable **gun** is set with value **true** for 2 second then is set to **false**. After the variable is set with **true** execution continue with next instruction because optional parameter advance is set to **true**. So movement **MLIN** instruction is started after **gun** is set to **true**.

Example:

```
BOOL gun
…
PULSE (gun, true, 2) ;
MLIN (p1, v500, z20, tool1) ;
```

Variable **gun** is set with value **true** for 2 second then is set to **false**. Then movement **MLIN** instruction is started.

RESTART

Instruction for restart the execution of program from first instruction of Main subroutine. Remember that no variable is set with initialization value after the execution of this instruction.

```
RESTART ;
```

Example:

```
BOOL firstRun
…
IF NOT firstRun THEN
    firstRun := true ;
    MJOINT (p0, v500, fine, tool0) ;
    …
END_IF
…
MLIN (p1, v500, z20, tool1) ;
…
IF pieceLost THEN
    RESTART ;
END_IF
…
```

This example shows how to skip code if piece is lost during working. If **piece** is **true** program is restarted without reinitializing variables. So only first time the **MJOINT** is executed.

RETURN

Breaks the execution of a program or subroutine.

Example:

```
…
IF input < 0 THEN
    RETURN ;
END_IF ;
output := input + 1 ;
```

In this example if ***input*** is less than zero the execution of the subroutine is interrupted.

CASE

Runs one of several sequence of statements, depending on the value of a ***test_expression***.

CASE ***test_expression*** OF
expression_1, [expression_2], [expression_3]:
… sequence of statements …
[expression from] .. [expression to]:
… sequence of statements …
ELSE
… sequence of statements …
END_CASE ;

The first expression inside **CASE** that match the expression statement of **CASE** is executed.
It is possible to define list of value or a range of value.
The **ELSE** statement must be at the end of the **CASE** block and is executed if no expression inside **CASE** are met.

Example:

```
CASE i OF
    1:
        MLIN (P1, v100, fine, tool1) ;
    2 .. NumPoints - 1:
        CALL MoveToPoint (i) ;
    99, var, var + 1:
        MJOINT (HomePos, v100, fine, tool1) ;
        i := 0 ;
    ELSE
        i := 99 ;
END_CASE ;
```

STARTMOVE

Restore previous holded movement.Optional parameter set the unhold time in seconds, if parameter is not present it is used system's defined unhold time (normally 0.5 second).

STARTMOVE ([***unhold time***]) ;

Example:

```
…
STARTMOVE (0.5) ;
…
```

Restore movement in 0.5 second.

STOPPROG

Stop program execution after all pending movement are finished. Program execution can be resumed. After execution the program pointer is at next instruction.

STOPPROG ;

Example

```
STOPPROG ;
MLIN (target1, v100, fine, tool1) ;
```

After **STOPPROG** program pointer will be at **MLIN** instruction. With a new start request the movement will be executed.

STOPMOVE

STOPMOVE is used to stop robot movements. Optional parameter sets the hold time in seconds. If parameter is omitted it is used system's defined hold time (normally 0.5 second). Pending movements can be resumed with instruction **STARTMOVE** or canceled with instruction **CLEARMOVE**.

STOPMOVE ([***hold time***]) ;

Example:

```
…
STOPMOVE (0.75) ;
…
```

Hold movement in 0.75 second.

THREAD

Launch execution of a subroutine in parallel with normal flow of program. First parameter of instruction must be numeric and contain a numeric reference of subroutine after execution of **THREAD** instruction. This numeric reference can be used to check state of execution of subroutine for example with instruction **JOIN**.

THREAD **pid**, **subroutine**() ;

Example:

```
UDINT pid
…
(* Start open grimper *)
THREAD pid, OpenGrimper() ;
(* meanwhile do something else *)
MLIN (p23, v250, fine, tool1, wobj0) ;
…
```

With above instruction the execution of subroutine **OpenGrimper** start and program continue execution without waiting end of subroutine like as in **CALL** instruction. The instruction **THREAD** is useful for execution of subroutine that can be executed in parallel with normal execution of program. Subroutine started by **THREAD** **cannot contains**:

- movement instructions;

- instruction that can modify ROBOT parameters.

TRIGCALL

Set a variable of type ***TRIGGER***. When condition of trigger is satisfied specific subroutine is called. Subroutine must not have input nor output parameters.

TRIGCALL ***trigger_var***, ***type, ref_value, subroutinetocall***() ***;***

type of trigger can be:

- **Distance**: distance from target
- **TimeToEnd**: time before reaching target.

Value of trigger can be (depending on trigger type):

- Distance in unit, normally millimeters.
- Time in seconds.

Example:

```
TRIGCALL trig2, Distance, 150, OpenGrimper() ;
…
TRIGON (trig2) ;
MLIN (p84, v500, fine, tool1, wobj4) ;
```

When the trigger ***trig2*** is fired the subroutine ***OpenGrimper*** will be executed. Trigger ***trig2*** is fired at a position 150 mm before target **p84**. So when robot is at distance of 150 mm subroutine **OpenGrimper** is executed.

TRIGON

Activates triggers for the next move. For each move can be set a maximum of 4 events with one instruction **TRIGON**. If you call more than one time the instruction **TRIGON,** the used triggers are the triggers activated with last instruction **TRIGON** before the movement instruction. After calling a movement instruction there is no trigger set for the next move, so you must use one more instruction **TRIGON** to activate triggers for the next move. Triggers are handled only for **MLIN** and **MCIRC**. After all other movement instructions (for example **MJOINT** or **EPATH** that don't handle events), the triggers for the next move are reset.

TRIGON (***trigger1***, [***trigger2***], [***trigger3***], [***trigger4***]) ;

Example:

```
trig1 := TRIGSET when Distance is 50 do (grimper = 1) ;
TRIGON (trig1) ;
MLIN (pa, v100, fine, tool1) ;
trig2 := TRIGSET when TimeToEnd is 1 do (airgun = 1) ;
TRIGON (trig2) ;
MLIN(pb, v100, fine, tool1) ;
```

In this example the variable **grimper** is set to **1** at a distance of **50** mm from point **pa**. Then variable **airgun** is set to **1** at **1** second before reaching point **pb**.

For the movement to point **pa** is active only event **trig1**, for the movement to point **pb** is active only event **trig2**.

TRIGSET

Set a variable of type ***TRIGGER***. When condition of trigger is fired specified variable is set with value of expression. Value of expression is calculated when **TRIGSET** instruction is executed.

trigger_var := TRIGSET when ***type*** is ***ref_value*** do (***var_to_set :***= ***expr_to_set***)

type of trigger can be:

- **Distance**: distance from target
- **TimeToEnd**: time before reaching target.

ref_value can be (depending on trigger type):

- Distance in unit, normally millimeters.
- Time in seconds.

var_to_set must be a numeric variable.

expr_to_set is the value that must be set in variable when event is fired.

The event is one shoot.

Example:

```
TRIGGER trig1
…
trig1 := TRIGSET when Distance is 100 do (glue := 1) ;
…
TRIGON (trig1) ;
MLIN (p12, v100, fine, tool1, wobj0) ;
```

When the trigger ***trig1*** is fired the variable ***glue*** is set to value ***1***. Trigger **trig1** is fired for example when robot is at 100 mm before **p12**.

WAIT

Waits for the execution of a condition and optionally interrupts the wait if a timeout is set. The result of the optional parameter will be ***0*** if WAIT is correctly terminated or ***not equal to zero*** in case of timeout. The ***timeout*** and ***result*** parameters are optional.

WAIT (***condition***, [***timeout***], [***result***]) ;

Example:

```
BOOL boxtokeep
…
WAIT (boxtokeep) ;
MLIN (pick, v100, fine, tool1) ;
```

In this example the linear move is executed after ***boxtokeep*** is ***true***.

UNLOAD

Unload a module from memory previously loaded with instruction **LOAD**. If the optional parameter save is defined as true and the module was changed after was loading the module will be saved before unload.

UNLOAD (**filename**, **save**);

Example:

```
LOAD ("parprog.xpl) ;
…
UNLOAD ("parprog.xpl", true) ;
```

In this example module loaded from "**parprog.xpl**" will be unloaded from memory. If the module was changed after was loaded the module will be saved.

WAIT_POS

Waits for the end the execution of the movements and optionally stops the waiting if a timeout is set. The ***result*** optional parameter will be set to ***0*** if WAIT_POS is correctly terminated or ***not equal to zero*** in case of timeout. All parameters are optional.

WAIT_POS ([***timeout***], [***result***]) ;

Example:

```
KMAXT (50, 15) ;
MJOINT (POINTC(400, 0, 0, 0, 180, 0), v100, z200, tool1) ;
MLIN (POINTC(600, 600, 0, 0, 180, 0), v100, z200, tool1) ;
WAIT_POS () ;
MLIN (POINTC(800, 400, 0, 0, 180, 0), v100, z200, tool1) ;
MLIN (POINTC(800, 600, 0, 0, 180, 0), v100, z200, tool1) ;
MLIN (POINTC(600, 800, 0, 0, 180, 0), v100, z200, tool1) ;
```

In this example the first linear move is not overlapped to the next movement, the other movements will overlap at a distance of 200 mm.

WHILE

Executes a block of code while the ***condition*** is ***true.***

WHILE ***condition*** DO
...
END_WHILE ;

Example:

```
WHILE i <= 10 DO
    i := i + 1 ;
    MLIN (POINTC(200, -500, 200, 0, 180, 0)) ;
    MLIN (POINTC(200, -900, 200, 0, 180, 0)) ;
END_WHILE ;
```

In this example the code in while loop is executed until the value of variable ***i*** is greater than 10.
When you enter a WHILE instruction a line with END_WHILE is automatically added.

Functions that can be used in expressions

ABS (*value*)

Returns the absolute value of a quantity.

```
i := ABS(-20) ;
```

In this example value 20 is assigned to variable ***i.***

ASIN (*value*)

Arcsin trigonometric function, the result is expressed in radiant.

```
val := ASIN(1) / pi ;
```

Variable **val** will be 0.5.

ACOS (*value*)

Arccos trigonometric function, the result is expressed in radiant.

```
val := ACOS(-1) / pi ;
```

Variable **val** will be 1.

ATAN2 (*value1, value2*)

Arctan trigonometric function, the result is expressed in radiant.

```
val := ATAN2(SQRT(2), SQRT(2)) / pi ;
valB := ATAN2(SQRT(3), 1) / pi * 180 ;
```

Variable **val** will be 0.25 and **valB** will be 60.

COS (*radiant*)

Cosine trigonometric function.

```
val := COS(pi) ;
```

Variable val will be -1.

LIMIT (value, minimum value, maximum value)

Limits the value within the specified range.

```
valueA := LIMIT(45, 10, 100) ;
valueB := LIMIT(800, 10, 100) ;
valueC := LIMIT(-20, 10, 100) ;
```

Variable **valueA** will be **45, valueB** will be **100** and **valueC** will be 10.

MAX (value a, value b)

Returns ***a*** if ***b*** is less than ***a***, otherwise returns ***b***.

```
valueA := MAX(3, 7) ;
valueB := MAX(10, 2) ;
```

Variable **valueA** will be 7 and **valueB** will be 10.

MIN (value a, value b)

Returns ***a*** if ***b*** is greater than ***a***, otherwise returns ***b***.

```
valueA := MIN(3, 7) ;
valueB := MIN(10, 2) ;
```

Variable **valueA** will be 3 and **valueB** will be 2.

MOD (value, divisor)

Gives the remainder of a division.

```
i := MOD(10, 3) ;
```

In this example the value of variable **i** will be 1.

PI ()

Returns the PI Greek value (3.1415...).

```
len := 2 * PI() * r ;
```

RAND (minimum value, maximum value)

Returns a random number between ***minimum value*** and ***maximum value***.

```
val := RAND(10, 100) ;
```

Variable **val** can be any value between **10** and **100**, for example can be 18.71.

RANGE (value, minimum value, maximum value)

Used to test if value is between the minimum maximum values, it returns ***true*** if ***value*** is between ***minimum value*** and ***maximum value***.

```
testA := RANGE(45, 10, 100) ;
testB := RANGE(800, 10, 100) ;
```

Variable **testA** will be **true** and **testB** will be **false**.

ROUND (*value*)

Returns the nearest integer value.

```
temp := 4.29 ;
i := ROUND(temp) ;
temp := 5.5 ;
a := ROUND(temp) ;
```

In this example variable **i** will be 4 and **a** will be 6.

SIGN (*value*)

Returns -1 if ***value*** is negative or +1 otherwise.

```
absValue := SIGN(-24.5) * (-24.5) ;
```

Variable **absValue** will be 24.5.

SIN (*radiant*)

Sine trigonometric function.

```
val := SIN(pi / 2) ;
```

Variable **val** will be 1.

SQRT (*value*)

Square root.

```
val := SQRT(16) ;
```

Variable **val** will be 4.

TAN (*radiant*)

Tangent trigonometric function.

```
val := TAN(pi / 4) ;
```

Variable **val** will be 1.

TFB ()

Returns the time elapsed from boot in seconds. Can also be used for time measurement.

```
timeA := TFB() ;
...
timeB := TFB() ;
elapsed := timeB - timeA ;
```

TRUNC (*value*)

Returns the integer part of a value.

```
integerValue := TRUNC(3.45) ;
```

Variable **integerValue** will be 3.

Functions for vector and point variables

VECT3 (*x, y, z*)

Returns a vector variable with 3 real (floating point) values.

```
Pos := VECT3(400, 300, 100) ;
```

VPOINTC (position vector, rotation vector)

Returns a point composed of two vectors: position (x, y, z) and rotation (rotX, rotY, rotZ). Where ***a*** is the rotation around axis ***z***, ***b*** around axis ***y***, ***c*** around axis ***x***.

```
Point := VPOINTC(VECT3(400, 300, 100), VECT3(90, 180, 0)) ;
```

Variable **Point** will be (400, 300, 100, 0, 180, 90).

VEPOINTC (position vector, rotation vector, eaux point)

Returns a point composed of two vectors position (x, y, z), rotation (rotX, rotY, rotZ) and a point that store external auxiliary axes joint position.

```
pos := VECT3(400, 300, 100) ;
rot := VECT3(0, 180, 90) ;
eaux := POINTJ(1, 2, 3, 4, 5, 6) ;
Point := VEPOINTC(pos, rot, eaux) ;
```

POINTC (*x, y, z, a, b, c*)

Returns a point composed with 6 real values. ***a*** is the rotation around axis ***z***, ***b*** around axis ***y***, ***c*** around axis ***x***.

```
Point := POINTC(400, 300, 100, 0, 180, 0) ;
```

POINTJ (*j1, j2, j3, j4, j5, j6*)

Returns a point composed with 6 real values. ***j1,j2,j3,j4,j5,j6*** are the joint positions

```
Point := POINTJ(0, 90, 0, 0, 0, 0) ;
```

EPOINTC (*x, y, z, a, b, c, ea1, ea2, ea3, ea4, ea5, ea6*)

Returns an extended point composed with 12 real values. ***a*** is the rotation around axis ***z***, ***b*** around axis ***y***, ***c*** around axis ***x***. Value ***ea1, ea2, ea3, ea4, ea5, ea6*** are external auxiliary axes joint position.

```
Point := EPOINTC(40, 300, 100, 0, 0, 0, 1, 2, 3, 4, 5, 6) ;
```

EPOINTJ (*j1, j2, j3, j4, j5, j6, ea1, ea2, ea3, ea4, ea5, ea6*)

Returns an extended point composed with 12 real values. Values **j1,j2,j3,j4,j5,j6** are the joint positions and values ***ea1, ea2, ea3, ea4, ea5, ea6*** are external auxiliary axes joint position.

```
Point := EPOINTJ(0, 90, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6) ;
```

AJEPOINTC (*cartesian point, eaux point*)

Returns an extended point composed with a cartesian point and a point that store external auxiliary axes joint position.

```
pos_cart := POINTC(100, 200, 300, 0, 180, 0) ;
eaux := POINTJ(1, 2, 3, 4, 5, 6) ;
Point := AJEPOINTC(pos_cart, eaux) ;
```

AJEPOINTJ (*joint point, eaux point*)

Returns an extended point composed with a joint point and a point that store external auxiliary axes joint position.

```
pos_joint := POINTJ(10, 20, 30, 1, 2, 3) ;
eaux := POINTJ(4, 5, 6, 7, 8, 9) ;
Point := AJEPOINTJ(pos_joint, eaux) ;
```

POSC (reference system)

Returns the actual tool center position (TCP) as a POINTC (relative to ***reference*** system)**.**

```
lostPiecePosition := POSC(wobj) ;
…
MLIN (servicePosition, v100, fine, tool, wobj0) ;
…
MLIN (lostPiecePosition, v100, fine, tool, wobj) ;
```

In the example above actual cartesian position is stored in variable **lostPiecePosition** then after some other instruction robot will move back to this point.

POSJ ()

Returns the actual joints position as a POINTJ.

```
retryPosition := POSJ() ;
…
MJOINT (retryPosition, v100, fine, tool) ;
```

POSITION (*cartesian point*)

Extracts the position vector (x, y, z) from a cartesian point.

```
Point := POINTC(400, 300, 100, 0, 180, 90) ;
Pos := POSITION(Point) ;
```

Variable Pos will be (400, 300, 100).

ROTATION (*cartesian point*)

Extracts the rotation vector (rotX, rotY, rotZ) from a cartesian point.

```
Point := POINTC(400, 300, 100, 0, 180, 90) ;
Rot := ROTATION(Point) ;
```

Variable Rot will be (90, 180, 0).

MODULE (*vector*)

Returns the magnitude (length) of vector.

```
vect := VECT3(SQRT(3), SQRT(3), SQRT(3)) ;
len := MODULE(vect) ;
```

Variable **len** will be 3.

DOT (vector a, vector b)

Returns the scalar product of two vectors.

```
a := VECT3(100, 0, 0) ;
b := VECT3(-100, 0, 0) ;
cosang := DOT(a, b) / (MODULE(a) * MODULE(b)) ;
```

Variable **cosang** will be -1.

CROSS (*vector a, vector b*)

Returns the vectorial product.

```
a := VECT3(100, 0, 0) ;
b := VECT3(0, 100, 0) ;
c := NORMALIZE(CROSS(a, b)) ;
```

Variable **c** will be (0,0,1)

NORMALIZE (*vector*)

Returns the normalized vector. Normalized vector is a vector which module is 1.

```
a := VECT3(100, 0, 0) ;
Versor := NORMALIZE(a) ;
```

Variable **versor** will be (1, 0, 0).

TOPOS (*cartesian point, reference system*)

Return a point espressed in ***world*** coordinate system from a point related to a ***refsys*** coordinate system.

```
wobj := REFSYS(wobj0, 100, 200, 50, 0, 0, 0) ;
pointA := POINTC(200, 300, 100, 0, 0, 0) ;
pointB := TOPOS(pointA, wobj) ;
```

Variable **pointB** will be (300, 500, 150, 0, 0, 0).

TOLOCALPOS (*cartesian point, reference system*)

Return a point espressed in ***refsys*** coordinate system from a point related to a ***world*** coordinate system.

```
wobj := REFSYS(wobj0, 100, 200, 50, 0, 0, 0) ;
pointA := POINTC(300, 500, 150, 0, 0, 0) ;
pointB := TOLOCALPOS(pointA, wobj) ;
```

Variable **pointB** will be (200, 300, 100, 0, 0, 0).

CALCPOSJ (*cartesian point, reference system*)

Convert a cartesian point related to a ***refsys*** coordinate system in actual robot joint position. Returned point is a POINTJ that is valid if conversion is possible and inside joint limit.

```
jointP := CALCPOSJ(pointFromCamera, refsysCamera) ;
IF ISPOINTVALID(jointP) THEN
    MJOINT(jointP, v100, fine, tool0) ;
ELSE
    MESSAGE("Piece not reachable") ;
END_IF ;
```

In the example above robot make a joint movement only if pointFromCamera is reachable.

ISPOINTVALID (*point*)

Return a boolean value that indicates if a point is valid.

OFFSET (*cartesian point, x, y, z*)

Function for add an offset in x, y, z direction to a cartesian point.

```
point_low := POINTC(100,200,0,0,0,0) ;
point_high := OFFSET(point_low, 10, 20, 30) ;
```

In example above **point_high** will be (110, 220, 30, 0, 0, 0).

DISTANCE (*cartesian point, cartesian point*)

Function for calculate linear distance between two cartesian point. This value is always positive.

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

```
p1 := POINTC(0,0,0,0,0,0) ;
p2 := POINTC(100,100,100,0,0,0) ;
dist := DISTANCE(p1, p2) ;
```

In example above **dist** will be 173.205.

OFFSETTOOL (*cartesian point, x, y, z, a, b, c*)

Function for add an offset, expressed in the active TOOL coordinate system, to a cartesian point.

```
tool := STOOL(0, 0, 100, 0, 45, 0) ;
point := POINTC(300, 0, 400, 180, -45, 180) ;
point_s := OFFSETTOOL(point, 0, 0, 100, 0, 0, 0) ;
```

In example above **point_s** will be (229.289, 0, 329.289, 180, -45, 180).

Functions for reference systems

REFSYS (*parent reference, x, y, z, a, b, c*)

Makes a reference system from a parent system and six **LREAL** values.

```
wobj := REFSYS(wobj0,100,200,300,0,0,0) ;
```

VREFSYS (*parent reference, position vector, rotation vector*)

Makes a reference system from a parent system and two vectors. The first vector is the position vector and the second is the rotation vector.

```
pos := VECT3(100,200,300) ;
rot := VECT3(0,0,0) ;
wobj := VREFSYS(wobj0,pos,rot) ;
```

REFSYS3P (*parent reference, origin point, x direction point, y direction point*)

Makes a reference system from a parent system and 3 points. First point is the origin point, second is a point in x direction and third is a point in y direction.

```
p0 := POINTC(100,200,300,0,0,0) ;
p1 := POINTC(200,200,300,0,0,0) ;
p2 := POINTC(100,300,300,0,0,0) ;
wobj := REFSYS3P(wobj0,p0,p1,p2) ;
```

CWOBJ ()

Returns the current active reference system.

```
wobj := CWOBJ() ;
```

RWOBJ ()

Returns the active robot reference system.

```
wobj := RWOBJ() ;
```

GETWOBJ (*base/holder flag, robot*)

Returns the base/holder refsys of a specific robot.
If flag is FALSE return base refsys of robot otherwise return holder refsys.
Base reference system is a fixed reference system.
Holder reference system may be a movable reference system.
For example a conveyor has a fixed reference system that describe where conveyor is positioned in the system.
Holder reference system of conveyor describe a reference system that is moving with the belt of conveyor.

```
wobj := GETWOBJ(false, cvy) ;
p0 := POINTC(0,0,0,0,0,0) ;
MLIN(p0,spe,ovl,tool,wobj) ;
```

In this example robot will move at origin of **cvy**.

```
wobjM := GETWOBJ(true, cvy) ;
rsPhoto := REFSYS(wobjM, x, y, z, a, b, c) ;
wobj := GETTRKWOBJ(rsPhoto, cvyTrackData) ;
```

In this example we make a movable refsys for tracking an object on a conveyor.

GETTRKWOBJ (*reference system, tracking data*)

Sets tracking parameters for a specific reference system and returns the movable reference system.

```
wobj := GETTRKWOBJ(wobjcvy, cvyTrackdata) ;
```

SETROBOTWOBJ (robot, parent reference, reference point)

Sets and returns the base refsys of a specific robot.

```
cvy := ROBOT("conveyor") ;
p0 := POINTC(200,-400,200,0,0,0) ;
wobj := SETROBOTWOBJ(cvy,wobj0,p0) ;
```

Functions for setting complex data

ROBOT (*axesgroup name*)

Returns **ROBOT** data retrieved from system with specified axesgroup name.
This function is normally used when you need like in tracking application to access different axesgroup from actual robot.

```
conveyor := ROBOT("conveyor_belt") ;
```

In this example we connect **conveyor** variable with axesgroup **conveyor_belt**.

TRACKING (*maxspace, startspace, maxspe, maxacc, type, par1*)

Used for setting **TRACKING** data.
Parameter **maxspace** define total space where tracking is enabled, **startspace** define the space where tracking must start, **maxspe** define maximum tracking speed during coupling ramp, **maxacc** define maximum tracking acceleration during coupling ramp, **type** define if tracking is linear or rotating. Last parameter **par1** is used for filtering purpose on conveyor position, a value of 0 means no filter. In case of conveyor position read by external encoder this parameter can be useful. A starting value can be 0.1. Correct value is a tradeoff between reading stability and response to change in conveyor speed.

```
cvyTrackData := TRACKING(1000, 300, 200, 2000, Linear, 0.1) ;
```

In this example we set **cvyTrackData** with some parameters. For further detail see tracking application note document.

SSPEED (*tangential speed, orientation speed*)

This function can be used to set **SPEED** data type.
Tangential speed is expressed in mm/s and orientation speed in degree/s.

```
spe := SSPEED(500, 5) ;
```

In this example we set **spe** with 500 mm/s as tangential speed and 5 degree/s as orientation speed.

SZONE (*linear distance, reorientation angular distance*)

This function can be used to set **ZONE** data type.
Linear distance is expressed in mm and reorientation angular distance in degree.

```
ovl := SZONE(100, 3) ;
```

In this example we set **ovl** with 100 mm as linear distance and 3 degree as angular distance.

STOOL (*x, y, z, a, b, c*)

This function can be used to set **TOOL** data type.

```
tool := STOOL(0, 0, 150, 0, 180, 0) ;
```

In this example we set **tool** with some parameters.

String related functions

BOOL_TO_STRING (*BOOL val*)

Converts a **BOOL** in a string.

```
BOOL bool_val
STRING strTrue
STRING strFalse
…
bool_val := true ;
strTrue := BOOL_TO_STRING(bool_val) ;
bool_val:= false ;
strFalse := BOOL_TO_STRING(bool_val) ;
```

After execution **strTrue** will be “true” and **strFalse** will be “false”.

DINT_TO_STRING (*DINT val*)

Converts a **DINT** in a string.

```
DINT dint_val
…
dint_val := -120 ;
strVal := DINT_TO_STRING(dint_val) ;
```

After execution **strVal** will be “-120”.

LREAL_TO_STRING (*LREAL val*)

Converts a **LREAL** in a string.

```
LREAL lreal_val
…
lreal_val := 3.14 ;
strVal := LREAL_TO_STRING(lreal_val) ;
```

After execution **strVal** will be “3.140000”.

UDINT_TO_STRING (*UDINT val*)

Converts a **UDINT** in a string.

```
UDINT udint_val
…
udint_val := 456 ;
strVal := UDINT_TO_STRING(udint_val) ;
```

After execution **strVal** will be “456”.

LEN (*STRING str*)

Returns length of string.

```
strA := "value" ;
n := LEN(strA) ;
```

In above example **n** after execution will be 5.

LEFT (*STRING str, pos*)

Returns a string composed with leftmost **pos** characters of string **str**.

```
strA := "Hello world" ;
strB := LEFT(strA, 5) ;
```

String **strB** after execution will be "Hello".

RIGHT (*STRING str, pos*)

Returns a string composed with rightmost **pos** characters of string **str**.

```
strA := "Hello world" ;
strB := RIGHT(strA, 5) ;
```

String **strB** after execution will be "world".

MID (*STRING str, len, pos*)

Returns a string composed with **len** characters beginning from **pos**-th character of string **str**.

```
strA := "Time is over" ;
strB := MID(strA, 2, 6) ;
```

String **strB** after execution will be "is".

CONCAT (*STRING str1, STRING str2*)

Returns a string composed from concatenation of string **str1** and string **str2**.

```
strA := "air" ;
strB := "plane" ;
strC := CONCAT(strA, strB) ;
```

String **strC** after execution will be "airplane".

INSERT (*STRING str1, STRING str2, pos*)

Returns a string composed by first **pos** characters of string **str1** followed by **str2** followed by remain characters of **str1**.

```
strA := "I go to office" ;
strB := "post " ;
strC := INSERT(strA, strB, 8) ;
```

String **strC** after execution will be "I go to post office".

FIND (*STRING str1, STRING str2*)

Returns the character position of first occurrence of string **str2** in string **str1**. If in string **str1** there is no occurrence returns 0.

```
strA := "policeman" ;
strB := "man" ;
n := FIND(strA, strB) ;
```

After exeution **n** will be 7.

DELETE (*STRING str, len, pos*)

Returns a string that is obtained by deleting **len** characters from string **str** beginning from position **pos**.

```
strA := "policeman" ;
strB := DELETE(strA, 3, 7) ;
```

String **strB** after execution will be "police".

REPLACE (*STRING str1, STRING str2, len, pos*)

Returns a string that is the result of replacing **len** characters at position **pos** of string **str1** with string **str2**.

```
strA := "grandmother" ;
strB := "fa" ;
strC := REPLACE(strA, strB, 2, 6) ;
```

String **strC** after execution will be "granfather".

STRING_TO_LREAL (*STRING str*)

Converts a string in a LREAL.

```
STRING strVal
LREAL lreal_val
…
strVal := "1.5" ;
lreal_val := STRING_TO_LREAL(strVal) ;
```

After execution **lreal_val** is 1.5;

STRING_TO_UDINT (*STRING str*)

Converts a string in a UDINT.

```
STRING strVal
UDINT udint_val
…
strVal := "124" ;
udint_val := STRING_TO_UDINT(strVal) ;
```

After execution **udint_val** is 124;

STRING_TO_DINT (*STRING str*)

Converts a string in a DINT.

```
STRING strVal
DINT dint_val
…
strVal := "-78" ;
dint_val := STRING_TO_DINT(strVal) ;
```

After execution **dint_val** is -78;

STRING_TO_BOOL (*STRING str*)

Converts a string in a BOOL.

```
STRING strVal
BOOL bool_val
…
strVal := "true" ;
bool_val := STRING_TO_BOOL(strVal) ;
```

After execution **bool_val** is true;

Other functions

RANDOM ()

Returns a random positive integer value between 0 and 32767.

```
val := RANDOM() ;
```

Variable **val** can have a value between 0 and 32767, for example **val** will be 27236.

R_AND (*UDINT a, UDINT b*)

Returns the binary AND of two values.

```
binA := 7 ;
binB := 5 ;
binC := R_AND(binA, binB) ;
```

Variable **binC** will be 5.

R_OR (*UDINT a, UDINT b*)

Returns the binary OR of two integer values.

```
binA := 7 ;
binB := 5 ;
binC := R_OR(binA, binB) ;
```

Variable **binC** will be 7.

R_XOR (*UDINT a, UDINT b*)

Returns the binary exclusive OR of two integer values.

```
binA := 7 ;
binB := 5 ;
binC := R_XOR(binA, binB) ;
```

Variable **binC** will be 2.

R_NOT (*UDINT*)

Returns the binary inversion of an integer value.

```
binA := 13 ;
binB := 255 ;
binC := R_AND(R_NOT(binA), binB) ;
```

Variable **binC** will be 242.

ABS_MOD (value, divisor)

Returns the remainder after the division of value by divisor, if value is negative a divisor is added to the result to make it positive.

```
valA := ABS_MOD(42, 5) ;
valB := ABS_MOD(-42, 5) ;
```

Variable **valA** will be 2 and **valB** will be 3.

CLOCKREAD (CLOCK variable)

Returns time elapsed contained in a **CLOCK** variable. Time is expressed in seconds.

```
CLOCKRESET(clk) ;
CLOCKSTART(clk) ;
…
CLOCKSTOP(clk) ;
MESSAGE ("Cycle time is %1", CLOCKREAD(clk)) ;
```

In the example above a possible message can be “Cycle time is 3.234”.

Revision history

Date	Revision	Changes
13-Oct-2016	1	removed **SET_AG** instruction and modify **USING_AG** instruction.
17-Oct-2016	2	Added **TRIGGER** datatype and **TRIGON** instruction.
17-Nov-2016	3	Added **TRIGSET**, **TRIGCALL**, **MESSAGE, ERROR** instruction. Removed **TRIGGER** function.
07-Dec-2016	4	Added instruction **STARTMOVE** and **STOPMOVE**. Renamed function **TRANSFORM** in **FROMLOCAL**, added function **TOLOCAL**.
25-Jan-2017	5	Changed syntax of instruction to met IEC61131-3 style. Removed example program.
03-Aug-2017	6	Renamed instruction **PCALL** as **THREAD**. Added instruction **JOIN**. Added instruction **ALIAS**. Added function **TOPOS**, **TOLOCALPOS**, **CWOBJ**, **RWOBJ**. Modified instruction **TRIGCALL**. Added function **CALCPOSJ** and **ISVALIDPOINT**.
10-Oct-2017	7	Added data type **SPEED**, **ZONE**, **TOOL**. Added parameters for instructions **MLIN**, **MCIRC**, **MJOINT**, **EPATH**. Added function **SSPEED**, **SZONE**, **STOOL**.
06-Nov-2017	8	Renamed function **REFSYS** as **VREFSYS**. Added new function **REFSYS**. Added instruction **CLEARMOVE**. Added new kind of variable: **TASK** and **GLOBAL**. Added attribute **CONST** and **RETAIN** for variable. Removed teachgun interface explanation from this manual.
08-Feb-2018	9	Removed parameter from **TRIGCALL**. Added data type called **INTR**. Added function **OFFSET**. Added instruction **PULSE, INTRSET, INTRDIS, INTRENA, INTRCOND, INTRERRNO, EXIT, RESTART, INTRALLOW, INTRDENY.** Changed instruction **MJOINT**, **MLIN**, **MCIRC**, **EPATH**.

Robox S.P.A.
Castelletto S.Ticino, Italy – www.robox.it

08-Mar-2018	10	Added data type called **CLOCK**. Added function **CLOCKREAD**. Added instruction **CLOCKRESET**, **CLOCKSTART**, **CLOCKSTOP**. Added data type called **TRACKING**. Added function **ROBOT**, **GETWOBJ**, **TRACKING**, **GETTRKWOBJ**, **SETTRKWOBJ**, **SETROBOTWOBJ**. Removed data type **AXES_GROUP** from types available for extern variables.
12-Jul-2018	11	Changed function **TRACKING**, **SETROBOTWOBJ**, **GETTRKWOBJ.** Removed function **SETTRKWOBJ**. Removed instruction **MBREAK, SETREF, DOVL, USING_AG**. Renamed instruction **KABST** as **KMAXT**, **KOVRJ** as **KMAXJ**, **KOVRW** as **KMAXW**. Changed instruction **MCIRC**. Added function **DISTANCE**.
08-Aug-2018	12	Added some examples.
26-Jun-2019	13	Removed instruction **USING_AG, _IF.** Added instruction **LOAD**, **UNLOAD**, **EXEC, ENDPROG, STOPPROG**. Added data types **EPOINTC**, **EPOINTJ**. Added functions **VEPOINTC**, **EPOINTC**, **EPOINTJ**, **AJEPOINTC**, **AJEPOINTJ, OFFSETTOOL**. Added string related functions. Added some examples. Removed function **TOLOCAL**, **FROMLOCAL**, **REF3P**.