RAPID reference manual

BaseWare

RAPID reference part 2, Functions and data types A-Z

RobotWare-OS 4.0

RAPID reference manual
3HAC 7774-1
Revision B

BaseWare
RAPID reference part 2, Functions and data types A-Z

RobotWare-OS 4.0

Table of contents

Functions A-Z

Data types A-Z

Index

ABB Automation Technology Products AB
Robotics
SE-721 68 Västerås
Sweden

Abs - Gets the absolute value

Abs is used to get the absolute value, i.e. a positive value of numeric data.

Example

reg1 := Abs(reg2);

Reg1 is assigned the absolute value of *reg2*.

Return value Data type: *num*

The absolute value, i.e. a positive numeric value.

e.g.

Input value	Returned value
3	3
-3	3
-2.53	2.53

Arguments

Abs (Input)

Input Data type: *num*

The input value.

Example

TPReadNum no_of_parts, "How many parts should be produced? ";
no_of_parts := Abs(no_of_parts);

The operator is asked to input the number of parts to be produced. To ensure that the value is greater than zero, the value given by the operator is made positive.

Syntax

Abs ’(’
  [Input ’:=’] < expression (**IN**) of *num* > ’)’

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

ACos - Calculates the arc cosine value

ACos (Arc Cosine) is used to calculate the arc cosine value.

Example

VAR num angle;
VAR num value;
.
.
angle := ACos(value);

Return value

ACos Data type: *num*

The arc cosine value, expressed in degrees, range [0, 180].

Arguments

ACos (Value)

Value Data type: *num*

The argument value, range [-1, 1].

Syntax

Acos'('
[Value ':='] <expression (**IN**) of *num*>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

AOutput - Reads the value of an analog output signal

AOutput is used to read the current value of an analog output signal.

Example

IF AOutput(ao4) > 5 THEN ...

If the current value of the signal *ao4* is greater than *5,* then ...

Return value

Data type: *num*

The current value of the signal.

The current value is scaled (in accordance with the system parameters) before it is read by the RAPID program. See Figure 1.

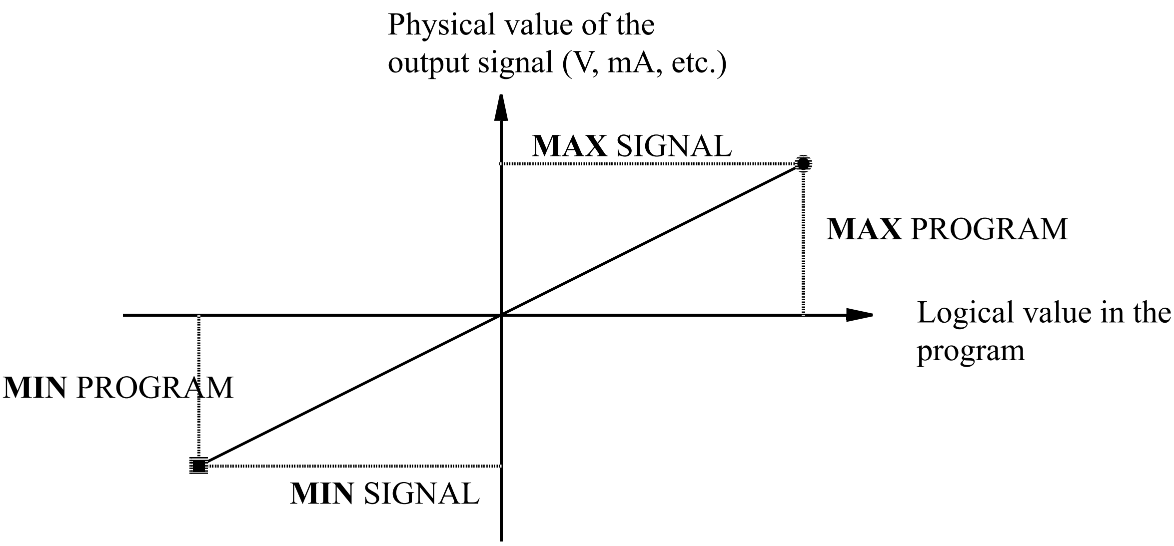


Figure 1 Diagram of how analog signal values are scaled.

Arguments

AOutput (Signal)

Signal Data type: *signalao*

The name of the analog output to be read.

Syntax

AOutput '('
[Signal ':='] < variable (**VAR**) of *signalao* > ')'

A function with a return value of data type *num*.

Related information

	Described in:
Input/Output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general *I/O Principles*	Motion and I/O Principles -
Configuration of I/O	User's Guide - *System Parameters*

ASin - Calculates the arc sine value

ASin (Arc Sine) is used to calculate the arc sine value.

Example

```
VAR num angle;
VAR num value;
.
.
angle := ASin(value);
```

Return value Data type: *num*

The arc sine value, expressed in degrees, range [-90, 90].

Arguments

ASin (Value)

Value Data type: *num*

The argument value, range [-1, 1].

Syntax

```
ASin'('
    [Value ':='] <expression (IN) of num>
    ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

ATan - Calculates the arc tangent value

ATan (Arc Tangent) is used to calculate the arc tangent value.

Example

```
VAR num angle;
VAR num value;
.
.
angle := ATan(value);
```

Return value Data type: *num*

The arc tangent value, expressed in degrees, range [-90, 90].

Arguments

ATan (Value)

Value Data type: *num*

The argument value.

Syntax

```
ATan'('
    [Value ':='] <expression (IN) of num>
    ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Arc tangent with a return value in the range [-180, 180]	Functions - *ATan2*

ATan2 - Calculates the arc tangent2 value

ATan2 (Arc Tangent2) is used to calculate the arc tangent2 value.

Example

```
VAR num angle;
VAR num x_value;
VAR num y_value;
.
.
angle := ATan2(y_value, x_value);
```

Return value Data type: *num*

The arc tangent value, expressed in degrees, range [-180, 180].

The value will be equal to ATan(y/x), but in the range [-180, 180], since the function uses the sign of both arguments to determine the quadrant of the return value.

Arguments

ATan2 (Y X)

Y Data type: *num*

The numerator argument value.

X Data type: *num*

The denominator argument value.

Syntax

ATan2'('
    [Y ':='] <expression (**IN**) of *num*> ','
    [X ':='] <expression (**IN**) of *num*>
    ')'

A function with a return value of the data type *num*.

Related information

	<u>Described in:</u>
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Arc tangent with only one argument	Functions - *ATan*

ByteToStr - Converts a byte to a string data

ByteToStr (Byte To String) is used to convert a *byte* into a *string* data with a defined byte data format.

Example

VAR string con_data_buffer{5};
VAR byte data1 := 122;

con_data_buffer{1} := ByteToStr(data1);

The content of the array component *con_data_buffer{1}* will be "122" after the *ByteToStr* ... function.

con_data_buffer{2} := ByteToStr(data1\Hex);

The content of the array component *con_data_buffer{2}* will be "7A" after the *ByteToStr* ... function.

con_data_buffer{3} := ByteToStr(data1\Okt);

The content of the array component *con_data_buffer{3}* will be "172" after the *ByteToStr* ... function.

con_data_buffer{4} := ByteToStr(data1\Bin);

The content of the array component *con_data_buffer{4}* will be "01111010"after the *ByteToStr* ... function.

con_data_buffer{5} := ByteToStr(data1\Char);

The content of the array component *con_data_buffer{5}* will be "z" after the *ByteToStr* ... function.

Return value

ByteToStr Data type: *string*

The result of the conversion operation with the following format:

Format:	Characters:	String length:	Range:
Dec .....:	’0’ - ’9’	1-3	"0" - "255"
Hex .....:	’0’ - ’9’, ’A’ -’F’	2	"00" - "FF"
Okt ......:	’0’ - ’7’	3	"000" - "377"
Bin ......:	’0’ - ’1’	8	"00000000" - "11111111"
Char ....:	Any ASCII char (*)	1	One ASCII char

(*) If non-writable ASCII char, the return format will be RAPID character code format (e.g. “\07” for BEL control character).

Arguments

ByteToStr (BitData [\Hex] | [\Okt] | [\Bin] | [\Char])

BitData Data type: *byte*

The bit data to be converted.

If the optional switch argument is omitted, the data will be converted in *decimal* (Dec) format.

[\Hex] *(Hexadecimal)* Data type: *switch*

The data will be converted in *hexadecimal* format.

[\Okt] *(Octal)* Data type: *switch*

The data will be converted in *octal* format.

[\Bin] *(Binary)* Data type: *switch*

The data will be converted in *binary* format.

[\Char] *(Character)* Data type: *switch*

The data will be converted in *ASCII character* format.

Limitations

The range for a data type *byte* is 0 to 255 decimal.

Syntax

ByteToStr'('
[BitData ':='] <expression (**IN**) of *byte*>
['\' Hex] | ['\' Okt] | ['\' Bin] | ['\' Char]
')' ';'

A function with a return value of the data type *string*.

Related information

	Described in:
Convert a string to a byte data	Instructions - *StrToByte*
Other bit (byte) functions	RAPID Summary - Bit Functions
Other string functions	RAPID Summary - String Functions

CalcJointT - Calculates joint angles from robtarget

CalcJointT (Calculate Joint Target) is used to calculate joint angles of the robot axes and external axes from a specified *robtarget* data.

The input *robtarget* data should be specified in the same coordinate system as specified in argument for *Tool*, *WObj* and at execution time active program displacement (*ProgDisp*) and external axes offset (*EOffs)*.
The returned *jointtarget* data is expressed in the calibration coordinate system.

Example

VAR jointtarget jointpos1;
CONST robtarget p1 := [...];

jointpos1 **:= CalcJointT(p1, tool1 \WObj:=wobj1);**

The jointtarget value corresponding to the robtarget value *p1* is stored in *jointpos1*. The tool *tool1* and work object *wobj1* are used for calculating the joint angles *jointpos1*.

Return value Data type: *jointtarget*

The angles in degrees for the axes of the robot on the arm side.

The values for the external axes, in mm for linear axes, in degrees for rotational axes.

The returned values are always related to the calibration position.

Arguments

CalcJointT (Rob_target Tool [\WObj])

Rob_target Data type: *robtarget*

The position of the robot and external axes in **the outermost coordinate system, related to the specified tool and work object and at execution time active program displacement (***ProgDisp***) and/or external axes offset (***EOffs***).**

Tool Data type: *tooldata*

The tool used for calculation of the robot joint angles.

[\WObj] **(***Work Object***)** Data type: *wobjdata*

The work object (coordinate system) to which the robot position is related.

If this argument is omitted the work object *wobj0* is used.
This argument must be specified when using stationary tool, coordinated external axes, or conveyor

Program execution

The returned *jointtarget* is calculated from the input *robtarget*.
To calculate the robot joint angles, the specified *Tool*, *WObj* (including coordinated user frame) and the *ProgDisp* active at execution time, are taken into consideration.
To calculate the external axis position at the execution time, active *EOffs* is taken into consideration.

The calculation always selects the robot configuration according to the specified configuration data in the input *robtarget* data. Instructions *ConfL* and *ConfJ* do not affect this calculation principle. When wrist singularity is used, robot axis 4 will be set to 0 degrees.

If there is any active program displacement (*ProgDisp*) and/or external axis offset (*EOffs*) at the time *the robtarget* is stored, then the same program displacement and/or external axis offset must be active when *CalcJointT* is executed.

Error handling

If at least one axis is outside the working area or the limits are exeeded for at least one coupled joint, the system variable ERRNO is set to ERR_ROBLIMIT and the execution continues in the error handler.

The error handler can then deal with the situation.

Syntax

CalcJointT'('
    [Rob_target ':='] <expression (**IN**) of *robtarget*> ','
    [Tool ':='] <persistent (**PERS**) of *tooldata*>
    ['\'WObj ':=' <persistent (**PERS**) of *wobjdata*>] ')'

A function with a return value of the data type *jointtarget*.

Related information

	Described in:
Calculate robtarget from jointtarget	Functions - *CalcRobT*
Definition of position	Data Types - *robtarget*
Definition of joint position	Data Types - *jointtarget*
Definition of tools	Data Types- *tooldata*
Definition of work objects	Data Types - *wobjdata*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Program displacement coordinate system	Instructions - *PDispOn*
External axis offset coordinate system	Instructions - *EOffsOn*

CalcRobT - Calculates robtarget from jointtarget

CalcRobT (Calculate Robot Target) is used to calculate a robtarget data from a given jointtarget data.

This function returns a *robtarget* value with position (x, y, z), orientation (q1 ... q4), robot axes configuration and external axes position.

The input *jointtarget* data should be specified in the calibration coordinate system. The returned *robtarget* data is expressed in the outermost coordinate system, taking the specified tool, work object and at execution time active program displacement (*ProgDisp*) and external axis offset (*EOffs*) into consideration.

Example

```
VAR robtarget p1;
CONST jointtarget jointpos1 := [...];
```

p1 := CalcRobT(jointpos1, tool1 \WObj:=wobj1);

The robtarget value corresponding to the jointtarget value *jointpos1* is stored in *p1*. The tool *tool1* and work object *wobj1* are used for calculating of the position *p1*.

Return value Data type: *robtarget*

The robot and external axis position is returned in data type *robtarget* and expressed in the outermost coordinate system, taking the specified tool, work object and at execution time active program displacement (*ProgDisp*) and external axes offset (*EOffs*) into consideration.

If there is no active *ProgDisp,* the robot position is expressed in the object coordinate system.
If there are no active *EOffs,* the external axis position is expressed in the calibration coordinate system.

Arguments

CalcRobT (Joint_target Tool [\WObj])

Joint_target Data type: *jointtarget*

The joint position for the robot axes and external axes related to the calibration coordinate system.

Tool Data type: *tooldata*

The tool used for calculation of the robot position.

[\WObj] **(***Work Object***)** Data type: *wobjdata*

The work object (coordinate system) to which the robot position returned by the function is related.

If this argument is omitted the work object *wobj0* is used.
This argument must be specified when using stationary tool, coordinated external axes, or conveyor.

Program execution

The returned *robtarget* is calculated from the input *jointtarget*.
To calculate the cartesian robot position, the specified *Tool*, *WObj* (including coordinated user frame) and at the execution time active *ProgDisp* are taken into consideration.
To calculate the external axes position, the *EOffs* active at execution time is taken into consideration.

Syntax

CalcRobT'('
 [Joint_target ':='] <expression (**IN**) of *jointtarget*> ','
 [Tool ':='] <persistent (**PERS**) of *tooldata*>
 ['\'WObj ':=' <persistent (**PERS**) of *wobjdata*>] ')'

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Calculate jointtarget from robtarget	Functions - *CalcJointT*
Definition of position	Data Types - *robtarget*
Definition of joint position	Data Types - *jointtarget*
Definition of tools	Data Types- *tooldata*
Definition of work objects	Data Types - *wobjdata*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Program displacement coordinate system	Instructions - *PDispOn*
External axes offset coordinate system	Instructions - *EOffsOn*

CalcRotAxisFrame - Calculate a rotational axis frame

CalcRotAxisFrame (Calculate Rotational Axis Frame) is used to calculate the user coordinate system of a rotational axis type mechanical unit.

Description

The definition of a user frame for a rotational external axis requires that the turntable (or similar mechanical structure) on the external axis has a marked reference point. Moreover the master robot's base frame and TCP must be calibrated. The calibration procedure consists of a number of positionings for the robot's TCP on the reference point when the turntable is rotated to different angles. See Figure 2.

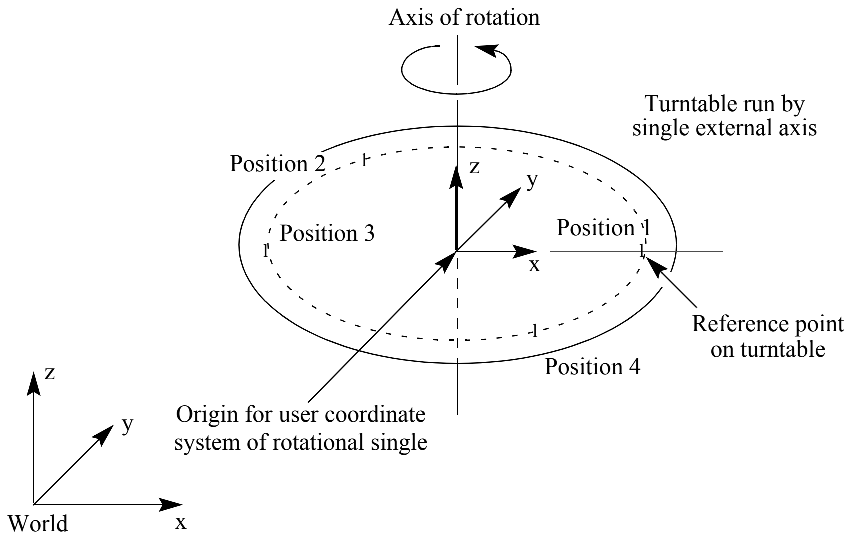


Figure 2 Definition of points for a rotational axis

The user coordinate system for the rotational axis has its origin in the centre of the turntable. The z direction coincides with the axis of rotation and the x axis goes through the reference point. Figure 3 shows the user coordinate system for two different positionings of the turntable (turntable seen from above).

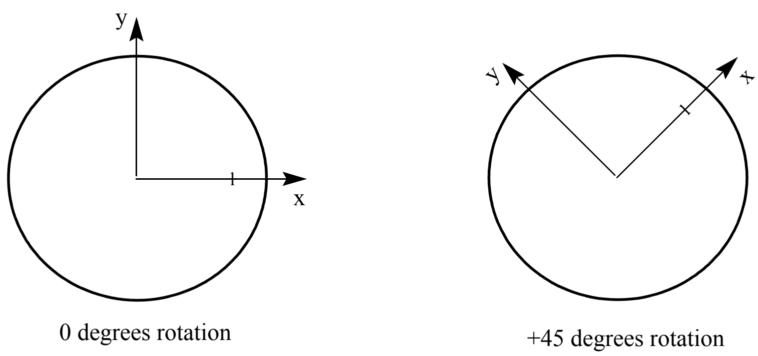


Figure 3 The user coordinate system at various angles of rotation

Example

```
CONST robtarget pos1 := [...];
CONST robtarget pos2 := [...];
CONST robtarget pos3 := [...];
CONST robtarget pos4 := [...];
VAR robtarget targetlist{10};
VAR num max_err := 0;
VAR num mean_err := 0;
VAR pose resFr:=[...];
PERS tooldata tMyTool:= [...];

! Instructions for creating/ModPos pos1 - pos4 with TCP pointing at the turntable.
MoveJ pos1, v10, fine, tMyTool;
MoveJ pos2, v10, fine, tMyTool;
MoveJ pos3, v10, fine, tMyTool;
MoveJ pos4, v10, fine, tMyTool;

! Add the targets to the array
targetlist{1}:= pos1;
targetlist{2}:= pos2;
targetlist{3}:= pos3;
targetlist{4}:= pos4;

resFr:=CalcRotAxisFrame(STN_1 , targetlist, 4, max_err, mean_err);

! Update the system parameters.
IF (max_err < 1.0) AND (mean_err < 0.5) THEN
    WriteCfgData "/MOC/SINGLE/STN_1", "base_frame_pos_x",resFr.trans.x;
    WriteCfgData "/MOC/SINGLE/STN_1", "base_frame_pos_y",resFr.trans.y;
    WriteCfgData "/MOC/SINGLE/STN_1", "base_frame_pos_z",resFr.trans.z;
    WriteCfgData "/MOC/SINGLE/STN_1", "base_frame_orient_u0",resFr.rot.q1;
    WriteCfgData "/MOC/SINGLE/STN_1", "base_frame_orient_u1",resFr.rot.q2;
    WriteCfgData "/MOC/SINGLE/STN_1", "base_frame_orient_u2",resFr.rot.q3;
    WriteCfgData "/MOC/SINGLE/STN_1", "base_frame_orient_u3",resFr.rot.q4;
     TPReadFK reg1,"Warmstart required for calibration to take effect."
              ,stEmpty,stEmpty,stEmpty, stEmpty,"OK";
    WarmStart;
ENDIF
```

Four positions, *pos1 - pos4*, are created/modposed so that the robots tool *tMyTool* points to the same reference point on the external axis *STN_1* but with different external axis rotation. The points are then used for calculating the external axis base frame, *resFr*, in relation to the world coordinate system. Finally, the frame is written to the configuration file and a warmstart is made to let the change to take effect.

Return value

CalcRotAxisFrame Data type: *pose*

The calculated frame.

Arguments

CalcRotAxisFrame (MechUnit [\AxisNo] TargetList TargetsInList MaxErr MeanErr)

MechUnit Data type: *mecunit*

Name of the mechnical unit to be calibrated.

[\AxisNo] Data type: *num*

Optional argument defining the axis number for which a frame should be determined. Default value is 1 applying to single rotational axis. For mechanical units with several axes, the axis number should be supplied with this argument.

TargetList Data type: *robtarget*

Array of robtargets holding the positions defined by pointing out the turntable. Minimum number of robtargets is 4, maximum 10.

TargetsInList Data type: *num*

Number of robtargets in array.

MaxErr Data type: *num*

The estimated maximum error in mm.

MeanErr Data type: *num*

The estimated mean error in mm.

Error handling

If the positions don’t have the required relation or are not specified with enough accuracy, the system variable ERRNO is set to ERR_FRAME. This error can then be handled in an error handler.

Syntax

CalcRotAxisFrame'('
[MechUnit ':='] <variable (**VAR**) of *mecunit*>
[\AxisNo ':=' <expression (**IN**) of *num*>]','
[TargetList ':='] <array {*} (**IN**) of *robtarget*> ','
[TargetsInList ':='] <expression (**IN**) of *num*> ','
[MaxErr ':='] <variable (**VAR**) of *num*> ','
[MeanErr ':='] <variable (**VAR**) of *num*>')'

A function with a return value of the data type *pose*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

CDate - Reads the current date as a string

CDate (Current Date) is used to read the current system date.

This function can be used to present the current date to the operator on the teach pendant display or to paste the current date into a text file that the program writes to.

Example

```
VAR string date;

date := CDate();
```

The current date is stored in the variable *date*.

Return value

Data type: *string*

The current date in a string.

The standard date format is “year-month-day”, e.g. ”1998-01-29”.

Example

```
date := CDate();
TPWrite “The current date is: “+date;
Write logfile, date;
```

The current date is written to the teach pendant display and into a text file.

Syntax

```
CDate ’(’ ’)’
```

A function with a return value of the type *string*.

Related information

	Described in:
Time instructions	RAPID Summary - *System & Time*
Setting the system clock	User’s Guide - *Service*

CJointT - Reads the current joint angles

CJointT (Current Joint Target) is used to read the current angles of the robot axes and external axes.

Example

VAR jointtarget joints;

joints **:= CJointT();**

The current angles of the axes for the robot and external axes are stored in *joints*.

Return value Data type: *jointtarget*

The current angles in degrees for the axes of the robot on the arm side.

The current values for the external axes, in mm for linear axes, in degrees for rotational axes.

The returned values are related to the calibration position.

Syntax

CJointT'(")'

A function with a return value of the data type *jointtarget*.

Related information

	Described in:
Definition of joint	Data Types - *jointtarget*
Reading the current motor angle	Functions - *ReadMotor*

ClkRead - Reads a clock used for timing

ClkRead is used to read a clock that functions as a stop-watch used for timing.

Example

reg1:=ClkRead(clock1);

The clock *clock1* is read and the time in seconds is stored in the variable *reg1*.

Return value

Data type: *num*

The time in seconds stored in the clock. Resolution 0.010 seconds.

Argument

ClkRead (Clock)

Clock Data type: *clock*

The name of the clock to read.

Program execution

A clock can be read when it is stopped or running.

Once a clock is read it can be read again, started again, stopped, or reset.

Error handling

If the clock runs for 4,294,967 seconds (49 days 17 hours 2 minutes 47 seconds) it becomes overflowed and the system variable ERRNO is set to ERR_OVERFLOW.

The error can be handled in the error handler.

Syntax

ClkRead '('
    [Clock ':='] < variable (**VAR**) of *clock* > ')'

A function with a return value of the type *num*.

Related information

	Described in:
Clock instructions	RAPID Summary - *System & Time*
Clock overflow	Data Types - *clock*
More examples	Instructions - *ClkStart*

Cos - Calculates the cosine value

Cos (Cosine) is used to calculate the cosine value from an angle value.

Example

```
VAR num angle;
VAR num value;
.
.
value := Cos(angle);
```

Return value Data type: *num*

The cosine value, range = [-1, 1] .

Arguments

Cos (Angle)

Angle Data type: *num*

The angle value, expressed in degrees.

Syntax

Cos'('
    [Angle ':='] <expression (**IN**) of *num*>
    ')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

CPos - Reads the current position (pos) data

CPos (Current Position) is used to read the current position of the robot.

This function returns the x, y, and z values of the robot TCP as data of type *pos*. If the complete robot position (*robtarget*) is to be read, use the function *CRobT* instead.

Example

VAR pos pos1;

MoveL *, v500, fine \Inpos := inpos50, tool1;
pos1 := CPos(\Tool:=tool1 \WObj:=wobj0);

The current position of the robot TCP is stored in variable *pos1*. The tool *tool1* and work object *wobj0* are used for calculating the position.

Note that the robot is standing still before the position is read and calculated. This is achieved by using the stop point *fine* within position accuracy *inpos50* in the preceding movement instruction.

Return value Data type: *pos*

The current position (pos) of the robot with x, y, and z in **the outermost coordinate system, taking the specified tool, work object and active** ProgDisp coordinate system **into consideration.**

Arguments

CPos ([\Tool] [\WObj])

[\Tool] Data type: *tooldata*

The tool used for calculation of the current robot position.

If this argument is omitted the current active tool is used.

[\WObj] **(***Work Object***)** Data type: *wobjdata*

The work object (coordinate system) to which the current robot position returned by the function is related.

If this argument is omitted the current active work object is used.

It is very sensible to always specify the arguments \Tool and \WObj during programming. The function will then always return the wanted position even if some other tool or work object has been activated manually.

Program execution

The coordinates returned represent the TCP position in the ProgDisp coordinate system.

Example

VAR pos pos2;
VAR pos pos3;
VAR pos pos4;

pos2 := CPos(**Tool:=grip3 \WObj:=fixture**);
.
.
pos3 := CPos(**Tool:=grip3 \WObj:=fixture**);
pos4 := pos3-pos2;

The x, y, and z position of the robot is captured at two places within the program using the *CPos* function. The tool *grip3* and work object *fixture* are used for calculating the position. The x, y and z distances travelled between these positions are then calculated and stored in the *pos* variable *pos4*.

Syntax

CPos '('
    ['\'Tool ':=' <persistent (**PERS**) of *tooldata*>]
    ['\'WObj ':=' <persistent (**PERS**) of *wobjdata*>] ')'

A function with a return value of the data type *pos*.

Related information

	Described in:
Definition of position	Data Types - *pos*
Definition of tools	Data Types- *tooldata*
Definition of work objects	Data Types - *wobjdata*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Reading the current *robtarget*	Functions - *CRobT*

CRobT - Reads the current position (robtarget) data

CRobT (Current Robot Target) is used to read the current position of the robot and external axes.

This function returns a *robtarget* value with position (x, y, z), orientation (q1 ... q4), robot axes configuration and external axes position. If only the x, y, and z values of the robot TCP (*pos*) are to be read, use the function *CPos* instead.

Example

```
VAR robtarget p1;

MoveL *, v500, fine \Inpos := inpos50, tool1;
p1 := CRobT(\Tool:=tool1 \WObj:=wobj0);
```

The current position of the robot and external axes is stored in *p1*. The tool *tool1* and work object *wobj0* are used for calculating the position.

Note that the robot is standing still before the position is read and calculated. This is achieved by using the stop point *fine* within position accuracy *inpos50* in the preceding movement instruction.

Return value Data type: *robtarget*

The current position of the robot and external axes in **the outermost coordinate system, taking the specified tool, work object and active** ProgDisp/ExtOffs coordinate system **into consideration.**

Arguments

CRobT ([\Tool] [\WObj])

[\Tool] Data type: *tooldata*

The tool used for calculation of the current robot position.

If this argument is omitted the current active tool is used.

[\WObj] **(***Work Object***)** Data type: *wobjdata*

The work object (coordinate system) to which the current robot position returned by the function is related.

If this argument is omitted the current active work object is used.

It is very sensible to always specify the arguments \Tool and \WObj during programming. The function will then always return the wanted position even if some other tool or work object has been activated manually.

Program execution

The coordinates returned represent the TCP position in the ProgDisp coordinate system. External axes are represented in the ExtOffs coordinate system.

Example

```
VAR robtarget p2;

p2 := ORobT( RobT(\Tool:=grip3 \WObj:=fixture) );
```

The current position in the object coordinate system (without any ProgDisp or ExtOffs) of the robot and external axes is stored in *p2*. The tool *grip3* and work object *fixture* are used for calculating the position.

Syntax

CRobT'('
    ['\'Tool ':=' <persistent (**PERS**) of *tooldata*>]
    ['\'WObj ':=' <persistent (**PERS**) of *wobjdata*>] ')'

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Definition of position	Data Types - *robtarget*
Definition of tools	Data Types- *tooldata*
Definition of work objects	Data Types - *wobjdata*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
ExtOffs coordinate system	Instructions - *EOffsOn*
Reading the current *pos* (x, y, z only)	Functions - *CPos*

CSpeedOverride - Reads the current override speed

CSpeedOverride is used to read the speed override set by the operator from the Program or Production Window. The return value is displayed as a percentage where 100% corresponds to the programmed speed.

Note! Must not be mixed up with the argument *Override* in the RAPID instruction *VelSet*.

Example

```
VAR num myspeed;

myspeed := CSpeedOverride();
```

The current override speed will be stored in the variable *myspeed*.
E.g if the value is 100 this is equivalent to 100%.

Return value Data type: *num*

The override speed value in percent of the programmed speed. This will be a numeric value in the range 0 - 10**0.**

Syntax

CSpeedOverride'('')'

A function with a return value of the data type *num*.

Related information

	Described in:
Changing the Override Speed	Users Guide *Programming and Testing Production Running*

CTime - Reads the current time as a string

CTime is used to read the current system time.

This function can be used to present the current time to the operator on the teach pendant display or to paste the current time into a text file that the program writes to.

Example

VAR string time;

time := CTime();

The current time is stored in the variable *time*.

Return value

Data type: *string*

The current time in a string.

The standard time format is "hours:minutes:seconds", e.g. "18:20:46".

Example

time := CTime();
TPWrite “The current time is: “+time;
Write logfile, time;

The current time is written to the teach pendant display and written into a text file.

Syntax

CTime ’(’ ’)’

A function with a return value of the type *string*.

Related Information

	Described in:
Time and date instructions	RAPID Summary - *System & Time*
Setting the system clock	User's Guide - *System Parameters*

CTool - Reads the current tool data

CTool (Current Tool) is used to read the data of the current tool.

Example

PERS tooldata temp_tool:= [TRUE, [[0, 0, 0], [1, 0, 0 ,0]],
[0.001, [0, 0, 0.001], [1, 0, 0, 0], 0, 0, 0]];

temp_tool **:= CTool();**

The value of the current tool is stored in the variable *temp_tool*.

Return value Data type: *tooldata*

This function returns a *tooldata* value holding the value of the current tool, i.e. the tool last used in a movement instruction.

The value returned represents the TCP position and orientation in the wrist centre coordinate system, see *tooldata*.

Syntax

CTool'(")'

A function with a return value of the data type *tooldata*.

Related information

	Described in:
Definition of tools	Data Types- *tooldata*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

CWObj - Reads the current work object data

CWObj (Current Work Object) is used to read the data of the current work object.

Example

```
PERS wobjdata temp_wobj;

temp_wobj := CWObj();
```

The value of the current work object is stored in the variable *temp_wobj*.

Return value Data type: *wobjdata*

This function returns a *wobjdata* value holding the value of the current work object, i.e. the work object last used in a movement instruction.

The value returned represents the work object position and orientation in the world coordinate system, see *wobjdata*.

Syntax

CWObj'('')'

A function with a return value of the data type *wobjdata*.

Related information

	Described in:
Definition of work objects	Data Types- *wobjdata*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

DefAccFrame - Define an accurate frame

DefAccFrame (Define Accurate Frame) is used to define a frame from three to ten original positions and the same number of displaced positions.

Description

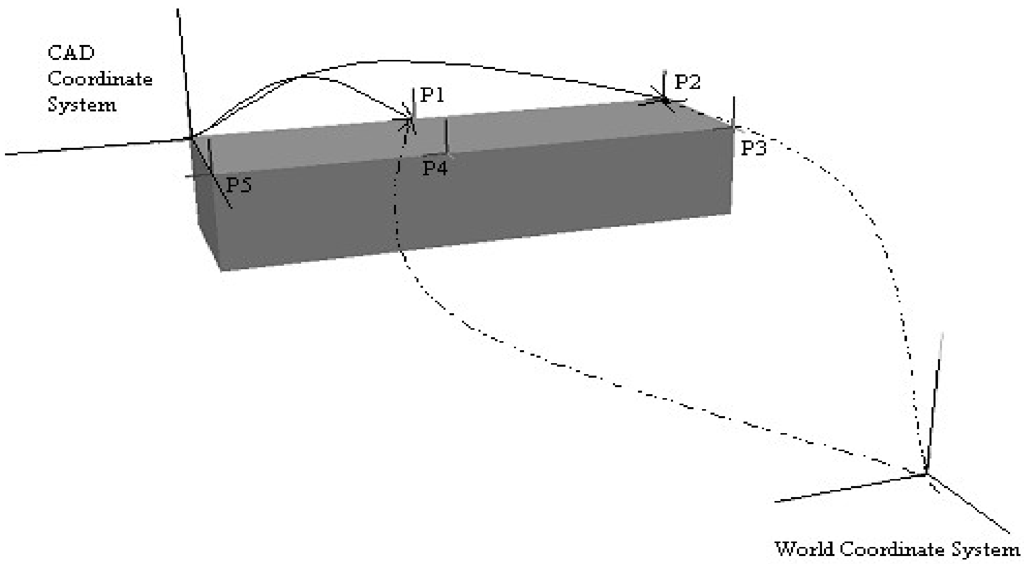
A frame can be defined when a set of targets are known at two different locations. Thus, *the same physical positions* are used but expressed differently.
Consider it in two different approaches:

I: The same physical positions are expressed in relation to different coordinate systems. For example, a number of positions are retrieved from a CAD drawing, thus the positions are expressed in CAD local coordinate system. The same positions are then expressed in robot world coordinate system. From these two sets of positions the frame between CAD coordinate system and robot world coordinate system is calculated.

II: A number of positions are related to an object in an original position. After a displacement of the object, the positions are determined again (often searched for). From these two sets of positions (old positions, new positions) the displacement frame is calculated.

Three targets are enough to define a frame, but to improve accuracy several points should be used.

Example



```
CONST robtarget p1 := [...];
CONST robtarget p2 := [...];
CONST robtarget p3 := [...];
CONST robtarget p4 := [...];
CONST robtarget p5 := [...];

VAR robtarget p6 := [...];
VAR robtarget p7 := [...];
VAR robtarget p8 := [...];
VAR robtarget p9 := [...];
VAR robtarget p10 := [...];
VAR robtarget pWCS{5};
VAR robtarget pCAD{5};

VAR pose frame1;
VAR num max_err;
VAR num mean_err;

! Add positions to robtarget arrays
pCAD{1}:=p1;
....
pCAD{5}:=p5;

pWCS{1}:=p6;
....
pWCS{5}:=p10;
```

```
frame1 := DefAccFrame (pCAD, pWCS, 5, max_err, mean_err);
```

Five positions *p1- p5*, related to an object, have been stored. The five positions are also stored in relation to world coordinate system as p6-p10. From these 10 positions the frame, *frame1,* between the object and the world coordinate system is calculated. The frame will be the CAD frame expressed in the world coordinate system. If the input order of the targetlists is exchanged, i.e. DefAccFrame(pWCS, pCAD....) the world frame will be expressed in the CAD coordinate system.

Return value

DefAccFrame Data type: *pose*

The calculated frame.

Arguments

DefAccFrame (TargetListOne TargetListTwo TargetsInList MaxErr MeanErr)

TargetListOne Data type: *robtarget*

Array of robtargets holding the positions defined in coordinate system one. Minimum number of robtargets is 3, maximum 10

TargetListTwo Data type: *robtarget*

Array of robtargets holding the positions defined in coordinate system two. Minimum number of robtargets is 3, maximum 10.

TargetsInList Data type: *num*

Number of robtargets in array.

MaxErr Data type: *num*

The estimated maximum error in mm.

MeanErr Data type: *num*

The estimated mean error in mm.

Error handling

If the positions don't have the required relation or are not specified with enough accuracy, the system variable ERRNO is set to ERR_FRAME. This error can then be handled in an error handler.

Syntax

DefAccFrame'('
    [TargetListOne ':='] <array {*} (**IN**) of *robtarget*> ','
    [TargetListTwo ':='] <array {*} (**IN**) of *robtarget*> ','
    [TargetsInList ':='] <expression (**IN**) of *num*> ','
    [MaxErr ':='] <variable (**VAR**) of *num*> ','
    [MeanErr ':='] <variable (**VAR**) of *num*>')'

A function with a return value of the data type *pose*.

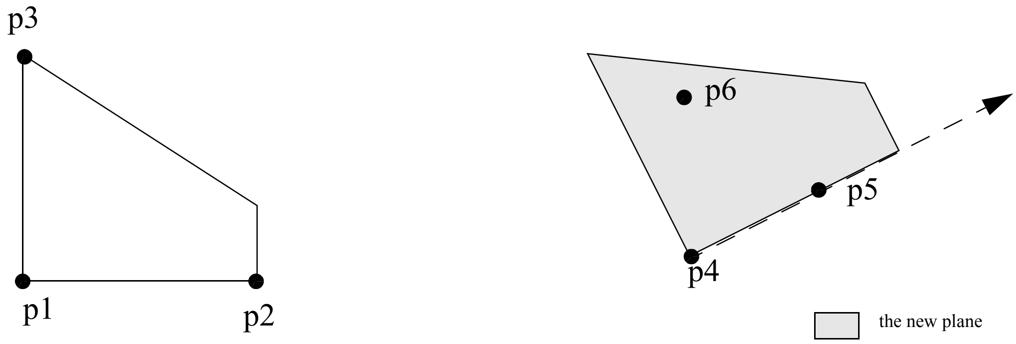
Related information

	Described in:
Calculating a frame from three positions	Functions - *DefFrame*
Calculate a displacement frame	Functions - *DefDFrame*

DefDFrame - Define a displacement frame

DefDFrame (Define Displacement Frame) is used to calculate a displacement frame from three original positions and three displaced positions.

Example




```
CONST robtarget p1 := [...];
CONST robtarget p2 := [...];
CONST robtarget p3 := [...];
VAR robtarget p4;
VAR robtarget p5;
VAR robtarget p6;
VAR pose frame1;
.
!Search for the new positions
SearchL sen1, p4, *, v50, tool1;
.
SearchL sen1, p5, *, v50, tool1;
.
SearchL sen1, p6, *, v50, tool1;
frame1 := DefDframe (p1, p2, p3, p4, p5, p6);
.
!activation of the displacement defined by frame1
PDispSet frame1;
```

Three positions *p1- p3*, related to an object in an original position, have been stored. After a displacement of the object, three new positions are searched for and stored as *p4-p6*. The displacement frame is calculated from these six positions. Then the calculated frame is used to displace all the stored positions in the program.

Return value Data type: *pose*

The displacement frame.

Arguments

DefDFrame (OldP1 OldP2 OldP3 NewP1 NewP2 NewP3)

OldP1 Data type: *robtarget*

The first original position.

OldP2 Data type: *robtarget*

The second original position.

OldP3 Data type: *robtarget*

The third original position.

NewP1 Data type: *robtarget*

The first displaced position. The difference between *OldP1* and *NewP1* will define the translation part of the frame and must be measured and determined with great accuracy.

NewP2 Data type: *robtarget*

The second displaced position. The line *NewP1* ... *NewP2* will define the rotation of the old line OldP1 ... OldP2.

NewP3 Data type: *robtarget*

The third displaced position. This position will define the rotation of the plane, e.g. it should be placed on the new plane of *NewP1*, *NewP2* and *NewP3*.

Error handling

If it is not possible to calculate the frame because of bad accuracy in the positions, the system variable ERRNO is set to ERR_FRAME. This error can then be handled in the error handler.

Syntax

DefDFrame'('
    [OldP1 ':='] <expression (**IN**) of *robtarget*> ','
    [OldP2 ':='] <expression (**IN**) of *robtarget*> ','
    [OldP3 ':='] <expression (**IN**) of *robtarget*> ','
    [NewP1 ':='] <expression (**IN**) of *robtarget*> ','
    [NewP2 ':='] <expression (**IN**) of *robtarget*> ','
    [NewP3 ':='] <expression (**IN**) of *robtarget*> ')'

A function with a return value of the data type *pose*.

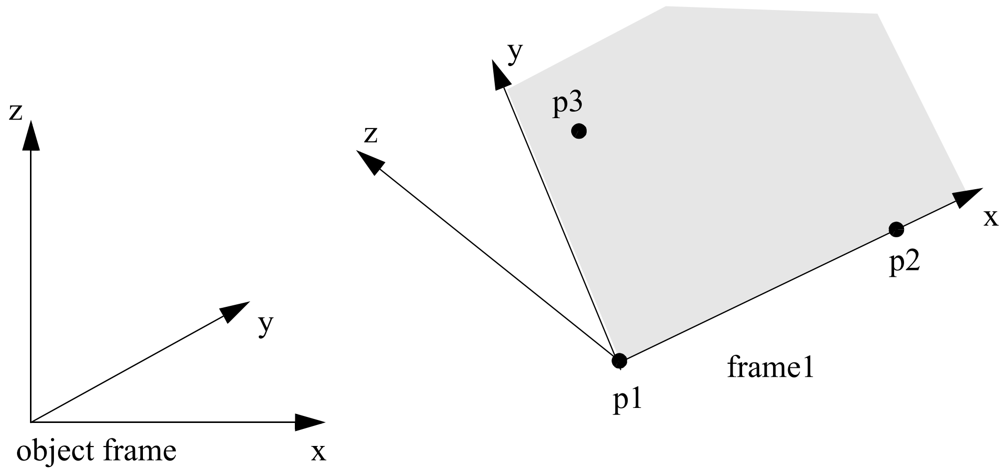
Related information

	Described in:
Activation of displacement frame	Instructions - *PDispSet*
Manual definition of displacement frame	User's Guide - *Calibration*

DefFrame - Define a frame

DefFrame (Define Frame) is used to calculate a frame, from three positions defining the frame.

Example



Three positions, *p1- p3*, related to the object coordinate system, are used to define the new coordinate system, *frame1*. The first position, *p1*, is defining the origin of *frame1*, the second position, *p2*, is defining the direction of the x-axis and the third position, *p3*, is defining the location of the xy-plane. The defined *frame1* may be used as a displacement frame, as shown in the example below:

```
CONST robtarget p1 := [...];
CONST robtarget p2 := [...];
CONST robtarget p3 := [...];
VAR pose frame1;
.
.
frame1 := DefFrame (p1, p2, p3);
.
.
!activation of the displacement defined by frame1
PDispSet frame1;
```

Return value Data type: *pose*

The calculated frame.

The calculation is related to the active object coordinate system.

Arguments

DefFrame (NewP1 NewP2 NewP3 [\Origin])

NewP1 Data type: *robtarget*

The first position, which will define the origin of the new frame.

NewP2 Data type: *robtarget*

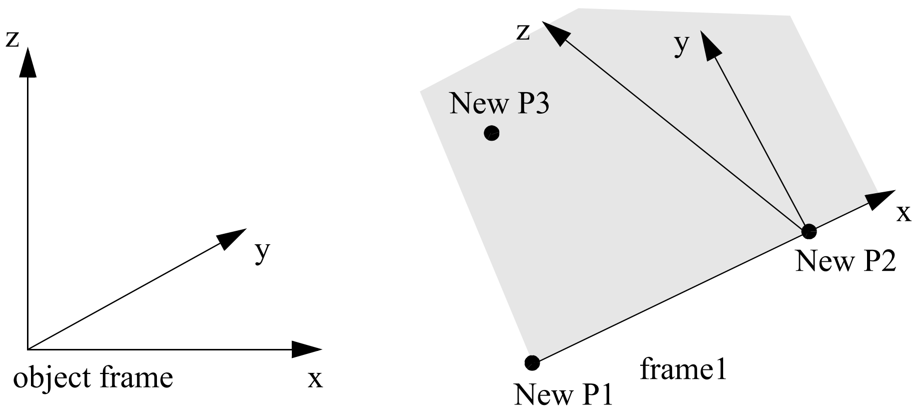
The second position, which will define the direction of the x-axis of the new frame.

NewP3 Data type: *robtarget*

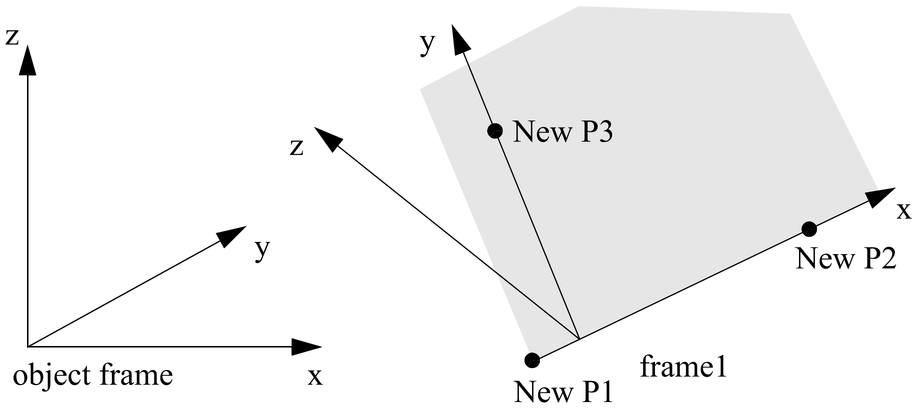
The third position, which will define the xy-plane of the new frame. The position of point 3 will be on the positive y side, see the figure above.

[**Origin**] Data type: *num*

Optional argument, which will define how the origin of the frame will be placed. Origin = 1, means that the origin is placed in NewP1, i.e. the same as if this argument is omitted. Origin = 2 means that the origin is placed in NewP2, see the figure below.



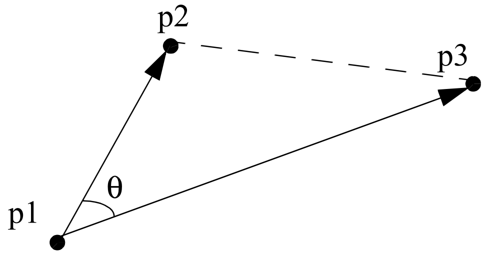
Origin = 3 means that the origin is placed on the line going through NewP1 and NewP2 and so that NewP3 will be placed on the y axis, see the figure below.



Other values, or if Origin is omitted, will place the origin in NewP1.

Limitations

The three positions *p1 - p3*, defining the frame, must define a well shaped triangle. The most well shaped triangle is the one with all sides of equal length.



The triangle is not considered to be well shaped if the angle θ a is too small. The angle θ is too small if:

$$|\cos\Theta| < 1 - 10^{-4}$$

The triangle *p1, p2, p3* must not be too small, i.e. the positions cannot be too close. The distances between the positions *p1 - p2* and *p1 - p3* must not be less than 0.1 mm.

Error handling

If the frame cannot be calculated because of the above limitations, the system variable ERRNO is set to ERR_FRAME. This error can then be handled in the error handler.

Syntax

DefFrame'('
    [NewP1 ':='] <expression (**IN**) of *robtarget*> ','
    [NewP2 ':='] <expression (**IN**) of *robtarget*> ','
    [NewP3 ':='] <expression (**IN**) of *robtarget*>
    ['\'Origin ':=' <expression (**IN**) of *num*>]')'

A function with a return value of the data type *pose*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Activation of displacement frame	Instructions - *PDispSet*

Dim - Obtains the size of an array

Dim (Dimension) is used to obtain the number of elements in an array.

Example

```
PROC arrmul(VAR num array{*}, num factor)

    FOR index FROM 1 TO Dim(array, 1) DO
        array{index} := array{index} * factor;
    ENDFOR

ENDPROC
```

All elements of a num array are multiplied by a factor.
This procedure can take any one-dimensional array of data type *num* as an input.

Return value Data type: *num*

The number of array elements of the specified dimension.

Arguments

Dim (ArrPar DimNo)

ArrPar *(Array Parameter)* Data type: Any type

The name of the array.

DimNo *(Dimension Number)* Data type: *num*

The desired array dimension: 1 = first dimension
2 = second dimension
3 = third dimension

Example

```
PROC add_matrix(VAR num array1{*,*,*}, num array2{*,*,*})

    IF Dim(array1,1) <> Dim(array2,1) OR Dim(array1,2) <> Dim(array2,2) OR
        Dim(array1,3) <> Dim(array2,3) THEN
        TPWrite "The size of the matrices are not the same";
        Stop;
    ELSE
        FOR i1 FROM 1 TO Dim(array1, 1) DO
```

```
            FOR i2 FROM 1 TO Dim(array1, 2) DO
                FOR i3 FROM 1 TO Dim(array1, 3) DO
                    array1{i1,i2,i3} := array1{i1,i2,i3} + array2{i1,i2,i3};
                ENDFOR
            ENDFOR
        ENDFOR
    ENDIF
    RETURN;

ENDPROC
```

Two matrices are added. If the size of the matrices differs, the program stops and an error message appears.
This procedure can take any three-dimensional arrays of data type *num* as an input.

Syntax

```
Dim '('
    [ArrPar':='] <reference (REF) of any type> ','
    [DimNo':='] <expression (IN) of num> ')'
```

A REF parameter requires that the corresponding argument be either a constant, a variable or an entire persistent. The argument could also be an IN parameter, a VAR parameter or an entire PERS parameter.

A function with a return value of the data type *num*.

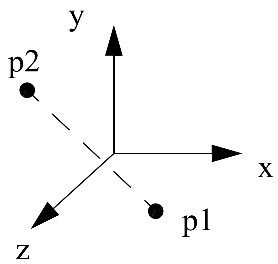
Related information

	Described in:
Array parameters	Basic Characteristics - *Routines*
Array declaration	Basic Characteristics - *Data*

Distance - Distance between two points

Distance is used to calculate the distance between two points in the space.

Example




```
VAR num dist;
CONST pos p1 := [4,0,4];
CONST pos p2 := [-4,4,4];
...
dist := Distance(p1, p2);
```

The distance in space between the points *p1* and *p2* is calculated and stored in the variable *dist*.

Return value Data type: *num*

The distance (always positive) between the points.

Arguments

Distance (Point1 Point2)

Point1 Data type: *pos*

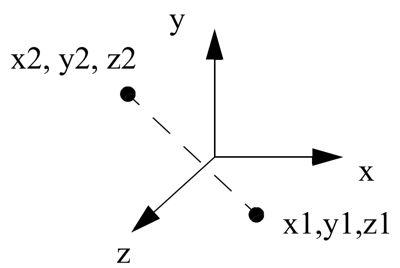
The first point described by the *pos* data type.

Point2 Data type: *pos*

The second point described by the *pos* data type.

Program execution

Calculation of the distance between the two points:



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

Syntax

Distance'('
    [Point1 ':='] <expression (**IN**) of *pos*> ','
    [Point2 ':='] <expression (**IN**) of *pos*>
    ')'

A function with a return value of the data type *num*.

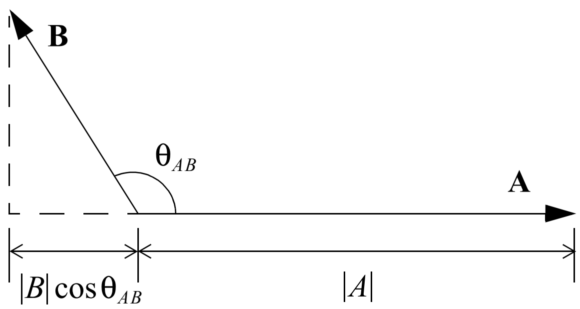
Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Definition of pos	Data Type - *pos*

DotProd - Dot product of two pos vectors

DotProd (Dot Product) is used to calculate the dot (or scalar) product of two pos vectors. The typical use is to calculate the projection of one vector upon the other or to calculate the angle between the two vectors.

Example



The dot or scalar product of two vectors **A** and **B** is a scalar, which equals the products of the magnitudes of **A** and **B** and the cosine of the angle between them.

$$A \cdot B = |A||B|\cos\theta_{AB}$$

The dot product:

- is less than or equal to the product of their magnitudes.
- can be either a positive or a negative quantity, depending whether the angle between them is smaller or larger then 90 degrees.
- is equal to the product of the magnitude of one vector and the projection of the other vector upon the first one.
- is zero when the vectors are perpendicular to each other.

The vectors are described by the data type *pos* and the dot product by the data type *num*:

```
VAR num dotprod;
VAR pos vector1;
VAR pos vector2;
.
.
vector1 := [1,1,1];
vector2 := [1,2,3];
dotprod := DotProd(vector1, vector2);
```

Return value Data type: *num*

The value of the dot product of the two vectors.

Arguments

DotProd (Vector1 Vector2)

Vector1 Data type: *pos*

The first vector described by the *pos* data type.

Vector2 Data type: *pos*

The second vector described by the *pos* data type.

Syntax

DotProd'('
[Vector1 ':='] <expression (**IN**) of *pos*> ','
[Vector2 ':='] <expression (**IN**) of *pos*>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

DOutput - Reads the value of a digital output signal

DOutput is used to read the current value of a digital output signal.

Example

IF DOutput(do2) = 1 THEN . . .

If the current value of the signal *do2* is equal to *1,* then . . .

Return value Data type: *dionum*

The current value of the signal (0 or 1).

Arguments

DOutput (Signal)

Signal Data type: *signaldo*

The name of the signal to be read.

Program execution

The value read depends on the configuration of the signal. If the signal is inverted in the system parameters, the value returned by this function is the opposite of the true value of the physical channel.

Example

IF DOutput(auto_on) <> active THEN . . .

If the current value of the system signal *auto_on* is *not active*, then ..., i.e. if the robot is in the manual operating mode, then ... Note that the signal must first be defined as a system output in the system parameters.

Syntax

DOutput '('
[Signal ':='] < variable (**VAR**) of *signaldo* > ')'

DOutput

Function

A function with a return value of the data type *dionum*.

Related information

	Described in:
Input/Output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	User's Guide - *System Parameters*

EulerZYX - Gets euler angles from orient

EulerZYX (Euler ZYX rotations) is used to get an Euler angle component from an orient type variable.

Example

```
VAR num anglex;
VAR num angley;
VAR num anglez;
VAR pose object;
.
.
anglex := EulerZYX(\X, object.rot);
angley := EulerZYX(\Y, object.rot);
anglez   := EulerZYX(\Z, object.rot);
```

Return value Data type: *num*

The corresponding Euler angle, expressed in degrees, range [-180, 180].

Arguments

EulerZYX ([\X] | [\Y] | [\Z] Rotation)

The arguments \X, \Y and \Z are mutually exclusive. If none of these are specified, a run-time error is generated.

[\X] Data type: *switch*

Gets the rotation around the X axis.

[**Y**] Data type: *switch*

Gets the rotation around the Y axis.

[**Z**] Data type: *switch*

Gets the rotation around the Z axis.

Rotation Data type: *orient*

The rotation in its quaternion representation.

EulerZYX

Function

Syntax

EulerZYX’(’
['\'X ’,’] | ['\'Y ’,’] | ['\'Z ’,’]
[Rotation ’:=’] <expression (**IN**) of *orient*>
’)’

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Exp - Calculates the exponential value

Exp (Exponential) is used to calculate the exponential value, ex.

Example

```
VAR num x;
VAR num value;
.
.
value:= Exp( x);
```

Return value Data type: _num_

The exponential value e^{x} .

Arguments

Exp (Exponent)

Exponent Data type: _num_

The exponent argument value.

Syntax

Exp'('
[Exponent ':='] <expression (**IN**) of _num_>
')'

A function with a return value of the data type _num_.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - _Mathematics_

FileTime - Retrieve time information about a file

FileTime is used to retrieve the last time for modification, access or file status change of a file. The time is measured in secs since 00:00:00 GMT, Jan. 1 1970. The time is returned as a num.

Example

```
Load "HOME:/notmymod.mod";
WHILE TRUE DO
    ! Call some routine in notmymod
    notmymodrout;
    IF FileTime("HOME:/notmymod.mod" \ModifyTime)
        > ModTime("notmymod") THEN
        UnLoad "HOME:notmymod.mod";
        Load "HOME:notmymod.mod";
    ENDIF
ENDWHILE
```

This program reloads a module if there is a newer at the source. It uses the *ModTime* to retrieve the latest loading time for the specified module, and to compare it to the *FileTime\ModifyTime* at the source. Then, if the source is newer, the program unloads and loads the module again.

Return value Data type: *num*

The time measured in secs since 00:00:00 GMT, Jan 1 1970.

Arguments

FileTime (Path [\ModifyTime] | [\AccessTime] | [\StatCTime])

Path Data type: *string*

The file specified with a full or relative path.

ModifyTime Data type: *switch*

Last modification time.

AccessTime Data type: *switch*

Time of last access (read, execute of modify).

StatCTime Data type: *switch*

Last file status (access qualification) change time.

Program execution

This function returns a numeric that specifies the time since the last:

- Modification
- Access
- File status change

of the specified file.

Example

This is a complete example that implements an alert service for maximum 10 files.

```
LOCAL RECORD falert
    string filename;
    num ftime;
ENDRECORD

LOCAL VAR falert myfiles[10];
LOCAL VAR num currentpos:=0;
LOCAL VAR intnum timeint;

LOCAL TRAP mytrap
    VAR num pos:=1;
    WHILE pos <= currentpos DO
        IF FileTime(myfiles{pos}.filename \ModifyTime) > myfiles{pos}.ftime THEN
            TPWrite "The file "+myfiles{pos}.filename+" is changed";
        ENDIF
        pos := pos+1;
    ENDWHILE
ENDTRAP

PROC alertInit(num freq)
    currentpos:=0;
    CONNECT timeint WITH mytrap;
    ITimer freq,timeint;
ENDPROC

PROC alertFree()
    IDelete timeint;
```

```
ENDPROC

PROC alertNew(string filename)
    currentpos := currentpos+1;
    IF currentpos <= 10 THEN
        myfiles{currentpos}.filename := filename;
        myfiles{currentpos}.ftime := FileTime (filename \ModifyTime);
    ENDIF
ENDPROC
```

Error handling

If the file does not exist, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Syntax

FileTime '('
    [Path ':='] < expression (**IN**) of *string*>
    ['\'ModifyTime] |
    ['\'AccessTime] |
    ['\'StatCTime] ')'

A function with a return value of the data type *num*.

Related information

	Described in:
Last time a module was loaded	Functions - *ModTime*

GetNextMechUnit - Get name of mechanical units

GetNextMechUnit is used for retrieving name of mechanical units in the robot system.

Examples

```
VAR num listno := 0;
VAR string name := "";

TPWrite "List of mechanical units:";
WHILE GetNextMechUnit(listno, name) DO
    TPWrite name;
    ! listno := listno + 1is done by GetNextMechUnit
ENDWHILE
```

The name of all mechanical units available in the system, will be displayed on the Teach Pendant.

Return Value

Data type: *bool*

TRUE if a mechanical unit was found, otherwise FALSE.

Arguments

GetNextMechUnit (ListNumber UnitName)

ListNumber Data type: *num*

This specifies which items in the list of mechanical units are to be retrieved. At return, this variable is always incremented by one to make it easy to access the next unit in the list.
The first mechanical unit in the list has index 0.

UnitName Data type: *string*

The name of the mechanical unit.

Example

```
VAR num listno := 4;
VAR string name := "";
VAR bool found := FALSE;

found := GetNextMechUnit (listno, name);
```

If *found* equal to TRUE, the name of mechanical unit number 4 will be in the variable *name,* else *name* contains only an empty string.

Syntax

GetNextMechUnit '('
[ListNumber':='] < variable (**VAR**) of *num*> ','
[UnitName':='] < variable (**VAR**) of *string*> ')'

A function with a return value of the data type *bool*.

Related information

	Described in:
Mechanical unit	Data Types - *mecunit*
Activating/Deactivating mechanical units	Instructions - *ActUnit, DeactUnit*
Configuration of mechanical units	User's Guide - *System Parameters*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

GetNextSym - Get next matching symbol

GetNextSym (Get Next Symbol) is used together with SetDataSearch to retrieve data objects from the system.

Example

```
VAR datapos block;
VAR string name;
VAR bool truevar:=TRUE;
...
SetDataSearch "bool" \Object:="^my" \InMod:="mymod"\LocalSym;
WHILE GetNextSym(name,block) DO
    SetDataVal name\Block:=block,truevar;
ENDWHILE
```

This session will set all local *bool* **data objects that begin with** *my* **in the module**
mymod **to TRUE**.

Return value Data type: *bool*

TRUE if a new object has been retrieved, the object name and its enclosed block is then returned in its arguments.

FALSE if no more objects match.

Arguments

GetNextSym Object Block [\Recursive]

Object Data type: *string*

Variable (VAR or PERS) to store the name of the data object that will be retrieved.

Block Data type: *datapos*

The enclosed block to the object.

[\Recursive] Data type: *switch*

This will force the search to enter the block below, e.g. if the search session has begun at the task level, it will also search modules and routines below the task.

Syntax

GetNextSym
[Object ':='] < variable or persistent (**INOUT**) of *string* > ','
[Block ':='] <variable (**VAR**) of *datapos*>
['\'Recursive] ';'

A function with a return value of the data type *bool*.

Related information

	Described in:
Define a symbol set in a search session	Instructions - *SetDataSearch*
Get the value of a data object	Instructions - G*etDataVal*
Set the value of a data object	Instructions - *SetDataVal*
Set the value of many data objects	Instructions - *SetAllDataVal*
The related data type datapos	Data Types - *datapos*

GetTaskName - Gets the name of current task

GetTaskName is used to get the identity of the current program task, with its name and number.

Example

```
VAR string mytaskname;
VAR num mytaskno;

mytaskname:=GetTaskName(\TaskNo:=mytaskno);
```

The current task name is stored in the variable *mytaskname*. The numerical identity of the task is stored in *mytaskno*.

Return value Data type: *string*

The name of the task in which the function is executed.

Arguments

GetTaskName ([\TaskNo])

[\TaskNo] Data type: *num*

The identity of the task represented as a numeric value. The numbers returned will be in the range 1-10 where 1 is the identity of the main task.

Syntax

GetTaskName'('
[\TaskNo ':='] < variable (**VAR**) of *num* > ')'

A function with a return value of the data type *string*.

Related information

	Described in:
Multitasking	RAPID Overview - *RAPID Summary Multitasking, Basic Characteristics Multitasking*.

GetTime - Reads the current time as a numeric value

GetTime is used to read a specified component of the current system time as a numeric value.

GetTime can be used to :

- have the program perform an action at a certain time
- perform certain activities on a weekday
- abstain from performing certain activities on the weekend
- respond to errors differently depending on the time of day.

Example

hour := GetTime(\Hour);

The current hour is stored in the variable *hour*.

Return value

Data type: *num*

One of the four time components specified below.

Argument

GetTime ([**WDay**] | [**Hour**] | [**Min**] | [**Sec**])

[\WDay] Data type: *switch*

Return the current weekday.
Range: 1 to 7 (Monday to Sunday).

[\Hour] Data type: *switch*

Return the current hour.
Range: 0 to 23.

[\Min] Data type: *switch*

Return the current minute.
Range: 0 to 59.

[\Sec] Data type: *switch*

Return the current second.
Range: 0 to 59.

One of the arguments must be specified, otherwise program execution stops with an error message.

Example

```
weekday := GetTime(\WDay);
hour := GetTime(\Hour);
IF weekday < 6 AND hour >6 AND hour < 16 THEN
                    production;
ELSE
                    maintenance;
ENDIF
```

If it is a weekday and the time is between 7:00 and 15:59 the robot performs production. At all other times, the robot is in the maintenance mode.

Syntax

```
GetTime '('
    ['\' WDay ]
    | [ '\' Hour ]
    | [ '\' Min ]
    | [ '\' Sec ] ')'
```

A function with a return value of the type *num*.

Related information

	Described in:
Time and date instructions	RAPID Summary - *System & Time*
Setting the system clock	User's Guide - *System Parameters*

GOutput - Reads the value of a group of digital output signals

G*Output* is used to read the current value of a group of digital output signals.

Example

IF GOutput(go2) = 5 THEN ...

If the current value of the signal *go2* is equal to *5,* then ...

Return value

Data type: *num*

The current value of the signal (a positive integer).

The values of each signal in the group are read and interpreted as an unsigned binary number. This binary number is then converted to an integer.

The value returned lies within a range that is dependent on the number of signals in the group.

No. of signals	Return value	No. of signals	Return value
1	0 - 1	9	0 - 511
2	0 - 3	10	0- 1023
3	0 - 7	11	0 - 2047
4	0 - 15	12	0 - 4095
5	0 - 31	13	0 - 8191
6	0 - 63	14	0 - 16383
7	0 - 127	15	0 - 32767
8	0 - 255	16	0 - 65535

Arguments

GOutput (Signal)

Signal Data type: *signalgo*

The name of the signal group to be read.

Syntax

GOutput '('
  [Signal ':='] < variable (**VAR**) of *signalgo* > ')'

A function with a return value of data type *num*.

Related information

	Described in:
Input/Output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	User's Guide - *System Parameters*

IsMechUnitActive - Is mechanical unit active

IsMechUnitActive (Is Mechanical Unit Active) is used to check whether a mechanical unit is activated or not.

Example

IF IsMechUnitActive(SpotWeldGun) CloseGun SpotWeldGun;

If the mechanical unit *SpotWeldGun* is active, the routine *CloseGun* will be invoked, where the gun is closed.

Return value Data type: *bool*

The function returns:

- TRUE, if the mechanical unit is active

- FALSE, if the mechanical unit is deactive

Arguments

IsMechUnitActive (MechUnit)

MechUnit *(Mechanical Unit)* Data type: *mecunit*

The name of the mechanical unit.

Syntax

IsMechUnitActive ’(’
[MechUnit ’:=’] < variable (VAR) of *mecunit***>**
’)’

A function with a return value of the data type *bool*.

Related information

	Described in:
Activating mechanical units	Instructions - *ActUnit*
Deactivating mechanical units	Instructions - *DeactUnit*
Mechanical units	Data Types - *mecunit*

IsPers - Is persistent

IsPers is used to test if a data object is a persistent variable or not.

Example

```
PROC procedure1 (INOUT num parameter1)
    IF IsVar(parameter1) THEN
        ! For this call reference to a variable
        ...
    ELSEIF IsPers(parameter1) THEN
        ! For this call reference to a persistent variable
        ...
    ELSE
        ! Should not happen
        EXIT;
    ENDIF
ENDPROC
```

The procedure *procedure1* will take different actions depending on whether the actual parameter *parameter1* is a variable or a persistent variable.

Return value Data type: *bool*

TRUE if the tested actual INOUT parameter is a persistent variable.
FALSE if the tested actual INOUT parameter is not a persistent variable.

Arguments

IsPers (DatObj)

DatObj (*Data Object*) Data type: any type

The name of the formal INOUT parameter.

Syntax

IsPers'('
    [DatObj ':='] < var or pers (**INOUT**) of any type > ')'

A function with a return value of the data type *bool*.

Related information

	Described in:
Test if variable	Function - *IsVar*
Types of parameters (access modes)	RAPID Characteristics - *Routines*

IsSysId - Test system identity

IsSysId (System Identity) can be used to test the system identity.

Example

```
IF NOT IsSysId("6400-1234") THEN
        ErrWrite "System identity fault","Faulty system identity for this program";
        EXIT;
ENDIF
```

The program is made for a special robot system and can't be used of another one.

Return value Data type: *bool*

TRUE = The system identity is the same as specified in the test.

FALSE = The system identity is not the same as specified in the test.

Arguments

IsSysId (SystemId)

SystemId Data type: *string*

The system identity.

Syntax

IsSysId '('
[SystemId':='] < expression (**IN**) of *string*> ')'

A function with a return value of the data type *bool.'*

IsVar - Is variable

IsVar is used to test whether a data object is a variable or not.

Example

```
PROC procedure1 (INOUT num parameter1)
   IF IsVAR(parameter1) THEN
      ! For this call reference to a variable
      ...
   ELSEIF IsPers(parameter1) THEN
      ! For this call reference to a persistent variable
      ...
   ELSE
      ! Should not happen
      EXIT;
   ENDIF
ENDPROC
```

The procedure *procedure1* will take different actions, depending on whether the actual parameter *parameter1* is a variable or a persistent variable.

Return value Data type: *bool*

TRUE if the tested actual INOUT parameter is a variable.
FALSE if the tested actual INOUT parameter is not a variable.

Arguments

IsVar (DatObj)

DatObj (*Data Object*) Data type: any type

The name of the formal INOUT parameter.

Syntax

IsVar'('
   [DatObj ':='] < var or pers (**INOUT**) of any type > ')'

A function with a return value of the data type *bool*.

Related information

	Described in:
Test if persistent	Function - *IsPers*
Types of parameters (access modes)	RAPID Characteristics - *Routines*

MaxRobSpeed - Maximum robot speed

MaxRobSpeed (Maximum Robot Speed) returns the maximum TCP speed for the used robot type.

Example

TPWrite “Max. TCP speed in mm/s for my robot = “ \Num:=MaxRobSpeed();

The message “*Max. TCP speed in mm/s for my robot = 5000*” is written on the Teach Pendant.

Return value Data type: *num*

Return the max. TCP speed in mm/s for the used robot type and normal pratical TCP values.

If use of extreme big TCP values in tool frame, create own speeddata with bigger TCP speed than returned by *MaxRobSpeed*.

Syntax

MaxRobSpeed ’(’ ’)’

A function with a return value of the data type *num*.

Related information

	Described in:
Definition of velocity	Data Types - *speeddata*
Definition of maximum velocity	Instructions - *VelSet*

MirPos - Mirroring of a position

MirPos (Mirror Position) is used to mirror the translation and rotation parts of a position.

Example

```
CONST robtarget p1;
VAR robtarget p2;
PERS wobjdata mirror;
.
.
p2 := MirPos(p1, mirror);
```

p1 is a robtarget storing a position of the robot and an orientation of the tool. This position is mirrored in the xy-plane of the frame defined by *mirror*, relative to the world coordinate system. The result is new robtarget data, which is stored in *p2*.

Return value Data type: *robtarget*

The new position which is the mirrored position of the input position.

Arguments

MirPos (Point MirPlane [\WObj] [\MirY])

Point Data type: *robtarget*

The input robot position. The orientation part of this position defines the current orientation of the tool coordinate system.

MirPlane *(Mirror Plane)* Data type: *wobjdata*

The work object data defining the mirror plane. The mirror plane is the xy-plane of the object frame defined in *MirPlane*. The location of the object frame is defined relative to the user frame, also defined in *MirPlane*, which in turn is defined relative to the world frame.

[\WObj] *(Work Object)* Data type: *wobjdata*

The work object data defining the object frame, and user frame, relative to which the input position, *Point*, is defined. If this argument is left out, the position is defined relative to the World coordinate system.
Note. If the position is created with a work object active, this work object must be referred to in the argument.

[\MirY] *(Mirror Y)* Data type: *switch*

If this switch is left out, which is the default rule, the tool frame will be mirrored as regards the x-axis and the z-axis. If the switch is specified, the tool frame will be mirrored as regards the y-axis and the z-axis.

Limitations

No recalculation is done of the robot configuration part of the input robtarget data.

Syntax

MirPos'('
  [Point ':='] < expression (**IN**) of *robtarget*>','
  [MirPlane ':='] <expression (**IN**) of *wobjdata*> ','
  ['\'WObj ':=' <expression (**IN**) of *wobjdata*>]
  ['\'MirY]')'

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

ModTime - Get time of load for a loaded module

ModTime (*Module Time*) is used to retrieve the time of loading a specified module. The module is specified by its name and must be in the task memory. The time is measured in secs since 00:00:00 GMT, Jan 1 1970. The time is returned as a num.

Example

```
MODULE mymod
    VAR num mytime;
    PROC printMyTime()
        mytime := ModTime("mymod");
        TPWrite "My time is "+NumToStr(mytime,0);
    ENDPROC
```

Return value Data type: *num*

The time measured in secs since 00:00:00 GMT, Jan 1 1970.

Arguments

ModTime (Object)

Object Data type: *string*

The name of the module.

Program execution

This function return a numeric that specify the time when the module was loaded.

Example

This is a complete example that implements an “update if newer” service.

```
MODULE updmod
    PROC callrout()
        Load "HOME:/mymod.mod";
        WHILE TRUE DO
            ! Call some routine in mymod
            mymodrout;
            IF FileTime("HOME:/mymod.mod" \ModifyTime)
            > ModTime("mymod") THEN
            UnLoad "HOME:/mymod.mod";
            Load "HOME:/mymod.mod";
            ENDIF
        ENDWHILE
    ENDPROC
ENDMODULE
```

This program reloads a module if there is a newer one at the source. It uses the *ModTime* to retrieve the latest loading time for the specified module, and compares it to the *FileTime\ModifyTime* at the source. Then, if the source is newer, the program unloads and loads the module again.

Syntax

ModTime ’(’
  [Object ’:=’] < expression (**IN**) of *string*>’)’

A function with a return value of the data type *num*.

Related information

	Described in:
Retrieve time info. about a file	Functions - *FileTime*

NOrient - Normalise orientation

NOrient (Normalise Orientation) is used to normalise unnormalised orientation (quaternion).

Description

An orientation must be normalised, i.e. the sum of the squares must equal 1:

$$q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1$$

If the orientation is slightly unnormalised, it is possible to normalise it.
The normalisation error is the absolute value of the sum of the squares of the orientation components.
The orientation is considered to be slightly unnormalised if the normalisation error is greater then 0.00001 and less then 0.1. If the normalisation error is greater then 0.1 the orient is unusable.

$$ABS(\sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} - 1) = normerr$$

normerr > 0.1	Unusable
normerr > 0.00001 AND err <= 0.1	Slightly unnormalised
normerr <= 0.00001	Normalised

Example

We have a slightly unnormalised position (0.707170, 0, 0, 0.707170)

$$ABS(\sqrt{0.707170^2 + 0^2 + 0^2 + 0.707170^2} - 1) = 0.0000894$$

$$0.0000894 > 0.00001 \Rightarrow unnormalized$$

```
VAR orient unnormorient := [0.707170, 0, 0, 0.707170];
VAR orient normorient;
.
.
normorient := NOrient(unnormorient);
```

The normalisation of the orientation (0.707170, 0, 0, 0.707170) becomes (0.707107, 0, 0, 0.707107).

Return value Data type: *orient*

The normalised orientation.

Arguments

NOrient (Rotation)

Orient Data type: *orient*

The orientation to be normalised.

Syntax

NOrient'('
[Rotation ':='] <expression (**IN**) of *orient*>
')'

A function with a return value of the data type *orient*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

NumToStr - Converts numeric value to string

NumToStr (Numeric To String) is used to convert a numeric value to a string.

Example

```
VAR string str;

str := NumToStr(0.38521,3);
```

The variable *str* is given the value "0.385".

```
reg1 := 0.38521

str := NumToStr(reg1, 2\Exp);
```

The variable *str* is given the value "3.85E-01".

Return value Data type: *string*

The numeric value converted to a string with the specified number of decimals, with exponent if so requested. The numeric value is rounded if necessary. The decimal point is suppressed if no decimals are included.

Arguments

NumToStr (Val Dec [\Exp])

Val (*Value*) Data type: *num*

The numeric value to be converted.

Dec (*Decimals*) Data type: *num*

Number of decimals. The number of decimals must not be negative or greater than the available precision for numeric values.

[\Exp] (*Exponent*) Data type: *switch*

To use exponent.

Syntax

NumToStr'('
    [Val ':='] <expression (**IN**) of *num*> ','
    [Dec ':='] <expression (**IN**) of *num*>
    [\Exp]
    ')'

A function with a return value of the data type *string*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

Offs - Displaces a robot position

Offs is used to add an offset to a robot position.

Examples

MoveL Offs(p2, 0, 0, 10), v1000, z50, tool1;

The robot is moved to a point *10* mm from the position *p2* (in the z-direction).

p1 := Offs (p1, 5, 10, 15);

The robot position *p1* is displaced *5* mm in the x-direction, *10* mm in the y-direction and *15* mm in the z-direction.

Return value Data type: *robtarget*

The displaced position data.

Arguments

Offs (Point XOffset YOffset ZOffset)

Point Data type: *robtarget*

The position data to be displaced.

XOffset Data type: *num*

The displacement in the x-direction.

YOffset Data type: *num*

The displacement in the y-direction.

ZOffset Data type: *num*

The displacement in the z-direction.

Example

PROC pallet (num row, num column, num distance, PERS tooldata tool, PERS wobjdata wobj)

```
    VAR robtarget palletpos:=[[0, 0, 0], [1, 0, 0, 0], [0, 0, 0, 0],
[9E9, 9E9, 9E9, 9E9, 9E9, 9E9]];

    palettpos := Offs (palettpos, (row-1)*distance, (column-1)*distance, 0);
    MoveL palettpos, v100, fine, tool\WObj:=wobj;

ENDPROC
```

A routine for picking parts from a pallet is made. Each pallet is defined as a work object (see Figure 4). The part to be picked (row and column) and the distance between the parts are given as input parameters.
Incrementing the row and column index is performed outside the routine.

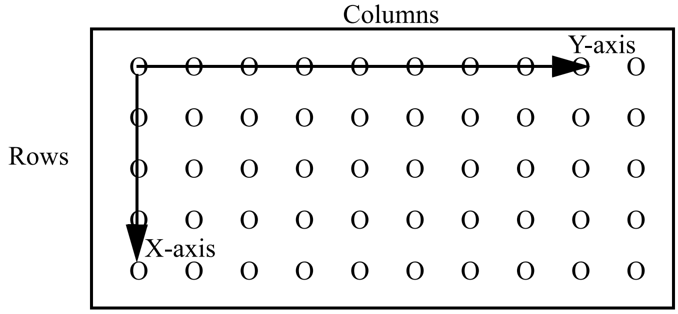


Figure 4 The position and orientation of the pallet is specified by defining a work object.

Syntax

Offs '('
    [Point ':='] <expression (**IN**) of *robtarget*> ','
    [XOffset ':='] <expression (**IN**) of *num*> ','
    [YOffset ':='] <expression (**IN**) of *num*> ','
    [ZOffset ':='] <expression (**IN**) of *num*> ')'

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Position data	Data Types - *robtarget*

OpMode - Read the operating mode

OpMode (Operating Mode) is used to read the current operating mode of the system.

Example

```
TEST OpMode()
CASE OP_AUTO:
                         ...
CASE OP_MAN_PROG:
                         ...
CASE OP_MAN_TEST:
                         ...
DEFAULT:
                         ...
ENDTEST
```

Different program sections are executed depending on the current operating mode.

Return value Data type: *symnum*

The current operating mode as defined in the table below.

Return value	Symbolic constant	Comment
0	OP_UNDEF	Undefined operating mode
1	OP_AUTO	Automatic operating mode
2	OP_MAN_PROG	Manual operating mode max. 250 mm/s
3	OP_MAN_TEST	Manual operating mode full speed, 100 %

Syntax

OpMode'(' ')'

A function with a return value of the data type *symnum*.

Related information

	Described in:
Different operating modes	User's Guide - *Starting up*
Reading running mode	Functions - *RunMode*

OrientZYX - Builds an orient from euler angles

OrientZYX (Orient from Euler ZYX angles) is used to build an orient type variable out of Euler angles.

Example

```
VAR num anglex;
VAR num angley;
VAR num anglez;
VAR pose object;
.
object.rot := OrientZYX(anglez, angley, anglex)
```

Return value Data type: *orient*

The orientation made from the Euler angles.

The rotations will be performed in the following order:
-rotation around the z axis,
-rotation around the new y axis
-rotation around the new x axis.

Arguments

OrientZYX (ZAngle YAngle XAngle)

ZAngle Data type: *num*

The rotation, in degrees, around the Z axis.

YAngle Data type: *num*

The rotation, in degrees, around the Y axis.

XAngle Data type: *num*

The rotation, in degrees, around the X axis.

The rotations will be performed in the following order:
-rotation around the z axis,
-rotation around the new y axis
-rotation around the new x axis.

Syntax

OrientZYX'('
    [ZAngle ':='] <expression (**IN**) of *num*> ','
    [YAngle ':='] <expression (**IN**) of *num*> ','
    [XAngle ':='] <expression (**IN**) of *num*>
    ')'

A function with a return value of the data type *orient*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

ORobT - Removes a program displacement from a position

ORobT (Object Robot Target) is used to transform a robot position from the program displacement coordinate system to the object coordinate system and/or to remove an offset for the external axes.

Example

```
VAR robtarget p10;
VAR robtarget p11;

p10 := CRobT();
p11 := ORobT(p10);
```

The current positions of the robot and the external axes are stored in *p10* and *p11*. The values stored in *p10* are related to the ProgDisp/ExtOffs coordinate system. The values stored in *p11* are related to the object coordinate system without any offset on the external axes.

Return value

Data type: *robtarget*

The transformed position data.

Arguments

ORobT (OrgPoint [\InPDisp] | [\InEOffs])

OrgPoint *(Original Point)* Data type: *robtarget*

The original point to be transformed.

[\InPDisp] *(In Program Displacement)* Data type: *switch*

Returns the TCP position in the ProgDisp coordinate system, i.e. removes external axes offset only.

[\InEOffs] *(In External Offset)* Data type: *switch*

Returns the external axes in the offset coordinate system, i.e. removes program displacement for the robot only.

Examples

p10 := ORobT(p10 \InEOffs);

The ORobT function will remove any program displacement that is active, leaving the TCP position relative to the object coordinate system. The external axes will remain in the offset coordinate system.

p10 := ORobT(p10 \InPDisp);

The ORobT function will remove any offset of the external axes. The TCP position will remain in the ProgDisp coordinate system.

Syntax

ORobT '('
    [OrgPoint ':='] < expression (**IN**) of *robtarget*>
    ['\'InPDisp] | ['\'InEOffs]')'

A function with a return value of the data type *robtarget*.

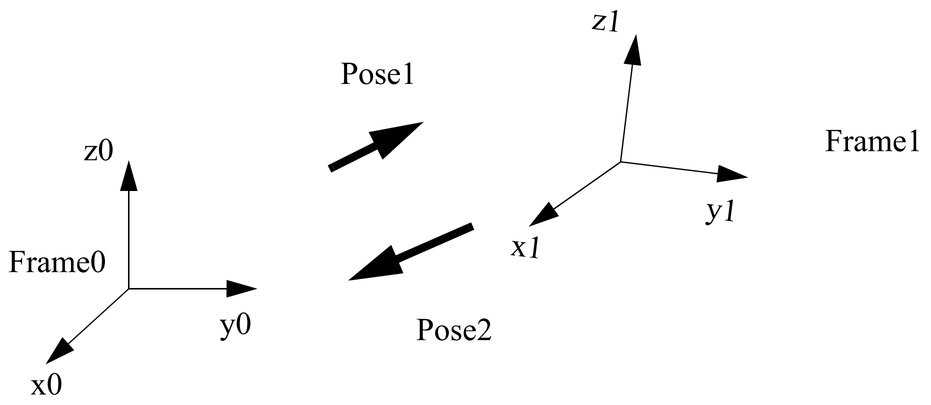
Related information

	Described in:
Definition of program displacement for the robot	Instructions - *PDispOn*, *PDispSet*
Definition of offset for external axes	Instructions - *EOffsOn*, *EOffsSet*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

PoseInv - Inverts the pose

PoseInv (Pose Invert) calculates the reverse transformation of a pose.

Example



Pose1 represents the coordinates of Frame1 related to Frame0.
The transformation giving the coordinates of Frame0 related to Frame1 is obtained by the reverse transformation:

```
VAR pose pose1;
VAR pose pose2;
.
.
pose2 := PoseInv(pose1);
```

Return value Data type: *pose*

The value of the reverse pose.

Arguments

PoseInv (Pose)

Pose Data type: *pose*

The pose to invert.

Syntax

PoseInv'('
[Pose ':='] <expression (**IN**) of *pose*>
')'

A function with a return value of the data type *pose*.

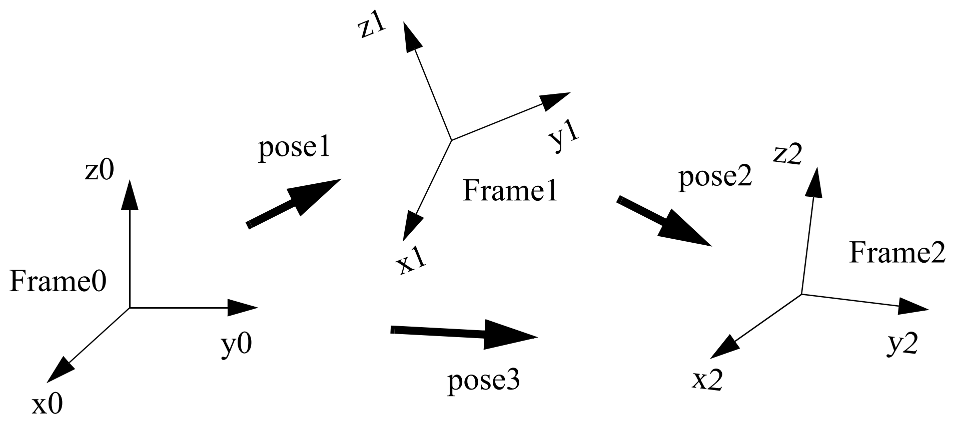
Related information

	<u>Described in:</u>
Mathematical instructions and functions	RAPID Summary - *Mathematics*

PoseMult - Multiplies pose data

PoseMult (Pose Multiply) is used to calculate the product of two frame transformations. A typical use is to calculate a new frame as the result of a displacement acting on an original frame.

Example



pose1 represents the coordinates of Frame1 related to Frame0.
pose2 represents the coordinates of Frame2 related to Frame1.

The transformation giving pose3, the coordinates of Frame2 related to Frame0, is obtained by the product of the two transformations:

```
VAR pose pose1;
VAR pose pose2;
VAR pose pose3;
.
.
pose3 := PoseMult(pose1, pose2);
```

Return value Data type: *pose*

The value of the product of the two poses.

Arguments

PoseMult (Pose1 Pose2)

Pose1 Data type: *pose*

The first pose.

Pose2 Data type: *pose*

The second pose.

Syntax

PoseMult'('
[Pose1 ':='] <expression (**IN**) of *pose*> ','
[Pose2 ':='] <expression (**IN**) of *pose*>
')'

A function with a return value of the data type *pose*.

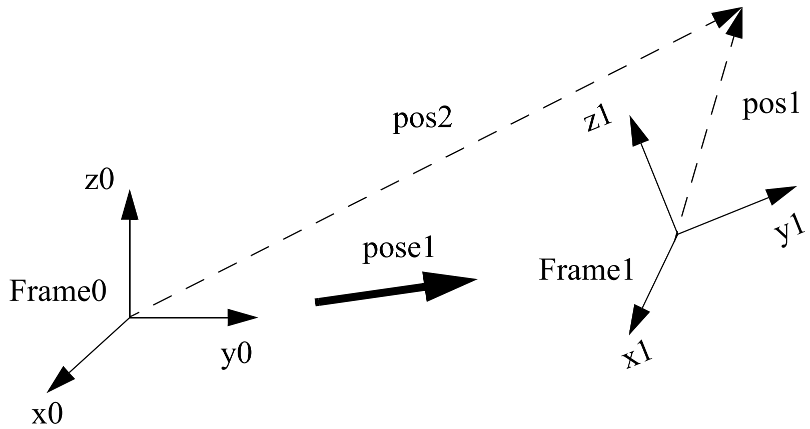
Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

PoseVect - Applies a transformation to a vector

PoseVect (Pose Vector) is used to calculate the product of a pose and a vector.
It is typically used to calculate a vector as the result of the effect of a displacement on an original vector.

Example



pose1 represents the coordinates of Frame1 related to Frame0.
pos1 is a vector related to Frame1.

The corresponding vector related to Frame0 is obtained by the product:

```
VAR pose pose1;
VAR pos pos1;
VAR pos pos2;
.
.
pos2:= PoseVect(pose1, pos1);
```

Return value Data type: *pos*

The value of the product of the pose and the original pos.

Arguments

PoseVect (Pose Pos)

Pose Data type: *pose*

The transformation to be applied.

Pos Data type: *pos*

The pos to be transformed.

Syntax

PoseVect'('
[Pose ':='] <expression (**IN**) of *pose*> ','
[Pos ':='] <expression (**IN**) of *pos*>
')'

A function with a return value of the data type *pos*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Pow - Calculates the power of a value

Pow (Power) is used to calculate the exponential value in any base.

Example

VAR num x;
VAR num y
VAR num reg1;
.
reg1:= Pow(x, y);

reg1 is assigned the value *x*y.

Return value Data type: *num*

The value of the base x raised to the power of the exponent y (xy).

Arguments

Pow (Base Exponent)

Base Data type: *num*

The base argument value.

Exponent Data type: *num*

The exponent argument value.

Limitations

The execution of the function x^y will give an error if:

. x < 0 and y is not an integer;
. x = 0 and y < 0.

Syntax

Pow'('
[Base ':='] <expression (**IN**) of *num*> ','
[Exponent ':='] <expression (**IN**) of *num*>
')'

Pow

Function

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Present - Tests if an optional parameter is used

Present is used to test if an optional argument has been used when calling a routine.

An optional parameter may not be used if it was not specified when calling the routine. This function can be used to test if a parameter has been specified, in order to prevent errors from occurring.

Example

```
PROC feeder (\switch on | \switch off)

    IF Present (on) Set do1;
    IF Present (off) Reset do1;

ENDPROC
```

The output *do1*, which controls a feeder, is set or reset depending on the argument used when calling the routine.

Return value Data type: *bool*

TRUE = The parameter value or a switch has been defined when calling the routine.

FALSE = The parameter value or a switch has not been defined.

Arguments

Present (OptPar)

OptPar *(Optional Parameter)* Data type: Any type

The name of the optional parameter to be tested.

Example

```
PROC glue (\switch on, num glueflow, robtarget topoint, speeddata speed,
zonedata zone, PERS tooldata tool, \PERS wobjdata wobj)

    IF Present (on) PulseDO glue_on;
    SetAO gluesignal, glueflow;
    IF Present (wobj) THEN
        MoveL topoint, speed, zone, tool \WObj=wobj;
    ELSE
        MoveL topoint, speed, zone, tool;
    ENDIF

ENDPROC
```

A glue routine is made. If the argument *\on* is specified when calling the routine, a pulse is generated on the signal *glue_on*. The robot then sets an analog output *gluesignal*, which controls the glue gun, and moves to the end position. As the wobj parameter is optional, different MoveL instructions are used depending on whether this argument is used or not.

Syntax

Present '('
    [OptPar':='] <reference (**REF**) of any type> ')'

A REF parameter requires, in this case, the optional parameter name.

A function with a return value of the data type *bool*.

Related information

	Described in:
Routine parameters	Basic Characteristics - *Routines*

ReadBin - Reads a byte from a file or serial channel

ReadBin (Read Binary) is used to read a byte (8 bits) from a file or serial channel.

This function works on both binary and character-based files or serial channels.

Example

```
VAR num character;
VAR iodev inchannel;
...
Open "com2:", inchannel\Bin;
character := ReadBin(inchannel);
```

A byte is read from the binary serial channel *inchannel*.

Return value

Data type: *num*

A byte (8 bits) is read from a specified file or serial channel. This byte is converted to the corresponding positive numeric value and returned as a *num* data type.
If a file is empty (end of file), the number -1 is returned.

Arguments

ReadBin (IODevice [\Time])

IODevice Data type: *iodev*

The name (reference) of the file or serial channel to be read.

[\Time] Data type: *num*

The max. time for the reading operation (timeout) in seconds. If this argument is not specified, the max. time is set to 60 seconds.

If this time runs out before the reading operation is finished, the error handler will be called with the error code ERR_DEV_MAXTIME. If there is no error handler, the execution will be stopped.

The timeout function is in use also during program stop and will be noticed in the RAPID program at program start.

Program execution

Program execution waits until a byte (8 bits) can be read from the file or serial channel.

Example

```
VAR num bindata;
VAR iodev file;

Open "HOME:/myfile.bin", file \Read \Bin;
bindata := ReadBin(file);
WHILE bindata <> EOF_BIN DO
    TPWrite ByteToStr(bindata\Char);
    bindata := ReadBin(file);
ENDWHILE
```

Read the contents of a binary file *myfile.bin* from the beginning to the end and displays the received binary data converted to chars on the teach pendant (one char on each line).

Limitations

The function can only be used for files and serial channels that have been opened with read access (*\Read* for character based files, *\Bin* or *\Append \Bin* for binary files).

Error handling

If an error occurs during reading, the system variable ERRNO is set to ERR_FILEACC.

If time out before the read operation is finished, the system variable ERRNO is set to ERR_DEV_MAXTIME.

These errors can then be dealt with by the error handler.

Predefined data

The constant *EOF_BIN* can be used to stop reading at the end of the file.

CONST num EOF_BIN := -1;

Syntax

ReadBin'('
    [IODevice ':='] <variable (**VAR**) of *iodev*>
    ['\'Time':=' <expression (**IN**) of *num*>]')'

A function with a return value of the type *num*.

Related information

	Described in:
Opening (etc.) files or serial channels	RAPID Summary - *Communication*
Convert a byte to a string data	Functions - *ByteToStr*

ReadMotor - Reads the current motor angles

ReadMotor is used to read the current angles of the different motors of the robot and external axes. The primary use of this function is in the calibration procedure of the robot.

Example

VAR num motor_angle2;

motor_angle2 **:= ReadMotor(2);**

The current motor angle of the second axis of the robot is stored in *motor_angle2*.

Return value Data type: *num*

The current motor angle in radians of the stated axis of the robot or external axes.

Arguments

ReadMotor [\MecUnit] Axis

MecUnit *(Mechanical Unit)* Data type: *mecunit*

The name of the mechanical unit for which an axis is to be read. If this argument is omitted, the axis for the robot is read. (Note, in this release only robot is permitted for this argument).

Axis Data type: *num*

The number of the axis to be read (1 - 6).

Program execution

The motor angle returned represents the current position in radians for the motor and independently of any calibration offset. The value is not related to a fix position of the robot, only to the resolver internal zero position, i.e. normally the resolver zero position closest to the calibration position (the difference between the resolver zero position and the calibration position is the calibration offset value). The value represents the full movement of each axis, although this may be several turns.

Example

VAR num motor_angle3;

motor_angle3 **:= ReadMotor(\MecUnit:=robot, 3)**;

The current motor angle of the third axis of the robot is stored in *motor_angle3*.

Syntax

ReadMotor'('
['\'MecUnit ':=' < variable (VAR) of *mecunit*>',']
[Axis ':='] < expression (IN) of num>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Reading the current joint angle	Functions - *CJointT*

ReadNum - Reads a number from a file or serial channel

ReadNum (Read Numeric) is used to read a number from a character-based file or serial channel.

Example

```
VAR iodev infile;
...
Open "HOME:/file.doc", infile\Read;
reg1 := ReadNum(infile);
```

Reg1 is assigned a number read from the file *file.doc*.

Return value

Data type: *num*

The numeric value read from a specified file or serial channel.
If the file is empty (end of file), the number 9.999E36 is returned.

Arguments

ReadNum (IODevice [\Delim] [\Time])

IODevice Data type: *iodev*

The name (reference) of the file or serial channel to be read.

[\Delim] (*Delimiters*) Data type: *string*

A string containing the delimiters to use when parsing a line in the file or serial channel. By default (without *Delim*), the file is read line by line and the line-feed character (\0A) is the only delimiter considered by the parsing. When the *Delim* argument is used, any character in the specified string argument will be considered to determine the significant part of the line.

When using the argument *Delim*, the control system always adds the characters carriage return (\0D) and line-feed (\0A) to the delimiters specified by the user.

To specify non-alphanumeric characters, use \xx, where xx is the hexadecimal representation of the ASCII code of the character (example: TAB is specified by \09).

[\Time] Data type: *num*

The max. time for the reading operation (timeout) in seconds. If this argument is not specified, the max. time is set to 60 seconds.

If this time runs out before the read operation is finished, the error handler will be called with the error code ERR_DEV_MAXTIME. If there is no error handler, the execution will be stopped.

The timeout function is also in use during program stop and will be noticed in the RAPID program at program start.

Program execution

Starting at the current file position, the function reads and discards any heading delimiters. A heading delimiter without the argument *\Delim* is a line-feed character. Heading delimiters with the argument *\Delim* are any characters specified in the *\Delim* argument plus carriage return and line-feed characters. It then reads everything up to and including the next delimiter character (will be discarded), but not more than 80 characters. If the significant part exceeds 80 characters, the remainder of the characters will be read on the next reading.

The string that is read is then converted to a numeric value; e.g. “234.4” is converted to the numeric value 234.4.

Example

```
reg1 := ReadNum(infile\Delim:="\09  ");
IF reg1 > EOF_NUM THEN
   TPWrite "The file is empty";
   ...
```

Reads a number in a line where numbers are separated by TAB (“\09”) or SPACE (“ “) characters.
Before using the number read from the file, a check is performed to make sure that the file is not empty.

Limitations

The function can only be used for character based files that have been opened for reading.

Error handling

If an access error occurs during reading, the system variable ERRNO is set to ERR_FILEACC.

If there is an attempt to read non-numeric data, the system variable ERRNO is set to ERR_RCVDATA.

If time out before the read operation is finished, the system variable ERRNO is set to ERR_DEV_MAXTIME.

These errors can then be dealt with by the error handler.

Predefined data

The constant *EOF_NUM* can be used to stop reading, at the end of the file.

CONST num EOF_NUM := 9.998E36;

Syntax

ReadNum ’(’
[IODevice ’:=’]<variable (**VAR**) of *iodev*>
[‘\’Delim’:=’<expression (**IN**) of *string*>]
[’\’Time’:=’<expression (**IN**) of *num*>]’)’

A function with a return value of the type *num*.

Related information

	Described in:
Opening (etc.) files or serial channels	RAPID Summary - *Communication*

ReadStr - Reads a string from a file or serial channel

ReadStr (Read String) is used to read a string from a character-based file or serial channel.

Example

```
VAR string text;
VAR iodev infile;
...
Open "HOME:/file.doc", infile\Read;
text := ReadStr(infile);
```

Text is assigned a string read from the file *file.doc*.

Return value

Data type: *string*

The string read from the specified file or serial channel.
If the file is empty (end of file), the string "EOF" is returned.

Arguments

ReadStr (IODevice [\Delim] [\RemoveCR] [\DiscardHeaders] [\Time])

IODevice Data type: *iodev*

The name (reference) of the file or serial channel to be read.

[\Delim] (*Delimiters*) Data type: *string*

A string containing the delimiters to use when parsing a line in the file or serial channel. By default the file is read line by line and the line-feed character (\0A) is the only delimiter considered by the parsing. When the *\Delim* argument is used, any character in the specified string argument plus by default line-feed character will be considered to determine the significant part of the line.

To specify non-alphanumeric characters, use \xx, where xx is the hexadecimal representation of the ASCII code of the character (example: TAB is specified by \09).

[\RemoveCR] Data type: *switch*

A switch used to remove the trailing carriage return character when reading PC files. In PC files, a new line is specified by carriage return and line feed (CRLF). When reading a line in such files, the carriage return character is by default read into the return string. When using this argument, the carriage return character will be read from the file but not included in the return string.

[\DiscardHeaders] Data type: *switch*

This argument specifies whether the heading delimiters (specified in *Delim* plus default line-feed) are skipped or not before transferring data to the return string. By default, if the first character at the current file position is a delimiter, it is read but not transferred to the return string, the line parsing is stopped and the return will be an empty string. If this argument is used, all delimiters included in the line will be read from the file but discarded, and the return string will contain the data starting at the first non-delimiter character in the line.

[\Time] Data type: *num*

The max. time for the reading operation (timeout) in seconds. If this argument is not specified, the max. time is set to 60 seconds.

If this time runs out before the read operation is finished, the error handler will be called with the error code ERR_DEV_MAXTIME. If there is no error handler, the execution will be stopped.

The timeout function is in use also during program stop and will be noticed in the RAPID program at program start.

Program execution

Starting at the current file position, if the *DiscardHeaders* argument is used, the function reads and discards any heading delimiters (line-feed characters and any character specified in the *Delim* argument). In all cases, it then reads everything up to the next delimiter character, but not more than 80 characters. If the significant part exceeds 80 characters, the remainder of the characters will be read on the next reading. The delimiter that caused the parsing to stop is read from the file but not transferred to the return string. If the last character in the string is a carriage return character and the *RemoveCR* argument is used, this character will be removed from the string.

Example 1

```
text := ReadStr(infile);
IF text = EOF THEN
    TPWrite "The file is empty";
    ...
```

Before using the string read from the file, a check is performed to make sure that the file is not empty.

Example 2

Consider a file containing:
<LF><SPACE><TAB>Hello<SPACE><SPACE>World<CR><LF>

```
text := ReadStr(infile);
```

text will be an empty string: the first character in the file is the default <LF> delimiter.

```
text := ReadStr(infile\DiscardHeaders);
```

text will contain <SPACE><TAB>Hello<SPACE><SPACE>World<CR>: the first character in the file, the default <LF> delimiter, is discarded.

```
text := ReadStr(infile\RemoveCR\DiscardHeaders);
```

text will contain <SPACE><TAB>Hello<SPACE><SPACE>World: the first character in the file, the default <LF> delimiter, is discarded; the final carriage return character is removed

```
text := ReadStr(infile\Delim:=” \09”\RemoveCR\DiscardHeaders);
```

text will contain “Hello”: the first characters in the file that match either the default <LF> delimiter or the character set defined by *\Delim* (space and tab) are discarded. Data is then transferred up to the first delimiter that is read from the file but not transferred into the string. A new invocation of the same statement will return “World”.

Example 3

Consider a file containing:
<CR><LF>Hello<CR><LF>

text := ReadStr(infile);

text will contain the <CR> (\0d) character: <CR> and <LF> characters are read from the file, but only <CR> is transferred to the string. A new invocation of the same statement will return “Hello\0d”.

text := ReadStr(infile\RemoveCR);

text will contain an empty string: <CR> and <LF> characters are read from the file; <CR> is transferred but removed from the string. A new invocation of the same statement will return “Hello”.

text := ReadStr(infile\Delim:=”\0d”);

text will contain an empty string: <CR> is read from the file but not transferred to the return string. A new invocation of the same instruction will return an empty string again: <LF> is read from the file but not transferred to the return string.

text := ReadStr(infile\Delim:=”\0d”\DiscardHeaders);

text will contain “Hello”. A new invocation of the same instruction will return “EOF” (end of file).

Limitations

The function can only be used for files or serial channels that have been opened for reading in a character-based mode.

Error handling

If an error occurs during reading, the system variable ERRNO is set to ERR_FILEACC.

If timeout before the read operation is finished, the system variable ERRNO is set to ERR_DEV_MAXTIME.

These errors can then be dealt with by the error handler.

Predefined data

The constant *EOF* can be used to check if the file was empty when trying to read from the file or to stop reading at the end of the file.

CONST string EOF := "EOF";

Syntax

ReadStr ’(’
[IODevice ’:=’] <variable (**VAR**) of *iodev*>
[‘\’Delim’:=’<expression (**IN**) of *string*>]
[‘\’RemoveCR]
[‘\’DiscardHeaders]
[’\’Time’:=’ <expression (**IN**) of *num*>]’)’

A function with a return value of the type *string*.

Related information

	Described in:
Opening (etc.) files or serial channels	RAPID Summary - *Communication*

ReadStrBin - Reads a string from a binary serial channel or file

ReadStrBin (Read String Binary) is used to read a string from a binary serial channel or file.

Example

```
VAR iodev channel2;
VAR string text;
...
Open “com2:”, channel2 \Bin;
text := ReadStrBin (channel2, 10);
```

Text is assigned a *10* characters text string read from the serial channel referred to by *channel2*.

Return value

Data type: *string*

The text string read from the specified serial channel or file. If the file is empty (end of file), the string "EOF" is returned.

Arguments

ReadStrBin (IODevice NoOfChars [\Time])

IODevice Data type: *iodev*

The name (reference) of the binary serial channel or file to be read.

NoOfChars Data type: *num*

The number of characters to be read from the binary serial channel or file.

[\Time] Data type: *num*

The max. time for the reading operation (timeout) in seconds. If this argument is not specified, the max. time is set to 60 seconds.

If this time runs out before the read operation is finished, the error handler will be called with the error code ERR_DEV_MAXTIME. If there is no error handler, the execution will be stopped.

The timeout function is in use also during program stop and will be noticed in the RAPID program at program start.

Program execution

The function reads the specified number of characters from the binary serial channel or file.

Example

```
text := ReadStrBin(infile,20);
IF text = EOF THEN
    TPWrite "The file is empty";
```

Before using the string read from the file, a check is performed to make sure that the file is not empty.

Limitations

The function can only be used for serial channels or files that have been opened for reading in a binary mode.

Error handling

If an error occurs during reading, the system variable ERRNO is set to ERR_FILEACC.

If timeout before the read operation is finished, the system variable ERRNO is set to ERR_DEV_MAXTIME.

These errors can then be dealt with by the error handler.

Predefined data

The constant *EOF* can be used to check if the file was empty, when trying to read from the file or to stop reading at the end of the file.

```
CONST string EOF := "EOF";
```

Syntax

ReadStrBin '('
  [IODevice ':='] <variable (**VAR**) of *iodev*>','
  [NoOfChars ':='] <expression (**IN**) of *num*>
  ['\'Time':=' <expression (**IN**) of *num*>]')'

A function with a return value of the type *string*.

Related information

	Described in:
Opening (etc.) serial channels or files	RAPID Summary - *Communication*
Write binary string	Instructions - *WriteStrBin*

RelTool - Make a displacement relative to the tool

RelTool (Relative Tool) is used to add a displacement and/or a rotation, expressed in the tool coordinate system, to a robot position.

Example

```
MoveL    RelTool (p1, 0, 0, 100), v100, fine, tool1;
```

The robot is moved to a position that is 100 mm from p1 in the direction of the tool.

```
MoveL    RelTool (p1, 0, 0, 0 \Rz:= 25), v100, fine, tool1;
```

The tool is rotated 25^o around its z-axis.

Return value Data type: *robtarget*

The new position with the addition of a displacement and/or a rotation, if any, relative to the active tool.

Arguments

RelTool (Point Dx Dy Dz [\Rx] [\Ry] [\Rz])

Point Data type: *robtarget*

The input robot position. The orientation part of this position defines the current orientation of the tool coordinate system.

Dx Data type: *num*

The displacement in mm in the x direction of the tool coordinate system.

Dy Data type: *num*

The displacement in mm in the y direction of the tool coordinate system.

Dz Data type: *num*

The displacement in mm in the z direction of the tool coordinate system.

[\Rx] Data type: *num*

The rotation in degrees around the x axis of the tool coordinate system.

[\Ry] Data type: *num*

The rotation in degrees around the y axis of the tool coordinate system.

[\Rz] Data type: *num*

The rotation in degrees around the z axis of the tool coordinate system.

In the event that two or three rotations are specified at the same time, these will be performed first around the x-axis, then around the new y-axis, and then around the new z-axis.

Syntax

RelTool'('
[Point ':='] < expression (**IN**) of *robtarget*>','
[Dx ':='] <expression (**IN**) of *num*> ','
[Dy ':='] <expression (**IN**) of *num*> ','
[Dz ':='] <expression (**IN**) of *num*>
['\'Rx ':=' <expression (**IN**) of *num*>]
['\'Ry ':=' <expression (**IN**) of *num*>]
['\'Rz ':=' <expression (**IN**) of *num*>]')'

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Positioning instructions	RAPID Summary - *Motion*

RobOS - Check if execution is on RC or VC

RobOS (Robot Operating System) can be used to check, if the execution is performed on Robot Controller RC or Virtual Controller VC (such as RobotStudio, Program-Maker, QuickTeach).

Example

```
IF RobOS() THEN
    ! Execution statements in RC
ELSE
    ! Execution statements in VC
ENDIF
```

Return value Data type: *bool*

TRUE if execution runs on Robot Controller, FALSE otherwise.

Syntax

RobOS '(")'

A function with a return value of the data type *bool*.

Round - Round is a numeric value

Round is used to round a numeric value to a specified number of decimals or to an integer value.

Example

VAR num val;

val := Round(0.38521\Dec:=3);

The variable *val* is given the value 0.385.

val := Round(0.38521\Dec:=1);

The variable *val* is given the value 0.4.

val := Round(0.38521);

The variable *val* is given the value 0.

Return value Data type: *num*

The numeric value rounded to the specified number of decimals.

Arguments

Round (Val [\Dec])

Val (*Value*) Data type: *num*

The numeric value to be rounded.

[\Dec] (*Decimals* Data type: *num*

Number of decimals.

If the specified number of decimals is 0 or if the argument is omitted, the value is rounded to an integer.

The number of decimals must not be negative or greater than the available precision for numeric values.

Syntax

Round'('
    [Val ':='] <expression (**IN**) of *num*>
    [\Dec ':=' <expression (**IN**) of *num*>]
    ')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Truncating a value	Functions - *Trunc*

RunMode - Read the running mode

RunMode (Running Mode) is used to read the current running mode of the program task.

Example

```
IF RunMode() = RUN_CONT_CYCLE THEN
..
ENDIF
```

The program section is executed only for continuous or cycle running.

Return value Data type: *symnum*

The current running mode as defined in the table below.

Return value	Symbolic constant	Comment
0	RUN_UNDEF	Undefined running mode
1	RUN_CONT_CYCLE	Continuous or cycle running mode
2	RUN_INSTR_FWD	Instruction forward running mode
3	RUN_INSTR_BWD	Instruction backward running mode
4	RUN_SIM	Simulated running mode
5	RUN_STEP_MOVE	Move instructions forward running, logical instructions continuous running mode

Arguments

RunMode ([\Main])

[\Main] Data type: *switch*

Return current running mode for program task *main*.
Used in multi-tasking system to get current running mode for program task *main* from some other program task.

If this argument is omitted, the return value always mirrors the current running mode for the program task which executes the function *RunMode*.

Syntax

RunMode '(' ['\'Main] ')'

A function with a return value of the data type *symnum*.

Related information

	Described in:
Reading operating mode	Functions - *OpMode*

Sin - Calculates the sine value

Sin (Sine) is used to calculate the sine value from an angle value.

Example

```
VAR num angle;
VAR num value;
.
.
value := Sin(angle);
```

Return value Data type: *num*

The sine value, range [-1, 1] .

Arguments

Sin (Angle)

Angle Data type: *num*

The angle value, expressed in degrees.

Syntax

Sin'('
  [Angle':='] <expression (**IN**) of *num*>
  ')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Sqrt - Calculates the square root value

Sqrt (Square root) is used to calculate the square root value.

Example

```
VAR num x_value;
VAR num y_value;
.
.
y_value := Sqrt( x_value);
```

Return value Data type: *num*

The square root value.

Arguments

Sqrt (Value)

Value Data type: *num*

The argument value for square root ($\sqrt{\ }$); it has to be ≥ 0.

Syntax

Sqrt'('
  [Value':='] <expression (**IN**) of *num*>
  ')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

StrFind - Searches for a character in a string

StrFind (String Find) is used to search in a string, starting at a specified position, for a character that belongs to a specified set of characters.

Example

VAR num found;

found := StrFind("Robotics",1,"aeiou");

The variable *found* is given the value 2.

found := StrFind("Robotics",1,"aeiou"\NotInSet);

The variable *found* is given the value 1.

found := StrFind("IRB 6400",1,STR_DIGIT);

The variable *found* is given the value 5.

found := StrFind("IRB 6400",1,STR_WHITE);

The variable *found* is given the value 4.

Return value Data type: *num*

The character position of the first character, at or past the specified position, that belongs to the specified set. If no such character is found, String length +1 is returned.

Arguments

StrFind (Str ChPos Set [\NotInSet])

Str (*String*) Data type: *string*

The string to search in.

ChPos (*Character Position*) Data type: *num*

Start character position. A runtime error is generated if the position is outside the string.

Set Data type: *string*

Set of characters to test against.

[\NotInSet] Data type: *switch*

Search for a character not in the set of characters.

Syntax

StrFind'('
[Str ':='] <expression (**IN**) of *string*> ','
[ChPos ':='] <expression (**IN**) of *num*> ','
[Set':='] <expression (**IN**) of *string*>
['\'NotInSet]
')'

A function with a return value of the data type *num*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrLen - Gets the string length

StrLen (String Length) is used to find the current length of a string.

Example

VAR num len;

len := StrLen("Robotics");

The variable *len* is given the value 8.

Return value Data type: *num*

The number of characters in the string (>=0).

Arguments

StrLen (Str)

Str (*String*) Data type: *string*

The string in which the number of characters is to be counted.

Syntax

StrLen'('
[Str ':='] <expression (**IN**) of *string*>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrMap - Maps a string

StrMap (String Mapping) is used to create a copy of a string in which all characters are translated according to a specified mapping.

Example

```
VAR string str;

str := StrMap("Robotics","aeiou","AEIOU");
```

The variable *str* is given the value "RObOtIcs".

```
str := StrMap("Robotics",STR_LOWER, STR_UPPER);
```

The variable *str* is given the value "ROBOTICS".

Return value Data type: *string*

The string created by translating the characters in the specified string, as specified by the "from" and "to" strings. Each character, from the specified string, that is found in the "from" string is replaced by the character at the corresponding position in the "to" string. Characters for which no mapping is defined are copied unchanged to the resulting string.

Arguments

StrMap (Str FromMap ToMap)

Str (*String*) Data type: *string*

The string to translate.

FromMap Data type: *string*

Index part of mapping.

ToMap Data type: *string*

Value part of mapping.

Syntax

StrMap'('
  [Str ':='] <expression (**IN**) of *string*> ','
  [FromMap':='] <expression (**IN**) of *string*> ','
  [ToMap':='] <expression (**IN**) of *string*>
  ')'

A function with a return value of the data type *string*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrMatch - Search for pattern in string

StrMatch (String Match) is used to search in a string, starting at a specified position, for a specified pattern.

Example

VAR num found;

found := StrMatch("Robotics",1,"bo");

The variable *found* is given the value 3.

Return value Data type: *num*

The character position of the first substring, at or past the specified position, that is equal to the specified pattern string. If no such substring is found, string length +1 is returned.

Arguments

StrMatch (Str ChPos Pattern)

Str (*String*) Data type: *string*

The string to search in.

ChPos (*Character Position*) Data type: *num*

Start character position. A runtime error is generated if the position is outside the string.

Pattern Data type: *string*

Pattern string to search for.

Syntax

StrMatch'('
[Str ':='] <expression (**IN**) of *string*> ','
[ChPos ':='] <expression (**IN**) of *num*> ','
[Pattern':='] <expression (**IN**) of *string*>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrMemb - Checks if a character belongs to a set

StrMemb (String Member) is used to check whether a specified character in a string belongs to a specified set of characters.

Example

```
VAR bool memb;
```

```
memb := StrMemb("Robotics",2,"aeiou");
```

The variable *memb* is given the value TRUE, as o is a member of the set "aeiou".

```
memb := StrMemb("Robotics",3,"aeiou");
```

The variable *memb* is given the value FALSE, as b is not a member of the set "aeiou".

```
memb := StrMemb("S-721 68  VÄSTERÅS",3,STR_DIGIT);
```

The variable *memb* is given the value TRUE.

Return value Data type: *bool*

TRUE if the character at the specified position in the specified string belongs to the specified set of characters.

Arguments

StrMemb (Str ChPos Set)

Str (*String*) Data type: *string*

The string to check in.

ChPos (*Character Position*) Data type: *num*

The character position to check. A runtime error is generated if the position is outside the string.

Set Data type: *string*

Set of characters to test against.

Syntax

StrMemb'('
[Str ':='] <expression (**IN**) of *string*> ','
[ChPos ':='] <expression (**IN**) of *num*> ','
[Set':='] <expression (**IN**) of *string*>
')'

A function with a return value of the data type *bool*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrOrder - Checks if strings are ordered

StrOrder (String Order) is used to check whether two strings are in order, according to a specified character ordering sequence.

Example

```
VAR bool le;

le := StrOrder("FIRST","SECOND",STR_UPPER);
```

The variable *le* is given the value TRUE, because "FIRST" comes before "SECOND" in the character ordering sequence STR_UPPER.

Return value Data type: *bool*

TRUE if the first string comes before the second string (Str1 <= Str2) when characters are ordered as specified.

Characters that are not included in the defined ordering are all assumed to follow the present ones.

Arguments

StrOrder (Str1 Str2 Order)

Str1 (*String 1*) Data type: *string*

First string value.

Str2 (*String 2*) Data type: *string*

Second string value.

Order Data type: *string*

Sequence of characters that define the ordering.

Syntax

StrOrder'('
[Str1 ':='] <expression (**IN**) of *string*> ','
[Str2 ':='] <expression (**IN**) of *string*> ','
[Order ':='] <expression (**IN**) of *string*>
')'

A function with a return value of the data type *bool*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrPart - Finds a part of a string

StrPart (String Part) is used to find a part of a string, as a new string.

Example

```
VAR string part;

part := StrPart("Robotics",1,5);
```

The variable *part* is given the value "Robot".

Return value Data type: *string*

The substring of the specified string, which has the specified length and starts at the specified character position.

Arguments

StrPart (Str ChPos Len)

Str (*String*) Data type: *string*

The string in which a part is to be found.

ChPos (*Character Position*) Data type: *num*

Start character position. A runtime error is generated if the position is outside the string.

Len (*Length*) Data type: *num*

Length of string part. A runtime error is generated if the length is negative or greater than the length of the string, or if the substring is (partially) outside the string.

Syntax

StrPart'('
  [Str ':='] <expression (**IN**) of *string*> ','
  [ChPos ':='] <expression (**IN**) of *num*> ','
  [Len':='] <expression (**IN**) of *num*>
  ')'

A function with a return value of the data type *string*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrToByte - Converts a string to a byte data

StrToByte (String To Byte) is used to convert a *string* with a defined byte data format into a *byte* data.

Example

```
VAR string con_data_buffer{5} := ["10", "AE", "176", "00001010", "A"];
VAR byte data_buffer{5};
```

```
data_buffer{1} := StrToByte(con_data_buffer{1});
```

The content of the array component *data_buffer{1}* will be 10 decimal after the *StrToByte* ... function.

```
data_buffer{2} := StrToByte(con_data_buffer{2}\Hex);
```

The content of the array component *data_buffer{2}* will be 174 decimal after the *StrToByte* ... function.

```
data_buffer{3} := StrToByte(con_data_buffer{3}\Okt);
```

The content of the array component *data_buffer{3}* will be 126 decimal after the *StrToByte* ... function.

```
data_buffer{4} := StrToByte(con_data_buffer{4}\Bin);
```

The content of the array component *data_buffer{4}* will be 10 decimal after the *StrToByte* ... function.

```
data_buffer{5} := StrToByte(con_data_buffer{5}\Char);
```

The content of the array component *data_buffer{5}* will be 65 decimal after the *StrToByte* ... function.

Return value Data type: *byte*

The result of the conversion operation in decimal representation.

Arguments

StrToByte (ConStr [\Hex] | [\Okt] | [\Bin] | [\Char])

ConStr *(Convert String)* Data type: *string*

The string data to be converted.

If the optional switch argument is omitted, the string to be converted has *decimal* (Dec) format.

[\Hex] *(Hexadecimal)* Data type: *switch*

The string to be converted has *hexadecimal* format.

[\Okt] *(Octal)* Data type: *switch*

The string to be converted has *octal* format.

[\Bin] *(Binary)* Data type: *switch*

The string to be converted has *binary* format.

[\Char] *(Character)* Data type: *switch*

The string to be converted has *ASCII* character format.

Limitations

Depending on the format of the string to be converted, the following string data is valid:

Format:	String length:	Range:
Dec .....: ’0’ - ’9’	3	"0" - "255"
Hex .....: ’0’ - ’9’, ’a’ -’f’, ’A’ - ’F’	2	"0" - "FF"
Okt ......: ’0’ - ’7’	3	"0" - "377"
Bin ......: ’0’ - ’1’	8	"0" - "11111111"
Char ....: Any ASCII character	1	One ASCII char

RAPID character codes (e.g. “\07” for BEL control character) can be used as arguments in *ConStr.*

Syntax

StrToByte'('
    [ConStr ':='] <expression (**IN**) of *string*>
    ['\' Hex] | ['\' Okt] | ['\' Bin] | ['\' Char]
    ')' ';'

A function with a return value of the data type *byte*.

Related information

	Described in:
Convert a byte to a string data	Instructions - *ByteToStr*
Other bit (byte) functions	RAPID Summary - *Bit Functions*
Other string functions	RAPID Summary - *String Functions*

StrToVal - Converts a string to a value

StrToVal (String To Value) is used to convert a string to a value of any data type.

Example

```
VAR bool ok;
VAR num nval;

ok := StrToVal("3.85",nval);
```

The variable *ok* is given the value TRUE and *nval* is given the value *3.85*.

Return value Data type: *bool*

TRUE if the requested conversion succeeded, FALSE otherwise.

Arguments

StrToVal (Str Val)

Str (*String*) Data type: *string*

A string value containing literal data with format corresponding to the data type used in argument *Val*. Valid format as for RAPID literal aggregates.

Val (*Value*) Data type: *ANYTYPE*

Name of the variable or persistent of any data type for storage of the result from the conversion. The data is unchanged if the requested conversion failed.

Example

VAR string 15 := "[600, 500, 225.3]";
```
VAR bool ok;
VAR pos pos15;

ok := StrToVal(str15,pos15);
```

The variable *ok* is given the value TRUE and the variable *p15* is given the value that are specified in the string *str15*.

Syntax

StrToVal'('
[Str ':='] <expression (**IN**) of *string*> ','
[Val ':='] <var or pers (**INOUT**) of *ANYTYPE*>
')'

A function with a return value of the data type *bool*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

Tan - Calculates the tangent value

Tan (Tangent) is used to calculate the tangent value from an angle value.

Example

VAR num angle;
VAR num value;
.
.
value := Tan(angle);

Return value Data type: *num*

The tangent value.

Arguments

Tan (Angle)

Angle Data type: *num*

The angle value, expressed in degrees.

Syntax

Tan'('
[Angle ':='] <expression (**IN**) of *num*>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Arc tangent with return value in the range [-180, 180]	*Functions - ATan2*

TestAndSet - Test variable and set if unset

TestAndSet can be used together with a normal data object of the type *bool*, as a binary semaphore, to retrieve exclusive right to specific RAPID code areas or system resources. The function could be used both between different program tasks and different execution levels (TRAP or Event Routines) within the same program task.

Example of resources that can need protection from access at the same time:

- Use of some RAPID routines with function problems when executed in parallel.
- Use of the Teach Pendant - Operator Output & Input

Example

MAIN program task:

```
PERS bool tproutine_inuse := FALSE;
....
WaitUntil TestAndSet(tproutine_inuse);
TPWrite “First line from MAIN”;
TPWrite “Second line from MAIN”;
TPWrite “Third line from MAIN”;
tproutine_inuse := FALSE;
```

BACK1 program task:

```
PERS bool tproutine_inuse := FALSE;
.....
WaitUntil TestAndSet(tproutine_inuse);
TPWrite “First line from BACK1”;
TPWrite “Second line from BACK1”;
TPWrite “Third line from BACK1”;
tproutine_inuse := FALSE;
```

To avoid mixing up the lines, one from MAIN and one from BACK1, the use of the TestAndSet function guarantees that all three lines from each task are not separated.

If program task MAIN takes the semaphore *TestAndSet(tproutine_inuse)* first, then program task BACK1 must wait until the program task MAIN has left the semaphore.

Return value Data type: *num*

TRUE if the semaphore has been taken by me (executor of TestAndSet function), otherwise FALSE. ???

Arguments

TestAndSet Object

Object Data type: *bool*

User defined data object to be used as semaphore. The data object could be a VAR or a PERS. If TestAndSet are used between different program tasks, the object must be a PERS or an installed VAR (intertask objects).

Program execution

This function will in one indivisible step check the user defined variable and, if it is unset, will set it and return TRUE, otherwise it will return FALSE.

```
IF Object = FALSE THEN
        Object := TRUE;
        RETURN TRUE;
ELSE
        RETURN FALSE;
ENDIF
```

Example

```
LOCAL VAR bool doit_inuse := FALSE;
...
PROC doit(...)
        WaitUntil TestAndSet (doit_inuse);
        ....
        doit_inuse := FALSE;
ENDPROC
```

If a module is installed built-in and shared, it is possible to use a local module variable for protection of access from different program tasks at the same time.

Note in this case: If program execution is stopped in the routine *doit* and the program pointer is moved to *main*, the variable *doit_inuse* will not be reset. To avoid this, reset the variable *doit_inuse* to FALSE in the START event routine.

Syntax

TestAndSet '('
  [Object ':='] < variable or persistent (**INOUT**) of *bool*> ')'

A function with a return value of the data type *bool*.

Related information

	Described in:
Built-in and shared module	User's Guide - *System parameters*
Intertask objects	RAPID Developer's Manual - RAPID Kernel Reference Manual - *Intertask objects*

TestDI - Tests if a digital input is set

TestDI is used to test whether a digital input is set.

Examples

IF TestDI (di2) THEN . . .

If the current value of the signal *di2* is equal to *1*, then . . .

IF NOT TestDI (di2) THEN . . .

If the current value of the signal *di2* is equal to *0*, then . . .

WaitUntil TestDI(di1) AND TestDI(di2);

Program execution continues only after both the *di1* input and the *di2* input have been set.

Return value Data type: *bool*

TRUE = The current value of the signal is equal to 1.

FALSE = The current value of the signal is equal to 0.

Arguments

TestDI (Signal)

Signal Data type: *signaldi*

The name of the signal to be tested.

Syntax

TestDI '('
[Signal ':='] < variable (**VAR**) of *signaldi* > ')'

A function with a return value of the data type *bool*.

Related information

	Described in:
Reading the value of a digital input signal	Functions - *DInput*
Input/Output instructions	RAPID Summary - *Input and Output Signals*

TestSignRead - Read test signal value

TestSignRead is used to read the actual test signal value.

This function returns the momentary value or the mean value of the latest samples, depending on channel specification in instruction *TestSignDefine*.

Example

```
CONST num speed_channel;
VAR num speed_value;
...
TestSignDefine speed_channel, speed, orbit, 1, 0;
...
! During some movements with orbit's axis 1
speed_value := TestSignRead(speed_channel);
...
TestSignReset;
```

speed_value is assigned the mean value of the latest 8 samples generated each 0.5ms of the test signal *speed* on channel *speed_channel*.
The channel *speed_channel* measures the speed of axis *1* on the mechanical unit *orbit*.

Return value

Data type: *num*

The numeric value in SI units on the motor side for the specified channel according to the definition in instruction *TestSignDefine*.

Arguments

TestSignRead (Channel)

Channel Data type: *num*

The channel number 1-12 for the test signal to be read.
The same number must be used in the definition instruction *TestSignDefine*.

Program execution

Returns the momentary value or the mean value of the latest samples, depending on the channel specification in the instruction *TestSignDefine*.

For predefined test signals with valid SI units for external manipulator axes, see data type *testsignal*.

Example

```
CONST num torque_channel;
VAR num torque_value;
VAR intnum timer_int;
CONST jointtarget psync := [...];
...
CONNECT timer_int WITH TorqueTrap;
ITimer \Single, 0.05, timer_int;
TestSignDefine torque_channel, torque_ref, IRBP_K, 2, 0.001;
...
MoveAbsJ psync \NoEOffs, v5, fine, tool0;
...
IDelete timer_int;
TestSignReset;

TRAP TorqueTrap
   IF (TestSignRead(torque_channel) > 6) THEN
      TPWrite "Torque pos = " + ValToStr(CJointT());
      Stop;
      EXIT;
   ELSE
      IDelete timer_int;
      CONNECT timer_int WITH TorqueTrap;
      ITimer \Single, 0.05, timer_int;
   ENDIF
ENDTRAP
```

The joint position, when the torque reference for manipulator *IRBP_K* axis *2* is for the first time greater than *6* Nm on the motor side during the slow movement to position *psync*, is displayed on the Operators Window on the TP.

Syntax

TestSignRead'('
   [Channel ':='] <expression (**IN**) of *num*>')'

A function with a return value of the type *num*.

Related information

	Described in:
Define test signal	Instructions - *TestSignDefine*
Reset test signals	Instructions - *TestSignReset*

Trunc - Truncates a numeric value

Trunc (Truncate) is used to truncate a numeric value to a specified number of decimals or to an integer value.

Example

```
VAR num val;

val := Trunc(0.38521\Dec:=3);
```

The variable *val* is given the value 0.385.

```
reg1 := 0.38521

val := Trunc(reg1\Dec:=1);
```

The variable *val* is given the value 0.3.

```
val := Trunc(0.38521);
```

The variable *val* is given the value 0.

Return value Data type: *num*

The numeric value truncated to the specified number of decimals.

Arguments

Trunc (Val [\Dec])

Val (*Value*) Data type: *num*

The numeric value to be truncated.

[\Dec] (*Decimals*) Data type: *num*

Number of decimals.

If the specified number of decimals is 0 or if the argument is omitted, the value is truncated to an integer.

The number of decimals must not be negative or greater than the available precision for numeric values.

Syntax

Trunc'('
[Val ':='] <expression (**IN**) of *num*>
[\Dec ':=' <expression (**IN**) of *num*>]
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Rounding a value	Functions - *Round*

ValToStr - Converts a value to a string

ValToStr (Value To String) is used to convert a value of any data type to a string.

Example

```
VAR string str;
VAR pos p := [100,200,300];

str := ValToStr(1.234567);
```

The variable *str* is given the value "1.23457".

```
str := ValToStr(TRUE);
```

The variable *str* is given the value "TRUE".

```
str := ValToStr(p);
```

The variable *str* is given the value "[100,200,300]".

Return value Data type: *string*

The value is converted to a string with standard RAPID format. This means in principle 6 significant digits. If the decimal part is less than 0.000005 or greater than 0.999995, the number is rounded to an integer.

A runtime error is generated if the resulting string is too long.

Arguments

ValToStr (Val)

Val (*Value*) Data type: *ANYTYPE*

A value of any data type.

Syntax

ValToStr'('
[Val ':='] <expression (**IN**) of *ANYTYPE*>
')'

A function with a return value of the data type *string*.

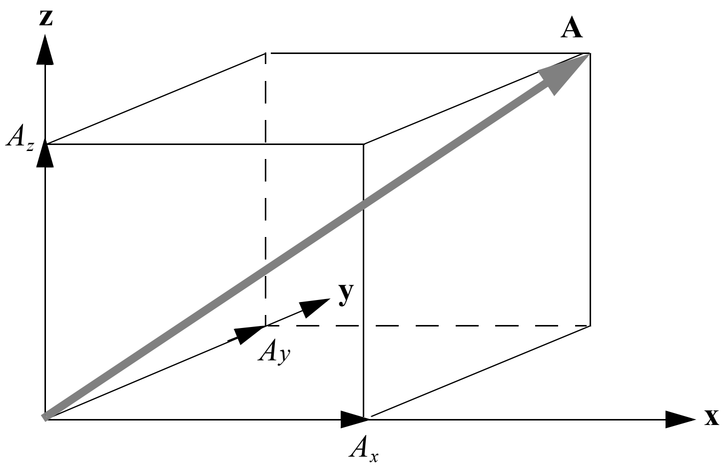
Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

VectMagn - Magnitude of a pos vector

VectMagn (Vector Magnitude) is used to calculate the magnitude of a *pos* vector.

Example



A vector **A** can be written as the sum of its components in the three orthogonal directions:

$$A = A_x x + A_y y + A_z z$$

The magnitude of **A** is:

$$|A| = \sqrt{A_x^2 + A_y^2 + A_z^2}$$

The vector is described by the data type *pos* and the magnitude by the data type *num*:

```
VAR num magnitude;
VAR pos vector;
.
.
vector := [1,1,1];
magnitude := VectMagn(vector);
```

Return value Data type: *num*

The magnitude of the vector (data type *pos*).

Arguments

VectMagn (Vector)

Vector Data type: *pos*

The vector described by the data type *pos*.

Syntax

VectMagn'('
[Vector ':='] <expression (**IN**) of *pos*>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

aiotrigg - Analog I/O trigger condition

aiotrigg (Analog I/O Trigger) is used to define the condition to generate an interrupt for an analog input or output signal.

Description

Data of the type *aiotrigg* defines the way a low and a high threshold will be used to determine whether the logical value of an analog signal satisfies a condition to generate an interrupt.

Example

```
VAR intnum sig1int;
CONNECT sig1int WITH iroutine1;
ISignalAI \Single, ai1, AIO_BETWEEN, 1.5, 0.5, 0, sig1int;
```

Orders an interrupt which is to occur the first time the logical value of the analog input signal *ai1* is between *0.5* and *1.5*. A call is then made to the *iroutine1* trap routine.

Predefined data

The following symbolic constants of the data type aiotrigg are predefined and can be used when specifying a condition for the instructions *ISignalAI* and *ISignalAO*.

Value	Symbolic constant	Comment
1	AIO_ABOVE_HIGH	Signal will generate interrupts if above specified high value
2	AIO_BELOW_HIGH	Signal will generate interrupts if below specified high value
3	AIO_ABOVE_LOW	Signal will generate interrupts if above specified low value
4	AIO_BELOW_LOW	Signal will generate interrupts if below specified low value
5	AIO_BETWEEN	Signal will generate interrupts if between specified low and high values
6	AIO_OUTSIDE	Signal will generate interrupts if below specified low value or above specified high value
7	AIO_ALWAYS	Signal will always generate interrupts

Characteristics

aiotrigg is an alias data type for num and consequently inherits its characteristics.

Related information

	Described in:
Interrupt from analog input signal	Instructions - *ISignalAI*
Interrupt from analog output signal	Instructions - *ISignalAO*
Data types in general, alias data types	Basic Characteristics - *Data Types*

bool - Logical values

Bool is used for logical values (true/false).

Description

The value of data of the type *bool* can be either *TRUE* or *FALSE*.

Examples

```
flag1 := TRUE;
```

flag is assigned the value TRUE.

```
VAR bool highvalue;
VAR num reg1;
    .
highvalue := reg1 > 100;
```

highvalue is assigned the value *TRUE* if *reg1 is greater than 100*; otherwise, *FALSE* is assigned.

```
IF highvalue Set do1;
```

The *do1* signal is set if *highvalue* is *TRUE*.

```
highvalue := reg1 > 100;
mediumvalue := reg1 > 20 AND NOT highvalue;
```

mediumvalue is assigned the value *TRUE* if *reg1* is between *20* and *100*.

Related information

	Described in:
Logical expressions	Basic Characteristics - *Expressions*
Operations using logical values	Basic Characteristics - *Expressions*

byte - Decimal values 0 - 255

Byte is used for decimal values (0 - 255) according to the range of a byte.

This data type is used in conjunction with instructions and functions that handle the bit manipulations and convert features.

Description

Data of the type *byte* represents a decimal byte value.

Examples

CONST num parity_bit := 8;

VAR byte data1 := 130;

Definition of a variable *data1* with a decimal value 130.

BitClear data1, parity_bit;

Bit number 8 (**parity_bit) in the variable data1 will be set to 0, e.g. the** content of the variable *data1* will be changed from 130 to 2 (decimal representation).

Error handling

If an argument of the type *byte* has a value that is not in the range between 0 and 255, an error is returned on program execution.

Characteristics

Byte is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Alias data types	Basic Characteristics- *Data Types*
Bit functions	RAPID Summary - *Bit Functions*

clock - Time measurement

Clock is used for time measurement. A *clock* functions like a stopwatch used for timing.

Description

Data of the type *clock* stores a time measurement in seconds and has a resolution of 0.01 seconds.

Example

```
VAR clock clock1;

    ClkReset clock1;
```

The clock, *clock1,* is declared and reset. Before using *ClkReset*, *ClkStart*, *ClkStop* and *ClkRead*, you must declare a variable of data type *clock* in your program.

Limitations

The maximum time that can be stored in a clock variable is approximately 49 days (4,294,967 seconds). The instructions *ClkStart*, *ClkStop* and *ClkRead* report clock overflows in the very unlikely event that one occurs.

A clock must be declared as a *VAR* variable type, not as a *persistent* variable type.

Characteristics

Clock is a non-value data type and cannot be used in value-oriented operations.

Related information

	Described in:
Summary of Time and Date Instructions	RAPID Summary - *System & Time*
Non-value data type characteristics	Basic Characteristics - *Data Types*

confdata - Robot configuration data

Confdata is used to define the axis configurations of the robot.

Description

All positions of the robot are defined and stored using rectangular coordinates. When calculating the corresponding axis positions, there will often be two or more possible solutions. This means that the robot is able to achieve the same position, i.e. the tool is in the same position and with the same orientation, with several different positions or configurations of the robots axes.

Some robot types use iterative numerical methods to determine the robot axes positions. In these cases the configuration parameters may be used to define good starting values for the joints to be used by the iterative procedure.

To unambiguously denote one of these possible configurations, the robot configuration is specified using four axis values. For a rotating axis, the value defines the current quadrant of the robot axis. The quadrants are numbered 0, 1, 2, etc. (they can also be negative). The quadrant number is connected to the current joint angle of the axis. For each axis, quadrant 0 is the first quarter revolution, 0 to 90°, in a positive direction from the zero position; quadrant 1 is the next revolution, 90 to 180°, etc. Quadrant -1 is the revolution 0° to (-90°), etc. (see Figure 5).

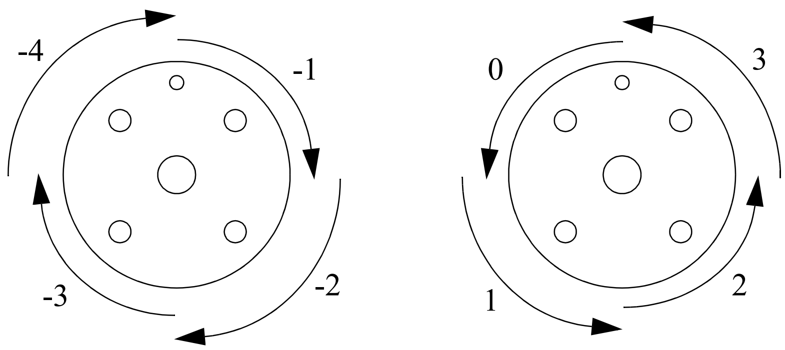


Figure 5 The configuration quadrants for axis 6.

For a linear axis, the value defines a meter interval for the robot axis. For each axis, value 0 means a position between 0 and 1 meters, 1 means a position between 1 and 2 meters. For negative values, -1 means a position between -1 and 0 meters, etc. (see Figure 6).

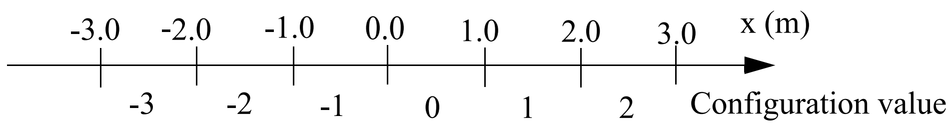


Figure 6 Configuration values for a linear axis.

Robot configuration data for IRB140

There are three singularities within the robots working range (See *Motion and I/O Principles - Singularities*).

cf1 is the quadrant number for axis 1.

cf4 is the quadrant number for axis 4.

cf6 is the quadrant number for axis 6.

cfx is used to select one of eight possible robot configurations numbered from 0 through 7. The table below describes each one of them in terms of how the robot is positioned relative to the three singularities.

cfx	Wrist center relative to axis 1	Wrist center relative to lower arm	Axis 5 angle
0	In front of	In front of	Positive
1	In front of	In front of	Negative
2	In front of	Behind	Positive
3	In front of	Behind	Negative
4	Behind	In front of	Positive
5	Behind	In front of	Negative
6	Behind	Behind	Positive
7	Behind	Behind	Negative

The pictures below give an example of how the same tool position and orientation is attained by using the eight different configurations:

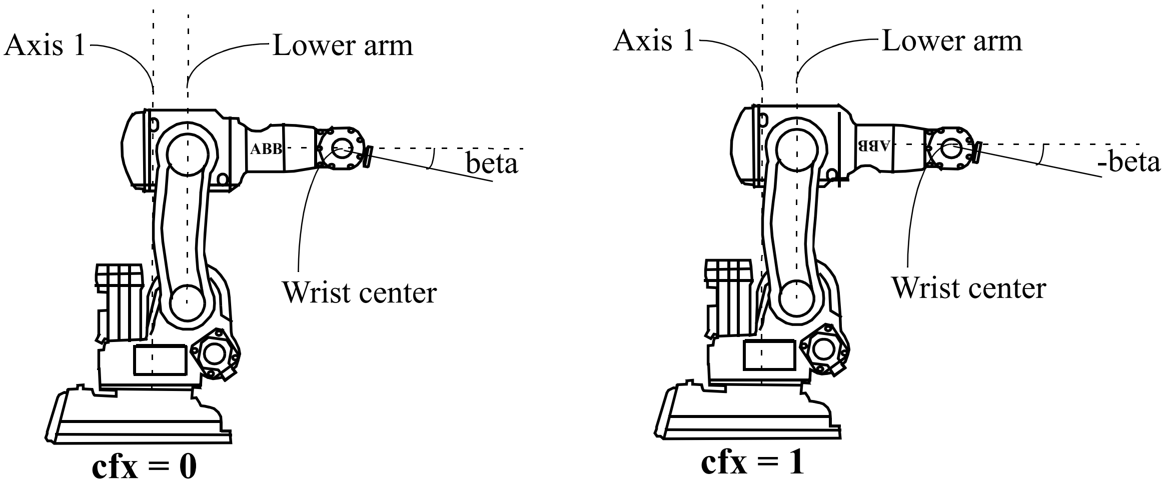


Figure 7 Example of robot configuration 0 and 1. Note the different signs of the axis 5 angle.

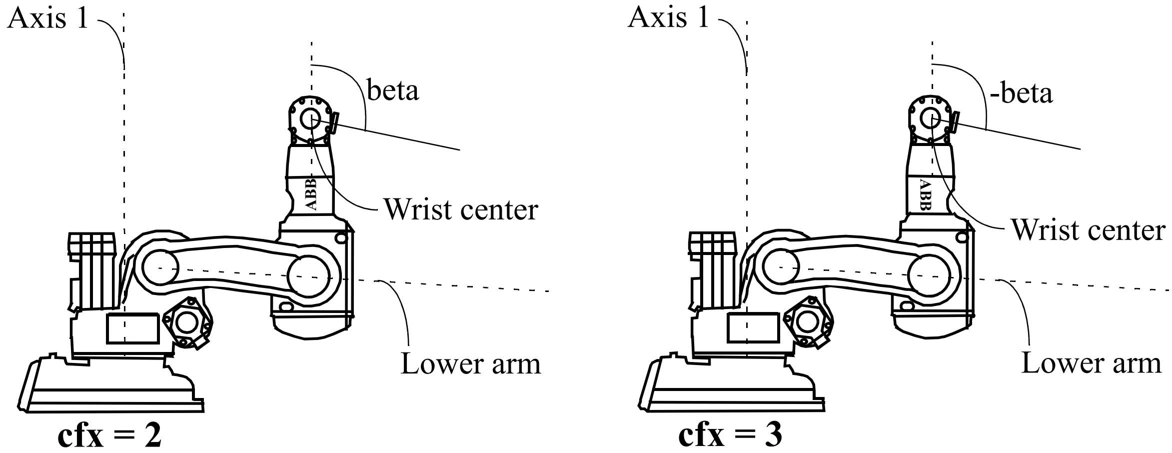


Figure 8 Example of robot configuration 2 and 3. Note the different signs of the axis 5 angle.

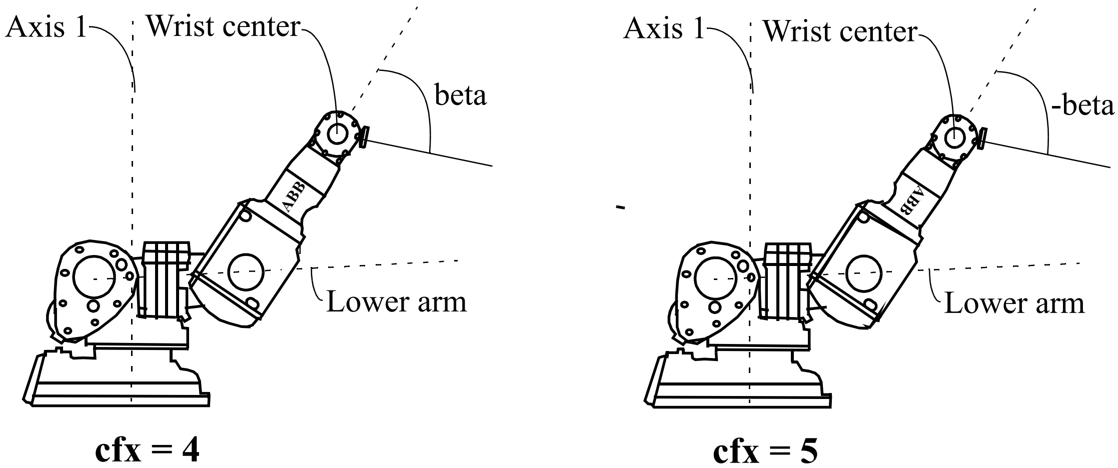


Figure 9 Example of robot configuration 4 and 5. Note the different signs of the axis 5 angle.

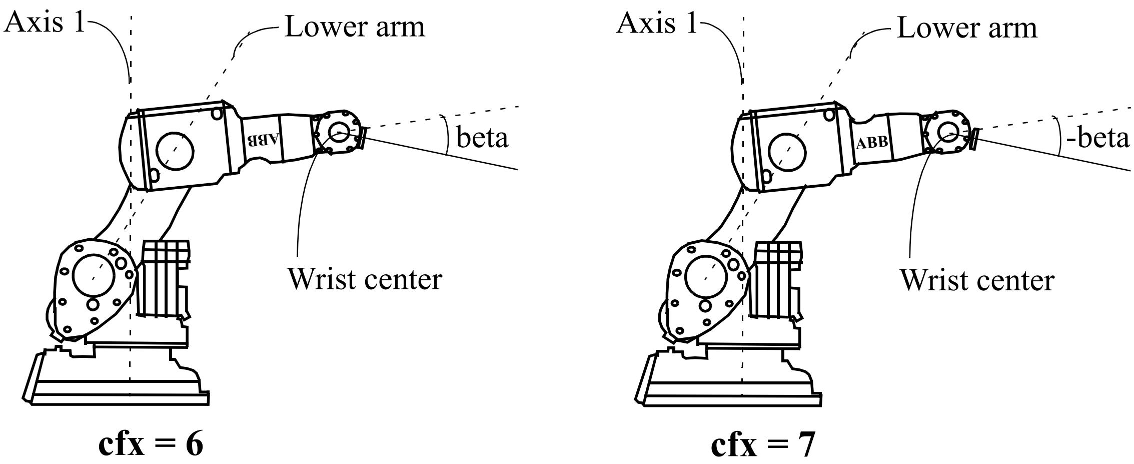


Figure 10 Example of robot configuration 6 and 7. Note the different signs of the axis 5 angle.

Robot configuration data for IRB340

Only the configuration parameter cf4 is used.

Robot configuration data for IRB540, 640

Only the configuration parameter cf6 is used.

Robot configuration data for IRB1400, 2400, 3400, 4400, 6400

Only the three configuration parameters cf1, cf4 and cf6 are used.

Robot configuration data for IRB5400

All four configuration parameters are used. cf1, cf4, cf6 for joints 1, 4, and 6 respectively and cfx for joint 5.

Robot configuration data for IRB5404, 5406

The robots have two rotation axes (arms 1 and 2) and one linear axis (arm 3).

cf1 is used for the rotating axis 1

cfx is used for the rotating axis 2

cf4 and cf6 are not used

Robot configuration data for IRB5413, 5414, 5423

The robots have two linear axes (arms 1 and 2) and one or two rotating axes (arms 4 and 5) (Arm 3 locked)

cf1 is used for the linear axis 1

cfx is used for the linear axis 2

cf4 is used for the rotating axis 4

cf6 is not used

Robot configuration data for IRB840

The robot has three linear axes (arms 1, 2 and 3) and one rotating axis (arm 4).

cf1 is used for the linear axis 1

cfx is used for the linear axis 2

cf4 is used for the rotating axis 4

cf6 is not used

Because of the robot's mainly linear structure, the correct setting of the configuration parameters c1, cx is of less importance.

Components

cf1 Data type: *num*

Rotating axis:

The current quadrant of axis 1, expressed as a positive or negative integer.

Linear axis:

The current meter interval of axis 1, expressed as a positive or negative integer.

cf4 Data type: *num*

Rotating axis:

The current quadrant of axis 4, expressed as a positive or negative integer.

Linear axis:

The current meter interval of axis 4, expressed as a positive or negative integer.

cf6 Data type: *num*

Rotating axis:

The current quadrant of axis 6, expressed as a positive or negative integer.

Linear axis:

The current meter interval of axis 6, expressed as a positive or negative integer.

cfx Data type: *num*

Rotating axis:

For the IRB140, the current robot configuration, expressed as an integer in the range from 0 to 7.

For the IRB5400, the current quadrant of axis 5, expressed as a positive or negative integer.

For other robots, using the current quadrant of axis 2, expressed as a positive or negative integer.

Linear axis:

The current meter interval of axis 2, expressed as a positive or negative integer.

Example

VAR confdata conf15 := [1, -1, 0, 0]

A robot configuration *conf15* is defined as follows:

- The axis configuration of the robot axis 1 is quadrant *1*, i.e. 90-180o.
- The axis configuration of the robot axis 4 is quadrant *-1*, i.e. 0-(-90o).
- The axis configuration of the robot axis 6 is quadrant *0*, i.e. 0 - 90o.
- The axis configuration of the robot axis 5 is quadrant *0*, i.e. 0 - 90o.

Structure

< dataobject of *confdata* >
< *cf1* of *num* >
< *cf4* of *num* >
< *cf6* of *num* >
< *cf*x of *num* >

Related information

	Described in:
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Handling configuration data	Motion and I/O Principles - *Robot Configuration*

dionum - Digital values 0 - 1

Dionum (digital input output numeric) is used for digital values (0 or 1).

This data type is used in conjunction with instructions and functions that handle digital input or output signals.

Description

Data of the type *dionum* represents a digital value 0 or 1.

Examples

CONST dionum close := 1;

Definition of a constant *close* with a value equal to *1*.

SetDO grip1, close;

The signal *grip1* is set to *close*, i.e. 1.

Predefined data

The constants *high, low and edge* are predefined in the system module *user.sys*:

CONST dionum low:=0;

CONST dionum high:=1;

CONST dionum edge:=2;

The constants *low* and *high* are designed for IO instructions.

Edge can be used together with the interrupt instructions *ISignalDI* and *ISignalDO*.

Characteristics

Dionum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Summary input/output instructions	RAPID Summary - *Input and Output Signals*
Configuration of I/O	User's Guide - *System Parameters*
Alias data types	Basic Characteristics- *Data Types*

errdomain - Error domain

errdomain (error domain) is used to specify an error domain.

Description

Data of the type *errdomain* represents the domain where the error, warning or state changed is logged.
Refer to *User Guide - Error Management, System and Error Messages* for more information.

Example

```
VAR errdomain err_domain;
VAR num err_number;
VAR errtype err_type;
VAR trapdata err_data;
...
TRAP trap_err
    GetTrapData err_data;
    ReadErrData err_data, err_domain, err_number, err_type;
ENDTRAP
```

When an error is trapped to the trap routine *trap_err*, the error domain, the error number and the error type are saved into appropriate variables.

Predefined data

The following predefined constants can be used to specify an error domain.

Tabell 1 Predefined error domains

Name	Error Domain	Value
COMMON_ERR	All error and state changed domains	0
OP_STATE	Operational state change	1
SYSTEM_ERR	System errors	2
HARDWARE_ERR	Hardware errors	3
PROGRAM_ERR	Program errors	4
MOTION_ERR	Motion errors	5
OPERATOR_ERROR	Operator errors	6
IO_COM_ERR	I/O and Communication errors	7
USER_DEF_ERR	User defined errors (raised by RAPID)	8
OPTION_PROD_ERR	Optional product errors	9
ARCWELD_ERR	ArcWelding Application errors	11
SPOTWELD_ERR	SpotWelding Application errors	12
PAINT_ERR	Paint Application errors	13
PICKWARE_ERR	Pickware Application errors	14

Characteristics

errdomain is an alias data type for num and consequently inherits its characteristics.

Related information

	Described in:
Ordering an interrupt on errors	Instructions - *IError*
Error numbers	User's Guide - *System and error messages*
Alias data types	Basic Characteristics - *Data Types*

errnum - Error number

Errnum is used to describe all recoverable (non fatal) errors that occur during program execution, such as division by zero.

Description

If the robot detects an error during program execution, this can be dealt with in the error handler of the routine. Examples of such errors are values that are too high and division by zero. The system variable *ERRNO*, of type *errnum*, is thus assigned different values depending on the nature of an error. The error handler may be able to correct an error by reading this variable and then program execution can continue in the correct way.

An error can also be created from within the program using the RAISE instruction. This particular type of error can be detected in the error handler by specifying an error number (within the range 1-90 or booked with instruction *BookErrNo*) as an argument to RAISE.

Examples

```
reg1 := reg2 / reg3;
.
ERROR
  IF ERRNO = ERR_DIVZERO THEN
    reg3 := 1;
    RETRY;
  ENDIF
```

If *reg3* = 0, the robot detects an error when division is taking place. This error, however, can be detected and corrected by assigning *reg3* the value *1*. Following this, the division can be performed again and program execution can continue.

```
CONST errnum machine_error := 1;
.
IF di1=0 RAISE machine_error;
.
ERROR
  IF ERRNO=machine_error RAISE;
```

An error occurs in a machine (detected by means of the input signal *di1*). A jump is made to the error handler in the routine which, in turn, calls the error handler of the calling routine where the error may possibly be corrected. The constant, *machine_error*, is used to let the error handler know exactly what type of error has occurred.

Predefined data

The system variable ERRNO can be used to read the latest error that occurred. A number of predefined constants can be used to determine the type of error that has occurred.

Name	Cause of error
ERR_ACC_TOO_LOW	Too low acceleration/deceleration specified in instruction *PathAccLim* or *WorldAccLim*
ERR_ALRDYCNT	The interrupt variable is already connected to a TRAP routine
ERR_ALRDY_MOVING	The robot is already moving when executing a *StartMove* instruction
ERR_AO_LIM	*ScaleLag* analog signal value outside limit in *TriggIO, TriggEquip* or *TriggSpeed*
ERR_ARGDUPCND	More than one present conditional argument for the same parameter
ERR_ARGNAME	Argument is expression, not present or of type switch when executing ArgName
ERR_ARGNOTPER	Argument is not a persistent reference
ERR_ARGNOTVAR	Argument is not a variable reference
ERR_AXIS_ACT	Axis is not active
ERR_AXIS_IND	Axis is not independent
ERR_AXIS_MOVING	Axis is moving
ERR_AXIS_PAR	Parameter axis in instruction TestSign and SetCurrRef is wrong.
ERR_BWDLIMIT	Limit StepBwdPath
ERR_CALLIO_INTER	If an IOEnable or IODisable request is interrupted by another request to the same unit
ERR_CALLPROC	Procedure call error (not procedure) at runtime (late binding)
ERR_CFG_ILLTYPE	Type mismatch - *ReadCfgData*, *WriteCfgData*
ERR_CFG_LIMIT	Data limit - *WriteCfgData*
ERR_CFG_NOTFND	Not found - *ReadCfgData*, *WriteCfgData*
ERR_CNTNOTVAR	CONNECT target is not a variable reference
ERR_CNV_NOT_ACT	The conveyor is not activated.
ERR_CNV_CONNECT	The *WaitWobj* instruction is already active.
ERR_CNV_DROPPED	The object that the instruction *WaitWobj* was waiting for has been dropped.

ERR_DEV_MAXTIME	Timeout when executing a ReadBin, ReadNum or a ReadStr instruction
ERR_DIPLAG_LIM	Too big *DipLag* in the instruction *TriggSpeed* connected to current *TriggL/TriggC/TriggJ*
ERR_DIVZERO	Division by zero
ERR_EXCRTYMAX	Maximum number of retries exceeded.
ERR_EXECPHR	An attempt was made to execute an instruction using a place holder
ERR_FILEACC	A file is accessed incorrectly
ERR_FILEEXIST	A file already exists
ERR_FILEOPEN	A file cannot be opened
ERR_FILNOTFND	File not found
ERR_FNCNORET	No return value
ERR_FRAME	Unable to calculate new frame
ERR_ILLDIM	Incorrect array dimension
ERR_ILLQUAT	Attempt to use illegal orientation (quaternion) value
ERR_ILLRAISE	Error number in RAISE out of range
ERR_INOMAX	No more interrupt numbers available
ERR_IODISABLE	Timeout when executing IODisable
ERR_IODN_TIMEOUT	Timeout when executing IODNGetAttr or IODNSetAttr
ERR_IOENABLE	Timeout when executing IOEnable
ERR_IOERROR	I/O Error from instruction Save
ERR_LOADED	The program module is already loaded
ERR_LOADID_FATAL	Only internal use in LoadId
ERR_LOADID_RETRY	Only internal use in LoadId
ERR_LOADNO_INUSE	The load session is in use in StartLoad
ERR_LOADNO_NOUSE	The load session is not in use in CancelLoad
ERR_MAXINTVAL	The integer value is too large
ERR_MODULE	Incorrect module name in instruction Save
ERR_MSG_PENDING	The unit is busy
ERR_NAME_INVALID	If the unit name does not exist or if the unit is not allowed to be disabled
ERR_NEGARG	Negative argument is not allowed
ERR_NOTARR	Data is not an array

ERR_NOTEQDIM	The array dimension used when calling the routine does not coincide with its parameters
ERR_NOTINTVAL	Not an integer value
ERR_NOTPRES	A parameter is used, despite the fact that the corresponding argument was not used at the routine call
ERR_OUTOFBND	The array index is outside the permitted limits
ERR_OVERFLOW	Clock overflow
ERR_PATH	Missing destination path in instruction Save
ERR_PATHDIST	Too long regain distance for StartMove instruction
ERR_PID_MOVESTOP	Only internal use in LoadId
ERR_PID_RAISE_PP	Error from ParIdRobValid or ParIdPosValid
ERR_RANYBIN_CHK	Check sum error detected at data transfer with instruction ReadAnyBin
ERR_RANYBIN_EOF	End of file is detected before all bytes are read in instruction ReadAnyBin
ERR_RCVDATA	An attempt was made to read non numeric data with ReadNum
ERR_REFUNKDAT	Reference to unknown entire data object
ERR_REFUNKFUN	Reference to unknown function
ERR_REFUNKPRC	Reference to unknown procedure at linking time or at run time (late binding)
ERR_REFUNKTRP	Reference to unknown trap
ERR_ROBLIMIT	Axis outside working area or limits exceeded for at least one coupled joint
ERR_SC_WRITE	Error when sending to external computer
ERR_SIGSUPSEARCH	The signal has already a positive value at the beginning of the search process
ERR_STRTOOLNG	The string is too long
ERR_SYM_ACCESS	Symbol read/write access error
ERR_TP_DIBREAK	A TPRead instruction was interrupted by a digital input
ERR_TP_MAXTIME	Timeout when executing a TPRead instruction
ERR_UNIT_PAR	Parameter Mech_unit in TestSign and SetCurrRef is wrong
ERR_UNKINO	Unknown interrupt number
ERR_UNKPROC	Incorrect reference to the load session in instruction WaitLoad

ERR_UNLOAD	Unload error in instruction UnLoad or WaitLoad
ERR_WAIT_MAXTIME	Timeout when executing a WaitDI or WaitUntil instruction
ERR_WHLSEARCH	No search stop

Characteristics

Errnum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Error recovery	RAPID Summary - *Error Recovery-* Basic Characteristics - *Error Recovery*
Data types in general, alias data types	Basic Characteristics - *Data Types*

errtype - Error type

errtype (error type) is used to specify an error type (gravity).

Description

Data of the type *errtype* represents the type (state change, warning, error) of an error message.
Refer to *User Guide - Error Management, System and Error Messages* for more information.

Example

```
VAR errdomain err_domain;
VAR num err_number;
VAR errtype err_type;
VAR trapdata err_data;
...
TRAP trap_err
    GetTrapData err_data;
    ReadErrData err_data, err_domain, err_number, err_type;
ENDTRAP
```

When an error is trapped to the trap routine *trap_err*, the error domain, the error number and the error type are saved into appropriate variables.

Predefined data

The following predefined constants can be used to specify an error type.

Tabell 2 Predefined error types

Name	Error Type	Value
TYPE_ALL	Any type of error (state change, warning, error)	0
TYPE_STATE	State change (operational message)	1
TYPE_WARN	Warning (such as RAPID recoverable error)	2
TYPE_ERR	Error	3

Characteristics

errtype is an alias data type for num and consequently inherits its characteristics.

Related information

	Described in:
Ordering an interrupt on errors	Instructions - *IError*
Error numbers	User's Guide - *System and error messages*
Alias data types	Basic Characteristics - *Data Types*

extjoint - Position of external joints

Extjoint is used to define the axis positions of external axes, positioners or workpiece manipulators.

Description

The robot can control up to six external axes in addition to its six internal axes, i.e. a total of twelve axes. The six external axes are logically denoted: a, b, c, d, e, f. Each such logical axis can be connected to a physical axis and, in this case, the connection is defined in the system parameters.

Data of the type *extjoint* is used to hold position values for each of the logical axes a - f.

For each logical axis connected to a physical axis, the positi*on is defin*ed as follows:

- For rotating axes – the position is defined as the rotation in degrees from the calibration position.
- For linear axes – the position is defined as the distance in mm from the calibration position.

If a logical axis is not connected to a physical one, the value 9E9 is used as a position value, indicating that the axis is not connected. At the time of execution, the position data of each axis is checked and it is checked whether or not the corresponding axis is connected. If the stored position value does not comply with the actual axis connection, the following applies:

- If the position is not defined in the position data (value is 9E9), the value will be ignored if the axis is connected and not activated. But if the axis is activated, it will result in an error.
- If the position is defined in the position data, although the axis is not connected, the value will be ignored.

If an external axis offset is used (instruction *EOffsOn* or *EOffsSet*), the positions are specified in the ExtOffs coordinate system.

Components

eax_a *(external axis a)* Data type: *num*

The position of the external logical axis “a”, expressed in degrees or mm (depending on the type of axis).

eax_b *(external axis b)* Data type: *num*

The position of the external logical axis “b”, expressed in degrees or mm (depending on the type of axis).

...

eax_f *(external axis f)* Data type: *num*

The position of the external logical axis “f”, expressed in degrees or mm (depending on the type of axis).

Example

VAR extjoint axpos10 := [11, 12.3, 9E9, 9E9, 9E9, 9E9] ;

The position of an external positioner, *axpos10,* is defined as follows:

- The position of the external logical axis “a” is set to 11, expressed in degrees or mm (depending on the type of axis).
- The position of the external logical axis “b” is set to 12.3, expressed in degrees or mm (depending on the type of axis).
- Axes c to f are undefined.

Structure

< dataobject of *extjoint* >
< *eax_a* of *num* >
< *eax_b* of *num* >
< *eax_c* of *num* >
< *eax_d* of *num* >
< *eax_e* of *num* >
< *eax_f* of *num* >

Related information

	Described in:
Position data	Data Types - *robtarget*
ExtOffs coordinate system	Instructions - *EOffsOn*

intnum - Interrupt identity

Intnum (interrupt numeric) is used to identify an interrupt.

Description

When a variable of type *intnum* is connected to a trap routine, it is given a specific value identifying the interrupt. This variable is then used in all dealings with the interrupt, such as when ordering or disabling an interrupt.

More than one interrupt identity can be connected to the same trap routine. The system variable *INTNO* can thus be used in a trap routine to determine the type of interrupt that occurs.

Examples

```
VAR intnum feeder_error;
.
CONNECT feeder_error WITH correct_feeder;
ISignalDI di1, 1, feeder_error;
```

An interrupt is generated when the input *di1* is set to *1*. When this happens, a call is made to the *correct_feeder* trap routine.

```
VAR intnum feeder1_error;
VAR intnum feeder2_error;
.
PROC init_interrupt();
.
   CONNECT feeder1_error WITH correct_feeder;
   ISignalDI di1, 1, feeder1_error;
   CONNECT feeder2_error WITH correct_feeder;
   ISignalDI di2, 1, feeder2_error;
.
ENDPROC
.
TRAP correct_feeder
   IF INTNO=feeder1_error THEN
   .
   ELSE
   .
   ENDIF
.
ENDTRAP
```

An interrupt is generated when either of the inputs *di1* or *di2* is set to *1*. A call is then made to the *correct_feeder* trap routine. The system variable INTNO is used in the trap routine to find out which type of interrupt has occurred.

Limitations

The maximum number of active variables of type *intnum* at any one time (between *CONNECT* and *IDelete*) is limited to 40.The maximum number of interrupts, in the queue for execution of *TRAP* routine at any one time, is limited to 30.

Characteristics

Intnum is an alias data type for *num* and thus inherits its properties.

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Alias data types	Basic Characteristics- *Data Types*

iodev - Serial channels and files

Iodev (I/O device) is used for serial channels, such as printers and files.

Description

Data of the type *iodev* contains a reference to a file or serial channel. It can be linked to the physical unit by means of the instruction *Open* and then used for reading and writing.

Example

```
VAR iodev file;
.
Open "HOME:/LOGDIR/INFILE.DOC", file\Read;
input := ReadNum(file);
```

The file *INFILE.DOC* is opened for reading. When reading from the file, *file* is used as a reference instead of the file name.

Characteristics

Iodev is a non-value data type.

Related information

	Described in:
Communication via serial channels	RAPID Summary - *Communication*
Configuration of serial channels	User's Guide - *System Parameters*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

jointtarget - Joint position data

Jointtarget is used to define the position that the robot and the external axes will move to with the instruction *MoveAbsJ*.

Description

Jointtarget defines each individual axis position, for both the robot and the external axes.

Components

robax *(robot axes)* Data type: *robjoint*

Axis positions of the robot axes in degrees.

Axis position is defined as the rotation in degrees for the respective axis (arm) in a positive or negative direction from the axis calibration position.

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

The position *is defin*ed as follows for each individual axis (*eax_a*, *eax_b* ... *eax_f*):

- For rotating axes, the position is defined as the rotation in degrees from the calibration position.
- For linear axes, the position is defined as the distance in mm from the calibration position.

External axes *eax_a* ... are logical axes. How the logical axis number and the physical axis number are related to each other is defined in the system parameters.

The value 9E9 is defined for axes which are not connected. If the axes defined in the position data differ from the axes that are actually connected on program execution, the following applies:

- If the position is not defined in the position data (value 9E9) the value will be ignored, if the axis is connected and not activated. But if the axis is activated it will result in error.
- If the position is defined in the position data yet the axis is not connected, the value is ignored.

Examples

CONST jointtarget calib_pos := [[0, 0, 0, 0, 0, 0], [0, 9E9, 9E9, 9E9, 9E9, 9E9]];

The normal calibration position for IRB2400 is defined in *calib_pos* by the data type *jointtarget.* The normal calibration position 0 (degrees or mm) is also defined for the external logical axis a. The external axes b to f are undefined.

Structure

< dataobject of *jointtarget* >
< *robax* of *robjoint* >
< *rax_1* of *num* >
< *rax_2* of *num* >
< *rax_3* of *num* >
< *rax_4* of *num* >
< *rax_5* of *num* >
< *rax_6* of *num* >
< *extax* of *extjoint* >
< *eax_a* of *num* >
< *eax_b* of *num* >
< *eax_c* of *num* >
< *eax_d* of *num* >
< *eax_e* of *num* >
< *eax_f* of *num* >

Related information

	Described in:
Move to joint position	Instructions - *MoveAbsJ*
Positioning instructions	RAPID Summary - *Motion*
Configuration of external axes	User's Guide - *System Parameters*

loaddata - Load data

Loaddata is used to describe loads attached to the mechanical interface of the robot (the robot's mounting flange).

Load data usually defines the payload (grip load is defined by the instruction *Grip-Load*) of the robot, i.e. the load held in the robot gripper. The tool load is specified in the tool data (*tooldata*) which includes load data.

Description

Specified loads are used to set up a model of the dynamics of the robot so that the robot movements can be controlled in the best possible way.

It is important to always define the actual tool load and when used, the payload of the robot too. Incorrect definitions of load data can result in overloading of the robot mechanical structure.

When incorrect load data is specified, it can often lead to the following consequences:

- If the value in the specified load data is greater than that of the value of the true load;
  -> The robot will not be used to its maximum capacity
  -> Impaired path accuracy including a risk of overshooting
  -> Risk of overloading the mechanical structure
- If the value in the specified load data is less than the value of the true load;
  -> Impaired path accuracy including a risk of overshooting
  -> Risk of overloading the mechanical structure

The payload is connected/disconnected using the instruction *GripLoad*.

Components

mass Data type: *num*

The weight of the load in kg.

cog *(centre of gravity)* Data type: *pos*

The centre of gravity of a tool load expressed in the wrist coordinate system. If a stationary tool is used, it means the centre of gravity for the tool holding the work object.

The centre of gravity of a payload expressed in the tool coordinate system. The object coordinate system when a stationary tool is used.

aom *(axes of moment)* Data type: *orient*

Tool load (Ref. to Figure 11)

The orientation of the coordinate system defined by the inertial axes of the tool load. Expressed in the wrist coordinate system as a quaternion (q1, q2, q3, q4). If a stationary tool is used, it means the inertial axes for the tool holding the work object.

The orientation of the tool load coordinate system must coincide with the orientation of the wrist coordinate system. **It must always be set to 1, 0, 0, 0.**

Pay load (Ref. to figure 1 and 2)

The orientation of the coordinate system defined by the inertial axes of the payload. Expressed in the tool coordinate system as a quaternion (q1, q2, q3, q4). The object coordinate system if a stationary tool is used.

The orientation of the payload coordinate system must coincide with the orientation of the wrist coordinate system. **It must always be set to 1, 0, 0, 0**.

Because of this limitation, the best way is to define the orientation of the tool coordinate system (*tool frame*) to coincide with the orientation of the wrist coordinate system.

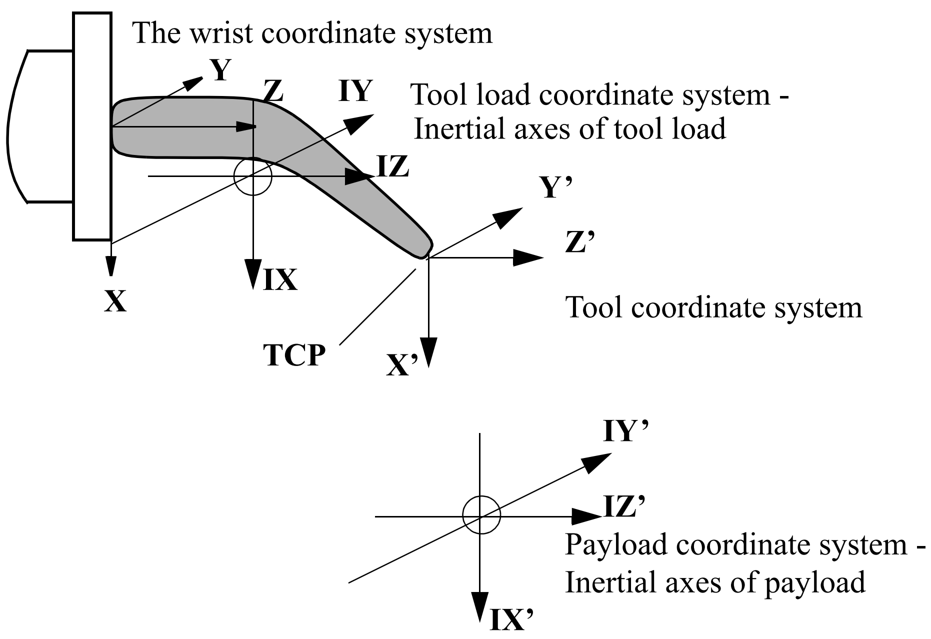


Figure 11 Restriction on the orientation of tool load and payload coordinate system.

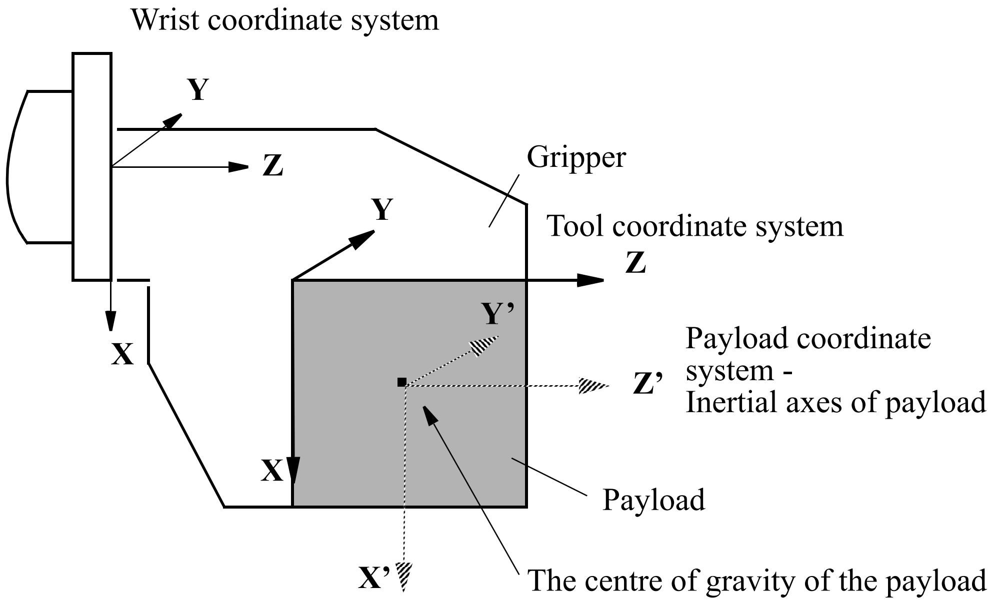


Figure 12 The centre of gravity and inertial axes of the payload.

ix *(inertia x)* Data type: *num*

The moment of inertia of the load around the x-axis of the tool load or payload coordinate system in kgm2.

Correct definition of the inertial moments will allow optimal utilisation of the path planner and axes control. This may be of special importance when handling large sheets of metal, etc. All inertial moments of inertia *ix*, *iy* and *iz* equal to 0 kgm2 imply a point mass.

Normally, the inertial moments must only be defined when the distance from the mounting flange to the centre of gravity is less than the dimension of the load (see Figure 13).

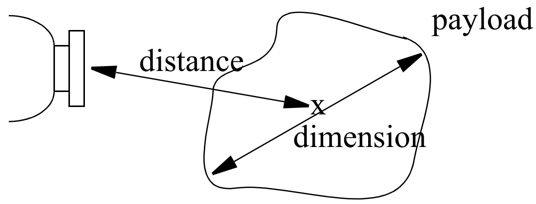


Figure 13 The moment of inertia must normally be defined when the distance is less than the load dimension.

iy *(inertia y)* Data type: *num*

The inertial moment of the load around the y-axis, expressed in kgm2.

For more information, see *ix*.

iz *(inertia z)* Data type: *num*

The inertial moment of the load around the z-axis, expressed in kgm2.

For more information, see *ix*.

Examples

PERS loaddata piece1 := [5, [50, 0, 50], [1, 0, 0, 0], 0, 0, 0];

The payload in Figure 11 is described using the following values:

- Weight *5* kg.
- The centre of gravity is x = *50*, y = *0* and z = *50* mm in the tool coordinate system.
- The payload is a point mass.

Set gripper;
WaitTime 0.3;
GripLoad piece1;

Connection of the payload, *piece1*, specified at the same time as the robot grips the load *piece1*.

Reset gripper;
WaitTime 0.3;
GripLoad load0;

Disconnection of a payload, specified at the same time as the robot releases a payload.

Limitations

The payload should only be defined as a persistent variable (PERS) and not within a routine. Current values are then saved when storing the program on diskette and are retrieved on loading.

Arguments of the type load data in the *GripLoad* instruction should only be an entire persistent (not array element or record component).

Predefined data

The load *load0* defines a payload, the weight of which is equal to 0 kg, i.e. no load at all. This load is used as the argument in the instruction *GripLoad* to disconnect a payload.

The load *load0* can always be accessed from the program, but cannot be changed (it is stored in the system module *BASE*).

PERS loaddata load0 := [0.001, [0, 0, 0.001], [1, 0, 0, 0],0, 0 ,0];

Structure

< dataobject of *loaddata* >
< *mass* of *num* >
< *cog* of *pos* >
< *x* of *num* >
< *y* of *num* >
< *z* of *num* >
< *aom* of *orient* >
< *q1* of *num* >
< *q2* of *num* >
< *q3* of *num* >
< *q4* of *num* >
< *ix* of *num* >
< *iy* of *num* >
< *iz* of *num* >

Related information

	Described in:
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Definition of tool loads	Data Types - *tooldata*
Activation of payload	Instructions - *GripLoad*

loadsession - Program load session

Loadsession is used to define different load sessions of RAPID program modules.

Description

Data of the type *loadsession* is used in the instructions *StartLoad* and *WaitLoad*, to identify the load session. *Loadsession* only contains a reference to the load session.

Characteristics

Loadsession is a *non-value* data type and cannot be used in value-oriented operations.

Related information

	Described in:
Loading program modules during execution	Instructions - *StartLoad, WaitLoad*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

mecunit - Mechanical unit

Mecunit is used to define the different mechanical units which can be controlled and accessed from the robot and the program.

The names of the mechanical units are defined in the system parameters and, consequently, must not be defined in the program.

Description

Data of the type *mecunit* only contains a reference to the mechanical unit.

Limitations

Data of the type *mecunit* must not be defined in the program. The data type can, on the other hand, be used as a parameter when declaring a routine.

Predefined data

The mechanical units defined in the system parameters can always be accessed from the program (installed data).

Characteristics

Mecunit is a *non-value* data type. This means that data of this type does not permit value-oriented operations.

Related information

	Described in:
Activating/Deactivating mechanical units	Instructions - *ActUnit, DeactUnit*
Configuration of mechanical units	User's Guide - *System Parameters*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

motsetdata - Motion settings data

Motsetdata is used to define a number of motion settings that affect all positioning instructions in the program:

- Max. velocity and velocity override
- Acceleration data
- Behavior around singular points
- Management of different robot configurations
- Override of path resolution
- Motion supervision
- Limitation of acceleration/deceleration
- Tool reorientation during circle path

This data type does not normally have to be used since these settings can only be set using the instructions *VelSet*, *AccSet*, *SingArea*, *ConfJ, ConfL*, *PathResol*, *MotionSup, PathAccLim, CirPathMode* and *WorldAccLim*.

The current values of these motion settings can be accessed using the system variable *C_MOTSET.*

Description

The current motion settings (stored in the system variable *C_MOTSET*) affect all movements.

Components

vel.oride Data type: *veldata/num*

Velocity as a percentage of programmed velocity.

vel.max Data type: *veldata/num*

Maximum velocity in mm/s.

acc.acc Data type: *accdata/num*

Acceleration and deceleration as a percentage of the normal values.

acc.ramp Data type: *accdata/num*

The rate by which acceleration and deceleration increases as a percentage of the normal values.

sing.wrist Data type: *singdata/bool*

The orientation of the tool is allowed to deviate somewhat in order to prevent wrist singularity.

sing.arm Data type: *singdata/bool*

The orientation of the tool is allowed to deviate somewhat in order to prevent arm singularity (not implemented).

sing.base Data type: *singdata/bool*

The orientation of the tool is not allowed to deviate.

conf.jsup Data type: *confsupdata/bool*

Supervision of joint configuration is active during joint movement.

conf.lsup Data type: *confsupdata/bool*

Supervision of joint configuration is active during linear and circular movement.

conf.ax1 Data type: *confsupdata/num*

Maximum permitted deviation in degrees for axis 1 (not used in this version).

conf.ax4 Data type: *confsupdata/num*

Maximum permitted deviation in degrees for axis 4 (not used in this version).

conf.ax6 Data type: *confsupdata/num*

Maximum permitted deviation in degrees for axis 6 (not used in this version).

pathresol Data type: *num*

Current override in percentage of the configured path resolution.

motionsup Data type: *bool*

Mirror RAPID status (TRUE = On and FALSE = Off) of motion supervision function.

tunevalue Data type: *num*

Current RAPID override as a percentage of the configured tunevalue for the motion supervision function.

acclim Data type**:** *bool*

Limitation of tool acceleration along the path. (TRUE = limitation and FALSE = no limitation).

accmax Data type: *num*

TCP acceleration limitation in m/s^2. If *acclim* is FALSE, the value is always set to -1.

decellim Data type: *bool*

Limitation of tool deceleration along the path. TRUE = limitation and FALSE = no limitation).

decelmax Data type: *num*

TCP deceleration limitation in m/s^2. If *decellim* is FALSE, the value is always set to -1.

cirpathreori Data type: *num*

Tool reorientation during circle path:
0 = Standard method with interpolation in path frame
1 = Modified method with interpolation in object frame
2 = Modified method with programmed tool orientation in *CirPoint*

worldacclim Data type: *bool*

Limitation of acceleration in world coordinate system. (TRUE = limitation and FALSE = no limitation).

worldaccmax Data type: *num*

Limitation of acceleration in world coordinate system in m/s^2. If *worldacclim* is FALSE, the value is always set to -1.

Limitations

One and only one of the components *sing.wrist*, *sing.arm* or *sing.base* may have a value equal to TRUE.

Example

```
IF C_MOTSET.vel.oride > 50 THEN
    ...
ELSE
    ...
ENDIF
```

Different parts of the program are executed depending on the current velocity override.

Predefined data

C_MOTSET describes the current motion settings of the robot and can always be accessed from the program (installed data). *C_MOTSET*, on the other hand, can only be changed using a number of instructions, not by assignment.

The following default values for motion parameters are set

- at a cold start-up
- when a new program is loaded
- when starting program execution from the beginning.

```
PERS motsetdata C_MOTSET := [
    [ 100, 500 ],                   -> veldata
    [ 100, 100 ],                   -> accdata
    [ FALSE, FALSE, TRUE ],         -> singdata
    [ TRUE, TRUE, 30, 45, 90],      -> confsupdata
    100 ,                           -> path resolution
    TRUE,                           -> motionsup
    100,                            -> tunevalue
    FALSE,                          -> acclim
    -1,                             -> accmax
    FALSE,                          -> decellim
    -1,                             -> decelmax
    0,                              -> cirpathreori
    FALSE,                          -> worldacclim
    -1];                            -> worldaccmax
```

Structure

<dataobject of *motsetdata*>

- <*vel* of *veldata* > -> Affected by instruction VelSet
    - < *oride* of *num* >
    - < *max* of *num* >
- <*acc* of *accdata* > -> Affected by instruction AccSet
    - < *acc* of *num* >
    - < *ramp* of *num* >
- <*sing* of *singdata* > -> Affected by instruction SingArea
    - < *wrist* of *bool* >
    - < *arm* of *bool* >
    - < *base* of *bool* >
- <*conf* of *confsupdata* > -> Affected by instructions ConfJ and ConfL
    - < *jsup* of *bool* >
    - <*lsup* of *bool* >
    - < *ax1* of *num* >
    - < *ax4* of *num* >
    - < *ax6* of *num* >
- <*pathresol* of *num*> -> Affected by instruction PathResol
- <*motionsup* of *bool*> -> Affected by instruction MotionSup
- <*tunevalue* of *num*> -> Affected by instruction MotionSup
- <*acclim* of *bool*> -> Affected by instruction PathAccLim
- <*accmax* of *num*> -> Affected by instruction PathAccLim
- <*decellim* of *bool*> -> Affected by instruction PathAccLim
- <*decelmax* of *num*> -> Affected by instruction PathAccLim
- <*cirpathreori* of *num*> -> Affected by instruction CirPathMode
- <*worldacclim* of *bool*> -> Affected by instruction WorldAccLim
- <*worldaccmax* of *num*> -> Affected by instruction WorldAccLim

Related information

	Described in:
Instructions for setting motion parameters	RAPID Summary - *Motion Settings*

num - Numeric values (registers)

Num is used for numeric values; e.g. counters.

Description

The value of the *num* data type may be

- an integer; e.g. -5,
- a decimal number; e.g. 3.45.

It may also be written exponentially; e.g.2E3 (= 2*103 = 2000), 2.5E-2 (= 0.025).

Integers between -8388607 and +8388608 are always stored as exact integers.

Decimal numbers are only approximate numbers and should not, therefore, be used in *is equal to* or *is not equal to* comparisons. In the case of divisions, and operations using decimal numbers, the result will also be a decimal number; i.e. not an exact integer.

E.g.

```
a := 10;
b := 5;
IF a/b=2 THEN      As the result of a/b is not an integer,
                   this condition is not necessarily
...                satisfied.
```

Example

```
VAR num reg1;
  .
reg1 := 3;
```

reg1 is assigned the value *3*.

```
a := 10 DIV 3;
b := 10 MOD 3;
```

Integer division where *a* is assigned an integer (=3) and *b* is assigned the remainder (=1).

Predefined data

The constant pi (π) is already defined in the system module *BASE*.

CONST num pi := 3.1415926;

The constants EOF_BIN and EOF_NUM are already defined in the system.

CONST num EOF_BIN := -1;

CONST num EOF_NUM := 9.998E36;

Related information

	Described in:
Numeric expressions	Basic Characteristics - *Expressions*
Operations using numeric values	Basic Characteristics - *Expressions*

o_jointtarget - Original joint position data

o_jointtarget (original joint target) is used in combination with the function *Absolute Limit Modpos*. When this function is used to modify a position, the original position is stored as a data of the type *o_jointtarget*.

Description

If the function *Absolute Limit Modpos* is activated and a named position in a movement instruction is modified with the function *Modpos, then the original* programmed position is saved.

Example of a program before *Modpos*:

```
CONST jointtarget jpos40:= [[0, 0, 0, 0, 0, 0],
                            [0, 9E9, 9E9, 9E9, 9E9, 9E9]];
...
MoveAbsJ jpos40, v1000, z50, tool1;
```

The same program after *ModPos in which* the point *jpos40 is* corrected to 2 degrees for robot axis 1:

```
CONST jointtarget jpos40     := [[2, 0, 0, 0, 0, 0],
                                [0, 9E9, 9E9, 9E9, 9E9, 9E9]];
CONST o_jointtarget o_jpos40 := [[0, 0, 0, 0, 0, 0],
                                [0, 9E9, 9E9, 9E9, 9E9, 9E9]];
...
MoveAbsJ jpos40, v1000, z50, tool1;
```

The original programmed point has now been saved in *o_jpos40* (by the data type *o_jointtarget*) and the modified point saved in *jpos40* (by the data type *jointtarget*).

By saving the original programmed point, the robot can monitor that further *Modpos* of the point in question are within the acceptable limits from the original programmed point.

The fixed name convention means that an original programmed point with the name *xxxxx* is saved with the name *o_xxxxx* by using *Absolute Limit Modpos*.

Components

robax *(robot axes)* Data type: *robjoint*

Axis positions of the robot axes in degrees.

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

Structure

< dataobject of *o_jointtarget* >
< *robax* of *robjoint*>
< *rax_1* of *num* >
< *rax_2* of *num* >
< *rax_3* of *num* >
< *rax_4* of *num* >
< *rax_5* of *num* >
< *rax_6* of *num* >
< *extax* of *extjoint* >
< *eax_a* of *num* >
< *eax_b* of *num* >
< *eax_c* of *num* >
< *eax_d* of *num* >
< *eax_e* of *num* >
< *eax_f* of *num* >

Related information

	Described in:
Position data	Data Types - *Jointtarget*
Configuration of Limit Modpos	User's Guide - *System Parameters*

o_robtarget - Original position data

o_robtarget (*original robot target*) is used in combination with the function *Absolute Limit Modpos*. When this function is used to modify a position, the original position is stored as a data of the type *o_robtarget*.

Description

If the function *Absolute Limit Modpos* is activated and a named position in a movement instruction is modified with the function *Modpos, then the original* programmed position is saved.

Example of a program before *Modpos*:

```
CONST robtarget p50  := [[500, 500, 500], [1, 0, 0, 0], [1, 1, 0, 0],
                          [500, 9E9, 9E9, 9E9, 9E9, 9E9] ];
...
MoveL p50, v1000, z50, tool1;
```

The same program after *ModPos in which* the point *p50 is* corrected to 502 in the x-direction:

```
CONST robtarget p50  := [[502, 500, 500], [1, 0, 0, 0], [1, 1, 0, 0],
                          [500, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST o_robtarget o_p50 := [[500, 500, 500], [1, 0, 0, 0], [1, 1, 0, 0],
                          [ 500, 9E9, 9E9, 9E9, 9E9, 9E9] ];
...
MoveL p50, v1000, z50, tool1;
```

The original programmed point has now been saved in *o_p50* (by the data type *o_robtarget*) and the modified point saved in *p50* (by the data type *robtarget*).

By saving the original programmed point, the robot can monitor that further *Modpos* of the point in question are within the acceptable limits from the original programmed point.

The fixed name convention means that an original programmed point with the name *xxxxx* is saved with the name *o_xxxxx* by using *Absolute Limit Modpos*.

Components

trans *(translation)* Data type: *pos*

The position (x, y and z) of the tool centre point expressed in mm.

rot *(rotation)* Data type: *orient*

The orientation of the tool, expressed in the form of a quaternion (q1, q2, q3 and q4).

robconf *(robot configuration)* Data type: *confdata*

The axis configuration of the robot (cf1, cf4, cf6 and cfx).

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

Structure

< dataobject of *o_robtarget* >
< *trans* of *pos* >
< *x* of *num* >
< *y* of *num* >
< *z* of *num* >
< *rot* of *orient* >
< *q1* of *num* >
< *q2* of *num* >
< *q3* of *num* >
< *q4* of *num* >
< *robconf* of *confdata* >
< *cf1* of *num* >
< *cf4* of *num* >
< *cf6* of *num* >
< *cfx* of *num* >
< *extax* of *extjoint* >
< *eax*_a of *num* >
< *eax*_b of *num* >
< *eax*_c of *num* >
< *eax*_d of *num* >
< *eax*_e of *num* >
< *eax*_f of *num* >

Related information

	Described in:
Position data	Data Types - *Robtarget*
Configuration of Limit Modpos	User's Guide - *System Parameters*

opnum - Comparison operator

opnum is used to represent an operator for comparisons in arguments to RAPID functions or instructions.

Description

An *opnum* constant is intended to be used to define the type of comparison, when checking values in generic instructions.

Example

TriggCheckIO checkgrip, 100, airok, EQ, 1, intno1;

Predefined data

The following symbolic constants of the data type *opnum* are predefined and can be used to define the type of comparison used for instance in instruction *TriggCheckIO*.

Value	Symbolic constant	Comment
1	LT	Less than
2	LTEQ	Less than or equal to
3	EQ	Equal to
4	NOTEQ	Not equal to
5	GTEQ	Greater than or equal to
6	GT	Greather than

Characteristics

opnum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Data types in general, alias data types	Basic Characteristics - *Data Types*

orient - Orientation

Orient is used for orientations (such as the orientation of a tool) and rotations (such as the rotation of a coordinate system).

Description

The orientation is described in the form of a quaternion which consists of four elements: *q1*, *q2*, *q3* and *q4*. For more information on how to calculate these, see below.

Components

q1 Data type: *num*

Quaternion 1.

q2 Data type: *num*

Quaternion 2.

q3 Data type: *num*

Quaternion 3.

q4 Data type: *num*

Quaternion 4.

Example

```
VAR orient orient1;
.
orient1 := [1, 0, 0, 0];
```

The *orient1* orientation is assigned the value q1=1, q2-q4=0; this corresponds to no rotation.

Limitations

The orientation must be normalised; i.e. the sum of the squares must equal 1:

$$q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1$$

What is a Quaternion?

The orientation of a coordinate system (such as that of a tool) can be described by a rotational matrix that describes the direction of the axes of the coordinate system in relation to a reference system (see Figure 14).

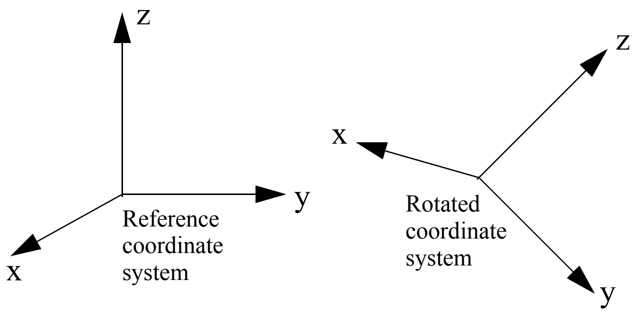


Figure 14 The rotation of a coordinate system is described by a quaternion.

The rotated coordinate systems axes (**x**, **y**, **z**) are vectors which can be expressed in the reference coordinate system as follows:

x = (x1, x2, x3)

y = (y1, y2, y3)

z = (z1, z2, z3)

This means that the x-component of the x-vector in the reference coordinate system will be x1, the y-component will be x2, etc.

These three vectors can be put together in a matrix, a rotational matrix, where each of the vectors form one of the columns:

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

A quaternion is just a more concise way to describe this rotational matrix; the quaternions are calculated based on the elements of the rotational matrix:

$$q1 = \frac{\sqrt{x_1 + y_2 + z_3 + 1}}{2}$$

$$q2 = \frac{\sqrt{x_1 - y_2 - z_3 + 1}}{2} \qquad \text{sign } q2 = \text{sign } (y3\text{-}z2)$$

$$q3 = \frac{\sqrt{y_2 - x_1 - z_3 + 1}}{2} \qquad \text{sign } q3 = \text{sign } (z1\text{-}x3)$$

$$q4 = \frac{\sqrt{z_3 - x_1 - y_2 + 1}}{2} \qquad \text{sign } q4 = \text{sign } (x2\text{-}y1)$$

Example 1

A tool is orientated so that its Z'-axis points straight ahead (in the same direction as the X-axis of the base coordinate system). The Y'-axis of the tool corresponds to the Y-axis of the base coordinate system (see Figure 15). How is the orientation of the tool defined in the position data (robtarget)?

The orientation of the tool in a programmed position is normally related to the coordinate system of the work object used. In this example, no work object is used and the base coordinate system is equal to the world coordinate system. Thus, the orientation is related to the base coordinate system.

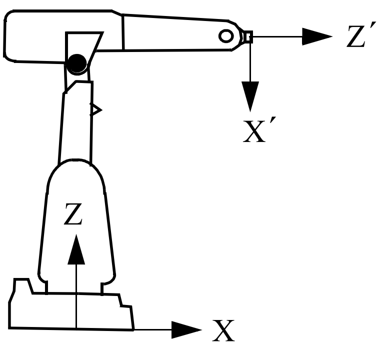


Figure 15 The direction of a tool in accordance with example 1.

The axes will then be related as follows:

- **x'** = -**z** = (0, 0, -1)
- **y'** = **y** = (0, 1, 0)
- **z'** = **x** = (1, 0, 0)

Which corresponds to the following rotational matrix: $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$

The rotational matrix provides a corresponding quaternion:

$$q1 = \frac{\sqrt{0+1+0+1}}{2} = \frac{\sqrt{2}}{2} = 0.707$$

$$q2 = \frac{\sqrt{0-1-0+1}}{2} = 0$$

$$q3 = \frac{\sqrt{1-0-0+1}}{2} = \frac{\sqrt{2}}{2} = 0.707$$

sign q3 = sign (1+1) = +

$$q4 = \frac{\sqrt{0-0-1+1}}{2} = 0$$

Example 2

The direction of the tool is rotated 30o about the X'- and Z'-axes in relation to the wrist coordinate system (see Figure 15). How is the orientation of the tool defined in the tool data?

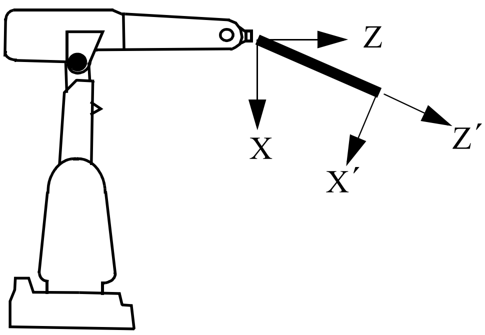


Figure 16 The direction of the tool in accordance with example 2.

The axes will then be related as follows:

- **x**' = (cos30o, 0, -sin30o)
- **x**' = (0, 1, 0)
- **x**' = (sin30o, 0, cos30o)

Which corresponds to the following rotational matrix: $\begin{bmatrix} \cos 30° & 0 & \sin 30° \\ 0 & 1 & 0 \\ -\sin 30° & 0 & \cos 30° \end{bmatrix}$

The rotational matrix provides a corresponding quaternion:

$$q1 = \frac{\sqrt{\cos 30° + 1 + \cos 30° + 1}}{2} = 0.965926$$

$$q2 = \frac{\sqrt{\cos 30° - 1 - \cos 30° + 1}}{2} = 0$$

$$q3 = \frac{\sqrt{1 - \cos 30° - \cos 30° + 1}}{2} = 0.258819$$

sign q3 = sign (sin30o+sin30o) = +

$$q4 = \frac{\sqrt{\cos 30° - \cos 30° - 1 + 1}}{2} = 0$$

Structure

<dataobject of *orient*>
<*q1* of *num*>
<*q2* of *num*>
<*q3* of *num*>
<*q4* of *num*>

Related information

	Described in:
Operations on orientations	Basic Characteristics - *Expressions*

pos - Positions (only X, Y and Z)

Pos is used for positions (only X, Y and Z).

The *robtarget* data type is used for the robot's position including the orientation of the tool and the configuration of the axes.

Description

Data of the type *pos* describes the coordinates of a position: X, Y and Z.

Components

x Data type: *num*

The X-value of the position.

y Data type: *num*

The Y-value of the position.

z Data type: *num*

The Z-value of the position.

Examples

```
VAR pos pos1;
.
pos1 := [500, 0, 940];
```

The *pos1* position is assigned the value: X=500 mm, Y=0 mm, Z=940 mm.

```
pos1.x := pos1.x + 50;
```

The *pos1* position is shifted *50* mm in the X-direction.

Structure

<dataobject of *pos*>
<*x* of *num*>
<*y* of *num*>
<*z* of *num*>

Related information

	Described in:
Operations on positions	Basic Characteristics - *Expressions*
Robot position including orientation	Data Types- *robtarget*

pose - Coordinate transformations

Pose is used to change from one coordinate system to another.

Description

Data of the type *pose* describes how a coordinate system is displaced and rotated around another coordinate system. The data can, for example, describe how the tool coordinate system is located and oriented in relation to the wrist coordinate system.

Components

trans *(translation)* Data type: *pos*

The displacement in position (x, y and z) of the coordinate system.

rot *(rotation)* Data type: *orient*

The rotation of the coordinate system.

Example

```
VAR pose frame1;
.
frame1.trans := [50, 0, 40];
frame1.rot := [1, 0, 0, 0];
```

The *frame1* coordinate transformation is assigned a value that corresponds to a displacement in position, where X=*50* mm, Y=*0* mm, Z=*40* mm; there is, however, no rotation.

Structure

<dataobject of *pose*>
<*trans* of *pos*>
<*rot* of *orient*>

Related information

	Described in:
What is a Quaternion?	*Data Types - orient*

progdisp - Program displacement

Progdisp is used to store the current program displacement of the robot and the external axes.

This data type does not normally have to be used since the data is set using the instructions *PDispSet, PDispOn, PDispOff, EOffsSet, EOffsOn and EOffsOff*. It is only used to temporarily store the current value for later use.

Description

The current values for program displacement can be accessed using the system variable *C_PROGDISP*.

For more information, see the instructions *PDispSet, PDispOn, EOffsSet and EOffsOn*.

Components

pdisp *(program displacement)* Data type: *pose*

The program displacement for the robot, expressed using a translation and an orientation. The translation is expressed in mm.

eoffs *(external offset)* Data type: *extjoint*

The offset for each of the external axes. If the axis is linear, the value is expressed in mm; if it is rotating, the value is expressed in degrees.

Example

```
VAR progdisp progdisp1;
.
SearchL sen1, psearch, p10, v100, tool1;
PDispOn \ExeP:=psearch, *, tool1;
EOffsOn \ExeP:=psearch, *;
.
progdisp1:=C_PROGDISP;
PDispOff;
EOffsOff;
.
PDispSet progdisp1.pdisp;
EOffsSet progdisp1.eoffs;
```

First, a program displacement is activated from a searched position. Then, it is temporarily deactivated by storing the value in the variable *progdisp1* and, later on, re-activated using the instructions *PDispSet* and *EOffsSet*.

Predefined data

The system variable *C_PROGDISP* describes the current program displacement of the robot and external axes, and can always be accessed from the program (installed data). *C_PROGDISP*, on the other hand, can only be changed using a number of instructions, not by assignment.

Structure

< dataobject of *progdisp* >

<*pdisp* of *pose*>

< *trans* of *pos* >

< *x* of *num* >

< *y* of *num* >

< *z* of *num* >

< *rot* of *orient* >

< *q1* of *num* >

< *q2* of *num* >

< *q3* of *num* >

< *q4* of *num* >

< *eoffs* of *extjoint* >

< *eax_a* of *num* >

< *eax_b* of *num* >

< *eax_c* of *num* >

< *eax_d* of *num* >

< *eax_e* of *num* >

< *eax_f* of *num* >

Related information

	Described in:
Instructions for defining program displacement	RAPID Summary - *Motion Settings*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

robjoint - Joint position of robot axes

Robjoint is used to define the axis position in degrees of the robot axes.

Description

Data of the type *robjoint* is used to store axis positions in degrees of the robot axes 1 to 6. Axis position is defined as the rotation in degrees for the respective axis (arm) in a positive or negative direction from the axis calibration position.

Components

rax_1 *(robot axis 1)* Data type: *num*

The position of robot axis 1 in degrees from the calibration position.

...

rax_6 *(robot axis 6)* Data type: *num*

The position of robot axis 6 in degrees from the calibration position.

Structure

< dataobject of *robjoint* >
< *rax_1* of *num* >
< *rax_2* of *num* >
< *rax_3* of *num* >
< *rax_4* of *num* >
< *rax_5* of *num* >
< *rax_6* of *num* >

Related information

	Described in:
Joint position data	Data Types - *jointtarget*
Move to joint position	Instructions - *MoveAbsJ*

robtarget - Position data

Robtarget (*robot target*) is used to define the position of the robot and external axes.

Description

Position data is used to define the position in the positioning instructions to which the robot and external axes are to move.

As the robot is able to achieve the same position in several different ways, the axis configuration is also specified. This defines the axis values if these are in any way ambiguous, for example:

- if the robot is in a forward or backward position,
- if axis 4 points downwards or upwards,
- if axis 6 has a negative or positive revolution.

The position is defined based on the coordinate system of the work object, including any program displacement. If the position is programmed with some other work object than the one used in the instruction, the robot will not move in the expected way. Make sure that you use the same work object as the one used when programming positioning instructions. Incorrect use can injure someone or damage the robot or other equipment.

Components

trans *(translation)* Data type: *pos*

The position (x, y and z) of the tool centre point expressed in mm.

The position is specified in relation to the current object coordinate system, including program displacement. If no work object is specified, this is the world coordinate system.

rot *(rotation)* Data type: *orient*

The orientation of the tool, expressed in the form of a quaternion (q1, q2, q3 and q4).

The orientation is specified in relation to the current object coordinate system, including program displacement. If no work object is specified, this is the world coordinate system.

robconf *(robot configuration)* Data type: *confdata*

The axis configuration of the robot (cf1, cf4, cf6 and cfx). This is defined in the form of the current quarter revolution of axis 1, axis 4 and axis 6. The first positive quarter revolution 0 to 90 o is defined as 0. The component cfx is only used for the robot model IRB5400.

For more information, see data type *confdata*.

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

The position *is defin*ed as follows for each individual axis (*eax_a*, *eax_b* ... *eax_f*):

- For rotating axes, the position is defined as the rotation in degrees from the calibration position.
- For linear axes, the position is defined as the distance in mm from the calibration position.

External axes *eax_a* ... are logical axes. How the logical axis number and the physical axis number are related to each other is defined in the system parameters.

The value 9E9 is defined for axes which are not connected. If the axes defined in the position data differ from the axes that are actually connected on program execution, the following applies:

- If the position is not defined in the position data (value 9E9), the value will be ignored, if the axis is connected and not activated. But if the axis is activated, it will result in an error.
- If the position is defined in the position data although the axis is not connected, the value is ignored.

Examples

CONST robtarget p15 := [[600, 500, 225.3], [1, 0, 0, 0], [1, 1, 0, 0],
[11, 12.3, 9E9, 9E9, 9E9, 9E9]];

A position *p15* is defined as follows:

- The position of the robot: *x = 600*, *y = 500* and *z = 225.3* mm in the object coordinate system.
- The orientation of the tool in the same direction as the object coordinate system.
- The axis configuration of the robot: axes 1 and 4 in position 90-180o, axis 6 in position 0-90o.
- The position of the external logical axes, a and b, expressed in degrees or mm (depending on the type of axis). Axes c to f are undefined.

```
VAR robtarget p20;
. . .
p20 := CRobT();
p20 := Offs(p20,10,0,0);
```

The position *p20* is set to the same position as the current position of the robot by calling the function *CRobT*. The position is then moved *10* mm in the x-direction.

Limitations

When using the configurable edit function *Absolute Limit Modpos*, the number of characters in the name of the data of the type *robtarget*, is limited to 14 (in other cases 16).

Structure

< dataobject of *robtarget* >
    < *trans* of *pos* >
        < *x* of *num* >
        < *y* of *num* >
        < *z* of *num* >
    < *rot* of *orient* >
        < *q1* of *num* >
        < *q2* of *num* >
        < *q3* of *num* >
        < *q4* of *num* >
    < *robconf* of *confdata* >
        < *cf1* of *num* >
        < *cf4* of *num* >
        < *cf6* of *num* >
        < *cf*x of *num* >
    < *extax* of *extjoint* >
        < *eax*_a of *num* >
        < *eax*_b of *num* >
        < *eax*_c of *num* >
        < *eax*_d of *num* >
        < *eax*_e of *num* >
        < *eax*_f of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Handling configuration data	Motion and I/O Principles - *Robot Configuration*
Configuration of external axes	User's Guide - *System Parameters*
What is a quaternion?	Data Types - *Orient*

shapedata - World zone shape data

shapedata is used to describe the geometry of a world zone.

Description

World zones can be defined in 4different geometrical shapes:

- a straight box, with all sides parallel to the world coordinate system and defined by a *WZBoxDef* instruction
- a sphere, defined by a *WZSphDef* instruction
- a cylinder, parallel to the z axis of the world coordinate system and defined by a *WZCylDef* instruction
- a joint space area for robot and/or external axes, defined by the instruction *WZHomeJointDef* or *WZLimJointDef*

The geometry of a world zone *is defined by one of the previous instructions and the action of a world zone is defined by the instruction WZLimSup or WZDOSet.*

Example

```
VAR wzstationary pole;
VAR wzstationary conveyor;
...
PROC ...
    VAR shapedata volume;
    ...
    WZBoxDef \Inside, volume, p_corner1, p_corner2;
    WZLimSup \Stat, conveyor, volume;
    WZCylDef \Inside, volume, p_center, 200, 2500;
    WZLimSup \Stat, pole, volume;
ENDPROC
```

A *conveyor* is defined as a box and the supervision for this area is activated. A *pole* is defined as a cylinder and the supervision of this zone is also activated. If the robot reaches one of these areas, the motion is stopped.

Characteristics

shapedata is a non-value data type.

Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Define box-shaped world zone	Instructions - *WZBoxDef*
Define sphere-shaped world zone	Instructions - *WZSphDef*
Define cylinder-shaped world zone	Instructions - *WZCylDef*
Define a world zone for home joints	Instruction - *WZHomeJointDef*
Define a world zone for limit joints	Instruction - *WZLimJointDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

signal*xx* - Digital and analog signals

Data types within *signalxx* are used for digital and analog input and output signals.

The names of the signals are defined in the system parameters and are consequently not to be defined in the program.

Description

<u>Data type</u>	<u>Used for</u>
signalai	analog input signals
signalao	analog output signals
signaldi	digital input signals
signaldo	digital output signals
signalgi	groups of digital input signals
signalgo	groups of digital output signals

Variables of the type *signalxo* only contain a reference to the signal. The value is set using an instruction, e.g. *DOutput*.

Variables of the type *signalxi* contain a reference to a signal as well as the possibility to retrieve the value directly in the program, if used in value context.

The value of an input signal can be read directly in the program, e.g. :

```
! Digital input
IF di1 = 1 THEN ...

! Digital group input
IF gi1 = 5 THEN ...

! Analog input
IF ai1 > 5.2 THEN ...
```

Limitations

Data of the data type *signalxx* must not be defined in the program. However, if this is in fact done, an error message will be displayed as soon as an instruction or function that refers to this signal is executed. The data type can, on the other hand, be used as a parameter when declaring a routine.

Predefined data

The signals defined in the system parameters can always be accessed from the program by using the predefined signal variables (installed data). It should however be noted that if other data with the same name is defined, these signals cannot be used.

Characteristics

Signalxo is a *non-value* data type. Thus, data of this type does not permit value-oriented operations.

Signalxi is a semi-value data type.

Related information

	Described in:
Summary input/output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general I/O Principles	Motion and I/O Principles -
Configuration of I/O	User's Guide - *System Parameters*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

speeddata - Speed data

Speeddata is used to specify the velocity at which both the robot and the external axes move.

Description

Speed data defines the velocity:

- at which the tool centre point moves,
- of the reorientation of the tool,
- at which linear or rotating external axes move.

When several different types of movement are combined, one of the velocities often limits all movements. The velocity of the other movements will be reduced in such a way that all movements will finish executing at the same time.

The velocity is also restricted by the performance of the robot. This differs, depending on the type of robot and the path of movement.

Components

v_tcp *(velocity tcp)* Data type: *num*

The velocity of the tool centre point (TCP) in mm/s.

If a stationary tool or coordinated external axes are used, the velocity is specified relative to the work object.

v_ori *(velocity orientation)* Data type: *num*

The velocity of reorientation about the TCP expressed in degrees/s.

If a stationary tool or coordinated external axes are used, the velocity is specified relative to the work object.

v_leax *(velocity linear external axes)* Data type: *num*

The velocity of linear external axes in mm/s.

v_reax *(velocity rotational external axes)* Data type: *num*

The velocity of rotating external axes in degrees/s.

Example

VAR speeddata vmedium := [1000, 30, 200, 15];

The speed data *vmedium* is defined with the following velocities:

- *1000* mm/s for the TCP.
- *30* degrees/s for reorientation of the tool.
- *200* mm/s for linear external axes.
- *15* degrees/s for rotating external axes.

vmedium.v_tcp := 900;

The velocity of the TCP is changed to *900* mm/s.

Predefined data

A number of speed data are already defined in the system module *BASE*.

Name	TCP speed	Orientation	Linear ext. axis	Rotating ext. axis
v5	5 mm/s	500o/s	5000 mm/s	1000o/s
v10	10 mm/s	500o/s	5000 mm/s	1000o/s
v20	20 mm/s	500o/s	5000 mm/s	1000o/s
v30	30 mm/s	500o/s	5000 mm/s	1000o/s
v40	40 mm/s	500o/s	5000 mm/s	1000o/s
v50	50 mm/s	500o/s	5000 mm/s	1000o/s
v60	60 mm/s	500o/s	5000 mm/s	1000o/s
v80	80 mm/s	500o/s	5000 mm/s	1000o/s
v100	100 mm/s	500o/s	5000 mm/s	1000o/s
v150	150 mm/s	500o/s	5000 mm/s	1000o/s
v200	200 mm/s	500o/s	5000 mm/s	1000o/s
v300	300 mm/s	500o/s	5000 mm/s	1000o/s
v400	400 mm/s	500o/s	5000 mm/s	1000o/s
v500	500 mm/s	500o/s	5000 mm/s	1000o/s
v600	600 mm/s	500o/s	5000 mm/s	1000o/s
v800	800 mm/s	500o/s	5000 mm/s	1000o/s
v1000	1000 mm/s	500o/s	5000 mm/s	1000o/s
v1500	1500 mm/s	500o/s	5000 mm/s	1000o/s
v2000	2000 mm/s	500o/s	5000 mm/s	1000o/s
v2500	2500 mm/s	500o/s	5000 mm/s	1000o/s
v3000	3000 mm/s	500o/s	5000 mm/s	1000o/s
v4000	4000 mm/s	500o/s	5000 mm/s	1000o/s
v5000	5000 mm/s	500o/s	5000 mm/s	1000o/s
v6000	6000 mm/s	500o/s	5000 mm/s	1000o/s
v7000	7000 mm/s	500o/s	5000 mm/s	1000o/s
vmax	*)	500o/s	5000 mm/s	1000o/s

*) Max. TCP speed for the used robot type and normal pratical TCP values.
The RAPID function *MaxRobSpeed* returns the same value.
If use of extreme big TCP values in tool frame, create own speeddata with bigger TCP speed than returned by *MaxRobSpeed*.

Structure

< dataobject of *speeddata* >
< *v_tcp* of *num* >
< *v_ori* of *num* >
< *v_leax* of *num* >
< *v_reax* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Motion/Speed in general	Motion and I/O Principles - *Positioning during Program Execution*
Defining maximum velocity	Instructions - *VelSet*
Max. TCP speed for this robot	Function - *MaxRobSpeed*
Configuration of external axes	User's Guide - *System Parameters*
Motion performance	Product Specification

stoppointdata - Stop point data

Stoppointdata is used to specify how a position is to be terminated, i.e. how close to the programmed position the axes must be before moving towards the next position.

Description

A position can be terminated either in the form of a fly-by point or a stop point.

A fly-by point means that the programmed position is never reached. A zone is specified in the instruction for the movement, defining a corner path. Instead of heading for the programmed position, the direction of the motion is formed into the corner path before the position is reached. See data type *zonedata*.

A stop point means that the robot and external axes must reach the specified position before the robot/external axes continues with the next movement. The robot is considered to have reached a stop point when the convergence criteria of the point are satisfied. The convergence criteria are speed and position. It is also possible to specify timing criteria. For stop point *fine*, see also data type *zonedata*.

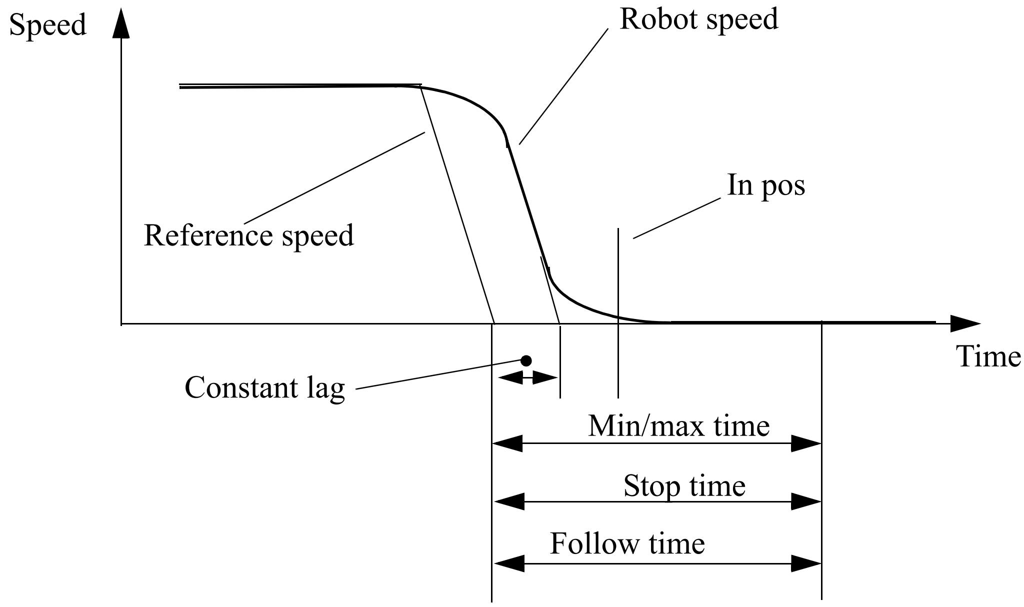
Three types of stop points can be defined by the *stoppointdata*.

- The **in position** type of stop point is defined as a percentage of the convergence criteria (position and speed) for the predefined stop point *fine*. The in-position type also uses a minimum and a maximum time. The robot waits for at least the minimum time, and at most the maximum time for the position and speed criteria to be satisfied.
- The **stop time** type of stop point always waits in the stop point for the given time.
- The **follow time** type of stop point is a special type of stop point used to coordinate the robot movement with a conveyor.

The *stoppointdata* also determines how the movement shall be synchronized with the RAPID execution. If the movement is synchronized, the RAPID execution waits for a “in pos” event when the robot is in position. If the movement is not synchronized, the RAPID execution gets a “prefetch” event almost a half second before the physical robot reach the programmed position. When the program executer gets an “in pos” or a “prefetch” event it continues with the next instruction. When the “prefetch” event arrives, the robot still have a long way to move. When the “in pos” event arrives the robot is close to the programmed position. Note that for the type **stop time** and **follow time**, the next instruction starts its execution at the same time as the stop time and follow time, respectively, start to count down. But for the type **in position**, the next instruction is started when the convergence criteria is fulfilled.

If use of move instructions with argument *\Conc*, no synchronization at all is done, so the actual move instruction will be ready at once.

Figure 17 Termination of a stop point



In the figure above, the termination of the stop points is described. The robots speed does not decrease linear. The robot servo is always ahead the physical robot. It is shown as the constant lag in the figure above. The constant lag is about 0.1 seconds. The timing elements of *stoppointdata* use the reference speed as trigger. When the reference speed is zero the time measurement starts. Therefore the time in the timing elements always include the constant lag. Consequently there is no sense in using values less than the constant lag.

Components

type *(type of stop point)* Data type: *stoppoint*

Defines the type of stoppoint.

1 (inpos) The movement terminates as an in-position type of stop point.
Enables the *inpos* element in *stoppointdata*.
The zone data in the instruction is not used, use *fine* or *z0*.

2 (stoptime) The movement terminates as a stop-time type of stop point.
Enables the *stoptime* element in *stoppointdata*.
The zone data in the instruction is not used, use *fine* or *z0*.

3 (followtime) The movement terminates as a conveyor follow-time type of fine point. The zone data in the instruction is used when the robot leaves the conveyor.
Enables the *followtime* element in *stoppointdata*.

Data type *stoppoint* is an alias data type for num. It is used to choose the type of stop point and which data elements to use in the *stoppointdata*. Predefined constants are:

Value	Symbolic constant	Comment
1	inpos	In position type number
2	stoptime	Stop time type number
3	fllwtime	Follow time type number

progsynch *(program synchronisation)* Data type: *bool*

Synchronisation with RAPID program execution.

- *TRUE*-> The movement is synchronized with RAPID execution. The program do not start to execute the next instruction until the stop point has been reached.
- *FALSE*-> The movement is **not** synchronized with RAPID execution. The program starts the execution of the next instruction before the stop point has been reached.

If use of move instructions with argument *\Conc*, no synchronization at all is done independent of the data in *progsynch*, so the actual move instruction will always be ready at once.

inpos.position *(position condition for TCP)* Data type: *num*

The position condition (the radius) for the TCP in percent of a normal *fine* stop point.

inpos.speed *(speed condition for TCP)* Data type: *num*

The speed condition for the TCP in percent of a normal *fine* stop point.

inpos.mintime *(minimum wait time)* Data type: *num*

The minimum wait time in seconds before in position. Used to make the robot wait at least the specified time in the point. Maximum value is 20.0 seconds.

inpos.maxtime *(maximum wait time)* Data type: *num*

The maximum wait time in seconds for convergence criteria to be satisfied. Used to assure that the robot does not get stuck in the point, if the speed and position conditions are set too tight. Maximum value is 20.0 seconds.

stoptime *(stop time)* Data type: *num*

The time in seconds, the TCP stands still in position before starting the next movement. Maximum value is 20.0 seconds.

followtime *(follow time)* Data type: *num*

The time in seconds the TCP follows the conveyor.

signal Data type: *string*

Reserved for future use.

relation Data type: *opnum*

Reserved for future use.

checkvalue Data type: *num*

Reserved for future use.

Examples

Inpos

VAR stoppointdata my_inpos := [inpos, TRUE, [25, 40, 0.1, 5], 0, 0, "", 0, 0];
MoveL *, v1000, z0 \Inpos:=my_inpos, grip4;

The stop point data **my_inpos** is defined by means of the following characteristics:

- The type of stop point is in-position type, *inpos*.
- The stop point will be synchronized with the RAPID program execution, *TRUE*.
- The stop point distance criteria is 25% of the distance defined for the stop point *fine*, *25*.
- The stop point speed criteria is 40% of the speed defined for the stop point *fine*, *40*.
- The minimum time to wait before convergence is 0,1 s, *0.1*.
- The maximum time to wait on convergence is 5 s, *5*.

The robot move towards the programmed position until one of the criteria position or speed is satisfied.

my_inpos.inpos.position := 40;
MoveL *, v1000, z0 \Inpos:=my_inpos, grip4;

The stop point distance criteria is adjusted to *40*%.

Stoptime

VAR stoppointdata my_stoptime := [stoptime, FALSE, [0, 0, 0, 0], 1.45, 0, "", 0, 0];

MoveL *, v1000, z0\Inpos:=my_stoptime, grip4;

The stop point data **my_stoptime** is defined by means of the following characteristics:

- The type of stop point is stop-time type, *stoptime*.
- The stop point will not be synchronized with the RAPID program execution, *FALSE*.
- The wait time in position is *1.45* s.

The robot moves towards the programmed position until the prefetch event arrives. The next RAPID instruction executes. If it is a move-instruction, the robot stops for *1.45* seconds before the next movement starts.

my_stoptime.stoptime := 6.66;
MoveL *, v1000, z0 \Inpos:=my_stoptime, grip4;

The stop point stop time is adjusted to *6.66* seconds If the next RAPID instruction is a move-instruction, the robot stops for *6.66* s.

Followtime

VAR stoppointdata my_followtime := [fllwtime, TRUE, [0, 0, 0, 0], 0, 0.5, ““, 0, 0];
MoveL *, v1000, z10 \Inpos:=my_followtime, grip6;

The stop point data **my_followtime** is defined by means of the following characteristics:

- The type of stop point is follow-time type, *fllwtime*.
- The stop point will be synchronized with the RAPID program execution, *TRUE*.
- The stop point follow time is 0.5 s, *0.5*.

The robot will follow the conveyor for 0.5 s before leaving it with the zone 10 mm, *z10*.

my_followtime.followtime := 0.4;

The stop point follow time is adjusted to *0.4* s.

Predefined data

A number of stop point data are already defined in the system module *BASE*.

In position stop points

Name	Progsynch	Position	Speed	Mintime	Maxtime	Stoptime	Followtime
inpos20	TRUE	20%	20%	0 s	20 s	-	-
inpos50	TRUE	50%	50%	0 s	20 s	-	-
inpos100	TRUE	100%	100%	0 s	20 s	-	-

(*inpos100* has same convergence criteria as stop point *fine*)

Stop time stop points

Name	Progsynch	Position	Speed	Mintime	Maxtime	Stoptime	Followtime
stoptime0_5	FALSE	-	-	-	-	0.5 s	-
stoptime1_0	FALSE	-	-	-	-	1.0 s	-
stoptime1_5	FALSE	-	-	-	-	1.5 s	-

Follow time stop points

Name	Progsynch	Position	Speed	Mintime	Maxtime	Stoptime	Followtime
fllwtime0_5	TRUE	-	-	-	-	-	0.5 s
fllwtime1_0	TRUE	-	-	-	-	-	1.0 s
fllwtime1_5	TRUE	-	-	-	-	-	1.5 s

Structure

< data object of *stoppointdata* >
< *type* of *stoppoint*>
< *progsynch* of *bool* >
< *inpos* of *inposdata* >
< *position* of *num* >
< *speed* of *num* >
< *mintime* of *num* >
< *maxtime* of *num* >
< *stoptime* of *num* >
< *followtime* of *num* >
< *signal* of *string* >
< *relation* of *opnum* >
< *checkvalue* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Movements/Paths in general	Motion and I/O Principles - *Positioning during Program Execution*
Configuration of external axes	User's Guide - *System Parameters*
Fly-by points	Data Types - *zonedata*

string - Strings

String is used for character strings.

Description

A character string consists of a number of characters (a maximum of 80) enclosed by quotation marks (“”),

e.g. “This is a character string”.

If the quotation marks are to be included in the string, they must be written twice,

e.g. “This string contains a ““character”.

If the back slash are to be included in the string, it must be written twice,

e.g. “This string contains a \\ character”.

Example

```
VAR string text;
.
text := “start welding pipe 1”;
TPWrite text;
```

The text *start welding pipe 1* is written on the teach pendant.

Limitations

A string may have from 0 to 80 characters; inclusive of extra quotation marks or back slash.

A string may contain any of the characters specified by ISO 8859-1 as well as control characters (non-ISO 8859-1 characters with a numeric code between 0-255).

Predefined data

A number of predefined string constants are available in the system and can be used together with string functions.

<u>Name</u> <u>Character set</u>

STR_DIGIT
<digit> ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

STR_UPPER
<upper case letter> ::=
A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S | T
| U | V | W | X | Y | Z | À | Á | Â | Ã
| Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í
| Î | Ï | 1) | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
| Ù | Ú | Û | Ü | 2) | 3)

STR_LOWER
<lower case letter> ::=
a | b | c | d | e | f | g | h | i | j
| k | l | m | n | o | p | q | r | s | t
| u | v | w | x | y | z | à | á | â | ã
| ä | å | æ | ç | è | é | ê | ë | ì | í
| î | ï | 1) | ñ | ò | ó | ô | õ | ö | ø
| ù | ú | û | ü | 2) | 3) | ß | ÿ

STR_WHITE
<blank character> ::=

1) Icelandic letter eth.
2) Letter Y with acute accent.
3) Icelandic letter thorn.

The following constants are already defined in the system module *BASE*:

```
CONST string diskhome := “HOME:”;

! For old programs from S4C system
CONST string ram1disk := “HOME:”;

CONST string disktemp := “TEMP:”;

CONST string flp1 := “flp1:”;

CONST string stEmpty := “”;
```

Related information

	<u>Described in:</u>
Operations using strings	Basic Characteristics - *Expressions*
String values	Basic Characteristics - *Basic Elements*

symnum - Symbolic number

Symnum is used to represent an integer with a symbolic constant.

Description

A *symnum* constant is intended to be used when checking the return value from the functions *OpMode* and *RunMode*. See example below.

Example

```
IF RunMode() = RUN_CONT_CYCLE THEN
.
.
ELSE
.
.
ENDIF
```

Predefined data

The following symbolic constants of the data type *symnum* are predefined and can be used when checking return values from the functions *OpMode* and *RunMode*

Value	Symbolic constant	Comment
0	RUN_UNDEF	Undefined running mode
1	RUN_CONT_CYCLE	Continuous or cycle running mode
2	RUN_INSTR_FWD	Instruction forward running mode
3	RUN_INSTR_BWD	Instruction backward running mode
4	RUN_SIM	Simulated running mode

Value	Symbolic constant	Comment
0	OP_UNDEF	Undefined operating mode
1	OP_AUTO	Automatic operating mode
2	OP_MAN_PROG	Manual operating mode max. 250 mm/s
3	OP_MAN_TEST	Manual operating mode full speed, 100%

Characteristics

Symnum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Data types in general, alias data types	Basic Characteristics - *Data Types*

System Data

System data is the internal data of the robot that can be accessed and read by the program. It can be used to read the current status, e.g. the current maximum velocity.

The following table contains a list of all system data.

Name	Description	Data Type	Changed by	See also
C_MOTSET	Current motion settings, i.e.: - max. velocity and velocity override - max. acceleration - movement about singular points - monitoring the axis configuration - path resolution - motion supervision with tunevalue -reduction of TCP acceleration/ deceleration along the movement path -modification of the tool orientation during circle interpolation	motsetdata	Instructions - VelSet - AccSet - SingArea - ConfL,ConfJ - PathResol - MotionSup - PathAccLim - CirPathReori	Data Types - *motsetdata* Instructions - *VelSet* Instructions - *AccSet* Instructions - *SingArea* Instructions - *ConfL*, *ConfJ* Instructions - *PathResol* Instructions - *MotionSup* Instructions - *PathAccLim* Instructions - *CirPathReori*
C_PROGDISP	Current program displacement for robot and external axes.	progdisp	Instructions - PDispSet - PDispOn - PDispOff - EOffsSet - EOffsOn - EOffsOff	Data Types - *progdisp* Instructions - *PDispSet* Instructions - *PDispOn* Instructions - *PDispOff* Instructions - *EOffsSet* Instructions - *EOffsOn* Instructions - *EOffsOff*
ERRNO	The latest error that occurred	errnum	The robot	Data Types - *errnum* RAPID Summary - Error Recovery
INTNO	The latest interrupt that occurred	intnum	The robot	Data Types - *intnum* RAPID Summary -*Interrupts*

taskid - Task identification

Taskid is used to identify available program tasks in the system.

The names of the program tasks are defined in the system parameters and, consequently, must not be defined in the program.

Description

Data of the type *taskid* only contains a reference to the program task.

Limitations

Data of the type *taskid* must not be defined in the program. The data type can, on the other hand, be used as a parameter when declaring a routine.

Predefined data

The program tasks defined in the system parameters can always be accessed from the program (installed data).

For all program tasks in the system, predefined variables of the data type *taskid* will be available. The variable identity will be "taskname"+"Id", e.g. for MAIN task the variable identity will be MAINId, TSK1 - TSK1Id etc.

Characteristics

Taskid is a *non-value* data type. This means that data of this type does not permit value-oriented operations.

Related information

	Described in:
Saving program modules	Instruction - *Save*
Configuration of program tasks	User's Guide - *System Parameters*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

testsignal - Test signal

The data type *testsignal* is used when a test of the robot motion system is performed.

Description

A number of predefined test signals are available in the robot system. The *testsignal* data type is available in order to simplify programming of instruction *TestSignDefine*.

Examples

TestSignDefine 2, speed, Orbit, 2, 0;

speed is a constant of the *testsignal* data type.

Predefined data

The following test signals for external manipulator axes are predefined in the system. All data is in SI units and measured on the motor side of the axis.

```
CONST testsignal speed          :=  6;     ! rad/s
CONST testsignal torque_ref     :=  9;     ! Nm
CONST testsignal resolver_angle :=  1;     ! rad
CONST testsignal speed_ref      :=  4;     ! rad/s
CONST testsignal dig_input1     := 102;    ! 0 or 1
CONST testsignal dig_input2     := 103;    ! 0 or 1
```

Characteristics

Testsignal is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Define test signal	Instructions - *TestSignDefine*
Read test signal	Functions - *TestSignRead*
Reset test signals	Instructions - *TestSignReset*

tooldata - Tool data

Tooldata is used to describe the characteristics of a tool, e.g. a welding gun or a gripper.

If the tool is fixed in space (a stationary tool), common tool data is defined for this tool and the gripper holding the work object.

Description

Tool data affects robot movements in the following ways:

- The tool centre point (TCP) refers to a point that will satisfy the specified path and velocity performance. If the tool is reorientated or if coordinated external axes are used, only this point will follow the desired path at the programmed velocity.
- If a stationary tool is used, the programmed speed and path will relate to the work object.
- Programmed positions refer to the position of the current TCP and the orientation in relation to the tool coordinate system. This means that if, for example, a tool is replaced because it is damaged, the old program can still be used if the tool coordinate system is redefined.

Tool data is also used when jogging the robot to:

- Define the TCP that is not to move when the tool is reorientated.
- Define the tool coordinate system in order to facilitate moving in or rotating about the tool directions.

It is important to always define the actual tool load and when used, the payload of the robot too.

Incorrect definitions of load data can result in overloading of the robot mechanical structure.

When incorrect tool load data is specified, it can often lead to the following consequences:

- If the value in the specified load is greater than that of the value of the true load;
  -> The robot will not be used to its maximum capacity
  -> Impaired path accuracy including a risk of overshooting
- If the value in the specified load is less than the value of the true load;
  -> Impaired path accuracy including a risk of overshooting
  -> Risk of overloading the mechanical structure

Components

robhold *(robot hold)* Data type: *bool*

Defines whether or not the robot is holding the tool:

- *TRUE*-> The robot is holding the tool.
- *FALSE* -> The robot is not holding the tool, i.e. a stationary tool.

tframe *(tool frame)* Data type: *pose*

The tool coordinate system, i.e.:

- The position of the TCP (x, y and z) in mm, expressed in the wrist coordinate system (See figure 1).
- The orientation of the tool coordinate system, expressed in the wrist coordinate system as a quaternion (q1, q2, q3 and q4) (See figure 1).

If a stationary tool is used, the definition is defined in relation to the world coordinate system.

If the direction of the tool is not specified, the tool coordinate system and the wrist coordinate system will coincide.

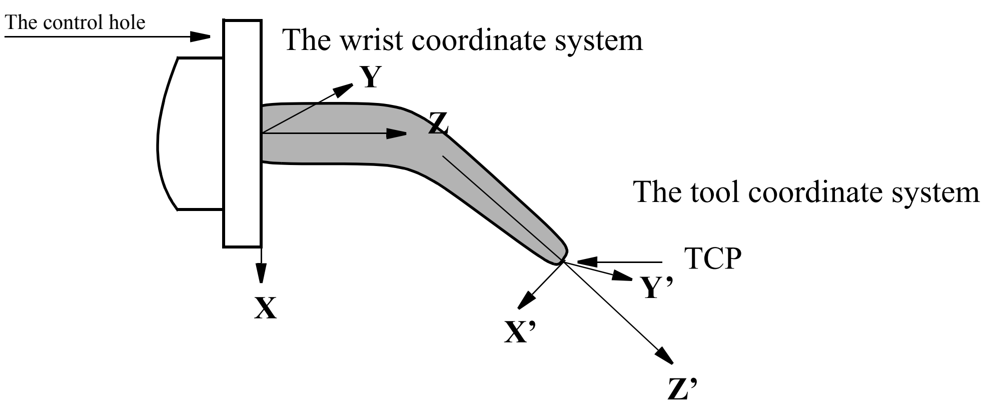


Figure 18 Definition of the tool coordinate system.

tload *(tool load)* Data type: *loaddata*

The load of the tool, i.e.:

- The weight of the tool in kg.
- The centre of gravity of the tool (x, y and z) in mm, expressed in the wrist coordinate system
- The orientation of the tool load coordinate system expressed in the wrist coordinate system, defining the inertial axes of the tool.
  The orientation of the tool load coordinate system must coincide with the orientation of the wrist coordinate system. **This must always be set to 1, 0, 0, 0**.
- The moments of inertia of the tool relative to its centre of mass around the tool load coordinate axes in kgm2.
  If all inertial components are defined as being 0 kgm2, the tool is handled as a point mass.

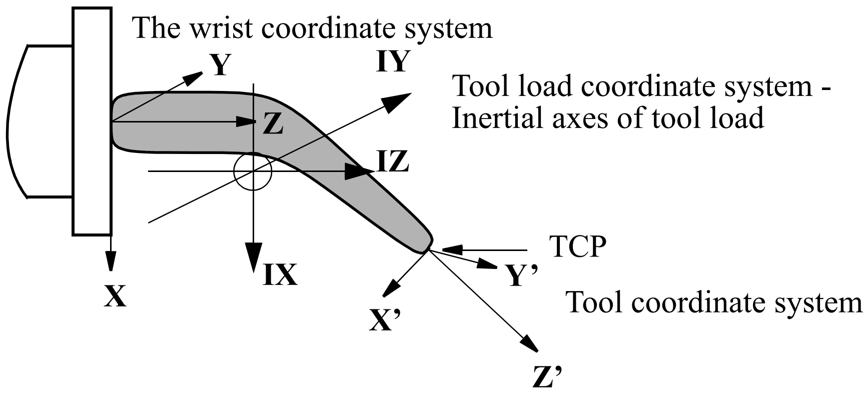


Figure 19 Tool load parameter definitions

For more information (such as coordinate system for stationary tool or restrictions), see the data type *loaddata*.

If a stationary tool is used, the load of the gripper holding the work object must be defined.

Note that only the load of the tool is to be specified. The payload handled by a gripper is connected and disconnected by means of the instruction *GripLoad*.

Examples

```
PERS tooldata gripper := [ TRUE, [[97.4, 0, 223.1], [0.924, 0, 0.383 ,0]],
  [5, [23, 0, 75], [1, 0, 0, 0], 0, 0, 0]];
```

The tool in Figure 18 is described using the following values:

- The robot is holding the tool.
- The TCP is located at a point *223.1* mm straight out from axis 6 and *97.4* mm along the X-axis of the wrist coordinate system.
- The X and Z directions of the tool are rotated 45o in relation to the wrist coordinate system.
- The tool weighs 5 kg.
- The centre of gravity is located at a point *75* mm straight out from axis 6 and *23* mm along the X-axis of the wrist coordinate system.
- The load can be considered a point mass, i.e. without any moment of inertia.

```
gripper.tframe.trans.z := 225.2;
```

The TCP of the tool, *gripper*, is adjusted to *225.2* in the z-direction.

Limitations

The tool data should be defined as a persistent variable *(PERS)* and should not be defined within a routine. The current values are then saved when the program is stored on diskette and are retrieved on loading.

Arguments of the type tool data in any motion instruction should only be an entire persistent (not array element or record component).

Predefined data

The tool *tool0* defines the wrist coordinate system, with the origin being the centre of the mounting flange. *Tool0* can always be accessed from the program, but can never be changed (it is stored in system module *BASE*).

```
PERS tooldata tool0 := [ TRUE, [ [0, 0, 0], [1, 0, 0 ,0] ],
                [0.001, [0, 0, 0.001], [1, 0, 0, 0], 0, 0, 0] ];
```

Structure

- < dataobject of *tooldata* >
  - < *robhold* of *bool* >
  - < *tframe* of *pose* >
    - < *trans* of *pos* >
      - < *x* of *num* >
      - < *y* of *num* >
      - < *z* of *num* >
    - < *rot* of *orient* >
      - < *q1* of *num* >
      - < *q2* of *num* >
      - < *q3* of *num* >
      - < *q4* of *num* >
  - < *tload* of *loaddata* >
    - < *mass* of *num* >
    - < *cog* of *pos* >
      - < *x* of *num* >
      - < *y* of *num* >
      - < *z* of *num* >
    - < *aom* of *orient* >
      - < *q1* of *num* >
      - < *q2* of *num* >
      - < *q3* of *num* >
      - < *q4* of *num* >
    - < *ix* of *num* >
    - < *iy* of *num* >
    - < *iz* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Definition of payload	Instructions - *Gripload*
Definition of load	Data types - *Load data*

tpnum - Teach pendant window number

tpnum is used to represent the Teach Pendant Window number with a symbolic constant.

Description

A *tpnum* constant is intended to be used in instruction *TPShow*. See example below.

Example

TPShow TP_PROGRAM;

The *Production Window* will be active if the system is in *AUTO* mode and the *Program Window* will be active if the system is in *MAN* mode, after execution of this instruction.

Predefined data

The following symbolic constants of the data type *tpnum* are predefined and can be used in instruction *TPShow*:

Value	Symbolic constant	Comment
1	TP_PROGRAM	AUTO: Production Window MAN: Program Window
2	TP_LATEST	Latest used Teach Pendant Window
3	TP_SCREENVIEWER	Screen Viewer window, if this option is active

Characteristics

tpnum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Data types in general, alias data types	Basic Characteristics - *Data Types*
Communicating using the teach pendant	RAPID Summary - *Communication*
Switch window on the teach pendant	Instructions - *TPShow*

triggdata - Positioning events - trigg

Triggdata is used to store data about a positioning event during a robot movement.

A positioning event can take the form of setting an output signal or running an interrupt routine at a specific position along the movement path of the robot.

Description

To define the conditions for the respective measures at a positioning event, variables of the type *triggdata* are used. The data contents of the variable are formed in the program using one of the instructions *TriggIO* or *TriggInt*, and are used by one of the instructions *TriggL*, *TriggC* or *TriggJ*.

Example

```
VAR triggdata gunoff;

TriggIO gunoff, 5 \DOp:=gun, off;

TriggL p1, v500, gunoff, fine, gun1;
```

The digital output signal *gun* is set to the value *off* when the TCP is at a position *5* mm before the point *p1*.

Characteristics

Triggdata is a non-value data type.

Related information

	Described in:
Definition of triggs	Instructions - *TriggIO*, *TriggInt*
Use of triggs	Instructions - *TriggL*, *TriggC*, *TriggJ*
Characteristics of non-value data types	Basic Characteristics- *Data Types*

trapdata - Interrupt data for current TRAP

trapdata (trap data) is used to contain the interrupt data that caused the current TRAP routine to be executed.

To be used in TRAP routines generated by instruction *IError,* before use of the instruction *ReadErrData*.

Description

Data of the type *trapdata* represents internal information related to the interrupt that caused the current trap routine to be executed.
Its content depends on the type of interrupt.

Example

```
VAR errdomain err_domain;
VAR num err_number;
VAR errtype err_type;
VAR trapdata err_data;
.
TRAP trap_err
    GetTrapData err_data;
    ReadErrData err_data, err_domain, err_number, err_type;
ENDTRAP
```

When an error is trapped to the trap routine *trap_err*, the error domain, the error number, and the error type are saved into appropriate non-value variables of type *trapdata*.

Characteristics

trapdata is a non-value data type.

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
More information on interrupt management	Basic Characteristics - *Interrupts*
Non value data types	Basic Characteristics - *Data Types*
Orders an interrupt on errors	Instructions - *IError*
Get interrupt data for current TRAP	Instructions- *GetTrapData*
Gets information about an error	Instructions - *ReadErrData*

tunetype - Servo tune type

Tunetype is used to represent an integer with a symbolic constant.

Description

A *tunetype* constant is intended to be used as an argument to the instruction *TuneServo*. See example below.

Example

```
TuneServo MHA160R1, 1, 110 \Type:= TUNE_KP;
```

Predefined data

The following symbolic constants of the data type *tunetype* are predefined and can be used as argument for the instruction *TuneServo.*

Value	Symbolic constant	Comment
0	TUNE_DF	Reduces overshoots
1	TUNE_KP	Affects position control gain
2	TUNE_KV	Affects speed control gain
3	TUNE_TI	Affects speed control integration time
4	TUNE_FRIC_LEV	Affects friction compensation level
5	TUNE_FRIC_RAMP	Affects friction compensation ramp
6	TUNE_DG	Reduces overshoots
7	TUNE_DH	Reduces vibrations with heavy loads
8	TUNE_DI	Reduces path errors
9	TUNE_DK	Only for ABB internal use
10	TUNE_DL	Only for ABB internal use

Characteristics

Tunetype is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Data types in general, alias data types	Basic Characteristics - *Data Types*
Use of data type tunetype	Instructions - *TuneServo*

wobjdata - Work object data

Wobjdata is used to describe the work object that the robot welds, processes, moves within, etc.

Description

If work objects are defined in a positioning instruction, the position will be based on the coordinates of the work object. The advantages of this are as follows:

- If position data is entered manually, such as in off-line programming, the values can often be taken from a drawing.
- Programs can be reused quickly following changes in the robot installation. If, for example, the fixture is moved, only the user coordinate system has to be redefined.
- Variations in how the work object is attached can be compensated for. For this, however, some sort of sensor will be required to position the work object.

If a stationary tool or coordinated external axes are used the work object must be defined, since the path and velocity would then be related to the work object instead of the TCP.

Work object data can also be used for jogging:

- The robot can be jogged in the directions of the work object.
- The current position displayed is based on the coordinate system of the work object.

Components

robhold *(robot hold)* Data type: *bool*

Defines whether or not the robot is holding the work object:

- *TRUE*-> The robot is holding the work object, i.e. using a stationary tool.
- *FALSE* -> The robot is not holding the work object, i.e. the robot is holding the tool.

ufprog *(user frame programmed)* Data type: *bool*

Defines whether or not a fixed user coordinate system is used:

- *TRUE* -> Fixed user coordinate system.
- *FALSE*-> Movable user coordinate system, i.e. coordinated external axes are used.

ufmec *(user frame mechanical unit)* Data type: *string*

The mechanical unit with which the robot movements are coordinated. Only specified in the case of movable user coordinate systems (*ufprog* is *FALSE*).

Specified with the name that is defined in the system parameters, e.g. "orbit_a".

uframe *(user frame)* Data type: *pose*

The user coordinate system, i.e. the position of the current work surface or fixture (see Figure 20):

- The position of the origin of the coordinate system (x, y and z) in mm.
- The rotation of the coordinate system, expressed as a quaternion (q1, q2, q3, q4).

If the robot is holding the tool, the user coordinate system is defined in the world coordinate system (in the wrist coordinate system if a stationary tool is used).

When coordinated external axes are used (*ufprog* is *FALSE*), the user coordinate system is defined in the system parameters.

oframe *(object frame)* Data type: *pose*

The object coordinate system, i.e. the position of the current work object (see Figure 20):

- The position of the origin of the coordinate system (x, y and z) in mm.
- The rotation of the coordinate system, expressed as a quaternion (q1, q2, q3, q4).

The object coordinate system is defined in the user coordinate system.

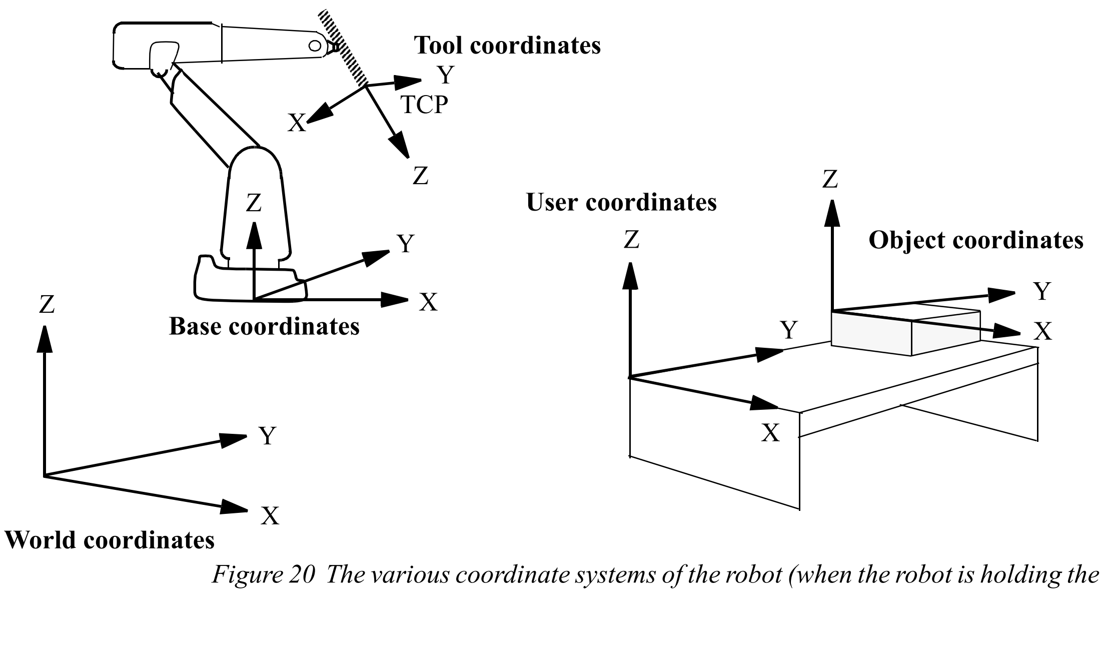


Figure 20 The various coordinate systems of the robot (when the robot is holding the tool).

Example

**PERS wobjdata wobj2 :=[FALSE, TRUE, "", [[300, 600, 200], [1, 0, 0 ,0]],
[[0, 200, 30], [1, 0, 0 ,0]]];**

The work object in Figure 20 is described using the following values:

- The robot is not holding the work object.
- The fixed user coordinate system is used.
- The user coordinate system is not rotated and the coordinates of its origin are x= *300*, y = *600* and z = *200* mm in the world coordinate system.
- The object coordinate system is not rotated and the coordinates of its origin are x= *0*, y= *200* and z= *30* mm in the user coordinate system.

wobj2.oframe.trans.z := 38.3;

- The position of the work object *wobj2* is adjusted to *38.3* mm in the z-direction.

Limitations

The work object data should be defined as a persistent variable (*PERS)* and should not be defined within a routine. The current values are then saved when the program is stored on diskette and are retrieved on loading.

Arguments of the type work object data in any motion instruction should only be an entire persistent (not array element or record component).

Predefined data

The work object data *wobj0* is defined in such a way that the object coordinate system coincides with the world coordinate system. The robot does not hold the work object.

Wobj0 can always be accessed from the program, but can never be changed (it is stored in system module *BASE*).

**PERS wobjdata wobj0 := [FALSE, TRUE, "", [[0, 0, 0], [1, 0, 0 ,0]],
[[0, 0, 0], [1, 0, 0 ,0]]];**

Structure

< dataobject of *wobjdata* >
< *robhold* of *bool* >
< *ufprog* of *bool*>
< *ufmec* of *string* >
< *uframe* of *pose* >
< *trans* of *pos* >
< *x* of *num* >
< *y* of *num* >
< *z* of *num* >
< *rot* of *orient* >
< *q1* of *num* >
< *q2* of *num* >
< *q3* of *num* >
< *q4* of *num* >
< *oframe* of *pose* >
< *trans* of *pos* >
< *x* of *num* >
< *y* of *num* >
< *z* of *num* >
< *rot* of *orient* >
< *q1* of *num* >
< *q2* of *num* >
< *q3* of *num* >
< *q4* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Coordinated external axes	Motion and I/O Principles - *Coordinate Systems*
Calibration of coordinated external axes	User's Guide - *System Parameters*

wzstationary - Stationary world zone data

wzstationary (world zone stationary) is used to identify a stationary world zone and can only be used in an event routine connected to the event POWER ON.

A world zone is supervised during robot movements both during program execution and jogging. If the robot's TCP reaches the world zone or if the robot/external axes reaches the world zone in joints , the movement is stopped or a digital output signal is set or reset.

Description

A *wzstationary world zone is defined and activated by a WZLimSup or a WZDOSet instruction.*

WZLimSup or *WZDOSet* gives the variable or the persistent of data type *stationary* a numeric value. The value identifies the world zone.

A stationary world zone is always active and is only erased by a warm start (switch power off then on, or change system parameters). It is not possible to deactivate, activate or erase a stationary world zone via RAPID instructions.

Stationary world zones should be active from power on and should be defined in a POWER ON event routine or a semistatic task.

Example

```
VAR wzstationary conveyor;
...
PROC ...
    VAR shapedata volume;
    ...
    WZBoxDef \Inside, volume, p_corner1, p_corner2;
    WZLimSup \Stat, conveyor, volume;
ENDPROC
```

A *conveyor* is defined as a straight box (the volume below the belt). If the robot reaches this volume, the movement is stopped.

Limitations

A *wzstationary* data can be defined as a variable (VAR) or as a persistent (PERS). It can be global in task or local within module, but not local within a routine.

Arguments of the type *wzstationary* should only be entire data (not array element or record component).

An init value for data of the type *wzstationary* is not used by the control system. When there is a need to use a persistent variable in a multi-tasking system, set the init value to 0 in both tasks,
e.g. PERS wzstationary share_workarea := [0];

Example

For a complete example see instruction *WZLimSup*.

Characteristics

wzstationary is an alias data type of *wztemporary* and inherits its characteristics.

Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Temporary world zone	Data Types - *wztemporary*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

wztemporary - Temporary world zone data

wztemporary (world zone temporary) is used to identify a temporary world zone and can be used anywhere in the RAPID program for the MAIN task.

A world zone is supervised during robot movements both during program execution and jogging. If the robot's TCP reaches the world zone or if the robot/external axes reaches the world zone in joints, the movement is stopped or a digital output signal is set or reset.

Description

A *wztemporary world zone is defined and activated by a WZLimSup or a WZDOSet instruction.*

WZLimSup or *WZDOSet* gives the variable or the persistent of data type *wztemporary* a numeric value. The value identifies the world zone.

Once defined and activated, a temporary world zone can be deactivated by *WZDisable*, activated again by *WZEnable*, and erased by *WZFree*.

All temporary world zones in the MAIN task are automatically erased and all data objects of type *wztemporary* in the MAIN task are set to 0:

- when a new program is loaded in the MAIN task
- when starting program execution from the beginning in the MAIN task

Example

```
VAR wztemporary roll;
...
PROC ...
    VAR shapedata volume;
    CONST pos t_center := [1000, 1000, 1000];
    ...
    WZCylDef \Inside, volume, t_center, 400, 1000;
    WZLimSup \Temp, roll, volume;
ENDPROC
```

A wztemporary variable, *roll,* is defined as a cylinder. If the robot reaches this volume, the movement is stopped.

Limitations

A *wztemporary* data can be defined as a variable (VAR) or as a persistent (PERS). It can be global in a task or local within a module, but not local within a routine.

Arguments of the type *wztemporary* must only be entire data, not an array element or record component.

A temporary world zone must only be defined (*WZLimSup* or *WZDOSet*) and free (*WZFree*) in the MAIN task. Definitions of temporary world zones in the background would affect the program execution in the MAIN task The instructions *WZDisable* and *WZEnable* can be used in the background task. When there is a need to use a persistent variable in a multi-tasking system, set the init value to 0 in both tasks,
e.g. PERS wztemporary share_workarea := [0];

Example

For a complete example see instruction *WZDOSet*.

Structure

<dataobject of *wztemporary*>
<*wz* of *num*>

Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Stationary world zone	Data Types - *wzstationary*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*
Deactivate world zone	Instructions - *WZDisable*
Activate world zone	Instructions - *WZEnable*
Erase world zone	Instructions - *WZFree*

zonedata - Zone data

Zonedata is used to specify how a position is to be terminated, i.e. how close to the programmed position the axes must be before moving towards the next position.

Description

A position can be terminated either in the form of a stop point or a fly-by point.

A stop point means that the robot and external axes must reach the specified position (stand still) before program execution continues with the next instruction. It is also possible to define stop points other than the predefined *fine.* The stop criteria, that tells if the robot is considered to have reached the point, can be manipulated using the *stoppointdata*.

A fly-by point means that the programmed position is never attained.
Instead, the direction of motion is changed before the position is reached.
Two different zones (ranges) can be defined for each position:

- The zone for the TCP path.
- The extended zone for reorientation of the tool and for external axes.

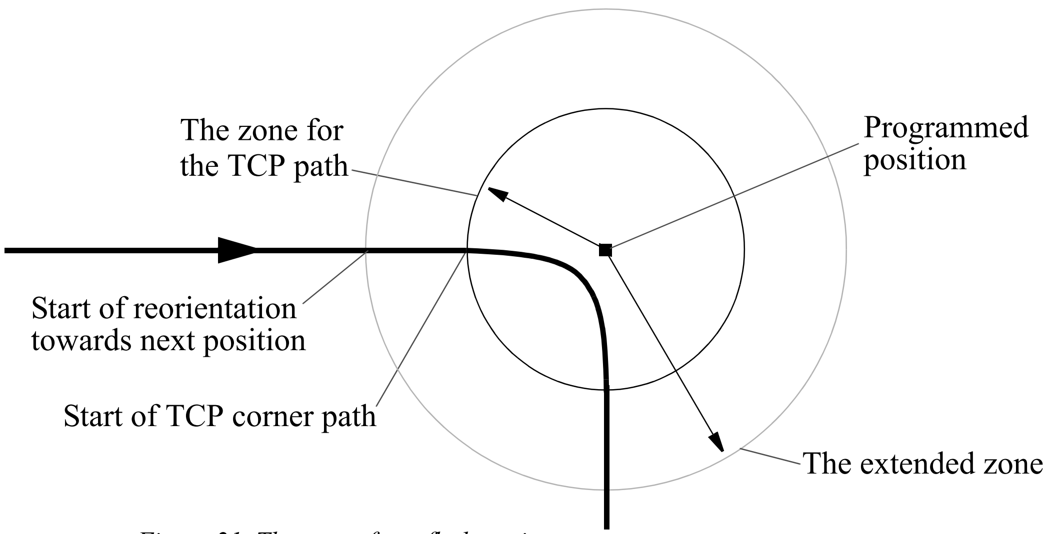


Figure 21 The zones for a fly-by point.

Zones function in the same way during joint movement, but the zone size may differ somewhat from the one programmed.

The zone size cannot be larger than half the distance to the closest position (forwards or backwards). If a larger zone is specified, the robot automatically reduces it.

The zone for the TCP path

A corner path (parabola) is generated as soon as the edge of the zone is reached (see Figure 21).

The zone for reorientation of the tool

Reorientation starts as soon as the TCP reaches the extended zone. The tool is reoriented in such a way that the orientation is the same leaving the zone as it would have been in the same position if stop points had been programmed. Reorientation will be smoother if the zone size is increased, and there is less of a risk of having to reduce the velocity to carry out the reorientation.

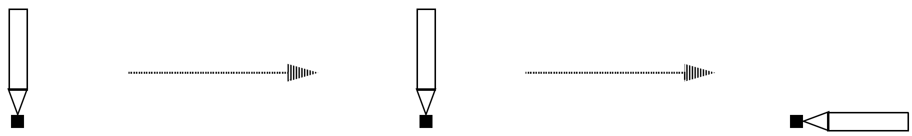


Figure 22 Three positions are programmed, the last with different tool orientation.

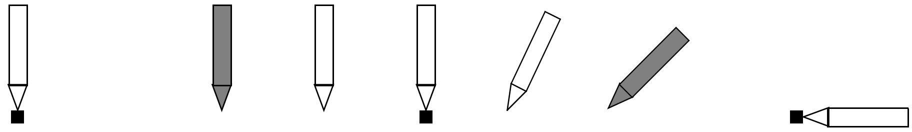


Figure 23 If all positions were stop points, program execution would look like this.

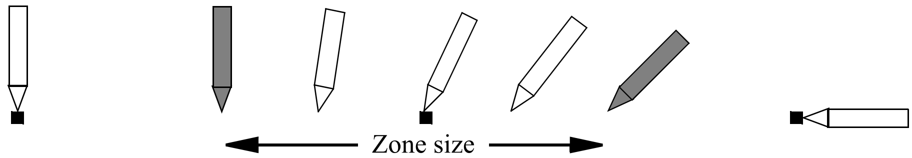


Figure 24 If the middle position was a fly-by point, program execution would look like this

The zone for external axes

External axes start to move towards the next position as soon as the TCP reaches the extended zone. In this way, a slow axis can start accelerating at an earlier stage and thus execute more evenly.

Reduced zone

With large reorientations of the tool or with large movements of the external axes, the extended zone and even the TCP zone can be reduced by the robot. The zone will be defined as the smallest relative size of the zone based upon the zone components (see next page) and the programmed motion.

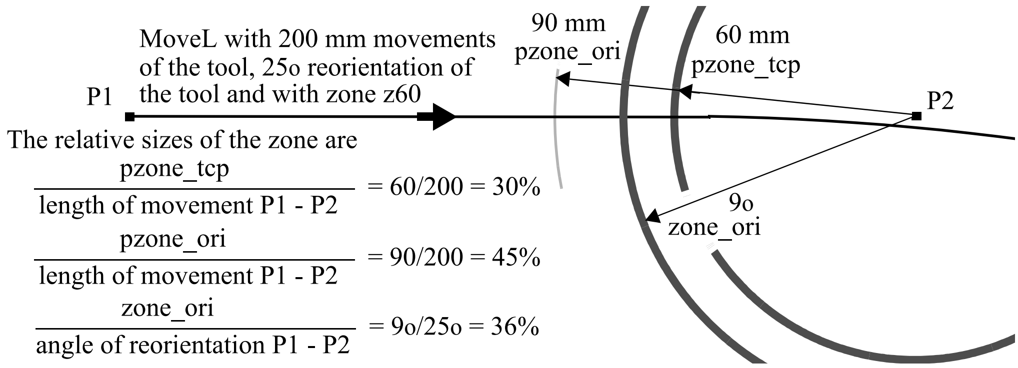


Figure 25 Example of reduced zone for reorientation of the tool to 36% of the motion due to zone_ori.

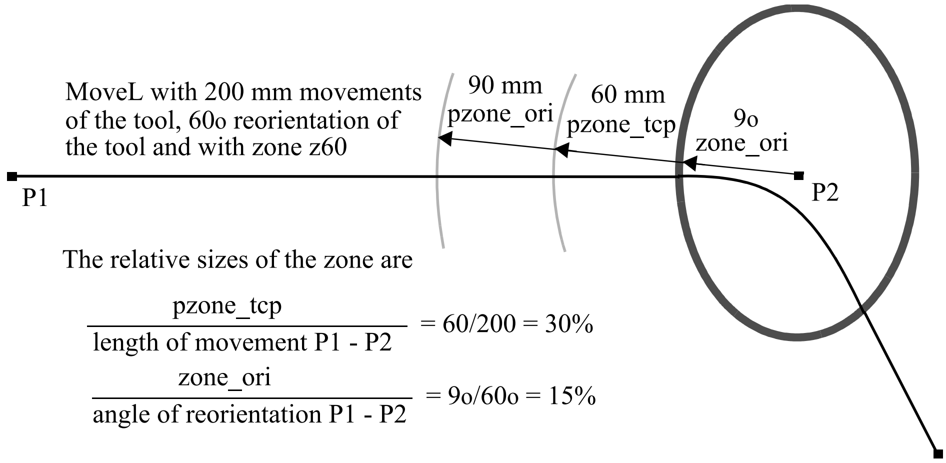


Figure 26 Example of reduced zone for reorientation of the tool and TCP path to 15% of the motion due to zone_ori.

When external axes are active they affect the relative sizes of the zone according to these formulas:

$$\frac{\text{pzone\_eax}}{\text{length of movement P1 - P2}}$$

$$\frac{\text{zone\_leax}}{\text{length of max linear ext. axis movement P1 - P2}}$$

$$\frac{\text{zone\_reax}}{\text{angle of max reorientation of rotating ext. axis P1 - P2}}$$

NOTE: If the TCP zone is reduced because of zone_ori, zone_leax or zone_reax the path planner enters a mode that can handle the case of no TCP movement. If there is a TCP movement when in this mode the speed is not compensated for the curvature of the path in a corner zone. For instance, this will cause a 30% speed reduction in a 90 degree corner. If this is a problem, increase the limiting zone component.

Components

finep *(fine point)* Data type: *bool*

Defines whether the movement is to terminate as a stop point (fine point) or as a fly-by point.

- *TRUE*-> The movement terminates as a stop point.
  The remaining components in the zone data are not used.
- *FALSE* -> The movement terminates as a fly-by point.

pzone_tcp *(path zone TCP)* Data type: *num*

The size (the radius) of the TCP zone in mm.

The extended zone will be defined as the smallest relative size of the zone based upon the following components and the programmed motion.

pzone_ori *(path zone orientation)* Data type: *num*

The zone size (the radius) for the tool reorientation. The size is defined as the distance of the TCP from the programmed point in mm.

The size must be larger than the corresponding value for *pzone_tcp*.
If a lower value is specified, the size is automatically increased to make it the same as *pzone_tcp*.

pzone_eax *(path zone external axes)* Data type: *num*

The zone size (the radius) for external axes. The size is defined as the distance of the TCP from the programmed point in mm.

The size must be larger than the corresponding value for *pzone_tcp*.
If a lower value is specified, the size is automatically increased to make it the same as *pzone_tcp*.

zone_ori *(zone orientation)* Data type: *num*

The zone size for the tool reorientation in degrees. If the robot is holding the work object, this means an angle of rotation for the work object.

zone_leax *(zone linear external axes)* Data type: *num*

The zone size for linear external axes in mm.

zone_reax *(zone rotational external axes)* Data type: *num*

The zone size for rotating external axes in degrees.

Examples

VAR zonedata path := [FALSE, 25, 40, 40, 10, 35, 5];

The zone data *path* is defined by means of the following characteristics:

- The zone size for the TCP path is *25* mm.
- The zone size for the tool reorientation is *40* mm (TCP movement).
- The zone size for external axes is *40* mm (TCP movement).

If the TCP is standing still, or there is a large reorientation, or there is a large external axis movement, with respect to the zone, the following apply instead:

- The zone size for the tool reorientation is *10* degrees.
- The zone size for linear external axes is *35* mm.
- The zone size for rotating external axes is *5* degrees.

path.pzone_tcp := 40;

The zone size for the TCP path is adjusted to *40* mm.

Predefined data

A number of zone data are already defined in the system module *BASE*.

Stop points

Name	
fine	0 mm

Fly-by points

	TCP movement		Tool reorientation			
Name	TCP path	Orientation	Ext. axis	Orientation	Linear axis	Rotating axis
z0	0.3 mm	0.3 mm	0.3 mm	0.03 o	0.3 mm	0.03 o
z1	1 mm	1 mm	1 mm	0.1 o	1 mm	0.1 o
z5	5 mm	8 mm	8 mm	0.8 o	8 mm	0.8 o
z10	10 mm	15 mm	15 mm	1.5 o	15 mm	1.5 o
z15	15 mm	23 mm	23 mm	2.3 o	23 mm	2.3 o
z20	20 mm	30 mm	30 mm	3.0 o	30 mm	3.0 o
z30	30 mm	45 mm	45 mm	4.5 o	45 mm	4.5 o
z40	40 mm	60 mm	60 mm	6.0 o	60 mm	6.0 o
z50	50 mm	75 mm	75 mm	7.5 o	75 mm	7.5 o
z60	60 mm	90 mm	90 mm	9.0 o	90 mm	9.0 o
z80	80 mm	120 mm	120 mm	12 o	120 mm	12 o
z100	100 mm	150 mm	150 mm	15 o	150 mm	15 o
z150	150 mm	225 mm	225 mm	23 o	225 mm	23 o
z200	200 mm	300 mm	300 mm	30 o	300 mm	30 o

Structure

< data object of *zonedata* >
< *finep* of *bool* >
< *pzone_tcp* of *num* >
< *pzone_ori* of *num* >
< *pzone_eax* of *num* >
< *zone_ori* of *num* >
< *zone_leax* of *num* >
< *zone_reax* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Movements/Paths in general	Motion and I/O Principles - *Positioning during Program Execution*
Configuration of external axes	User's Guide - *System Parameters*
Other Stop points	Data Types *stoppointdata*

ABB Automation Technology Products AB
Robotics
SE-721 68 Västerås
SWEDEN
Telephone: +46 (0) 21-34 40 00
Telefax: +46 (0) 21-13 25 92