RAPID Reference Manual

System Data Types and Routines On-line

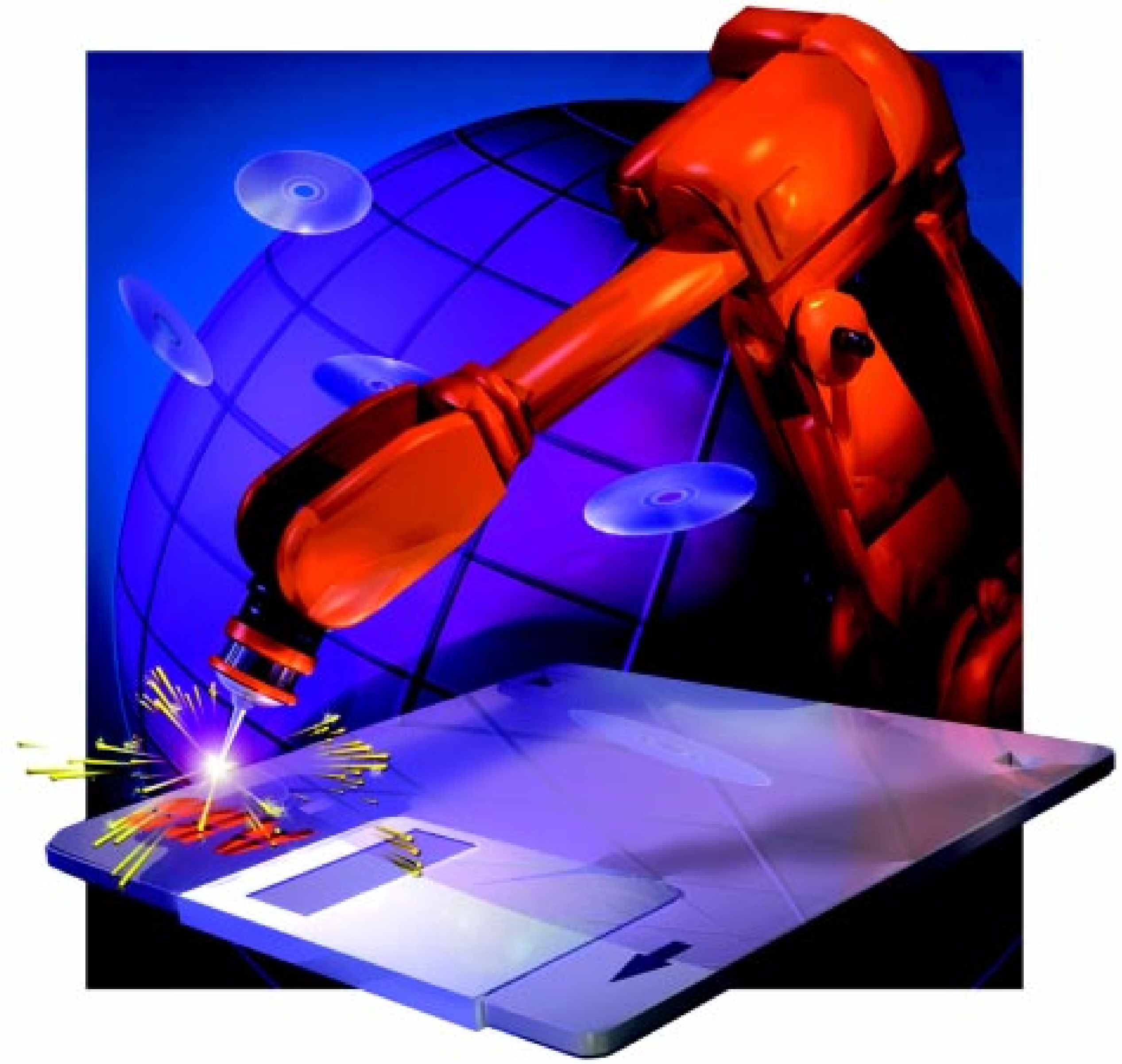


ABB Flexible Automation

ABB Robotics Products AB
DPT / MT
S-72168 VÄSTERÅS
SWEDEN
Telephone: (0) 21 344000
Telefax: (0) 21 132592

ABB

3HAC 0966-13

For BaseWare OS 3.1 Rev.1

System DataTypes and Routines

Data Types and System Data

Instructions

Functions

Index

Article number: 3HAC 0966-13
Issue: For BaseWare OS 3.1 Rev.1

ABB Robotics Products AB
S-721 68 Västerås
Sweden

CONTENTS

wztemporary	Temporary world zone data
zonedata	Zone data

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 1).

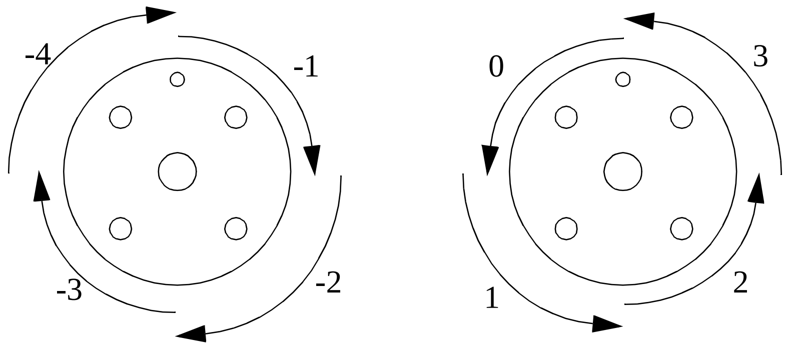


Figure 1 The configuration quadrants for axis 6.

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 2)

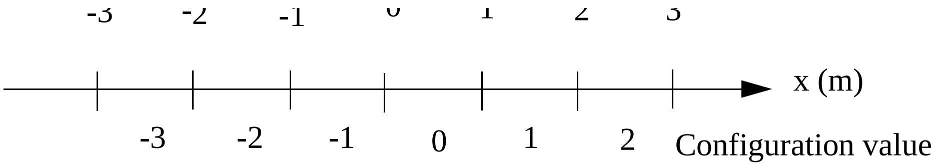


Figure 2 Configuration values for a linear axis

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 6400C

The IRB 6400C requires a slightly different way of unambiguously denoting one robot configuration. The difference lies in the interpretation of the confdata *cf1*.

cf1 is used to select one of two possible main axes (axes 1, 2 and 3) configurations:

- cf1 = 0 is the forward configuration
- cf1 = 1 is the backward configuration.

Figure 3 shows an example of a forward configuration and a backward configuration giving the same position and orientation.

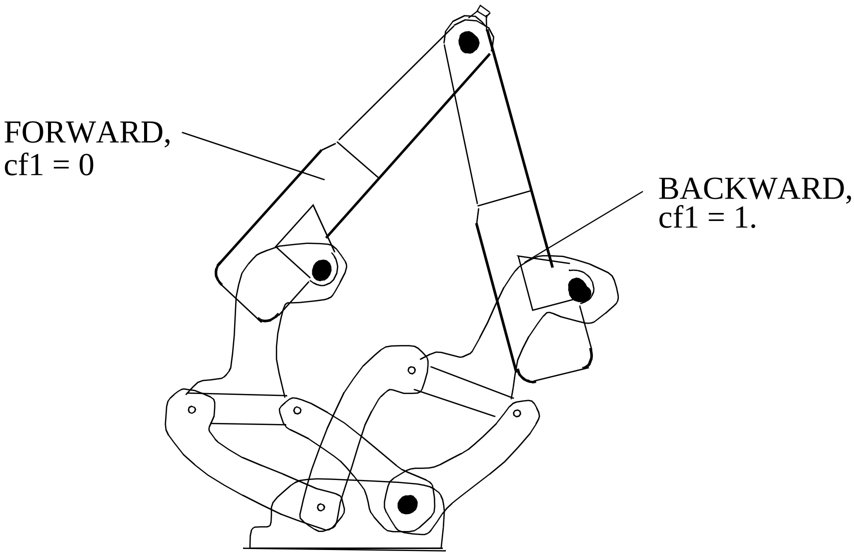


Figure 3 Same position and orientation with two different main axes configurations.

The forward configuration is the front part of the robot's working area with the arm directed forward. The backward configuration is the service part of the working area with the arm directed backwards.

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 IRB5400 robot, 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. 90-180^{o}.
- The axis configuration of the robot axis 4 is quadrant *-1*, i.e. 0-(-90^{o}).
- The axis configuration of the robot axis 6 is quadrant *0*, i.e. 0 - 90^{o}.
- The axis configuration of the robot axis 5 is quadrant *0*, i.e. 0 - 90^{o}.

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.

Error handling

If an argument of the type *dionum* has a value that is neither equal to 0 nor 1, an error is returned on program execution.

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*

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_ALRDYCNT	The interrupt variable is already connected to a TRAP routine
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_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_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_DIVZERO	Division by zero
ERR_EXCRTYMAX	Max. 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_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) valve

ERR_ILLRAISE	Error number in RAISE out of range
ERR_INOMAX	No more interrupt numbers available
ERR_IOENABLE	Timeout when executing IOEnable
ERR_IOERROR	I/O Error from instruction Save
ERR_IODISABLE	Timeout when executing IODisable
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_MAXINTVAL	The integer value is too large
ERR_MODULE	Incorrect module name in instruction Save
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_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_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_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_STEP_PAR	Parameter Step in SetCurrRef is wrong
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*

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 position is defined 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 "flp1: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 defined 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 *GripLoad*) 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 the payload (x, y, z) in mm, expressed in the tool coordinate system.
If a stationary tool is used, it means the centre of gravity for the tool holding the work object.

aom *(axes of moment)* Data type: *orient*

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).

If a stationary tool is used, it means the inertial axes for the tool holding the work object.

Restriction on orientation of atom and tool when an extended load is used, i e ix,iy,iz not all equal zero (point mass).

The orientation of the coordinate system defined by the inertial axes of the payload may only be rotated multiples of +/- 90 degrees about each coordinate axes of the tool coordinate system.

The same restriction applies to the orientation of the tool coordinate system relative to the wrist coordinate system.

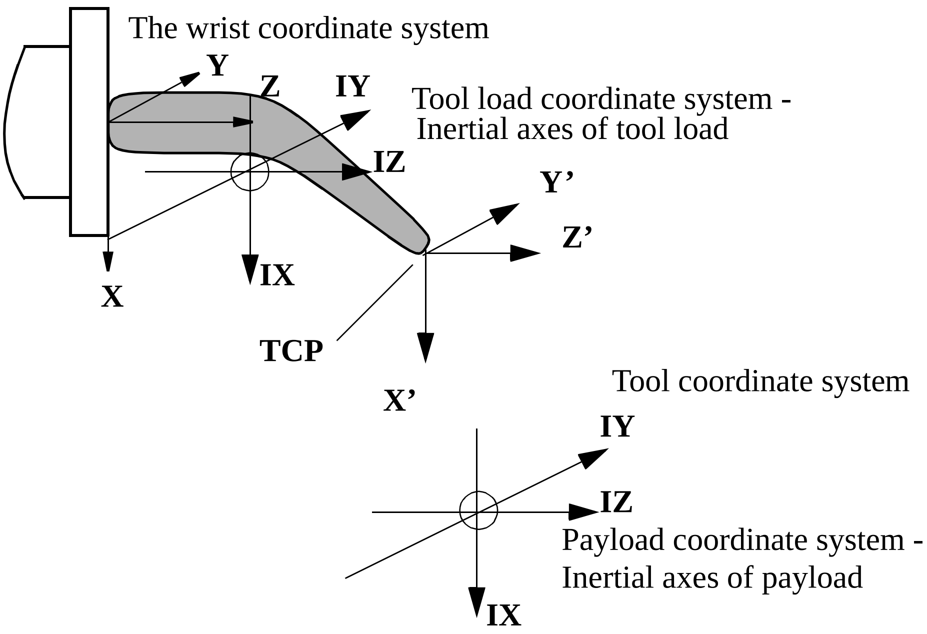


Figure 4 Restriction on the orientation of tool load and payload coordinate system.

ix *(inertia x)* Data type: *num*

The moment of inertia of the load about its IX-axis relative to its centre of mass in kgm^2.

Correct definition of the inertial moments of inertia 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 kgm^2 implies a point mass.

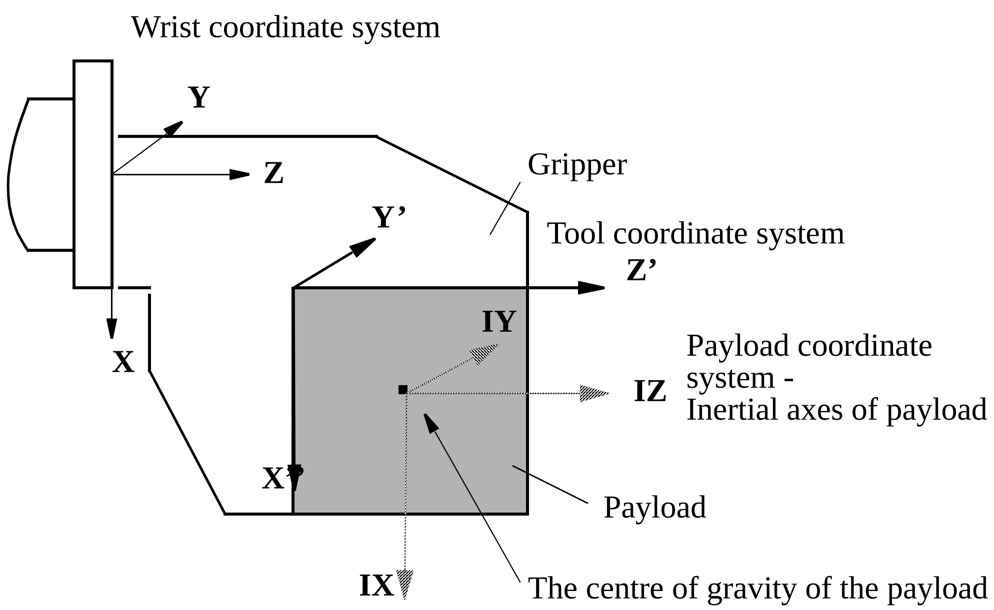


Figure 5 The centre of gravity and inertial axes of the payload.

Normally, the inertial moments of inertia 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 6).

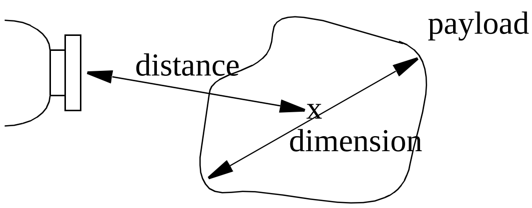


Figure 6 The moment of inertia must normally be defined when the distance is less than the load dimension.

iy *(inertia y)* Data type: *num*

The inertial moment of inertia of the load about its IY-axis, expressed in kgm^2.

For more information, see *ix*.

iz *(inertia z)* Data type: *num*

The inertial moment of inertia of the load about its IZ-axis, expressed in kgm^2.

For more information, see *ix*.

Examples

PERS loaddata piece1 := [5, [50, 0, 50], [1, 0, 0, 0], 0, 0, 0];

The payload in **Figure 4** 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
- Payload
- Override of path resolution
- Motion supervision

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*, *GripLoad*, *PathResol* and *MotionSup*.

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).

grip.load Data type:*gripdata/loaddata*

The payload of the robot (not including the gripper).

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.

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
    [ [ 0, [ 0, 0, 0 ], [ 1, 0, 0, 0 ], 0, 0, 0] ],-> gripdata
    [100 ],-> path resolution
    [TRUE ],-> motionsup
    [100 ] ];-> tunevalue
```

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 >
    <grip of gripdata >            -> Affected by instruction GripLoad
        < load 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 >
    <pathresol of num>             -> Affected by instruction PathResol
    <motionsup of bool>            -> Affected by instruction MotionSup
    <tunevalue of num>             -> Affected by instruction MotionSup
```

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 (= $2*10^3$ = 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*

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 7).

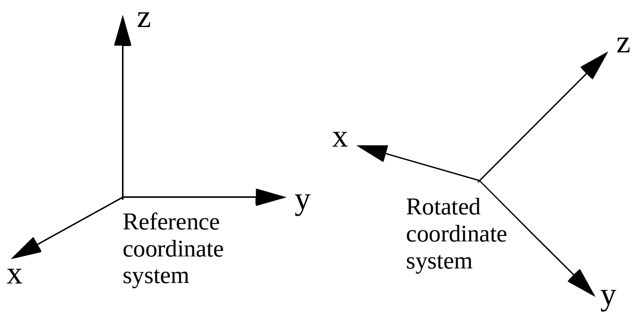


Figure 7 The rotation of a coordinate system is described by a quaternion.

The rotated coordinate systems axes (**x**, **y**, **z**) are vectors which can be expressed in the reference coordinate system as follows:

x = (x_1, x_2, x_3)

y = (y_1, y_2, y_3)

z = (z_1, z_2, z_3)

This means that the x-component of the x-vector in the reference coordinate system will be x_1, the y-component will be x_2, 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:

$$q1 = \frac{\sqrt{x_1 + y_2 + z_3 + 1}}{2}$$

$$q2 = \frac{\sqrt{x_1 - y_2 - z_3 + 1}}{2} \qquad \text{sign } q2 = \text{sign } (y_3 - z_2)$$

$$q3 = \frac{\sqrt{y_2 - x_1 - z_3 + 1}}{2} \qquad \text{sign } q3 = \text{sign } (z_1 - x_3)$$

$$q4 = \frac{\sqrt{z_3 - x_1 - y_2 + 1}}{2} \qquad \text{sign } q4 = \text{sign } (x_2 - y_1)$$

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 8). 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.

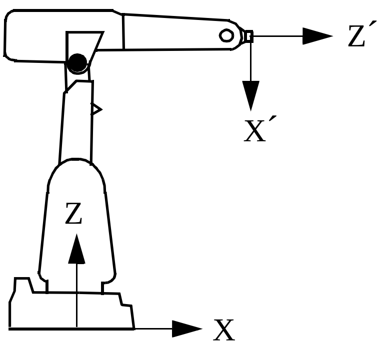


Figure 8 The direction of a tool in accordance with example 1.

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 30^{o} about the X'- and Z'-axes in relation to the wrist coordinate system (see Figure 8). How is the orientation of the tool defined in the tool data?

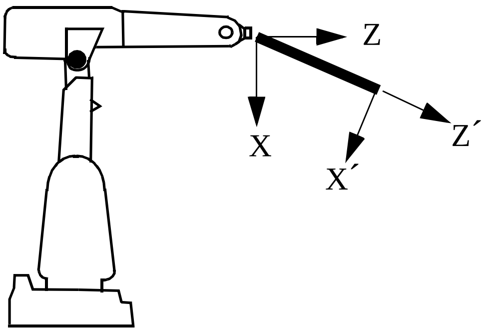


Figure 9 The direction of the tool in accordance with example 2.

The axes will then be related as follows:

x’ = (cos30^{o}, 0, -sin30^{o})

x’ = (0, 1, 0)

x’ = (sin30^{o}, 0, cos30^{o})

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 (sin30^{o}+sin30^{o}) = +

$$q4 = \frac{\sqrt{\cos 30° - \cos 30° - 1 + 1}}{2} = 0$$

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*

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*

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 defined 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 90-180^{o}, axis 6 in position 0-90^{o}.
- 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 3 different 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

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*
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

Data type	Used for
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 a method to retrieve the value. The value of the input signal is returned when a function is called, e.g. *DInput,* or when the variable is used in a value context, e.g. *IF signal_y=1 THEN*.

Limitations

Data of the data type *signalxx* may 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	Motion and I/O Principles - *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	500°/s	5000 mm/s	1000°/s
v10	10 mm/s	500°/s	5000 mm/s	1000°/s
v20	20 mm/s	500°/s	5000 mm/s	1000°/s
v30	30 mm/s	500°/s	5000 mm/s	1000°/s
v40	40 mm/s	500°/s	5000 mm/s	1000°/s
v50	50 mm/s	500°/s	5000 mm/s	1000°/s
v60	60 mm/s	500°/s	5000 mm/s	1000°/s
v80	80 mm/s	500°/s	5000 mm/s	1000°/s
v100	100 mm/s	500°/s	5000 mm/s	1000°/s
v150	150 mm/s	500°/s	5000 mm/s	1000°/s
v200	200 mm/s	500°/s	5000 mm/s	1000°/s
v300	300 mm/s	500°/s	5000 mm/s	1000°/s
v400	400 mm/s	500°/s	5000 mm/s	1000°/s
v500	500 mm/s	500°/s	5000 mm/s	1000°/s
v600	600 mm/s	500°/s	5000 mm/s	1000°/s
v800	800 mm/s	500°/s	5000 mm/s	1000°/s
v1000	1000 mm/s	500°/s	5000 mm/s	1000°/s
v1500	1500 mm/s	500°/s	5000 mm/s	1000°/s
v2000	2000 mm/s	500°/s	5000 mm/s	1000°/s
v2500	2500 mm/s	500°/s	5000 mm/s	1000°/s
v3000	3000 mm/s	500°/s	5000 mm/s	1000°/s
v4000	4000 mm/s	500°/s	5000 mm/s	1000°/s
v5000	5000 mm/s	500°/s	5000 mm/s	1000°/s
vmax	5000 mm/s	500°/s	5000 mm/s	1000°/s
v6000	6000 mm/s	500°/s	5000 mm/s	1000°/s
v7000	7000 mm/s	500°/s	5000 mm/s	1000°/s

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*
Configuration of external axes	User's Guide - *System Parameters*
Motion performance	Product Specification

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”.

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.

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.

Name	Character set
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 constants flp1, ram1disk and stEmpty are already defined in the system module *BASE*.

CONST string flp1 := “flp1:”;

CONST string ram1disk := “ram1disk:”;

CONST string stEmpty := “”;

Related information

	Described in:
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 - payload in gripper - path resolution - motion supervision with tunevalue	motsetdata	Instructions - VelSet - AccSet - SingArea - ConfL,ConfJ - GripLoad - PathResol - MotionSup	Data Types - *motsetdata* Instructions - *VelSet* Instructions - *AccSet* Instructions - *SingArea* Instructions - *ConfL, ConfJ* Instructions - *GripLoad* Instructions - *PathResol* Instructions - *MotionSup*
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*

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.

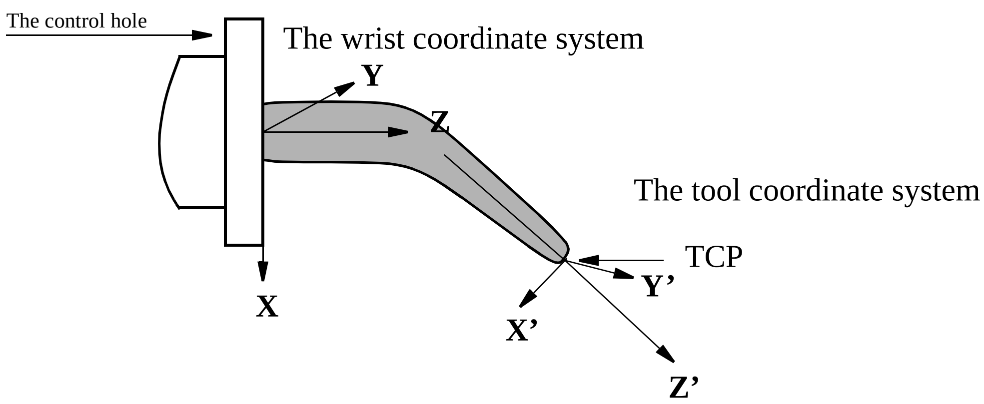


Figure 10 Definition of the tool coordinate system.

tload *(tool load)* Data type: *loaddata*

The load of the tool, i.e.

- mass

- The weight of the tool in kg.

- cog

- The centre of gravity of the tool (x, y and z) in mm, expressed in the wrist coordinate system

- aom

- The orientation of the coordinate system defined by the inertial axes of the tool, expressed in the wrist coordinate system. Note: Restriction on orientation when extended load is used (See loaddata)

- ix

- The moments of inertia of the tool relative to its centre of mass about its IX axis in kgm^2.

- iy

- The moments of inertia of the tool relative to its centre of mass about its IY axis in kgm^2.

- iz

- The moments of inertia of the tool relative to its centre of mass about its IZ axis in kgm^2.

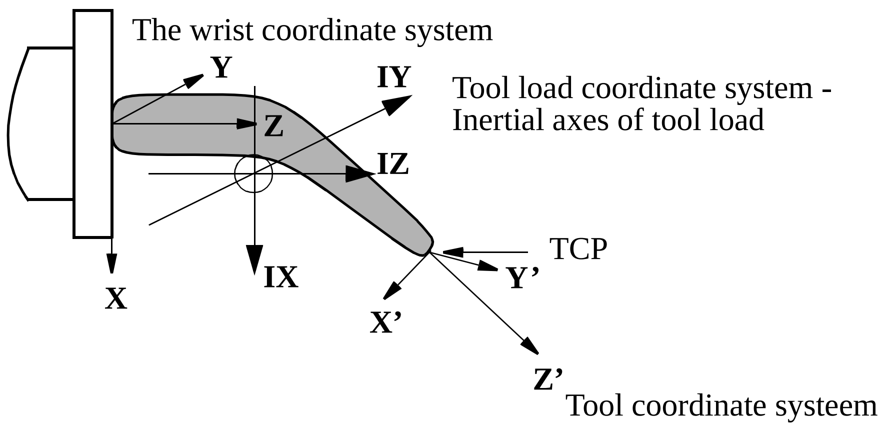


Figure 11 Tool load parameter definitions

If all inertial components are defined as being 0 kgm^2, the tool is handled as a point mass.

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 10 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 45^{o} 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

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*

tunetype Servo tune type

Tunetype is used to represent an integer with a symbolic constant.

Description

A *tunetype* constant is intented 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

The following symbolic constants of the data type *tunetype* are predefined and can be used as arguments for the instruction *SpeedPrioAct* (only available on request).

Value	Symbolic constant	Comment
1	SP_MODE1	Speed priority interpolation mode 1
2	SP_MODE2	Speed priority interpolation mode 2

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*

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 12):

- 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 12):

- 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.

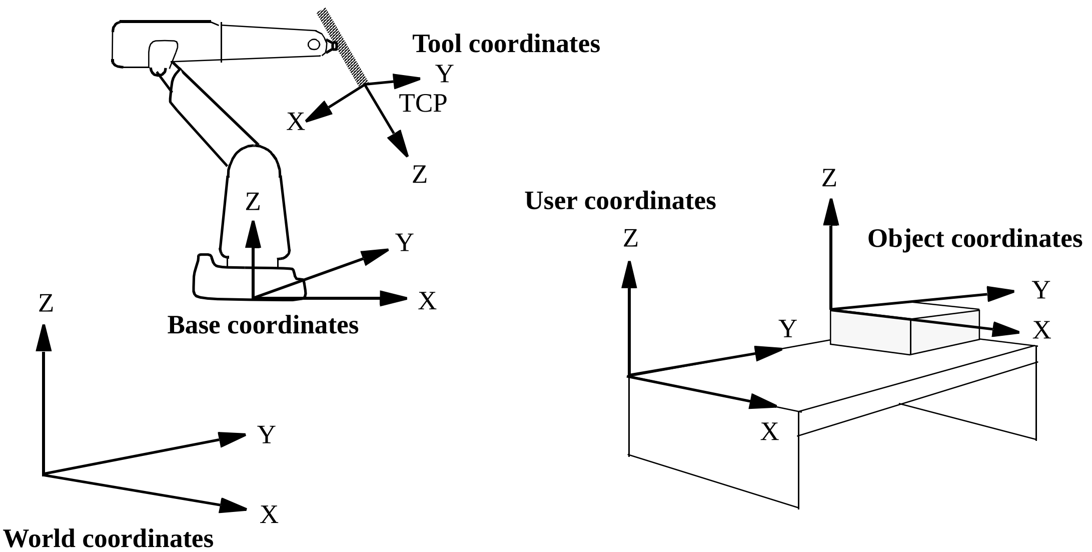


Figure 12 The various coordinate systems of the robot (when the robot is holding the tool).

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 12 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 this world zone, 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 persistent variable for *wzstationary* a numeric value that 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 only be defined as a global (not local within module or routine) variable (VAR) or as a persistent data (PERS).

Arguments of the type *wzstationary* should only be entire data (not array element or record component).

Init value for data of type *wzstationary* is not used by the system. When using a persistent variable in a multi-tasking system, set the init value to 0,
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 this world zone, 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 persistent variable for *wztemporary* a numeric value, that 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 system are automatically erased (erased in the system and all data objects of type *wztemporary* in 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;
    ...
    WZCylDef \Inside, volume, p_center, 400, 1000;
    WZLimSup \Temp, roll, volume;
ENDPROC
```

A *roll* (just being brought into the work area by the application) is defined as a cylinder. If the robot reaches this volume, the movement is stopped.

Limitations

A *wztemporary* data can only be defined as global (not local within module or routine) variable (VAR) or as a persistent data (PERS).

Arguments of the type *wztemporary* should only be entire data (not array element or record component).

A temporary world zone (instructions *WZLimSup* or *WZDOSet*) should not be defined in tasks other than MAIN because such a definition is affected by the program execution in the MAIN task.

Init value for data of type *wztemporary* is not used by the system.When using a persistent variable in a multi-tasking system, set the init value to 0,
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.

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.

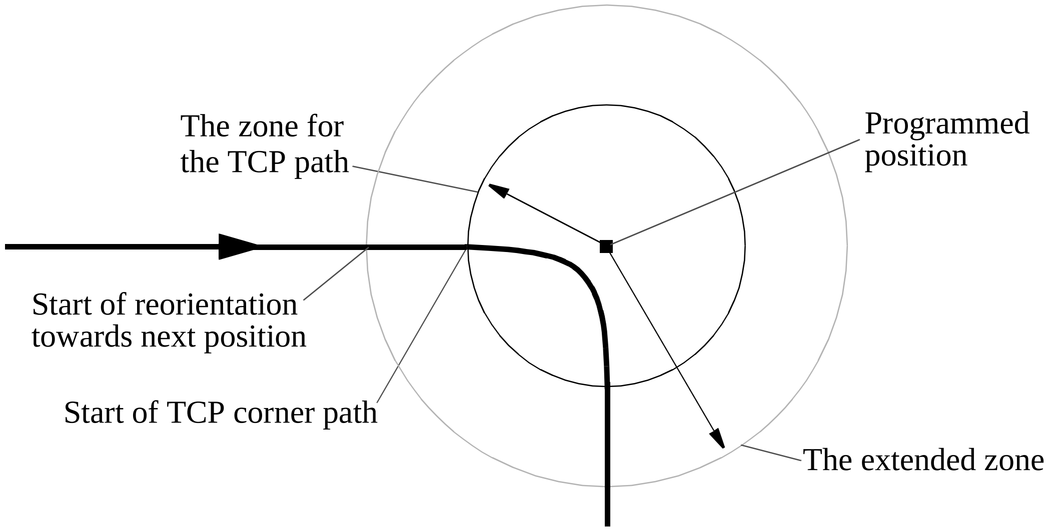


Figure 13 The zones for a fly-by point.

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 13).

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.

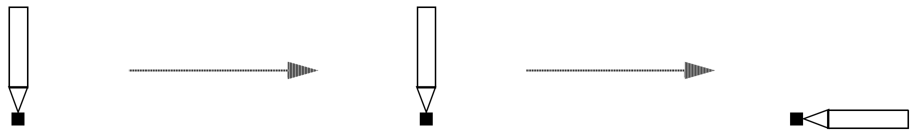


Figure 14a   Three positions are programmed, the last with different tool orientation.

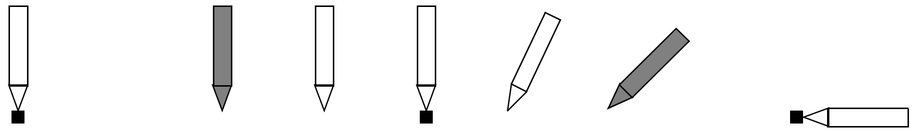


Figure 14b If all positions were stop points, program execution would look like this.

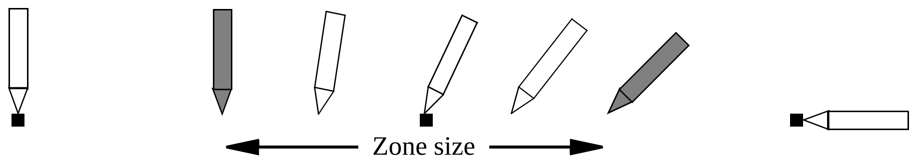


Figure 14c If the middle position was a fly-by point, program execution would look like this

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.

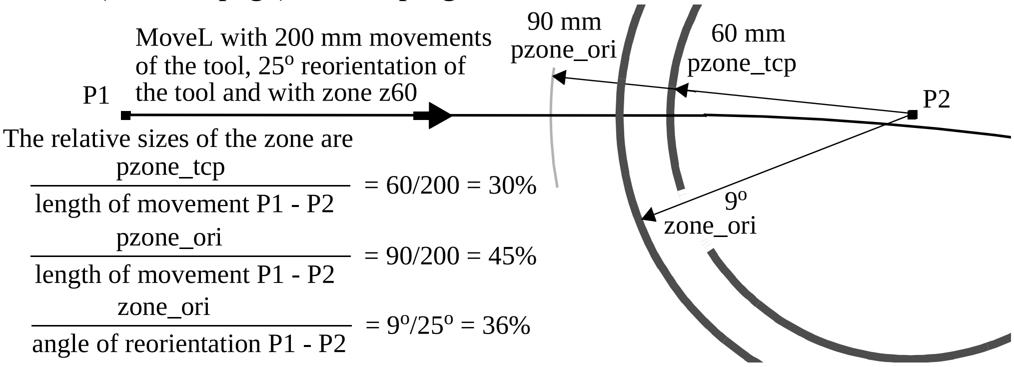


Figure 15 Example of reduced zone to 36% of the motion

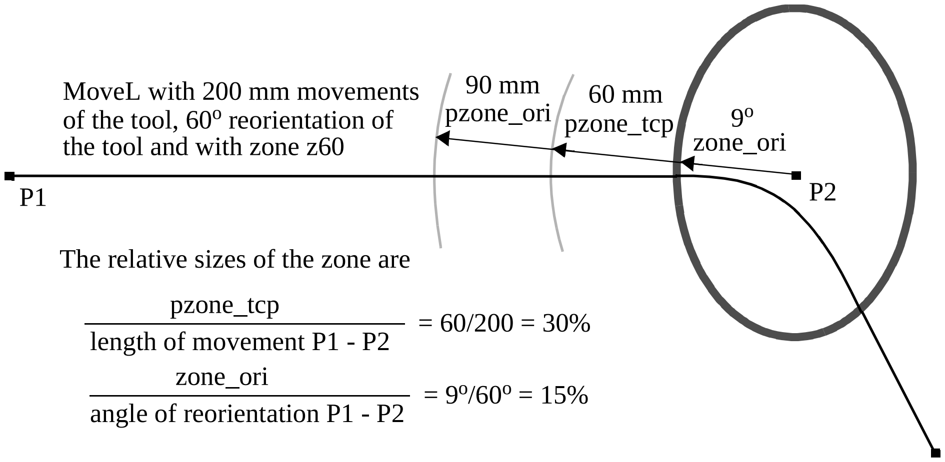


Figure 16 Example of reduced zone to 15% of the motion

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
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*

CONTENTS

IOEnable	Enable I/O unit
ISignalDI	Orders interrupts from a digital input signal
ISignalDO	Interrupts from a digital output signal
ISleep	Deactivates an interrupt
ITimer	Orders a timed interrupt
IVarValue	Orders a variable value interrupt
IWatch	Activates an interrupt
label	Line name
Load	Load a program module during execution
MoveAbsJ	Moves the robot to an absolute joint position
MoveC	Moves the robot circularly
MoveJ	Moves the robot by joint movement
MoveL	Moves the robot linearly
MoveCDO	Moves the robot circularly and sets digital output in the corner
MoveJDO	Moves the robot by joint movement and sets digital output in the corner
MoveLDO	Moves the robot linearly and sets digital output in the corner
MoveCSync	Moves the robot circularly and executes a RAPID procedure
MoveJSync	Moves the robot by joint movement and executes a RAPID procedure
MoveL Sync	Moves the robot linearly and executes a RAPID procedure
Open	Opens a file or serial channel
PathResol	Override path resolution
PDispOff	Deactivates program displacement
PDispOn	Activates program displacement
PDispSet	Activates program displacement using a value
PulseDO	Generates a pulse on a digital output signal
RAISE	Calls an error handler
Reset	Resets a digital output signal
RestoPath	Restores the path after an interrupt
RETRY	Restarts following an error
RETURN	Finishes execution of a routine
Rewind	Rewind file position
Save	Save a program module
SearchC	Searches circularly using the robot
SearchL	Searches linearly using the robot
Set	Sets a digital output signal
SetAO	Changes the value of an analog output signal

SetDO	Changes the value of a digital output signal
SetGO	Changes the value of a group of digital output signals
SingArea	Defines interpolation around singular points
SoftAct	Activating the soft servo
SoftDeact	Deactivating the soft servo
StartLoad	Load a program module during execution
StartMove	Restarts robot motion
Stop	Stops program execution
StopMove	Stops robot motion
StorePath	Stores the path when an interrupt occurs
TEST	Depending on the value of an expression ...
TPErase	Erases text printed on the teach pendant
TPReadFK	Reads function keys
TPReadNum	Reads a number from the teach pendant
TPShow	Switch window on the teach pendant
TPWrite	Writes on the teach pendant
TriggC	Circular robot movement with events
TriggEquip	Defines a fixed position-time I/O event
TriggInt	Defines a position related interrupt
TriggIO	Defines a fixed position I/O event
TriggJ	Axis-wise robot movements with events
TriggL	Linear robot movements with events
TRYNEXT	Jumps over an instruction which has caused an error
TuneReset	Resetting servo tuning
UnLoad	UnLoad a program module during execution
WaitDI	Waits until a digital input signal is set
WaitDO	Waits until a digital output signal is set
WaitLoad	Connect the loaded module to the task
VelSet	Changes the programmed velocity
WHILE	Repeats as long as ...
Write	Writes to a character-based file or serial channel
WriteBin	Writes to a binary serial channel
WriteStrBin	Writes a string to a binary serial channel
WaitTime	Waits a given amount of time
WaitUntil	Waits until a condition is met
WZBoxDef	Define a box-shaped world zone

WZCylDef	Define a cylinder-shaped world zone
WZDisable	Deactivate temporary world zone supervision
WZDOSet	Activate world zone to set digital output
WZEnable	Activate temporary world zone supervision
WZFree	Erase temporary world zone supervision
WZLimSup	Activate world zone limit supervision
WZSphDef	Define a sphere-shaped world zone

“:=” Assigns a value

The “:=” instruction is used to assign a new value to data. This value can be anything from a constant value to an arithmetic expression, e.g. *reg1+5*reg3*.

Examples

reg1 := 5;

reg1 is assigned the value *5*.

reg1 := reg2 - reg3;

reg1 is assigned the value that the *reg2-reg3* calculation returns.

counter := counter + 1;

counter is incremented by one.

Arguments

Data := Value

Data Data type: All

The data that is to be assigned a new value.

Value Data type: Same as Data

The desired value.

Examples

tool1.tframe.trans.x := tool1.tframe.trans.x + 20;

The TCP for *tool1* is shifted *20* mm in the X-direction.

pallet{5,8} := Abs(value);

An element in the *pallet* matrix is assigned a value equal to the absolute value of the *value* variable.

Limitations

The data (whose value is to be changed) must not be

- a constant
- a non-value data type.

The data and value must have similar (the same or alias) data types.

Syntax

(EBNF)
<assignment target> ’:=’ <expression> ’;’
<assignment target> ::=
    <variable>
    | <persistent>
    | <parameter>
    | **<VAR>**

Related information

	Described in:
Expressions	Basic Characteristics - *Expressions*
Non-value data types	Basic Characteristics - *Data Types*
Assigning an initial value to data	Basic Characteristics - *Data* Programming and Testing
Manually assigning a value to data	Programming and Testing

AccSet — Reduces the acceleration

AccSet is used when handling fragile loads. It allows slower acceleration and deceleration, which results in smoother robot movements.

Examples

```
AccSet 50, 100;
```

The acceleration is limited to 50% of the normal value.

```
AccSet 100, 50;
```

The acceleration ramp is limited to 50% of the normal value.

Arguments

AccSet Acc Ramp

Acc Data type: *num*

Acceleration and deceleration as a percentage of the normal values.
100% corresponds to maximum acceleration. Maximum value: 100%.
Input value < 20% gives 20% of maximum acceleration.

Ramp Data type: *num*

The rate at which acceleration and deceleration increases as a percentage of the normal values (see Figure 17). Jerking can be restricted by reducing this value.
100% corresponds to maximum rate. Maximum value: 100%.
Input value < 10% gives 10% of maximum rate.

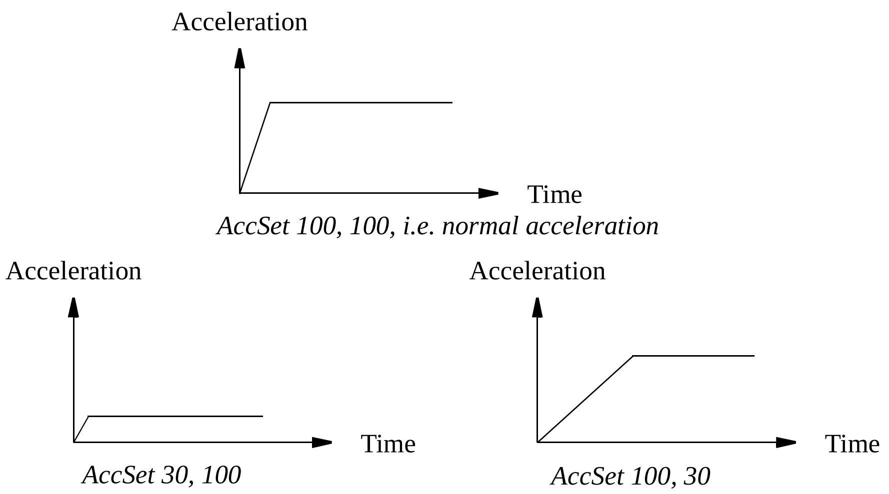


Figure 17 Reducing the acceleration results in smoother movements.

Program execution

The acceleration applies to both the robot and external axes until a new *AccSet* instruction is executed.

The default values (100%) are automatically set

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Syntax

AccSet
    [Acc ’:=’] < expression (**IN**) of *num* > ’,’
    [Ramp ’:=’] < expression (**IN**) of *num* > ’;’

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*

ActUnit Activates a mechanical unit

ActUnit is used to activate a mechanical unit.

It can be used to determine which unit is to be active when, for example, common drive units are used.

Example

ActUnit orbit_a;

Activation of the *orbit_a* mechanical unit.

Arguments

ActUnit MecUnit

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

The name of the mechanical unit that is to be activated.

Program execution

When the robot and external axes have come to a standstill, the specified mechanical unit is activated. This means that it is controlled and monitored by the robot.

If several mechanical units share a common drive unit, activation of one of these mechanical units will also connect that unit to the common drive unit.

Limitations

Instruction ActUnit cannot be used in

- program sequence StorePath ... RestoPath
- event routine RESTART

The movement instruction previous to this instruction, should be terminated with a stop point in order to make a restart in this instruction possible following a power failure.

Syntax

ActUnit
[MecUnit ':='] < variable (**VAR**) of *mecunit*> ';'

Related information

	Described in:
Deactivating mechanical units	Instructions - *DeactUnit*
Mechanical units	Data Types - *mecunit*
More examples	Instructions - *DeactUnit*

Add — Adds a numeric value

Add is used to add or subtract a value to or from a numeric variable or persistent.

Examples

Add reg1, 3;

3 is added to *reg1*, i.e. reg1:=reg1+3.

Add reg1, -reg2;

The value of *reg2* is subtracted from *reg1*, i.e. reg1:=reg1-reg2.

Arguments

Add Name AddValue

Name Data type: *num*

The name of the variable or persistent to be changed.

AddValue Data type: *num*

The value to be added.

Syntax

Add
[Name ’:=’] < var or pers (**INOUT**) of *num* > ’,’
[AddValue ’:=’] < expression (**IN**) of *num* > ’;’

Related information

	Described in:
Incrementing a variable by 1	Instructions - *Incr*
Decrementing a variable by 1	Instructions - *Decr*
Changing data using an arbitrary expression, e.g. multiplication	Instructions - *:=*

Break Break program execution

Break is used to make an immediate break in program execution for RAPID program code debugging purposes.

Example

```
..
Break;
...
```

Program execution stops and it is possible to analyse variables, values etc. for debugging purposes.

Program execution

The instruction stops program execution at once, without waiting for the robot and external axes to reach their programmed destination points for the movement being performed at the time. Program execution can then be restarted from the next instruction.

If there is a *Break* instruction in some event routine, the routine will be executed from the beginning of the next event.

Syntax

Break';'

Related information

	Described in:
Stopping for program actions	Instructions - *Stop*
Stopping after a fatal error	Instructions - *EXIT*
Terminating program execution	Instructions - *EXIT*
Only stopping robot movements	Instructions - *StopMove*

ProcCall Calls a new procedure

A procedure call is used to transfer program execution to another procedure. When the procedure has been fully executed, program execution continues with the instruction following the procedure call.

It is usually possible to send a number of arguments to the new procedure. These control the behaviour of the procedure and make it possible for the same procedure to be used for different things.

Examples

```
weldpipe1;
```

Calls the *weldpipe1* procedure.

```
errormessage;
Set do1;
    .

PROC errormessage()
    TPWrite "ERROR";
ENDPROC
```

The *errormessage* procedure is called. When this procedure is ready, program execution returns to the instruction following the procedure call, *Set do1*.

Arguments

Procedure { Argument }

Procedure Identifier

The name of the procedure to be called.

Argument Data type: In accordance with the procedure declaration

The procedure arguments (in accordance with the parameters of the procedure).

Example

```
weldpipe2 10, lowspeed;
```

Calls the *weldpipe2* procedure, including two arguments.

```
weldpipe3 10 \speed:=20;
```

Calls the *weldpipe3* procedure, including one mandatory and one optional argument.

Limitations

The procedure’s arguments must agree with its parameters:

- All mandatory arguments must be included.
- They must be placed in the same order.
- They must be of the same data type.
- They must be of the correct type with respect to the access-mode (input, variable or persistent).

A routine can call a routine which, in turn, calls another routine, etc. A routine can also call itself, i.e. a recursive call. The number of routine levels permitted depends on the number of parameters, but more than 10 levels are usually permitted.

Syntax

```
(EBNF)
<procedure> [ <argument list> ] ';'

<procedure> ::= <identifier>
```

Related information

	Described in:
Arguments, parameters	Basic Characteristics - *Routines*
More examples	Program Examples

CallByVar Call a procedure by a variable

CallByVar (*Call By Variable*) can be used to call procedures with specific names, e.g. *proc_name1*, *proc_name2*, *proc_name3* ... *proc_namex* via a variable.

Example

```
reg1 := 2;
CallByVar “proc”, reg1;
```

The procedure *proc2* is called.

Arguments

CallByVar Name Number

Name Data type: *string*

The first part of the procedure name, e.g. *proc_name*.

Number Data type: *num*

The numeric value for the number of the procedure. This value will be converted to a string and gives the 2:nd part of the procedure name e.g. *1*. The value must be a positive integer.

Example

Static selection of procedure call

```
TEST reg1
    CASE 1:
        lf_door door_loc;
    CASE 2:
        rf_door door_loc;
    CASE 3:
        lr_door door_loc;
    CASE 4:
        rr_door door_loc;
    DEFAULT:
        EXIT;
ENDTEST
```

Depending on whether the value of register *reg1* is 1, 2, 3 or 4, different procedures are called that perform the appropriate type of work for the selected door.

The door location in argument *door_loc*.

Dynamic selection of procedure call with RAPID syntax

```
reg1 := 2;
%"proc"+NumToStr(reg1,0)% door_loc;
```

The procedure *proc2* is called with argument *door_loc*.

Limitation: All procedures must have a specific name e.g. *proc1*, *proc2*, *proc3*.

Dynamic selection of procedure call with CallByVar

```
reg1 := 2;
CallByVar "proc",reg1;
```

The procedure *proc2* is called.

Limitation: All procedures must have specific name, e.g. *proc1*, *proc2*, *proc3*, and no arguments can be used.

Limitations

Can only be used to call procedures without parameters.

Execution of CallByVar takes a little more time than execution of a normal procedure call.

Error handling

In the event of a reference to an unknown procedure, the system variable ERRNO is set to ERR_REFUNKPRC.

In the event of the procedure call error (not procedure), the system variable ERRNO is set to ERR_CALLPROC.

These errors can be handled in the error handler.

Syntax

CallByVar
    [Name ':='] <expression (**IN**) of *string*>','
    [Number ':='] <expression (**IN**) of *num*>';'

Related information

	Described in:
Calling procedures	Basic Characteristic - *Routines* User's Guide - *The programming language RAPID*

Clear — Clears the value

Clear is used to clear a numeric variable or persistent , i.e. it sets it to 0.

Example

Clear reg1;

Reg1 is cleared, i.e. reg1:=0.

Arguments

Clear Name

Name — Data type: *num*

The name of the variable or persistent to be cleared.

Syntax

Clear
[Name ’:=’] < var or pers (**INOUT**) of *num* > ’;’

Related information

	Described in:
Incrementing a variable by 1	Instructions - *Incr*
Decrementing a variable by 1	Instructions - *Decr*

ClkReset Resets a clock used for timing

ClkReset is used to reset a clock that functions as a stop-watch used for timing.

This instruction can be used before using a clock to make sure that it is set to 0.

Example

ClkReset clock1;

The clock *clock1* is reset.

Arguments

ClkReset Clock

Clock Data type: *clock*

The name of the clock to reset.

Program execution

When a clock is reset, it is set to 0.

If a clock is running, it will be stopped and then reset.

Syntax

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

Related Information

	Described in:
Other clock instructions	RAPID Summary - *System & Time*

ClkStart Starts a clock used for timing

ClkStart is used to start a clock that functions as a stop-watch used for timing.

Example

```
ClkStart clock1;
```

The clock *clock1* is started.

Arguments

ClkStart Clock

Clock Data type: *clock*

The name of the clock to start.

Program execution

When a clock is started, it will run and continue counting seconds until it is stopped.

A clock continues to run when the program that started it is stopped. However, the event that you intended to time may no longer be valid. For example, if the program was measuring the waiting time for an input, the input may have been received while the program was stopped. In this case, the program will not be able to “see” the event that occurred while the program was stopped.

A clock continues to run when the robot is powered down as long as the battery back-up retains the program that contains the clock variable.

If a clock is running it can be read, stopped or reset.

Example

```
VAR clock clock2;

ClkReset clock2;
ClkStart clock2;
WaitUntil DInput(di1) = 1;
ClkStop clock2;
time:=ClkRead(clock2);
```

The waiting time for *di1* to become 1 is measured.

Syntax

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

Related Information

	Described in:
Other clock instructions	RAPID Summary - *System & Time*

ClkStop Stops a clock used for timing

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

Example

ClkStop clock1;

The clock *clock1* is stopped.

Arguments

ClkStop Clock

Clock Data type: *clock*

The name of the clock to stop.

Program execution

When a clock is stopped, it will stop running.

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

Syntax

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

Related Information

	Described in:
Other clock instructions	RAPID Summary - *System & Time*
More examples	Instructions - *ClkStart*

Close Closes a file or serial channel

Close is used to close a file or serial channel.

Example

Close channel2;

The serial channel referred to by *channel2* is closed.

Arguments

Close IODevice

IODevice Data type: *iodev*

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

Program execution

The specified file or serial channel is closed and must be re-opened before reading or writing. If it is already closed, the instruction is ignored.

Syntax

Close
[IODevice ':='] <variable (**VAR**) of *iodev*>';'

Related information

	Described in:
Opening a file or serial channel	RAPID Summary - *Communication*

comment Comment

Comment is only used to make the program easier to understand. It has no effect on the execution of the program.

Example

```
! Goto the position above pallet
MoveL p100, v500, z20, tool1;
```

A comment is inserted into the program to make it easier to understand.

Arguments

! Comment

Comment Text string

Any text.

Program execution

Nothing happens when you execute this instruction.

Syntax

(EBNF)
’!’ {<character>} <newline>

Related information

	Described in:
Characters permitted in a comment	Basic Characteristics- *Basic Elements*
Comments within data and routine declarations	Basic Characteristics- *Basic Elements*

ConfJ Controls the configuration during joint movement

ConfJ (Configuration Joint) is used to specify whether or not the robot's configuration is to be controlled during joint movement. If it is not controlled, the robot can sometimes use a different configuration than that which was programmed.

With ConfJ\Off, the robot cannot switch main axes configuration - it will search for a solution with the same main axes configuration as the current one. It moves to the closest wrist configuration for axes 4 and 6.

Examples

```
ConfJ \Off;
MoveJ *, v1000, fine, tool1;
```

The robot moves to the programmed position and orientation. If this position can be reached in several different ways, with different axis configurations, the closest possible position is chosen.

```
ConfJ \On;
MoveJ *, v1000, fine, tool1;
```

The robot moves to the programmed position, orientation and axis configuration. If this is not possible, program execution stops.

Arguments

ConfJ [\On] | [\Off]

\On

Data type: *switch*

The robot always moves to the programmed axis configuration. If this is not possible using the programmed position and orientation, program execution stops.

The IRB5400 robot will move to the pogrammed axis configuration or to an axis configuration close the the programmed one. Program execution will not stop if it is impossible to reach the programmed axis configuration.

\Off

Data type: *switch*

The robot always moves to the closest axis configuration.

Program execution

If the argument *\On* (or no argument) is chosen, the robot always moves to the programmed axis configuration. If this is not possible using the programmed position and orientation, program execution stops before the movement starts.

If the argument *\Off* is chosen, the robot always moves to the closest axis configuration. This may be different to the programmed one if the configuration has been incorrectly specified manually, or if a program displacement has been carried out.

The control is active by default. This is automatically set

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Syntax

ConfJ
  ['\' On] | ['\' Off] ';'

Related information

	Described in:
Handling different configurations	Motion Principles - *Robot Configuration*
Robot configuration during linear movement	Instructions - *ConfL*

ConfL Monitors the configuration during linear movement

ConfL (Configuration Linear) is used to specify whether or not the robot's configuration is to be monitored during linear or circular movement. If it is not monitored, the configuration at execution time may differ from that at programmed time. It may also result in unexpected sweeping robot movements when the mode is changed to joint movement.

NOTE: For the IRB5400 robot the monotoring is always off independant of the switch.

Examples

ConfL \On;
MoveL *, v1000, fine, tool1;

Program execution stops when the programmed configuration is not possible to reach from the current position.

SingArea \Wrist;
Confl \On;
MoveL *, v1000, fine, tool1;

The robot moves to the programmed position, orientation and wrist axis configuration. If this is not possible, program execution stops.

ConfL \Off;
MoveL *, v1000, fine, tool1;

No error message is displayed when the programmed configuration is not the same as the configuration achieved by program execution.

Arguments

ConfL [\On] | [\Off]

\On Data type: *switch*

The robot configuration is monitored.

\Off Data type: *switch*

The robot configuration is not monitored.

Program execution

During linear or circular movement, the robot always moves to the programmed position and orientation that has the closest possible axis configuration. If the argument *\On* (or no argument) is chosen, then the program execution stops as soon as:

- the configuration of the programmed position will not be attained from the current position.
- the needed reorientation of any one of the wrist axes to get to the programmed position from the current position exceeds a limit (140-180 degrees).

However, it is possible to restart the program again, although the wrist axes may continue to the wrong configuration. At a stop point, the robot will check that the configurations of all axes are achieved, not only the wrist axes.

If SingArea\Wrist is also used, the robot always moves to the programmed wrist axes configuration and at a stop point the remaining axes configurations will be checked.

If the argument *\Off* is chosen, there is no monitoring.

Monitoring is active by default. This is automatically set

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Syntax

```
ConfL
   [ '\' On] | [ '\' Off] ';'
```

Related information

	Described in:
Handling different configurations	Motion and I/O Principles- *Robot Configuration*
Robot configuration during joint movement	Instructions - *ConfJ*

CONNECT Connects an interrupt to a trap routine

CONNECT is used to find the identity of an interrupt and connect it to a trap routine.

The interrupt is defined by ordering an interrupt event and specifying its identity. Thus, when that event occurs, the trap routine is automatically executed.

Example

```
VAR intnum feeder_low;
CONNECT feeder_low WITH feeder_empty;
ISignalDI di1, 1 , feeder_low;
```

An interrupt identity *feeder_low* is created which is connected to the trap routine *feeder_empty*. The interrupt is defined as *input di1 is getting high*. In other words, when this signal becomes high, the *feeder_empty* trap routine is executed.

Arguments

CONNECT Interrupt WITH Trap routine

Interrupt Data type: *intnum*

The variable that is to be assigned the identity of the interrupt.
This must not be declared within a routine (routine data).

Trap routine Identifier

The name of the trap routine.

Program execution

The variable is assigned an interrupt identity which can then be used when ordering or disabling interrupts. This identity is also connected to the specified trap routine.

Note that before an event can be handled, an interrupt must also be ordered, i.e. the event specified.

Limitations

An interrupt (interrupt identity) cannot be connected to more than one trap routine. Different interrupts, however, can be connected to the same trap routine.

When an interrupt has been connected to a trap routine, it cannot be reconnected or transferred to another routine; it must first be deleted using the instruction *IDelete*.

Error handling

If the interrupt variable is already connected to a TRAP routine, the system variable ERRNO is set to ERR_ALRDYCNT.

If the interrupt variable is not a variable reference, the system variable ERRNO is set to ERR_CNTNOTVAR.

If no more interrupt numbers are available, the system variable ERRNO is set to ERR_INOMAX.

These errors can be handled in the ERROR handler.

Syntax

(EBNF)
CONNECT <connect target> **WITH** <trap>‘;’

<connect target> ::= <variable>
| <parameter>
| <VAR>
<trap> ::= <identifier>

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
More information on interrupt management	Basic Characteristics- *Interrupts*

DeactUnit Deactivates a mechanical unit

DeactUnit is used to deactivate a mechanical unit.

It can be used to determine which unit is to be active when, for example, common drive units are used.

Examples

```
DeactUnit orbit_a;
```

Deactivation of the *orbit_a* mechanical unit.

```
MoveL p10, v100, fine, tool1;
DeactUnit track_motion;
MoveL p20, v100, z10, tool1;
MoveL p30, v100, fine, tool1;
ActUnit track_motion;
MoveL p40, v100, z10, tool1;
```

The unit *track_motion* will be stationary when the robot moves to *p20* and *p30*. After this, both the robot and *track_motion* will move to *p40*.

```
MoveL p10, v100, fine, tool1;
DeactUnit orbit1;
ActUnit orbit2;
MoveL p20, v100, z10, tool1;
```

The unit *orbit1* is deactivated and *orbit2* activated.

Arguments

DeactUnit MecUnit

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

The name of the mechanical unit that is to be deactivated.

Program execution

When the robot and external axes have come to a standstill, the specified mechanical unit is deactivated. This means that it will neither be controlled nor monitored until it is re-activated.

If several mechanical units share a common drive unit, deactivation of one of the mechanical units will also disconnect that unit from the common drive unit.

Limitations

Instruction DeactUnit cannot be used

- in program sequence StorePath ... RestoPath
- in event routine RESTART
- when one of the axes in the mechanical unit is in independent mode.

The movement instruction previous to this instruction, should be terminated with a stop point in order to make a restart in this instruction possible following a power failure.

Syntax

DeactUnit
[MecUnit ':='] < variable (**VAR**) of *mecunit*> ';'

Related information

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

Decr Decrements by 1

Decr is used to subtract 1 from a numeric variable or persistent.

Example

```
Decr reg1;
```

1 is subtracted from *reg1*, i.e. reg1:=reg1-1.

Arguments

Decr Name

Name Data type: *num*

The name of the variable or persistent to be decremented.

Example

```
TPReadNum no_of_parts, "How many parts should be produced? ";
WHILE no_of_parts>0 DO
    produce_part;
    Decr no_of_parts;
ENDWHILE
```

The operator is asked to input the number of parts to be produced. The variable *no_of_parts* is used to count the number that still have to be produced.

Syntax

Decr
    [Name ’:=’] < var or pers (**INOUT**) of *num* > ’;’

Related information

	Described in:
Incrementing a variable by 1	Instructions - *Incr*
Subtracting any value from a variable	Instructions - *Add*
Changing data using an arbitrary expression, e.g. multiplication	Instructions - *:=*

EOffsOff Deactivates an offset for external axes

EOffsOff (External Offset Off) is used to deactivate an offset for external axes.

The offset for external axes is activated by the instruction *EOffsSet* or *EOffsOn* and applies to all movements until some other offset for external axes is activated or until the offset for external axes is deactivated.

Examples

```
EOffsOff;
```

Deactivation of the offset for external axes.

```
MoveL p10, v500, z10, tool1;
EOffsOn \ExeP:=p10, p11;
MoveL p20, v500, z10, tool1;
MoveL p30, v500, z10, tool1;
EOffsOff;
MoveL p40, v500, z10, tool1;
```

An offset is defined as the difference between the position of each axis at *p10* and *p11*. This displacement affects the movement to *p20* and *p30*, but not to *p40*.

Program execution

Active offsets for external axes are reset.

Syntax

```
EOffsOff ';'
```

Related information

	Described in:
Definition of offset using two positions	Instructions - *EOffsOn*
Definition of offset using values	Instructions - *EOffsSet*
Deactivation of the robot's motion displacement	Instructions - *PDispOff*

EOffsOn Activates an offset for external axes

EOffsOn (External Offset On) is used to define and activate an offset for external axes using two positions.

Examples

```
MoveL p10, v500, z10, tool1;
EOffsOn \ExeP:=p10, p20;
```

Activation of an offset for external axes. This is calculated for each axis based on the difference between positions *p10* and *p20*.

```
MoveL p10, v500, fine, tool1;
EOffsOn *;
```

Activation of an offset for external axes. Since a stop point has been used in the previous instruction, the argument \ExeP does not have to be used. The displacement is calculated on the basis of the difference between the actual position of each axis and the programmed point (*) stored in the instruction.

Arguments

EOffsOn [\ExeP] ProgPoint

[\ExeP] *(Executed Point)* Data type: *robtarget*

The new position of the axes at the time of the program execution. If this argument is omitted, the current position of the axes at the time of the program execution is used.

ProgPoint *(Programmed Point)* Data type: *robtarget*

The original position of the axes at the time of programming.

Program execution

The offset is calculated as the difference between *ExeP* and *ProgPoint* for each separate external axis. If *ExeP* has not been specified, the current position of the axes at the time of the program execution is used instead. Since it is the actual position of the axes that is used, the axes should not move when *EOffsOn* is executed.

This offset is then used to displace the position of external axes in subsequent positioning instructions and remains active until some other offset is activated (the instruction

EOffsSet or *EOffsOn*) or until the offset for external axes is deactivated (the instruction *EOffsOff*).

Only one offset for each individual external axis can be activated at any one time. Several *EOffsOn*, on the other hand, can be programmed one after the other and, if they are, the different offsets will be added.

The external axes' offset is automatically reset

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Example

```
SearchL sen1, psearch, p10, v100, tool1;
PDispOn \ExeP:=psearch, *, tool1;
EOffsOn \ExeP:=psearch, *;
```

A search is carried out in which the searched position of both the robot and the external axes is stored in the position *psearch*. Any movement carried out after this starts from this position using a program displacement of both the robot and the external axes. This is calculated based on the difference between the searched position and the programmed point (*) stored in the instruction.

Syntax

EOffsOn
  ['\' ExeP ':=' < expression (**IN**) of *robtarget* > ',']
  [ProgPoint ':='] < expression (**IN**) of *robtarget* > ';'

Related information

	Described in:
Deactivation of offset for external axes	Instructions - *EOffsOff*
Definition of offset using values	Instructions - *EOffsSet*
Displacement of the robot's movements	Instructions - *PDispOn*
Coordinate Systems	Motion Principles- *Coordinate Systems*

EOffsSet Activates an offset for external axes using a value

EOffsSet (External Offset Set) is used to define and activate an offset for external axes using values.

Example

```
VAR extjoint eax_a_p100 := [100, 0, 0, 0, 0, 0];
.
EOffsSet eax_a_p100;
```

Activation of an offset *eax_a_p100* for external axes, meaning (provided that the external axis "a" is linear) that:

- The ExtOffs coordinate system is displaced 100 mm for the logical axis "a" (see Figure 18).
- As long as this offset is active, all positions will be displaced 100 mm in the direction of the x-axis.

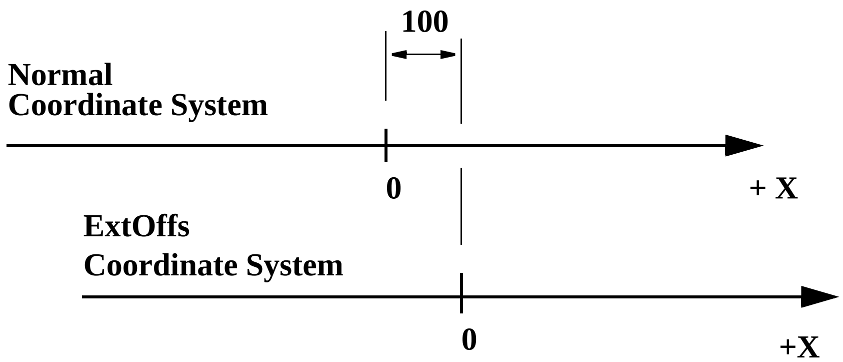


Figure 18 Displacement of an external axis.

Arguments

EOffsSet EAxOffs

EAxOffs *(External Axes Offset)* Data type: *extjoint*

The offset for external axes is defined as data of the type *extjoint*, expressed in:

- mm for linear axes
- degrees for rotating axes

Program execution

The offset for external axes is activated when the *EOffsSet* instruction is activated and remains active until some other offset is activated (the instruction *EOffsSet* or *EOffsOn*) or until the offset for external axes is deactivated (the *EOffsOff*).

Only one offset for external axes can be activated at any one time. Offsets cannot be added to one another using *EOffsSet*.

The external axes’ offset is automatically reset

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Syntax

EOffsSet
[EAxOffs ’:=’] < expression (**IN**) of *extjoint*> ’;’

Related information

	Described in:
Deactivation of offset for external axes	Instructions - *EOffsOff*
Definition of offset using two positions	Instructions - *EOffsSet*
Displacement of the robot’s movements	Instructions - *PDispOn*
Definition of data of the type *extjoint*	Data Types - *extjoint*
Coordinate Systems	Motion Principles- *Coordinate Systems*

ErrWrite Write an Error Message

ErrWrite (Error Write) is used to display an error message on the teach pendant and write it in the robot message log.

Example

ErrWrite “PLC error”, “Fatal error in PLC” \RL2:=”Call service”;
Stop;

A message is stored in the robot log. The message is also shown on the teach pendant display.

ErrWrite \ W, “ Search error”, “No hit for the first search”;
RAISE try_search_again;

A message is stored in the robot log only. Program execution then continues.

Arguments

ErrWrite [\W] Header Reason [\RL2] [\RL3] [\RL4]

[\W] *(Warning)* Data type: *switch*

Gives a warning that is stored in the robot error message log only (not shown directly on the teach pendant display).

Header Data type: *string*

Error message heading (max. 24 characters).

Reason Data type: string

Reason for error (line 1 of max. 40 characters).

[\RL2] *(Reason Line 2)* Data type: *string*

Reason for error (line 2 of max. 40 characters).

[\RL3] *(Reason Line 3)* Data type: *string*

Reason for error (line 3 of max. 40 characters).

[\RL4] *(Reason Line 4)* Data type: *string*

Reason for error (line 4 of max. 40 characters).

Program execution

An error message (max. 5 lines) is displayed on the teach pendant and written in the robot message log.

ErrWrite always generates the program error no. 80001 or in the event of a warning (argument \W) generates no. 80002.

Limitations

Total string length (Header+Reason+\RL2+\RL3+\RL4) is limited to 145 characters.

Syntax

ErrWrite
  [’\’ W ’,’]
  [Header ’:=’] < expression (**IN**) of *string*> ‘,’
  [Reason ’:=’] < expression (**IN**) of *string*>
  [’\’ RL2 ’:=’ < expression (**IN**) of *string*>]
  [’\’ RL3 ’:=’ < expression (**IN**) of *string*>]
  [’\’ RL4 ’:=’ < expression (**IN**) of *string*>] ‘;’

Related information

	Described in:
Display a message on the teach pendant only	Instructions - *TPWrite*
Message logs	Service

EXIT Terminates program execution

EXIT is used to terminate program execution. Program restart will then be blocked, i.e. the program can only be restarted from the first instruction of the main routine (if the start point is not moved manually).

The *EXIT* instruction should be used when fatal errors occur or when program execution is to be stopped permanently. The *Stop* instruction is used to temporarily stop program execution.

Example

```
ErrWrite "Fatal error","Illegal state";
EXIT;
```

Program execution stops and cannot be restarted from that position in the program.

Syntax

EXIT ';'

Related information

	Described in:
Stopping program execution temporarily	Instructions - *Stop*

ExitCycle Break current cycle and start next

ExitCycle is used to break the current cycle and move the PP back to the first instruction in the main routine. If the execution mode CONT is set, the execution will start to execute the next cycle.

Example

```
VAR num cyclecount:=0;
VAR intnum error_intno;

PROC main()
    IF cyclecount = 0 THEN
        CONNECT error_intno WITH error_trap;
        ISignalDI di_error,1,error_intno;
    ENDIF
    cyclecount:=cyclecount+1;
    ! start to do something intelligent
    ....

ENDPROC

TRAP error_trap
    TPWrite "ERROR, I will start on the next item";
    ExitCycle;
ENDTRAP
```

This will start the next cycle if the signal di_error is set.

Program Running

All variables, persistents, defined interrupts and motion settings are untouched.

Syntax

ExitCycle';'

Related information

	Described in:
Stopping after a fatal error	Instructions - *EXIT*
Terminating program execution	Instructions - *EXIT*
Stopping for program actions	Instructions - *Stop*
Finishing execution of a routine	Instructions - *RETURN*

FOR Repeats a given number of times

FOR is used when one or several instructions are to be repeated a number of times.

If the instructions are to be repeated as long as a given condition is met, the *WHILE* instruction is used.

Example

```
FOR i FROM 1 TO 10 DO
    routine1;
ENDFOR
```

Repeats the *routine1* procedure *10* times.

Arguments

FOR Loop counter FROM Start value TO End value [STEP Step value] DO ... ENDFOR

Loop counter Identifier

The name of the data that will contain the value of the current loop counter. The data is declared automatically and its name should therefore not be the same as the name of any data that exists already.

Start value Data type: *Num*

The desired start value of the loop counter. (usually integer values)

End value Data type: *Num*

The desired end value of the loop counter. (usually integer values)

Step value Data type: *Num*

The value by which the loop counter is to be incremented (or decremented) each loop. (usually integer values)

If this value is not specified, the step value will automatically be set to 1 (or -1 if the start value is greater than the end value).

Example

```
FOR i FROM 10 TO 2 STEP -1 DO
    a{i} := a{i-1};
ENDFOR
```

The values in an array are adjusted upwards so that a{10}:=a{9}, a{9}:=a{8} etc.

Program execution

1. The expressions for the start, end and step values are calculated.
2. The loop counter is assigned the start value.
3. The value of the loop counter is checked to see whether its value lies between the start and end value, or whether it is equal to the start or end value. If the value of the loop counter is outside of this range, the FOR loop stops and program execution continues with the instruction following ENDFOR.
4. The instructions in the FOR loop are executed.
5. The loop counter is incremented (or decremented) in accordance with the step value.
6. The FOR loop is repeated, starting from point 3.

Limitations

The loop counter (of data type *num*) can only be accessed from within the FOR loop and consequently hides other data and routines that have the same name. It can only be read (not updated) by the instructions in the FOR loop.

Decimal values for start, end or stop values, in combination with exact termination conditions for the FOR loop, cannot be used (undefined whether or not the last loop is running).

Syntax

(EBNF)
FOR <loop variable> **FROM** <expression> **TO** <expression>
  [**STEP** <expression>] **DO**
  <instruction list>
ENDFOR

<loop variable> ::= <identifier>

Related information

	Described in:
Expressions	Basic Characteristics - *Expressions*
Identifiers	Basic Characteristics - *Basic Elements*

GetSysData Get system data

GetSysData fetches the value and optional symbol name for the current system data of specified data type.

With this instruction it is possible to fetch data for and the name of the current active Tool or Work Object.

Example

```
PERS tooldata curtoolvalue := [TRUE, [[0, 0, 0], [1, 0, 0, 0]],
                                [0, [0, 0, 0], [1, 0, 0, 0], 0, 0, 0]];
VAR string curtoolname;

GetSysData curtoolvalue;
```

Copy current active tool data value to the persistent variable *curtoolvalue*.

```
GetSysData curtoolvalue \ObjectName := curtoolname;
```

Copy also current active tool name to the variable *curtoolname*.

Arguments

GetSysData DestObject [\ ObjectName]

DestObject Data type: *anytype*

Persistent for storage of current active system data value.

The data type of this argument also specifies the type of system data (Tool or Work Object) to fetch.

[\ObjectName] Data type: *string*

Option argument (variable or persistent) to also fetch the current active system data name.

Program execution

When running the instruction *GetSysData* the current data value is stored in the specified persistent in argument *DestObject*.

If argument *\ObjectName* is used, the name of the current data is stored in the specified variable or persistent in argument *ObjectName*.

Current system data for Tool or Work Object is activated by execution of any move instruction or can be manually set in the jogging window.

Syntax

GetSysData
[DestObject':='] < persistent(**PERS**) of *anytype*>
['\'ObjectName':=' < expression (**INOUT**) of *string*>] ';'

Related information

	Described in:
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*

GOTO Goes to a new instruction

GOTO is used to transfer program execution to another line (a label) within the same routine.

Examples

```
GOTO next;
    .
next:
```

Program execution continues with the instruction following *next*.

```
reg1 := 1;
next:
    .
reg1 := reg1 + 1;
IF reg1<=5 GOTO next;
```

The *next* program loop is executed five times.

```
IF reg1>100 GOTO highvalue;
lowvalue:
    .
GOTO ready;
highvalue:
    .
ready:
```

If *reg1 is greater than 100*, the *highvalue* program loop is executed; otherwise the *lowvalue* loop is executed.

Arguments

GOTO Label

Label Identifier

The label from where program execution is to continue.

Limitations

It is only possible to transfer program execution to a label within the same routine.

It is only possible to transfer program execution to a label within an IF or TEST instruction if the GOTO instruction is also located within the same branch of that instruction.

It is only possible to transfer program execution to a label within a FOR or WHILE instruction if the GOTO instruction is also located within that instruction.

Syntax

(EBNF)
GOTO <identifier>';'

Related information

	Described in:
Label	Instructions - *label*
Other instructions that change the program flow	RAPID Summary - *Controlling the Program Flow*

GripLoad Defines the payload of the robot

GripLoad is used to define the payload which the robot holds in its gripper.

Description

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

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

Examples

GripLoad piece1;

The robot gripper holds a load called *piece1*.

GripLoad load0;

The robot gripper releases all loads.

Arguments

GripLoad Load

Load Data type: *loaddata*

The load data that describes the current payload.

Program execution

The specified load affects the performance of the robot.

The default load, 0 kg, is automatically set

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Syntax

GripLoad
[Load ’:=’] < persistent (**PERS**) of *loaddata* > ’;’

Related information

	Described in:
Definition of load data	Data Types - *loaddata*
Definition of tool load	Data Types - *tooldata*

-

IDelete Cancels an interrupt

IDelete (Interrupt Delete) is used to cancel (delete) an interrupt.

If the interrupt is to be only temporarily disabled, the instruction *ISleep* or *IDisable* should be used.

Example

IDelete feeder_low;

The interrupt *feeder_low* is cancelled.

Arguments

IDelete Interrupt

Interrupt Data type: *intnum*

The interrupt identity.

Program execution

The definition of the interrupt is completely erased. To define it again, it must first be re-connected to the trap routine.

The instruction should be preceded by a stop point. Otherwise the interrupt will be deactivated before the end point is reached.

Interrupts do not have to be erased; this is done automatically when

- a new program is loaded
- the program is restarted from the beginning
- the program pointer is moved to the start of a routine

Syntax

IDelete
[Interrupt ':='] < variable (**VAR**) of *intnum* > ';'

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Temporarily disabling an interrupt	Instructions - *ISleep*
Temporarily disabling all interrupts	Instructions - *IDisable*

IDisable Disables interrupts

IDisable (Interrupt Disable) is used to disable all interrupts temporarily. It may, for example, be used in a particularly sensitive part of the program where no interrupts may be permitted to take place in case they disturb normal program execution.

Example

```
IDisable;
FOR i FROM 1 TO 100 DO
    character[i]:=ReadBin(sensor);
ENDFOR
IEnable;
```

No interrupts are permitted as long as the serial channel is reading.

Program execution

Interrupts which occur during the time in which an *IDisable* instruction is in effect are placed in a queue. When interrupts are permitted once more, the interrupt(s) of the program then immediately start generating, executed in "first in - first out" order in the queue.

Syntax

IDisable';'

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupt*
Permitting interrupts	Instructions - *IEnable*

IEnable — Enables interrupts

IEnable (Interrupt Enable) is used to enable interrupts during program execution.

Example

```
IDisable;
FOR i FROM 1 TO 100 DO
    character[i]:=ReadBin(sensor);
ENDFOR
IEnable;
```

No interrupts are permitted as long as the serial channel is reading. When it has finished reading, interrupts are once more permitted.

Program execution

Interrupts which occur during the time in which an *IDisable* instruction is in effect, are placed in a queue. When interrupts are permitted once more (*IEnable*), the interrupt(s) of the program then immediately start generating, executed in “first in - first out” order in the queue.Program execution then continues in the ordinary program and interrupts which occur after this are dealt with as soon as they occur.

Interrupts are always permitted when a program is started from the beginning,. Interrupts disabled by the *ISleep* instruction are not affected by the *IEnable* instruction.

Syntax

IEnable‘;’

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Permitting no interrupts	Instructions - *IDisable*

Compact IF   If a condition is met, then... (one instruction)

Compact IF is used when a single instruction is only to be executed if a given condition is met.

If different instructions are to be executed, depending on whether the specified condition is met or not, the *IF* instruction is used.

Examples

IF reg1 > 5 GOTO next;

If *reg1 is greater than 5*, program execution continues at the *next* label.

IF counter > 10 Set do1;

The *do1* signal is set if *counter > 10*.

Arguments

IF   Condition   ...

Condition Data type: *bool*

The condition that must be satisfied for the instruction to be executed.

Syntax

(EBNF)
IF <conditional expression> (<instruction> | **<SMT>**) ';'

Related information

	Described in:
Conditions (logical expressions)	Basic Characteristics - *Expressions*
IF with several instructions	Instructions - *IF*

IF If a condition is met, then ...; otherwise ...

IF is used when different instructions are to be executed depending on whether a condition is met or not.

Examples

```
IF reg1 > 5 THEN
    Set do1;
    Set do2;
ENDIF
```

The *do1* and *do2* signals are set only if *reg1 is greater than 5*.

```
IF reg1 > 5 THEN
    Set do1;
    Set do2;
ELSE
    Reset do1;
    Reset do2;
ENDIF
```

The *do1* and *do2* signals are set or reset depending on whether *reg1 is greater than 5* or not.

Arguments

IF Condition THEN ...
{ELSEIF Condition THEN ...}
[ELSE ...]
ENDIF

Condition Data type: *bool*

The condition that must be satisfied for the instructions between THEN and ELSE/ELSEIF to be executed.

Example

```
IF counter > 100 THEN
    counter := 100;
ELSEIF counter < 0 THEN
    counter := 0;
ELSE
    counter := counter + 1;
```

ENDIF

Counter is incremented by 1. However, if the value of *counter* is outside the limit *0-100*, *counter* is assigned the corresponding limit value.

Program execution

The conditions are tested in sequential order, until one of them is satisfied. Program execution continues with the instructions associated with that condition. If none of the conditions are satisfied, program execution continues with the instructions following ELSE. If more than one condition is met, only the instructions associated with the first of those conditions are executed.

Syntax

(EBNF)
IF <conditional expression> **THEN**
<instruction list>
{**ELSEIF** <conditional expression> **THEN** <instruction list> | **<EIF>**}
[**ELSE**
<instruction list>]
ENDIF

Related information

	Described in:
Conditions (logical expressions)	Basic Characteristics - *Expressions*

Incr Increments by 1

Incr is used to add 1 to a numeric variable or persistent.

Example

```
Incr reg1;
```

1 is added to *reg1*, i.e. reg1:=reg1+1.

Arguments

Incr Name

Name Data type: *num*

The name of the variable or persistent to be changed.

Example

```
WHILE stop_production=0 DO
    produce_part;
    Incr no_of_parts;
    TPWrite "No of produced parts= "\Num:=no_of_parts;
ENDWHILE
```

The number of parts produced is updated on the teach pendant each cycle. Production continues to run as long as the signal *stop_production* is not set.

Syntax

Incr
    [Name ':='] < var or pers (**INOUT**) of *num* > ';'

Related information

	Described in:
Decrementing a variable by 1	Instructions - *Decr*
Adding any value to a variable	Instructions - *Add*
Changing data using an arbitrary expression, e.g. multiplication	Instructions - *:=*

InvertDO Inverts the value of a digital output signal

InvertDO (Invert Digital Output) inverts the value of a digital output signal (0 -> 1 and 1 -> 0).

Example

InvertDO do15;

The current value of the signal *do15* is inverted.

Arguments

InvertDO Signal

Signal Data type: *signaldo*

The name of the signal to be inverted.

Program execution

The current value of the signal is inverted (see Figure 19).

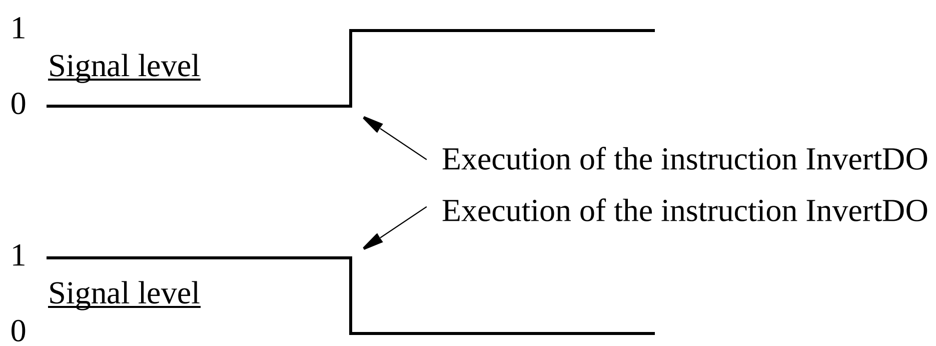


Figure 19 Inversion of a digital output signal.

Syntax

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

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	System Parameters

IODisable Disable I/O unit

IODisable is used to disable an I/O unit during program execution (only in the S4C system).

I/O units are automatically enabled after start-up if they are defined in the system parameters. When required for some reason, I/O units can be disabled or enabled during program execution.

Examples

IODisable “cell1”, 5;

Disable I/O unit with name *cell1*.Wait max. *5* s.

Arguments

IODisable UnitName MaxTime

UnitName Data type: *string*

The name of the I/O unit to be disabled (with same name as configured).

MaxTime Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the I/O unit has finished the disable steps, the error handler will be called, if there is one, with the error code ERR_IODISABLE. If there is no error handler, the execution will be stopped.

To disable an I/O unit takes about 2-5 s.

Program execution

The specified I/O unit starts the disable steps. The instruction is ready when the disable steps are finished. If the *MaxTime* runs out before the I/O unit has finished the disable steps, a recoverable error will be generated.

After disabling an I/O unit, any setting of outputs in this unit will result in an error.

Example

```
PROC go_home()
    VAR num recover_flag :=0;
    ...
    ! Start to disable I/O unit cell1
    recover_flag := 1;
    IODisable "cell1", 0;
    ! Move to home position
    MoveJ home, v1000,fine,tool1;
    ! Wait until disable of I/O unit cell1 is ready
    recover_flag := 2;
    IODisable "cell1", 5;
    ...
    ERROR
        IF ERRNO = ERR_IODISABLE THEN
            IF recover_flag = 1 THEN
                TRYNEXT;
            ELSEIF recover_flag = 2 THEN
                RETRY;
            ENDIF
        ELSEIF ERRNO = ERR_EXCRTYMAX THEN
            ErrWrite "IODisable error", "Not possible to disable I/O unit cell1";
            Stop;
        ENDIF
ENDPROC
```

To save cycle time, the I/O unit *cell1* is disabled during robot movement to the *home* position. With the robot at the *home* position, a test is done to establish whether or not the I/O unit *cell1* is fully disabled. After the max. number of retries (5 with a waiting time of *5* s), the robot execution will stop with an error message.

The same principle can be used with *IOEnable* (this will save more cycle time compared with *IODisable*).

Syntax

IODisable
    [UnitName ':='] < expression (**IN**) of *string*> ','
    [MaxTime ':='] < expression (**IN**) of *num* > ';'

Related information

	Described in:
Enabling an I/O unit	Instructions - *IOEnabl*e
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*

IOEnable Enable I/O unit

IOEnable is used to enable an I/O unit during program execution (only in the S4C system).

I/O units are automatically enabled after start-up if they are defined in the system parameters. When required for some reason, I/O units can be disabled or enabled during program execution.

Examples

IOEnable “cell1”, 5;

Enable I/O unit with name *cell1*. Wait max. *5* s.

Arguments

IOEnable UnitName MaxTime

UnitName Data type: *string*

The name of the I/O unit to be enabled (with same name as configured).

MaxTime Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the I/O unit has finished the enable steps, the error handler will be called, if there is one, with the error code ERR_IOENABLE. If there is no error handler, the execution will be stopped.

To enable an I/O unit takes about 2-5 s.

Program execution

The specified I/O unit starts the enable steps. The instruction is ready when the enable steps are finished. If the *MaxTime* runs out before the I/O unit has finished the enable steps, a recoverable error will be generated.

After a sequence of *IODisable - IOEnable*, all outputs for the current I/O unit will be set to the old values (before *IODisable*).

Example

IOEnable can also be used to check whether some I/O unit is disconnected for some reason.

```
VAR num max_retry:=0;
...
IOEnable "cell1", 0;
SetDO cell1_sig3, 1;
...
ERROR
    IF ERRNO = ERR_IOENABLE THEN
        IF max_retry < 5 THEN
            WaitTime 1;
            max_retry := max_retry + 1;
            RETRY;
        ELSE
            RAISE;
        ENDIF
    ENDIF
```

Before using signals on the I/O unit *cell1*, a test is done by trying to enable the I/O unit with timeout after *0* sec. If the test fails, a jump is made to the error handler. In the error handler, the program execution waits for *1* sec. and a new retry is made. After *5* retry attempts the error ERR_IOENABLE is propagated to the caller of this routine.

Syntax

IOEnable
[UnitName ':='] < expression (**IN**) of *string*> ','
[MaxTime ':='] < expression (**IN**) of *num* > ';'

Related information

	Described in:
More examples	Instructions - *IODisable*
Disabling an I/O unit	Instructions - *IODisable*
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*

ISignalDI Orders interrupts from a digital input signal

ISignalDI (Interrupt Signal Digital In) is used to order and enable interrupts from a digital input signal.

System signals can also generate interrupts.

Examples

```
VAR intnum sig1int;
CONNECT sig1int WITH iroutine1;
ISignalDI di1,1,sig1int;
```

Orders an interrupt which is to occur each time the digital input signal *di1* is set to *1*. A call is then made to the *iroutine1* trap routine.

```
ISignalDI di1,0,sig1int;
```

Orders an interrupt which is to occur each time the digital input signal *di1* is set to *0*.

```
ISignalDI \Single, di1,1,sig1int;
```

Orders an interrupt which is to occur only the first time the digital input signal *di1* is set to *1*.

Arguments

ISignalDI [\Single] Signal TriggValue Interrupt

[\Single] Data type: *switch*

Specifies whether the interrupt is to occur once or cyclically.

If the argument *Single* is set, the interrupt occurs once at the most. If the argument is omitted, an interrupt will occur each time its condition is satisfied.

Signal Data type: *signaldi*

The name of the signal that is to generate interrupts.

TriggValue Data type: *dionum*

The value to which the signal must change for an interrupt to occur.

The value is specified as 0 or 1 or as a symbolic value (e.g. *high/low*). The signal is edge-triggered upon changeover to 0 or 1.

TriggValue 2 or symbolic value *edge* can be used for generation of interrupts on both positive flank (0 -> 1) and negative flank (1 -> 0).

Interrupt Data type: *intnum*

The interrupt identity. This should have previously been connected to a trap routine by means of the instruction *CONNECT*.

Program execution

When the signal assumes the specified value, a call is made to the corresponding trap routine. When this has been executed, program execution continues from where the interrupt occurred.

If the signal changes to the specified value before the interrupt is ordered, no interrupt occurs (see Figure 20).

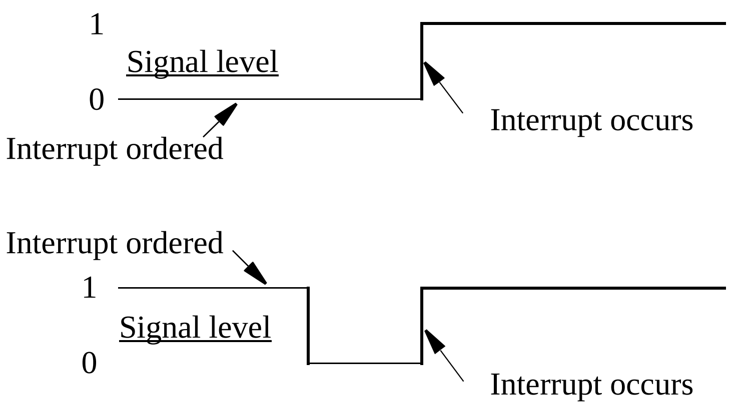


Figure 20 Interrupts from a digital input signal at signal level 1.

Limitations

The same variable for interrupt identity cannot be used more than once, without first deleting it. Interrupts should therefore be handled as shown in one of the alternatives below.

```
PROC main ( )
    VAR intnum sig1int;
    CONNECT sig1int WITH iroutine1;
    ISignalDI di1, 1, sig1int;
    WHILE TRUE DO
    :
    :
    ENDWHILE
ENDPROC
```

All activation of interrupts is done at the beginning of the program. These instructions are then kept outside the main flow of the program.

```
PROC main ( )
    VAR intnum sig1int;
    CONNECT sig1int WITH iroutine1;
    ISignalDI di1, 1, sig1int;
    :
    :
    IDelete sig1int;
ENDPROC
```

The interrupt is deleted at the end of the program, and is then reactivated. It should be noted, in this case, that the interrupt is inactive for a short period.

Syntax

ISignalDI
[’\’ Single’,’]
[Signal ’:=’] < variable (**VAR**) of *signaldi* > ’,’
[TriggValue ’:=’] < expression (**IN**) of *dionum* >’,’
[Interrupt ’:=’] < variable (**VAR**) of *intnum* > ’;’

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Interrupt from an output signal	Instructions - *ISignalDO*
More information on interrupt management	Basic Characteristics - *Interrupts*
More examples	Data Types - *intnum*

ISignalDO Interrupts from a digital output signal

ISignalDO (Interrupt Signal Digital Out) is used to order and enable interrupts from a digital output signal.

System signals can also generate interrupts.

Examples

```
VAR intnum sig1int;
CONNECT sig1int WITH iroutine1;
ISignalDO do1,1,sig1int;
```

Orders an interrupt which is to occur each time the digital output signal *do1* is set to *1*. A call is then made to the *iroutine1* trap routine.

```
ISignalDO do1,0,sig1int;
```

Orders an interrupt which is to occur each time the digital output signal *do1* is set to *0*.

```
ISignalDO\Single, do1,1,sig1int;
```

Orders an interrupt which is to occur only the first time the digital output signal *do1* is set to *1*.

Arguments

ISignalDO [\Single] Signal TriggValue Interrupt

[\Single] Data type: *switch*

Specifies whether the interrupt is to occur once or cyclically.

If the argument *Single* is set, the interrupt occurs once at the most. If the argument is omitted, an interrupt will occur each time its condition is satisfied.

Signal Data type: *signaldo*

The name of the signal that is to generate interrupts.

TriggValue Data type: *dionum*

The value to which the signal must change for an interrupt to occur.

The value is specified as 0 or 1 or as a symbolic value (e.g. *high/low*). The signal is edge-triggered upon changeover to 0 or 1.

TriggValue 2 or symbolic value *edge* can be used for generation of interrupts on both positive flank (0 -> 1) and negative flank (1 -> 0).

Interrupt Data type: *intnum*

The interrupt identity. This should have previously been connected to a trap routine by means of the instruction *CONNECT.*

Program execution

When the signal assumes the specified value 0 or 1, a call is made to the corresponding trap routine. When this has been executed, program execution continues from where the interrupt occurred.

If the signal changes to the specified value before the interrupt is ordered, no interrupt occurs (see Figure 21).

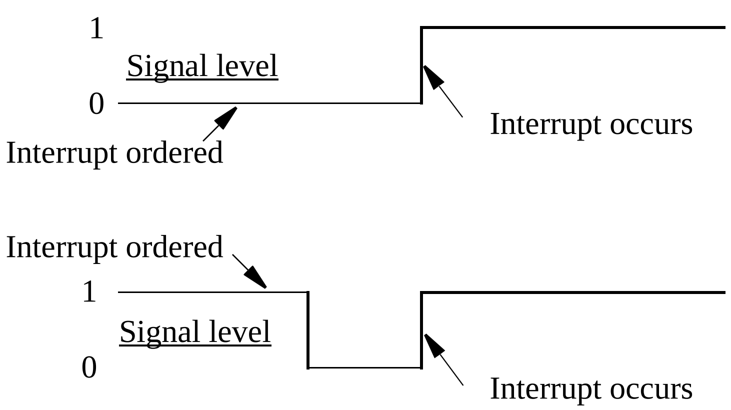


Figure 21 Interrupts from a digital output signal at signal level 1.

Limitations

The same variable for interrupt identity cannot be used more than once, without first deleting it. Interrupts should therefore be handled as shown in one of the alternatives below.

```
PROC main ( )
    VAR intnum sig1int;
    CONNECT sig1int WITH iroutine1;
    ISignalDO do1, 1, sig1int;
    WHILE TRUE DO
    :
    :
    ENDWHILE
ENDPROC
```

All activation of interrupts is done at the beginning of the program. These instructions are then kept outside the main flow of the program.

```
PROC main ( )
    VAR intnum sig1int;
    CONNECT sig1int WITH iroutine1;
    ISignalDO do1, 1, sig1int;
    :
    :
    IDelete sig1int;
ENDPROC
```

The interrupt is deleted at the end of the program, and is then reactivated. It should be noted, in this case, that the interrupt is inactive for a short period.

Syntax

ISignalDO
    [’\’ Single’,’]
    [Signal ’:=’] < variable (**VAR**) of *signaldo* > ’,’
    [TriggValue ’:=’] < expression (**IN**) of *dionum* >’,’
    [Interrupt ’:=’] < variable (**VAR**) of *intnum* > ’;’

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Interrupt from an input signal	Instructions - *ISignalDI*
More information on interrupt management	Basic Characteristics- *Interrupts*
More examples	Data Types - *intnum*

ISleep Deactivates an interrupt

ISleep (Interrupt Sleep) is used to deactivate an individual interrupt temporarily.

Example

```
ISleep sig1int;
```

The interrupt *sig1int* is deactivated.

Arguments

ISleep Interrupt

Interrupt Data type: *intnum*

The variable (interrupt identity) of the interrupt.

Program execution

The event connected to this interrupt does not generate any interrupts until the interrupt has been re-activated by means of the instruction *IWatch*. Interrupts which are generated whilst *ISleep* is in effect are ignored.

Example

```
VAR intnum timeint;
CONNECT timeint WITH check_serialch;
ITimer 60, timeint;
.
ISleep timeint;
WriteBin ch1, buffer, 30;
IWatch timeint;
.
TRAP check_serialch
    WriteBin ch1, buffer, 1;
    IF ReadBin(ch1\Time:=5) < 0 THEN
        TPWrite "The serial communication is broken";
        EXIT;
    ENDIF
ENDTRAP
```

Communication across the ch1 serial channel is monitored by means of interrupts which are generated every *60* seconds. The trap routine checks whether the communication is working. When, however, communication is in progress, these interrupts are not permitted.

Error handling

Interrupts which have neither been ordered nor enabled are not permitted. If the interrupt number is unknown, the system variable ERRNO will be set to ERR_UNKINO (see "Data types - errnum"). The error can be handled in the error handler.

Syntax

ISleep
[Interrupt ':='] < variable (**VAR**) of *intnum* > ';'

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Enabling an interrupt	Instructions - *IWatch*
Disabling all interrupts	Instructions - *IDisable*
Cancelling an interrupt	Instructions - *IDelete*

ITimer Orders a timed interrupt

ITimer (Interrupt Timer) is used to order and enable a timed interrupt.

This instruction can be used, for example, to check the status of peripheral equipment once every minute.

Examples

```
VAR intnum timeint;
CONNECT timeint WITH iroutine1;
ITimer 60, timeint;
```

Orders an interrupt that is to occur cyclically every *60* seconds. A call is then made to the trap routine *iroutine1*.

```
ITimer \Single, 60, timeint;
```

Orders an interrupt that is to occur once, after *60* seconds.

Arguments

ITimer [\Single] Time Interrupt

[\Single] Data type: *switch*

Specifies whether the interrupt is to occur once or cyclically.

If the argument *Single* is set, the interrupt occurs only once. If the argument is omitted, an interrupt will occur each time at the specified time.

Time Data type: *num*

The amount of time that must lapse before the interrupt occurs.

The value is specified in second if *Single* is set, this time may not be less than 0.05 seconds. The corresponding time for cyclical interrupts is 0.25 seconds.

Interrupt Data type: *intnum*

The variable (interrupt identity) of the interrupt. This should have previously been connected to a trap routine by means of the instruction *CONNECT.*

Program execution

The corresponding trap routine is automatically called at a given time following the interrupt order. When this has been executed, program execution continues from where the interrupt occurred.

If the interrupt occurs cyclically, a new computation of time is started from when the interrupt occurs.

Example

```
VAR intnum timeint;
CONNECT timeint WITH check_serialch;
ITimer 60, timeint;
.
TRAP check_serialch
    WriteBin ch1, buffer, 1;
    IF ReadBin(ch1\Time:=5) < 0 THEN
        TPWrite "The serial communication is broken";
        EXIT;
    ENDIF
ENDTRAP
```

Communication across the ch1 serial channel is monitored by means of interrupts which are generated every *60* seconds. The trap routine checks whether the communication is working. If it is not, program execution is interrupted and an error message appears.

Limitations

The same variable for interrupt identity cannot be used more than once, without being first deleted. See Instructions - *ISignalDI*.

Syntax

ITimer
    ['\'Single ',']
    [Time ':='] < expression (**IN**) of *num* >','
    [Interrupt ':='] < variable (**VAR**) of *intnum* > ';'

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
More information on interrupt management	Basic Characteristics- *Interrupts*

IVarValue Orders a variable value interrupt

IVarVal(Interrupt Variable Value) is used to order and enable an interrupt when the value of a variable accessed via the serial sensor interface has been changed.

This instruction can be used, for example, to get seam volume or gap values from a seam tracker.

Examples

```
LOCAL PERS num adtVlt{25}:=[1,1.2,1.4,1.6,1.8,2,2.16667,2.33333,2.5,...];
LOCAL PERS num adptWfd{25}:=[2,2.2,2.4,2.6,2.8,3,3.16667,3.33333,3.5,...];
LOCAL PERS num adptSpd{25}:=10,12,14,16,18,20,21.6667,23.3333,25[,...];
LOCAL CONST num GAP_VARIABLE_NO:=11;
PERS num gap_value;
VAR intnum IntAdap;

PROC main()
! Setup the interrupt. The trap routine AdapTrp will be called
! when the gap variable with number 'GAP_VARIABLE_NO' in
! the sensor interface has been changed. The new value will be available
! in the PERS gp_value variable.
    CONNECT IntAdap WITH AdapTrp;
    IVarValue GAP_VARIABLE_NO, gap_value, IntAdap;

    ! Start welding
    ArcL\On,*,v100,adaptSm,adaptWd,adaptWv,z10,tool\j\Track:=track;
    ArcL\On,*,v100,adaptSm,adaptWd,adaptWv,z10,tool\j\Track:=track;
ENDPROC

TRAP AdapTrap
    VAR num ArrInd;

    !Scale the raw gap value received
    ArrInd:=ArrIndx(gap_value);

    ! Update active welddata PERS variable 'adaptWd' with
    ! new data from the arrays of predefined parameter arrays.
    ! The scaled gap value is used as index in the voltage, wirefeed and speed arrays.
    adaptWd.weld_voltage:=adptVlt{ArrInd};
    adaptWd.weld_wirefeed:=adptWfd{ArrInd};
    adaptWd.weld_speed:=adptSpd{ArrInd};

    !Request a refresh of AW parameters using the new data i adaptWd
    ArcRefresh;
ENDTRAP
```

Arguments

IVarValue VarNo Value, Interrupt

VarNo Data type: *num*

The number of the variable to be supervised.

Value Data type: *num*

A PERS variable which will hold the new value of Varno.

Interrupt Data type: *intnum*

The variable (interrupt identity) of the interrupt. This should have previously been connected to a trap routine by means of the instruction *CONNECT*.

Program execution

The corresponding trap routine is automatically called at a given time following the interrupt order. When this has been executed, program execution continues from where the interrupt occurred.

Limitations

The same variable for interrupt identity cannot be used more than five times, without first being deleted.

Syntax

IVarValue
[VarNo ':='] < expression (**IN**) of *num* >','
[Value ':='] < persistent(**PERS**) of *num* >','
[Interrupt ':='] < variable (**VAR**) of *intnum* > ';'

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
More information on interrupt management	Basic Characteristics- *Interrupts*

IWatch Activates an interrupt

IWatch (Interrupt Watch) is used to activate an interrupt which was previously ordered but was deactivated with *ISleep*.

Example

```
IWatch sig1int;
```

The interrupt *sig1int* that was previously deactivated is activated.

Arguments

IWatch Interrupt

Interrupt Data type: *intnum*

Variable (interrupt identity) of the interrupt.

Program execution

The event connected to this interrupt generates interrupts once again. Interrupts generated during the time the *ISleep* instruction is in effect, however, are ignored.

Example

```
VAR intnum sig1int;
CONNECT sig1int WITH iroutine1;
ISignalDI di1,1,sig1int;
.
ISleep sig1int;
weldpart1;
IWatch sig1int;
```

During execution of the *weldpart1* routine, no interrupts are permitted from the signal *di1*.

Error handling

Interrupts which have not been ordered are not permitted. If the interrupt number is unknown, the system variable ERRNO is set to ERR_UNKINO (see “Date types - errnum”). The error can be handled in the error handler.

Syntax

IWatch
    [Interrupt ':='] < variable (**VAR**) of *intnum* > ';'

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Deactivating an interrupt	Instructions - *ISleep*

label Line name

Label is used to name a line in the program. Using the *GOTO* instruction, this name can then be used to move program execution.

Example

```
GOTO next;
    .
next:
```

Program execution continues with the instruction following *next*.

Arguments

Label:

Label Identifier

The name you wish to give the line.

Program execution

Nothing happens when you execute this instruction.

Limitations

The label must not be the same as

- any other label within the same routine,
- any data name within the same routine.

A label hides global data and routines with the same name within the routine it is located in.

Syntax

(EBNF)
<identifier>':'

Related information

	Described in:
Identifiers	Basic Characteristics- *Basic Elements*
Moving program execution to a label	Instructions - *GOTO*

Load — Load a program module during execution

Load is used to load a program module into the program memory during execution.

The loaded program module will be added to the already existing modules in the program memory.

Example

```
Load ram1disk \File:="PART_A.MOD";
```

Load the program module PART_A.MOD from the *ram1disk* into the program memory. (*ram1disk* is a predefined string constant "*ram1disk*:").

Arguments

Load FilePath [\File]

FilePath

Data type: *string*

The file path and the file name to the file that will be loaded into the program memory. The file name shall be excluded when the argument *\File* is used.

[\File]

Data type: *string*

When the file name is excluded in the argument *FilePath* then it must be defined with this argument.

Program execution

Program execution waits for the program module to finish loading before proceeding with the next instruction.

To obtain a good program structure, that is easy to understand and maintain, all loading and unloading of program modules should be done from the main module which is always present in the program memory during execution.

After the program module is loaded it will be linked and initialised. The initialisation of the loaded module sets all variables at module level to their init values. Unresolved references will be accepted if the system parameter for *Tasks* is set (BindRef = NO). However, when the program is started or the teach pendant function Program/File/Check is used, no check for unresolved references will be done if the parameter BindRef = NO. There will be a run time error on execution of an unresolved reference.

Examples

Load "ram1disk:DOORDIR/DOOR1.MOD";

Load the program module DOOR1.MOD from the *ram1disk* at the directory DOORDIR into the program memory.

Load "ram1disk:DOORDIR/" \File:="DOOR1.MOD";

Same as above but another syntax.

Limitations

Loading a program module that contains a main routine is not allowed.

Avoid ongoing robot movements during the loading.

Avoid using the floppy disk for loading since reading from the floppy drive is very time consuming.

Error handling

If the file in the *Load* instructions cannot be found, then the system variable ERRNO is set to ERR_FILNOTFND. If the module already is loaded into the program memory then the system variable ERRNO is set to ERR_LOADED (see "Data types - errnum"). The errors above can be handled in an error handler.

Syntax

Load
    [FilePath':=']<expression (**IN**) of *string*>
    ['\'File':=' <expression (**IN**) of *string*>]';'

Related information

	Described in:
Unload a program module	Instructions - *UnLoad*
Load a program module in parallel with another program execution	Instructions - *StartLoad-WaitLoad*
Accept unresolved references	System Parameters - *Controller* System Parameters - *Tasks* System Parameters - *BindRef*

MoveCSync Moves the robot circularly and executes a RAPID procedure

MoveCSync (*Move Circular Synchronously*) is used to move the tool centre point (TCP) circularly to a given destination. The specified RAPID procedure is executed at the middle of the corner path in the destination point. During the movement, the orientation normally remains unchanged relative to the circle.

Examples

MoveCSync p1, p2, v500, z30, tool2, “proc1”;

The TCP of the tool, *tool2*, is moved circularly to the position *p2*, with speed data *v500* and zone data *z30*. The circle is defined from the start position, the circle point *p1* and the destination point *p2*. Procedure *proc1* is executed in the middle of the corner path at *p2*.

Arguments

MoveCSync CirPoint ToPoint Speed [\T] Zone Tool [\WObj] ProcName

CirPoint Data type: *robtarget*

The circle point of the robot. The circle point is a position on the circle between the start point and the destination point. To obtain the best accuracy, it should be placed about halfway between the start and destination points. If it is placed too close to the start or destination point, the robot may give a warning. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction). The position of the external axes are not used.

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the TCP, the tool reorientation and external axes.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot and external axes move. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination point.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified.

ProcName *(Procedure Name)* Data type: *string*

Name of the RAPID procedure to be executed at the middle of the corner path in the destination point.

Program execution

See the instruction *MoveC* for more information about circular movements.

The specified RAPID procedure is executed when the TCP reaches the middle of the corner path in the destination point of the *MoveCSync* instruction, as shown in Figure 1:

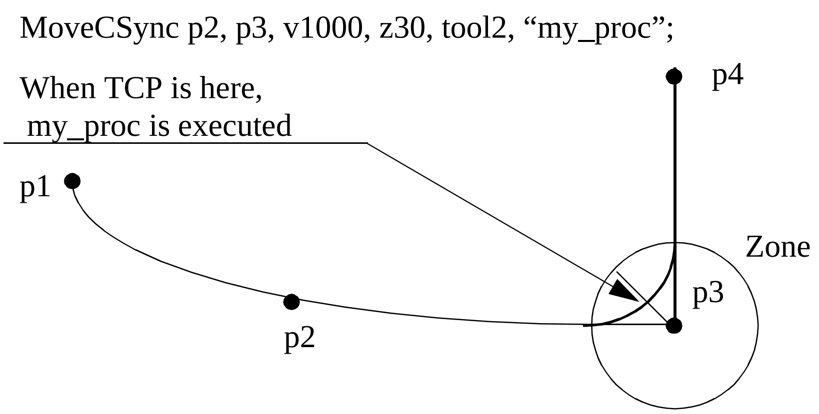


Figure 22 Execution of user-defined RAPID procedure at the middle of the corner path.

For stop points, we recommend the use of “normal” programming sequence with

MoveC + other RAPID instructions in sequence.

Execution of the specified RAPID procedure in different execution modes:

Execution mode:	Execution of RAPID procedure:
Continuously or Cycle	According to this description
Forward step	In the stop point
Backward step	Not at all

Limitation

General limitations according to instruction *MoveC*.

Switching execution mode after program stop from continuously or cycle to stepwise forward or backward results in an error. This error tells the user that the mode switch can result in missed execution of a RAPID procedure in the queue for execution on the path. This error can be avoided if the program is stopped with StopInstr before the mode switch.

Instruction *MoveCSync* cannot be used on TRAP level.
The specified RAPID procedure cannot be tested with stepwise execution.

Syntax

MoveCSync
[CirPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[Speed ’:=’] < expression (**IN**) of *speeddata* >
[’\’ T ’:=’ < expression (**IN**) of *num* >] ’,’
[Zone ’:=’] < expression (**IN**) of *zonedata* > ’,’
[Tool ’:=’] < persistent (**PERS**) of *tooldata* >
[’\’ WObj ’:=’ < persistent (**PERS**) of *wobjdata* >] ’,’
[ProcName ‘:=’] < expression (**IN**) of *string* >] ’;’

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

MoveAbsJMoves the robot to an absolute joint position

MoveAbsJ (*Move Absolute Joint*) is used to move the robot to an absolute position, defined in axes positions.

This instruction need only be used when:

- the end point is a singular point
- for ambiguous positions on the IRB 6400C, e.g. for movements with the tool over the robot.

The final position of the robot, during a movement with *MoveAbsJ,* is neither affected by the given tool and work object, nor by active program displacement. However, the robot uses these data to calculating the load, TCP velocity, and the corner path. The same tools can be used as in adjacent movement instructions.

The robot and external axes move to the destination position along a non-linear path. All axes reach the destination position at the same time.

Examples

MoveAbsJ p50, v1000, z50, tool2;

The robot with the tool *tool2* is moved along a non-linear path to the absolute axis position, *p50*, with velocity data *v1000* and zone data *z50*.

MoveAbsJ *, v1000\T:=5, fine, grip3;

The robot with the tool *grip3*, is moved along a non-linear path to a stop point which is stored as an absolute axis position in the instruction (marked with an *). The entire movement takes 5 s.

Arguments

MoveAbsJ [\Conc] ToJointPos Speed [\V] | [\T] Zone [\Z] Tool [\WObj]

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed while the robot is moving. The argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required.

Using the argument \Conc, the number of movement instructions in succession is limited to 5. In a program section that includes StorePath-RestoPath, movement instructions with the argument \Conc are not permitted.

If this argument is omitted and the ToPoint is not a stop point, the subsequent instruction is executed some time before the robot has reached the programmed zone.

ToJointPos (*To Joint Position*) Data type: *jointtarget*

The destination absolute joint position of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the tool reorientation and external axes.

[\V] *(Velocity)* Data type: *num*

This argument is used to specify the velocity of the TCP in mm/s directly in the instruction. It is then substituted for the corresponding velocity specified in the speed data.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

[\Z] *(Zone)* Data type: *num*

This argument is used to specify the position accuracy of the robot TCP directly in the instruction. The length of the corner path is given in mm, which is substituted for the corresponding zone specified in the zone data.

Tool Data type: *tooldata*

The tool in use during the movement.

The position of the TCP and the load on the tool are defined in the tool data. The TCP position is used to decide the velocity and the corner path for the movement.

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

The work object used during the movement.

This argument can be omitted if the tool is held by the robot. However, if the robot holds the work object, i.e. the tool is stationary, or with coordinated external axes, then the argument must be specified.

In the case of a stationary tool or coordinated external axes, the data used by the system to decide the velocity and the corner path for the movement, is defined in the work object.

Program execution

The tool is moved to the destination absolute joint position with interpolation of the axis angles. This means that each axis is moved with constant axis velocity and that all axes reach the destination joint position at the same time, which results in a non-linear path.

Generally speaking, the TCP is moved at approximate programmed velocity. The tool is reoriented and the external axes are moved at the same time as the TCP moves. If the programmed velocity for reorientation, or for the external axes, cannot be attained, the velocity of the TCP will be reduced.

A corner path is usually generated when movement is transferred to the next section of the path. If a stop point is specified in the zone data, program execution only continues when the robot and external axes have reached the appropriate joint position.

Examples

MoveAbsJ *, v2000\V:=2200, z40 \Z:=45, grip3;

> The tool, *grip3*, is moved along a non-linear path to a absolute joint position stored in the instruction. The movement is carried out with data set to *v2000* and *z40*, the velocity and zone size of the TCP are *2200* mm/s and *45* mm respectively.

MoveAbsJ \Conc, *, v2000, z40, grip3;

> The tool, *grip3*, is moved along a non-linear path to a absolute joint position stored in the instruction. Subsequent logical instructions are executed while the robot moves.

GripLoad obj_mass;
MoveAbsJ start, v2000, z40, grip3 \WObj:= obj;

> The robot moves the work object *obj* in relation to the fixed tool *grip3* along a non-linear path to an absolute axis position *start*.

Error handling

When running the program, a check is made that the arguments Tool and \WObj do not contain contradictory data with regard to a movable or a stationary tool respectively.

Limitations

A movement with *MoveAbsJ* is not affected by active program displacement, but is affected by active offset for external axes.

In order to be able to run backwards with the instruction *MoveAbsJ* involved, and avoiding problems with singular points or ambiguous areas, it is essential that the subsequent instructions fulfil certain requirements, as follows (see Figure 1).

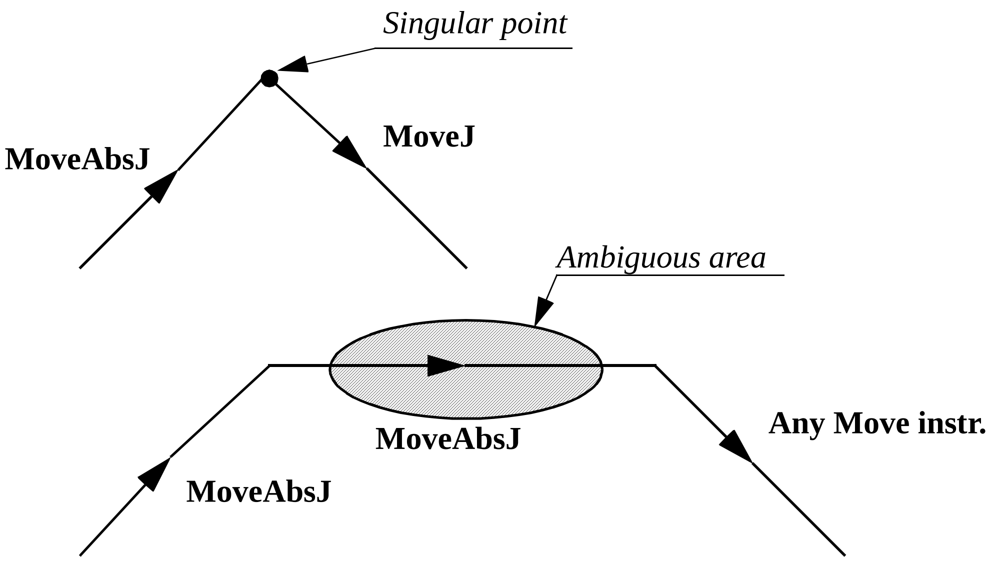


Figure 1 Limitation for backward execution with MoveAbsJ.

Syntax

MoveAbsJ
['\' Conc ',']
[ToJointPos ':='] < expression (**IN**) of *jointtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' V ':=' < expression (**IN**) of *num* >]
| ['\' T ':=' < expression (**IN**) of *num* >] ','
[Zone ':='] < expression (**IN**) of *zonedata* >
['\' Z ':=' < expression (**IN**) of *num* >] ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ';'

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of jointtarget	Data Types - *jointtarget*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Concurrent program execution	Motion and I/O Principles - *Synchronisation Using Logical Instructions*

MoveC Moves the robot circularly

MoveC is used to move the tool centre point (TCP) circularly to a given destination. During the movement, the orientation normally remains unchanged relative to the circle.

Examples

MoveC p1, p2, v500, z30, tool2;

The TCP of the tool, *tool*2, is moved circularly to the position *p2,* with speed data *v500* and zone data *z30*. The circle is defined from the start position, the circle point *p1* and the destination point *p2*.

MoveC *, *, v500 \T:=5, fine, grip3;

The TCP of the tool, *grip3*, is moved circularly to a fine point stored in the instruction (marked by the second *). The circle point is also stored in the instruction (marked by the first *). The complete movement takes *5* seconds.

MoveL p1, v500, fine, tool1;
MoveC p2, p3, v500, z20, tool1;
MoveC p4, p1, v500, fine, tool1;

A complete circle is performed if the positions are the same as those shown in Figure 2.

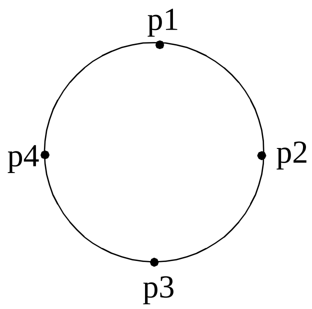


Figure 2 A complete circle is performed by two MoveC instructions.

Arguments

MoveC [\Conc] CirPoint ToPoint Speed [\V] | [\T] Zone [\Z] Tool [\WObj] [\Corr]

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed at once. This argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required.

Using the argument \Conc, the number of movement instructions in succession is limited to 5. In a program section that includes StorePath-RestoPath, movement instructions with the argument \Conc are not permitted.

If this argument is omitted, and the ToPoint is not a Stop point the subsequent instruction is executed some time before the robot has reached the programmed zone.

CirPoint Data type: *robtarget*

The circle point of the robot. The circle point is a position on the circle between the start point and the destination point. To obtain the best accuracy, it should be placed about halfway between the start and destination points. If it is placed too close to the start or destination point, the robot may give a warning. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction). The position of the external axes are not used.

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the TCP, the tool reorientation and external axes.

[\V] *(Velocity)* Data type: *num*

This argument is used to specify the velocity of the TCP in mm/s directly in the instruction. It is then substituted for the corresponding velocity specified in the speed data.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot and external axes move. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

[\Z] *(Zone)* Data type: *num*

This argument is used to specify the position accuracy of the robot TCP directly in the instruction. The length of the corner path is given in mm, which is substituted for the corresponding zone specified in the zone data.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination point.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified in order for a circle relative to the work object to be executed.

[\Corr] *(Correction)* Data type: *switch*

Correction data written to a corrections entry by the instruction *CorrWrite* will be added to the path and destination position, if this argument is present.

Program execution

The robot and external units are moved to the destination point as follows:

- The TCP of the tool is moved circularly at constant programmed velocity.
- The tool is reoriented at a constant velocity, from the orientation at the start position to the orientation at the destination point.
- The reorientation is performed relative to the circular path. Thus, if the orientation relative to the path is the same at the start and the destination points, the relative orientation remains unchanged during the movement (see Figure 3).

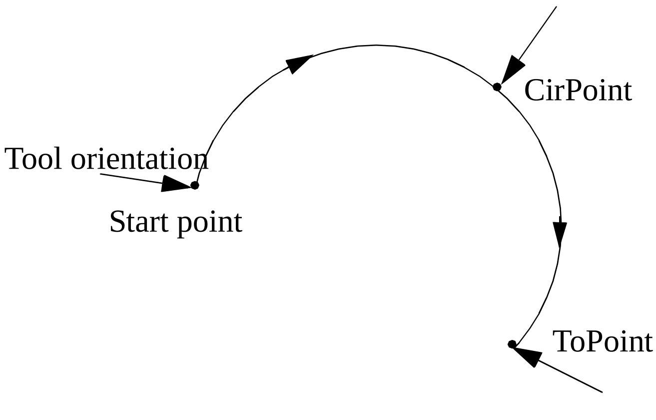


Figure 3 Tool orientation during circular movement.

- The orientation at the circle point is not critical; it is only used to distinguish between two possible directions of reorientation. The accuracy of the reorientation along the path depends only on the orientation at the start and destination points.
- Uncoordinated external axes are executed at constant velocity in order for them to arrive at the destination point at the same time as the robot axes. The position in the circle position is not used.

If it is not possible to attain the programmed velocity for the reorientation or for the external axes, the velocity of the TCP will be reduced.

A corner path is usually generated when movement is transferred to the next section of a path. If a stop point is specified in the zone data, program execution only continues when the robot and external axes have reached the appropriate position.

Examples

MoveC *, *, v500 \V:=550, z40 \Z:=45, grip3;

The TCP of the tool, *grip3*, is moved circularly to a position stored in the instruction. The movement is carried out with data set to *v500* and *z40*; the velocity and zone size of the TCP are *550* mm/s and *45* mm respectively.

MoveC \Conc, *, *, v500, z40, grip3;

The TCP of the tool, *grip3*, is moved circularly to a position stored in the instruction. The circle point is also stored in the instruction. Subsequent logical instructions are executed while the robot moves.

MoveC cir1, p15, v500, z40, grip3 \WObj:=fixture;

The TCP of the tool, *grip3*, is moved circularly to a position, *p15*, via the circle point *cir1*. These positions are specified in the object coordinate system for *fixture*.

Limitations

A change of execution mode from forward to backward or vice versa, while the robot is stopped on a circular path, is not permitted and will result in an error message.

The instruction *MoveC* (or any other instruction including circular movement) should never be started from the beginning, with TCP between the circle point and the end point. Otherwise the robot will not take the programmed path (positioning around the circular path in another direction compared with that programmed).

Make sure that the robot can reach the circle point during program execution and divide the circle segment if necessary.

Syntax

MoveC
[’\’ Conc ’,’]
[CirPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[Speed ’:=’] < expression (**IN**) of *speeddata* >
[’\’ V ’:=’ < expression (**IN**) of *num* >]
| [’\’ T ’:=’ < expression (**IN**) of *num* >] ’,’
[Zone ’:=’] < expression (**IN**) of *zonedata* >
[’\’ Z ’:=’ < expression (**IN**) of *num* >] ’,’
[Tool ’:=’] < persistent (**PERS**) of *tooldata* >
[’\’ WObj ’:=’ < persistent (**PERS**) of *wobjdata* >]
[’\’ Corr]’;’

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Writes to a corrections entry	Instructions - *CorrWrite*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Concurrent program execution	Motion and I/O Principles - *Synchronisation Using Logical Instructions*

MoveCDO Moves the robot circularly and sets digital output in the corner

MoveCDO (*Move Circular Digital Output*) is used to move the tool centre point (TCP) circularly to a given destination. The specified digital output is set/reset in the middle of the corner path at the destination point. During the movement, the orientation normally remains unchanged relative to the circle.

Examples

MoveCDO p1, p2, v500, z30, tool2, do1,1;

The TCP of the tool, *tool2*, is moved circularly to the position *p2*, with speed data *v500* and zone data *z30*. The circle is defined from the start position, the circle point *p1* and the destination point *p2*. Output *do1* is set in the middle of the corner path at *p2*.

Arguments

MoveCDO CirPoint ToPoint Speed [\T] Zone Tool [\WObj] Signal Value

CirPoint Data type: *robtarget*

The circle point of the robot. The circle point is a position on the circle between the start point and the destination point. To obtain the best accuracy, it should be placed about halfway between the start and destination points. If it is placed too close to the start or destination point, the robot may give a warning. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction). The position of the external axes are not used.

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the TCP, the tool reorientation and external axes.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot and external axes move. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination point.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified in order for a circle relative to the work object to be executed.

Signal Data type: *signaldo*

The name of the digital output signal to be changed.

Value Data type: *dionum*

The desired value of signal (0 or 1).

Program execution

See the instruction *MoveC* for more information about circular movement.

The digital output signal is set/reset in the middle of the corner path for flying points, as shown in Figure 1.

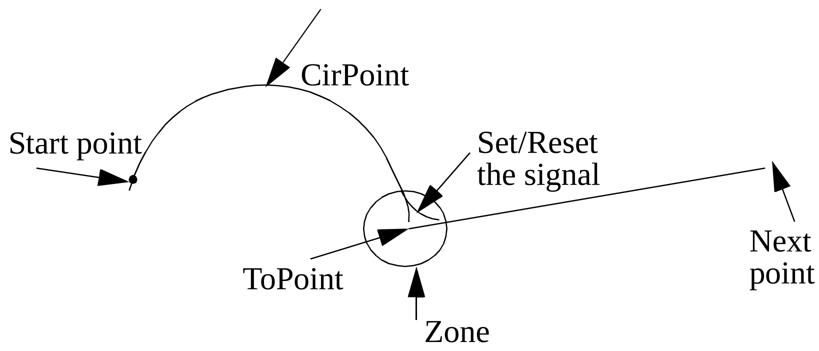


Figure 4 Set/Reset of digital output signal in the corner path with MoveCDO.

For stop points, we recommend the use of “normal” programming sequence with *MoveC* + *SetDO*. But when using stop point in instruction *MoveCDO*, the digital output signal is set/reset when the robot reaches the stop point.

The specified I/O signal is set/reset in execution mode continuously and stepwise forward but not in stepwise backward.

Limitations

General limitations according to instruction *MoveC.*

Syntax

MoveCDO
[CirPoint ':='] < expression (**IN**) of *robtarget* > ','
[ToPoint ':='] < expression (**IN**) of *robtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' T ':=' < expression (**IN**) of *num* >] ','
[Zone ':='] < expression (**IN**) of *zonedata* > ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ','
[Signal ':='] < variable (**VAR**) of *signaldo*>] ','
[Value ':='] < expression (**IN**) of *dionum* >] ';'

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Movements with I/O settings	Motion and I/O Principles - *Synchronisation Using Logical Instructions*

MoveJDO Moves the robot by joint movement and sets digital output in the corner

MoveJDO (*Move Joint Digital Output*) is used to move the robot quickly from one point to another when that movement does not have to be in a straight line. The specified digital output signal is set/reset at the middle of the corner path.

The robot and external axes move to the destination position along a non-linear path. All axes reach the destination position at the same time.

Examples

MoveJDO p1, vmax, z30, tool2, do1, 1;

The tool centre point (TCP) of the tool, *tool2*, is moved along a non-linear path to the position, *p1*, with speed data *vmax* and zone data *z30*. Output *do1* is set in the middle of the corner path at *p1*.

Arguments

MoveJDO ToPoint Speed [\T] Zone Tool [\WObj] Signal Value

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the tool reorientation and external axes.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination point.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified.

Signal Data type: *signaldo*

The name of the digital output signal to be changed.

Value Data type: *dionum*

The desired value of signal (0 or 1).

Program execution

See the instruction *MoveJ* for more information about joint movement.

The digital output signal is set/reset in the middle of the corner path for flying points, as shown in Figure 1.

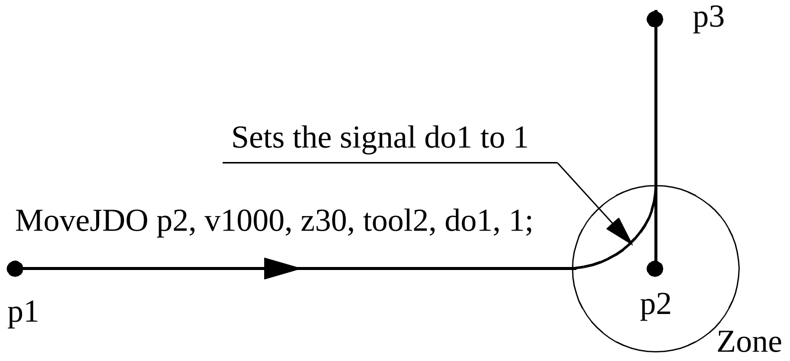


Figure 5 Set/Reset of digital output signal in the corner path with MoveJDO.

For stop points, we recommend the use of “normal” programming sequence with *MoveJ* + *SetDO*. But when using stop point in instruction *MoveJDO*, the digital output signal is set/reset when the robot reaches the stop point.

The specified I/O signal is set/reset in execution mode continuously and stepwise forward but not in stepwise backward.

Syntax

MoveJDO
[ToPoint ':='] < expression (**IN**) of *robtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' T ':=' < expression (**IN**) of *num* >] ','
[Zone ':='] < expression (**IN**) of *zonedata* > ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ','
[Signal ':='] < variable (**VAR**) of *signaldo*>] ','
[Value ':='] < expression (**IN**) of *dionum* >] ';'

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Movements with I/O settings	Motion and I/O Principles - *Synchronisation Using Logical Instructions*

MoveJ Moves the robot by joint movement

MoveJ is used to move the robot quickly from one point to another when that movement does not have to be in a straight line.

The robot and external axes move to the destination position along a non-linear path. All axes reach the destination position at the same time.

Examples

MoveJ p1, vmax, z30, tool2;

The tool centre point (TCP) of the tool, *tool*2, is moved along a non-linear path to the position, *p1*, with speed data *vmax* and zone data *z30*.

MoveJ *, vmax \T:=5, fine, grip3;

The TCP of the tool, *grip3*, is moved along a non-linear path to a stop point stored in the instruction (marked with an *). The entire movement takes *5* seconds.

Arguments

MoveJ [\Conc] ToPoint Speed [\V] | [\T] Zone [\Z] Tool [\WObj]

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed while the robot is moving. The argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required.

Using the argument \Conc, the number of movement instructions in succession is limited to 5. In a program section that includes StorePath-RestoPath, movement instructions with the argument \Conc are not permitted.

If this argument is omitted and the ToPoint is not a stop point, the subsequent instruction is executed some time before the robot has reached the programmed zone.

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the tool reorientation and external axes.

[\V] *(Velocity)* Data type: *num*

This argument is used to specify the velocity of the TCP in mm/s directly in the instruction. It is then substituted for the corresponding velocity specified in the speed data.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

[\Z] *(Zone)* Data type: *num*

This argument is used to specify the position accuracy of the robot TCP directly in the instruction. The length of the corner path is given in mm, which is substituted for the corresponding zone specified in the zone data.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination point.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified.

Program execution

The tool centre point is moved to the destination point with interpolation of the axis angles. This means that each axis is moved with constant axis velocity and that all axes reach the destination point at the same time, which results in a non-linear path.

Generally speaking, the TCP is moved at the approximate programmed velocity (regardless of whether or not the external axes are coordinated). The tool is reoriented and the external axes are moved at the same time as the TCP moves. If the programmed velocity for reorientation, or for the external axes, cannot be attained, the velocity of the TCP will be reduced.

A corner path is usually generated when movement is transferred to the next section of the path. If a stop point is specified in the zone data, program execution only continues when the robot and external axes have reached the appropriate position.

Examples

MoveJ *, v2000\V:=2200, z40 \Z:=45, grip3;

The TCP of the tool, *grip3*, is moved along a non-linear path to a position stored in the instruction. The movement is carried out with data set to *v2000* and *z40*; the velocity and zone size of the TCP are *2200* mm/s and *45* mm respectively.

MoveJ \Conc, *, v2000, z40, grip3;

The TCP of the tool, *grip3*, is moved along a non-linear path to a position stored in the instruction. Subsequent logical instructions are executed while the robot moves.

MoveJ start, v2000, z40, grip3 \WObj:=fixture;

The TCP of the tool, *grip3*, is moved along a non-linear path to a position, *start*. This position is specified in the object coordinate system for *fixture*.

Syntax

MoveJ
['\' Conc ',']
[ToPoint ':='] < expression (**IN**) of *robtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' V ':=' < expression (**IN**) of *num* >]
| ['\' T ':=' < expression (**IN**) of *num* >] ','
[Zone ':='] < expression (**IN**) of *zonedata* >
['\' Z ':=' < expression (**IN**) of *num* >] ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ';'

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Concurrent program execution	Motion and I/O Principles - *Synchronisation Using Logical Instructions*

MoveLDO Moves the robot linearly and sets digital output in the corner

MoveLDO (*Move Linearly Digital Output*) is used to move the tool centre point (TCP) linearly to a given destination. The specified digital output signal is set/reset at the middle of the corner path.

When the TCP is to remain stationary, this instruction can also be used to reorient the tool.

Example

MoveLDO p1, v1000, z30, tool2, do1,1;

The TCP of the tool, *tool2*, is moved linearly to the position *p1*, with speed data *v1000* and zone data *z30*. Output *do1* is set in the middle of the corner path at *p1*.

Arguments

MoveLDO ToPoint Speed [\T] Zone Tool
[\WObj] Signal Value

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity for the tool centre point, the tool reorientation and external axes.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination position.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified.

Signal Data type: *signaldo*

The name of the digital output signal to be changed.

Value Data type: *dionum*

The desired value of signal (0 or 1).

Program execution

See the instruction *MoveL* for more information about linear movements.

The digital output signal is set/reset in the middle of the corner path for flying points, as shown in Figure 1.

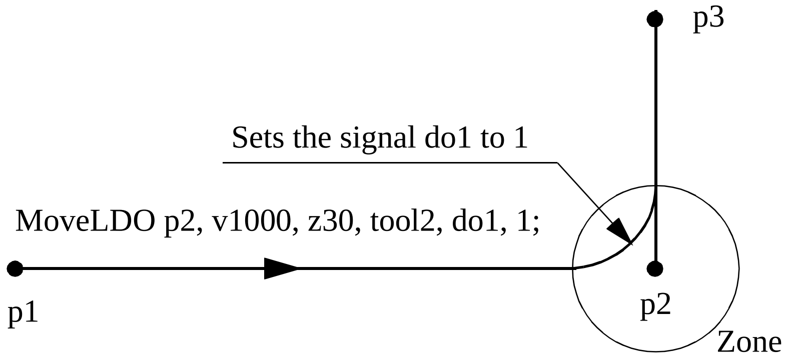


Figure 6 Set/Reset of digital output signal in the corner path with MoveLDO.

For stop points, we recommend the use of "normal" programming sequence with *MoveL* + *SetDO*. But when using stop point in instruction *MoveLDO*, the digital output signal is set/reset when the robot reaches the stop point.

The specified I/O signal is set/reset in execution mode continuously and stepwise forward but not in stepwise backward.

Syntax

MoveLDO
[ToPoint ':='] < expression (**IN**) of *robtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' T ':=' < expression (**IN**) of *num* >] ','
[Zone ':='] < expression (**IN**) of *zonedata* > ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ','
[Signal ':='] < variable (**VAR**) of *signaldo*>] ','
[Value ':='] < expression (**IN**) of *dionum* >] ';'

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Movements with I/O settings	Motion and I/O Principles - *Synchronisation Using Logical Instructions*

MoveL Moves the robot linearly

MoveL is used to move the tool centre point (TCP) linearly to a given destination. When the TCP is to remain stationary, this instruction can also be used to reorientate the tool.

Example

MoveL p1, v1000, z30, tool2;

The TCP of the tool, *tool*2, is moved linearly to the position *p1*, with speed data *v1000* and zone data *z30*.

MoveL *, v1000\T:=5, fine, grip3;

The TCP of the tool, *grip3*, is moved linearly to a fine point stored in the instruction (marked with an *). The complete movement takes *5* seconds.

Arguments

MoveL [\Conc] ToPoint Speed [\V] | [\T] Zone [\Z] Tool [\WObj] [\Corr]

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed at once. This argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required.

Using the argument \Conc, the number of movement instructions in succession is limited to 5. In a program section that includes StorePath-RestoPath, movement instructions with the argument \Conc are not permitted.

If this argument is omitted and the ToPoint is not a stop point, the subsequent instruction is executed some time before the robot has reached the programmed zone.

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity for the tool centre point, the tool reorientation and external axes.

[\V] _(Velocity)_ Data type: _num_

This argument is used to specify the velocity of the TCP in mm/s directly in the instruction. It is then substituted for the corresponding velocity specified in the speed data.

[\T] _(Time)_ Data type: _num_

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Zone Data type: _zonedata_

Zone data for the movement. Zone data describes the size of the generated corner path.

[\Z] _(Zone)_ Data type: _num_

This argument is used to specify the position accuracy of the robot TCP directly in the instruction. The length of the corner path is given in mm, which is substituted for the corresponding zone specified in the zone data.

Tool Data type: _tooldata_

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination position.

[\WObj] _(Work Object)_ Data type: _wobjdata_

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary tool or coordinated external axes are used, this argument must be specified in order to perform a linear movement relative to the work object.

[\Corr] _(Correction)_ Data type: _switch_

Correction data written to a corrections entry by the instruction _CorrWrite_ will be added to the path and destination position, if this argument is present.

Program execution

The robot and external units are moved to the destination position as follows:

- The TCP of the tool is moved linearly at constant programmed velocity.
- The tool is reoriented at equal intervals along the path.
- Uncoordinated external axes are executed at a constant velocity in order for them to arrive at the destination point at the same time as the robot axes.

If it is not possible to attain the programmed velocity for the reorientation or for the external axes, the velocity of the TCP will be reduced.

A corner path is usually generated when movement is transferred to the next section of a path. If a stop point is specified in the zone data, program execution only continues when the robot and external axes have reached the appropriate position.

Examples

MoveL *, v2000 \V:=2200, z40 \Z:=45, grip3;

> The TCP of the tool, *grip3*, is moved linearly to a position stored in the instruction. The movement is carried out with data set to *v2000* and *z40*; the velocity and zone size of the TCP are *2200* mm/s and *45* mm respectively.

MoveL \Conc, *, v2000, z40, grip3;

> The TCP of the tool, *grip3*, is moved linearly to a position stored in the instruction. Subsequent logical instructions are executed while the robot moves.

MoveL start, v2000, z40, grip3 \WObj:=fixture;

> The TCP of the tool, *grip3*, is moved linearly to a position, *start*. This position is specified in the object coordinate system for *fixture*.

Syntax

```
MoveL
   [ ’\’ Conc ’,’ ]
   [ ToPoint ’:=’ ] < expression (IN) of robtarget > ’,’
   [ Speed ’:=’ ] < expression (IN) of speeddata >
               [ ’\’ V ’:=’ < expression (IN) of num > ]
               | [ ’\’ T ’:=’ < expression (IN) of num > ] ’,’
   [Zone ’:=’ ] < expression (IN) of zonedata >
               [ ’\’ Z ’:=’ < expression (IN) of num > ] ’,’
   [ Tool ’:=’ ] < persistent (PERS) of tooldata >
   [ ’\’ WObj ’:=’ < persistent (PERS) of wobjdata > ]
   [ ’\’ Corr ]’;’
```

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Writes to a corrections entry	Instructions - *CorrWrite*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Concurrent program execution	Motion and I/O Principles - *Synchronisation Using Logical Instructions*

MoveJSync Moves the robot by joint movement and executes a RAPID procedure

MoveJSync (*Move Joint Synchronously)* is used to move the robot quickly from one point to another when that movement does not have to be in a straight line. The specified RAPID procedure is executed at the middle of the corner path in the destination point.

The robot and external axes move to the destination position along a non-linear path. All axes reach the destination position at the same time.

Examples

MoveJSync p1, vmax, z30, tool2, “proc1”;

The tool centre point (TCP) of the tool, *tool2*, is moved along a non-linear path to the position, *p1*, with speed data *vmax* and zone data *z30*. Procedure *proc1* is executed in the middle of the corner path at *p1*.

Arguments

MoveJSync ToPoint Speed [\T] Zone Tool [\WObj] ProcName

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the tool reorientation and external axes.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination point.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified.

ProcName *(Procedure Name)* Data type: *string*

Name of the RAPID procedure to be executed at the middle of the corner path in the destination point.

Program execution

See the instruction *MoveJ* for more information about joint movements.

The specified RAPID procedure is executed when the TCP reaches the middle of the corner path in the destination point of the *MoveJSync* instruction, as shown in Figure 1:

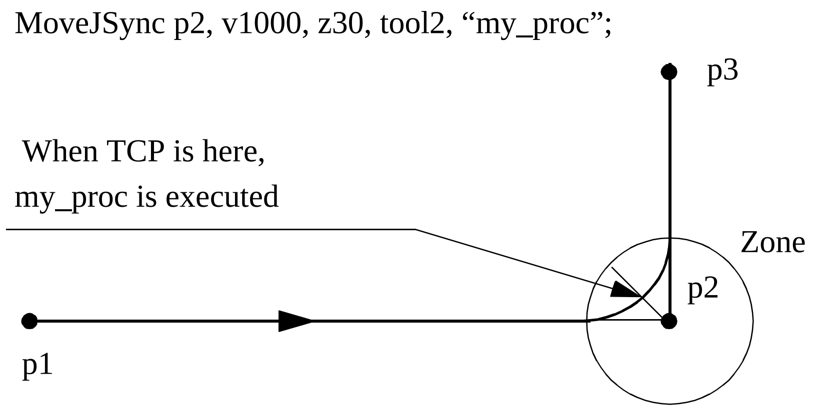


Figure 7 Execution of user-defined RAPID procedure in the middle of the corner path.

For stop points, we recommend the use of “normal” programming sequence with *MoveJ* + other RAPID instructions in sequence.

Execution of the specified RAPID procedure in different execution modes:

Execution mode:	Execution of RAPID procedure:
Continuously or Cycle	According to this description
Forward step	In the stop point
Backward step	Not at all

Limitation

Switching execution mode after program stop from continuously or cycle to stepwise forward or backward results in an error. This error tells the user that the mode switch can result in missed execution of a RAPID procedure in the queue for execution on the path. This error can be avoided if the program is stopped with StopInstr before the mode switch.

Instruction *MoveJSync* cannot be used on TRAP level.
The specified RAPID procedure cannot be tested with stepwise execution.

Syntax

MoveJSync
 [ToPoint ':='] < expression (**IN**) of *robtarget* > ','
 [Speed ':='] < expression (**IN**) of *speeddata* >
   ['\' T ':=' < expression (**IN**) of *num* >] ','
 [Zone ':='] < expression (**IN**) of *zonedata* >
   ['\' Z ':=' < expression (**IN**) of *num* >] ','
 [Tool ':='] < persistent (**PERS**) of *tooldata* >
 ['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ','
 [ProcName':='] < expression (**IN**) of *string* >] ';'

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

MoveL Sync Moves the robot linearly and executes a RAPID procedure

MoveLSync (*Move Linearly Synchronously*) is used to move the tool centre point (TCP) linearly to a given destination.The specified RAPID procedure is executed at the middle of the corner path in the destination point.

When the TCP is to remain stationary, this instruction can also be used to reorient the tool.

Example

MoveLSync p1, v1000, z30, tool2, “proc1”;

The TCP of the tool, *tool2*, is moved linearly to the position *p1*, with speed data *v1000* and zone data *z30*. Procedure *proc1* is executed in the middle of the corner path at *p1*.

Arguments

MoveLSync ToPoint Speed [\T] Zone Tool [\WObj] ProcName

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity for the tool centre point, the tool reorientation and external axes.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination position.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified.

ProcName *(Procedure Name)* Data type: *string*

Name of the RAPID procedure to be executed at the middle of the corner path in the destination point.

Program execution

See the instruction *MoveL* for more information about linear movements.

The specified RAPID procedure is executed when the TCP reaches the middle of the corner path in the destination point of the *MoveLSync* instruction, as shown in Figure 1:

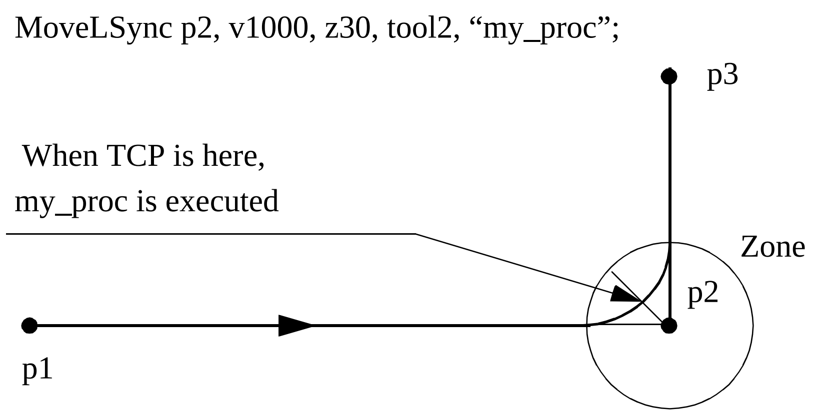


Figure 8 Execution of user-defined RAPID procedure in the middle of the corner path.

For stop points, we recommend the use of “normal” programming sequence with *MoveL* + other RAPID instructions in sequence.

Execution of the specified RAPID procedure in different execution modes:

Execution mode:	Execution of RAPID procedure:
Continuously or Cycle	According to this description
Forward step	In the stop point
Backward step	Not at all

Limitation

Switching execution mode after program stop from continuously or cycle to stepwise forward or backward results in an error. This error tells the user that the mode switch can result in missed execution of a RAPID procedure in the queue for execution on the path. This error can be avoided if the program is stopped with StopInstr before the mode switch.

Instruction *MoveLSync* cannot be used on TRAP level.
The specified RAPID procedure cannot be tested with stepwise execution.

Syntax

MoveLSync
[ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[Speed ’:=’] < expression (**IN**) of *speeddata* >
[’\’ T ’:=’ < expression (**IN**) of *num* >] ’,’
[Zone ’:=’] < expression (**IN**) of *zonedata* > ’,’
[Tool ’:=’] < persistent (**PERS**) of *tooldata* >
[’\’ WObj ’:=’ < persistent (**PERS**) of *wobjdata* >] ’,’
[ProcName‘:=’] < expression (**IN**) of *string* >] ‘;’

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

Open Opens a file or serial channel

Open is used to open a file or serial channel for reading or writing.

Example

```
VAR iodev logfile;
.
Open "flp1:LOGDIR/" \File:= "LOGFILE1.DOC",logfile;
```

The file *LOGFILE1.DOC* in unit *flp1:* (diskette), directory *LOGDIR*, is opened for writing. The reference name *logfile* is used later in the program when writing to the file.

Arguments

Open Object [\File] IODevice [\Read] | [\Write] | [\Append] | [\Bin]

Object Data type: *string*

The I/O object that is to be opened, e.g. "flp1:", "ram1disk:".

[\File] Data type: *string*

The name of the file. This name can also be specified in the argument *Object*, e.g. "flp1:LOGDIR/LOGFILE.DOC".

IODevice Data type: *iodev*

A reference to the file or serial channel to open. This reference is then used for reading from and writing to the file/channel.

The arguments *\Read*, *\Write*, *\Append* and *\Bin* are mutually exclusive. If none of these are specified, the instruction acts in the same way as the *\Write* argument.

[\Read] Data type: *switch*

Opens a character-based file or serial channel for reading. When reading from a file, the reading is started from the beginning of the file.

[\Write] Data type: *switch*

Opens a character-based file or serial channel for writing. If the selected file already exists, its contents are deleted. Anything subsequently written is written at the start of the file.

[\Append] Data type: *switch*

Opens a character-based file or serial channel for writing. If the selected file already exists, anything subsequently written is written at the end of the file.

[\Bin] Data type: *switch*

Opens a binary serial channel for reading and writing.
Works as append, i.e. file pointer at end of file.

Example

```
VAR iodev printer;
.
Open "sio1:", printer \Bin;
Write printer, "This is a message to the printer";
Close printer;
```

The serial channel *sio1:* is opened for binary reading and writing. The reference name *printer* is used later when writing to and closing the serial channel.

Program execution

The specified serial channel/file is activated so that it can be read from or written to. Several files can be open on the same unit at the same time.

Error handling

If a file cannot be opened, the system variable ERRNO is set to ERR_FILEOPEN. This error can then be handled in the error handler.

Syntax

Open
[Object ':='] <expression (**IN**) of *string*>
['\'File':=' <expression (**IN**) of *string*>']' ','
[IODevice ':='] <variable (**VAR**) of *iodev*>
['\'Read] | ['\'Write] | ['\'Append] | ['\'Bin] ';'

Related information

	Described in:
Writing to and reading from serial channels and files.	RAPID Summary - *Communication*

PathResol Override path resolution

PathResol (Path Resolution) is used to override the configured geometric path sample time defined in the system parameters for the manipulator.

Description

The path resolution affects the accuracy of the interpolated path and the program cycle time. The path accuracy is improved and the cycle time is often reduced when the parameter *PathSampleTime* is decreased. A value for parameter *PathSampleTime* which is too low, may however cause CPU load problems in some demanding applications. However, use of the standard configured path resolution (*PathSampleTime* 100%) will avoid CPU load problems and provide sufficient path accuracy in most situations.

Example of *PathResol* usage:

Dynamically critical movements (max payload, high speed, combined joint motions close to the border of the work area) may cause CPU load problems. Increase the parameter *PathSampleTime*.

Low performance external axes may cause CPU load problems during coordination. Increase the parameter *PathSampleTime*.

Arc-welding with high frequency weaving may require high resolution of the interpolated path. Decrease the parameter *PathSampleTime*.

Small circles or combined small movements with direction changes can decrease the path performance quality and increase the cycle time. Decrease the parameter *PathSampleTime*.

Gluing with large reorientations and small corner zones can cause speed variations. Decrease the parameter *PathSampleTime*.

Example

```
MoveJ p1,v1000,fine,tool1;
PathResol 150;
```

With the robot at a stop point, the path sample time is increased to *150%* of the configured.

Arguments

PathResol PathSampleTime

PathSampleTime Data type: *num*

Override as a percent of the configured path sample time.

100% corresponds to the configured path sample time.
Within the range 25-400%.

A lower value of the parameter *PathSampleTime* improves the path resolution (path accuracy).

Program execution

The path resolutions of all subsequent positioning instructions are affected until a new *PathResol* instruction is executed. This will affect the path resolution during all program execution of movements (default path level and path level after *StorePath*) and also during jogging.

The default value for override of path sample time is 100%. This value is automatically set

- at a cold start-up
- when a new program is loaded
- when starting program execution from the beginning.

The current override of path sample time can be read from the variable *C_MOTSET* (data type *motsetdata*) in the component *pathresol*.

Limitations

The robot must be standing still at a stop point before overriding the path sample time. When there is a corner path in the program, the system will instead create a stop point (warning 50146) and it is not possible to restart in this instruction following a power failure.

Syntax

PathResol
    [PathSampleTime ':='] < expression (**IN**) of *num*> ';'

Related information

	Described in:
Positioning instructions	Motion and I/O Principles- *Movements*
Motion settings	RAPID Summary - *Motion Settings*
Configuration of path resolution	System Parameters - *CPU Optimization*

PDispOff Deactivates program displacement

PDispOff (Program Displacement Off) is used to deactivate a program displacement.

Program displacement is activated by the instruction *PDispSet* or *PDispOn* and applies to all movements until some other program displacement is activated or until program displacement is deactivated.

Examples

```
PDispOff;
```

Deactivation of a program displacement.

```
MoveL p10, v500, z10, tool1;
PDispOn \ExeP:=p10, p11, tool1;
MoveL p20, v500, z10, tool1;
MoveL p30, v500, z10, tool1;
PDispOff;
MoveL p40, v500, z10, tool1;
```

A program displacement is defined as the difference between the positions *p10* and *p11*. This displacement affects the movement to *p20* and *p30*, but not to *p40*.

Program execution

Active program displacement is reset. This means that the program displacement coordinate system is the same as the object coordinate system, and thus all programmed positions will be related to the latter.

Syntax

PDispOff ';'

Related information

	Described in:
Definition of program displacement using two positions	Instructions - *PDispOn*
Definition of program displacement using values	Instructions - *PDispSet*

PDispOn Activates program displacement

PDispOn (Program Displacement On) is used to define and activate a program displacement using two robot positions.

Program displacement is used, for example, after a search has been carried out, or when similar motion patterns are repeated at several different places in the program.

Examples

```
MoveL p10, v500, z10, tool1;
PDispOn \ExeP:=p10, p20, tool1;
```

Activation of a program displacement (parallel movement). This is calculated based on the difference between positions *p10* and *p20*.

```
MoveL p10, v500, fine, tool1;
PDispOn *, tool1;
```

Activation of a program displacement (parallel movement). Since a stop point has been used in the previous instruction, the argument \ExeP does not have to be used. The displacement is calculated on the basis of the difference between the robot's actual position and the programmed point (*) stored in the instruction.

```
PDispOn \Rot \ExeP:=p10, p20, tool1;
```

Activation of a program displacement including a rotation. This is calculated based on the difference between positions *p10* and *p20*.

Arguments

PDispOn [\Rot] [\ExeP] ProgPoint Tool [\WObj]

[\Rot] *(Rotation)* Data type: *switch*

The difference in the tool orientation is taken into consideration and this involves a rotation of the program.

[\ExeP] *(Executed Point)* Data type: *robtarget*

The robot's new position at the time of the program execution.
If this argument is omitted, the robot's current position at the time of the program execution is used.

ProgPoint *(Programmed Point)* Data type: *robtarget*

The robot's original position at the time of programming.

Tool Data type: *tooldata*

The tool used during programming, i.e. the TCP to which the *ProgPoint* position is related.

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

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

This argument can be omitted and, if it is, the position is related to the world coordinate system. However, if a stationary TCP or coordinated external axes are used, this argument must be specified.

The arguments *Tool* and *\WObj* are used both to calculate the *ProgPoint* during programming and to calculate the current position during program execution if no ExeP argument is programmed.

Program execution

Program displacement means that the ProgDisp coordinate system is translated in relation to the object coordinate system. Since all positions are related to the ProgDisp coordinate system, all programmed positions will also be displaced. See Figure 9.

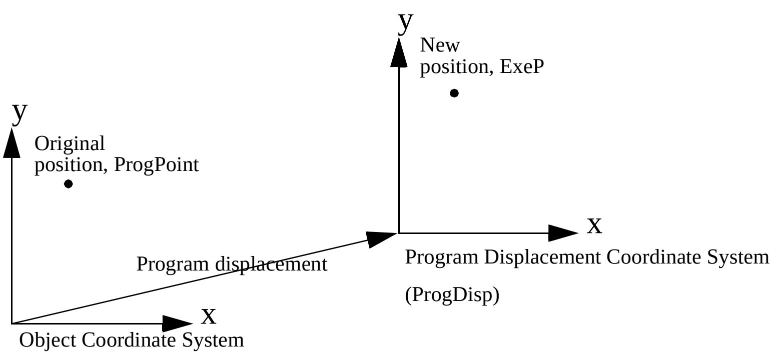


Figure 9 Displacement of a programmed position using program displacement.

Program displacement is activated when the instruction *PDispOn* is executed and remains active until some other program displacement is activated (the instruction *PDispSet* or *PDispOn*) or until program displacement is deactivated (the instruction *PDispOff*).

Only <u>one</u> program displacement can be active at any one time. Several *PDispOn* instructions, on the other hand, can be programmed one after the other and, in this case, the different program displacements will be added.

Program displacement is calculated as the difference between *ExeP* and *ProgPoint*. If *ExeP* has not been specified, the current position of the robot at the time of the program execution is used instead. Since it is the actual position of the robot that is used, the robot should not move when *PDispOn* is executed.

If the argument *\Rot* is used, the rotation is also calculated based on the tool orientation at the two positions. The displacement will be calculated in such a way that the new position (*ExeP)* will have the same position and orientation in relation to the displaced coordinate system, ProgDisp, as the old position (*ProgPoint*) had in relation to the original coordinate system (see Figure 10).

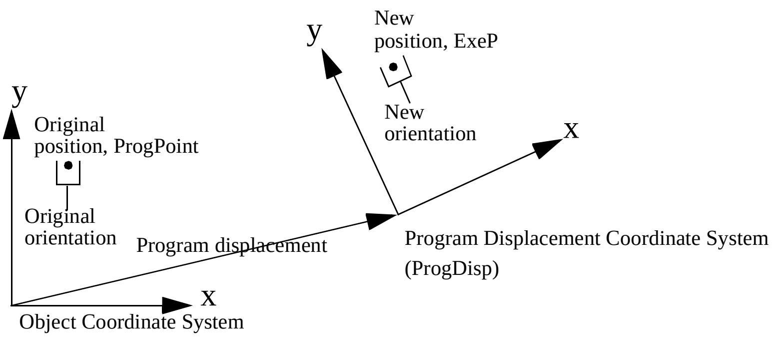


Figure 10 Translation and rotation of a programmed position.

The program displacement is automatically reset

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Example

```
PROC draw_square()
    PDispOn *, tool1;
    MoveL *, v500, z10, tool1;
    MoveL *, v500, z10, tool1;
    MoveL *, v500, z10, tool1;
    MoveL *, v500, z10, tool1;
    PDispOff;
ENDPROC
.
MoveL p10, v500, fine, tool1;
draw_square;
MoveL p20, v500, fine, tool1;
draw_square;
MoveL p30, v500, fine, tool1;
draw_square;
```

The routine *draw_square* is used to execute the same motion pattern at three different positions, based on the positions *p10*, *p20* and *p30*. See Figure 11.

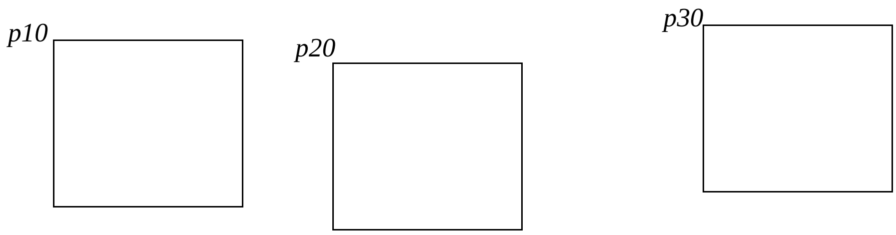


Figure 11 Using program displacement, motion patterns can be reused.

```
SearchL sen1, psearch, p10, v100, tool1\WObj:=fixture1;
PDispOn \ExeP:=psearch, *, tool1 \WObj:=fixture1;
```

A search is carried out in which the robot’s searched position is stored in the position *psearch*. Any movement carried out after this starts from this position using a program displacement (parallel movement). The latter is calculated based on the difference between the searched position and the programmed point (*) stored in the instruction. All positions are based on the *fixture1* object coordinate system.

Syntax

PDispOn
[[’\’ Rot]
[’\’ ExeP ’:=’ < expression (**IN**) of *robtarget* >] ’,’]
[ProgPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[Tool ’:=’] < persistent (**PERS**) of *tooldata*>
[‘\’WObj ’:=’ < persistent (**PERS**) of *wobjdata*>] ‘;’

Related information

	Described in:
Deactivation of program displacement	Instructions - *PDispOff*
Definition of program displacement using values	Instructions - *PDispSet*
Coordinate systems	Motion Principles - *Coordinate Systems*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
More examples	Instructions - *PDispOff*

PDispSet Activates program displacement using a value

PDispSet (Program Displacement Set) is used to define and activate a program displacement using values.

Program displacement is used, for example, when similar motion patterns are repeated at several different places in the program.

Example

```
VAR pose xp100 :=  [ [100, 0, 0], [1, 0, 0, 0] ];
.
PDispSet xp100;
```

Activation of the *xp100* program displacement, meaning that:

- The ProgDisp coordinate system is displaced 100 mm from the object coordinate system, in the direction of the positive x-axis (see Figure 12).
- As long as this program displacement is active, all positions will be displaced 100 mm in the direction of the x-axis.

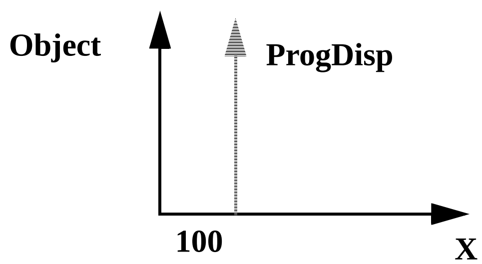


Figure 12 A 100 mm-program displacement along the x-axis.

Arguments

PDispSet DispFrame

DispFrame *(Displacement Frame)* Datatyp: *pose*

The program displacement is defined as data of the type *pose*.

Program execution

Program displacement involves translating and/or rotating the ProgDisp coordinate system relative to the object coordinate system. Since all positions are related to the ProgDisp coordinate system, all programmed positions will also be displaced. See Figure 13.

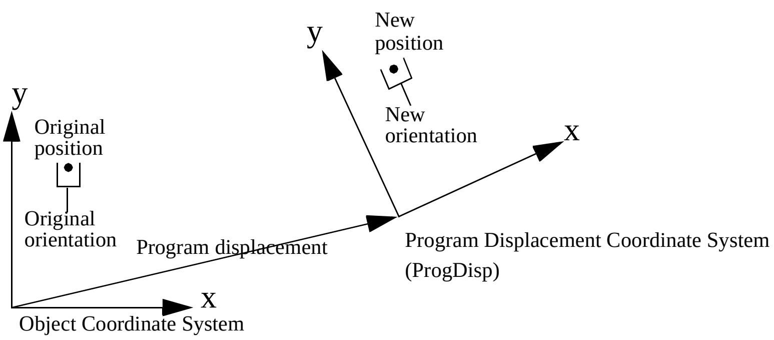


Figure 13 Translation and rotation of a programmed position.

Program displacement is activated when the instruction *PDispSet* is executed and remains active until some other program displacement is activated (the instruction *PDispSet* or *PDispOn*) or until program displacement is deactivated (the instruction *PDispOff*).

Only <u>one</u> program displacement can be active at any one time. Program displacements cannot be added to one another using *PDispSet*.

The program displacement is automatically reset

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Syntax

PDispSet
[DispFrame ':='] < expression (**IN**) of *pose*> ';'

Related information

	Described in:
Deactivation of program displacement	Instructions - *PDispOff*
Definition of program displacement using two positions	Instructions - *PDispOn*
Definition of data of the type *pose*	Data Types - *pose*
Coordinate systems	Motion Principles- *Coordinate Systems*
Examples of how program displacement can be used	Instructions - *PDispOn*

PulseDO Generates a pulse on a digital output signal

PulseDO is used to generate a pulse on a digital output signal.

Examples

PulseDO do15;

A pulse with a pulse length of 0.2 s is generated on the output signal *do15*.

PulseDO \PLength:=1.0, ignition;

A pulse of length *1.0 s* is generated on the signal *ignition*.

Arguments

PulseDO [\PLength] Signal

[\PLength] *(Pulse Length)* Data type: *num*

The length of the pulse in seconds (0.1 - 32s).
If the argument is omitted, a 0.2 second pulse is generated.

Signal Data type: *signaldo*

The name of the signal on which a pulse is to be generated.

Program execution

A pulse is generated with a specified pulse length (see Figure 14).

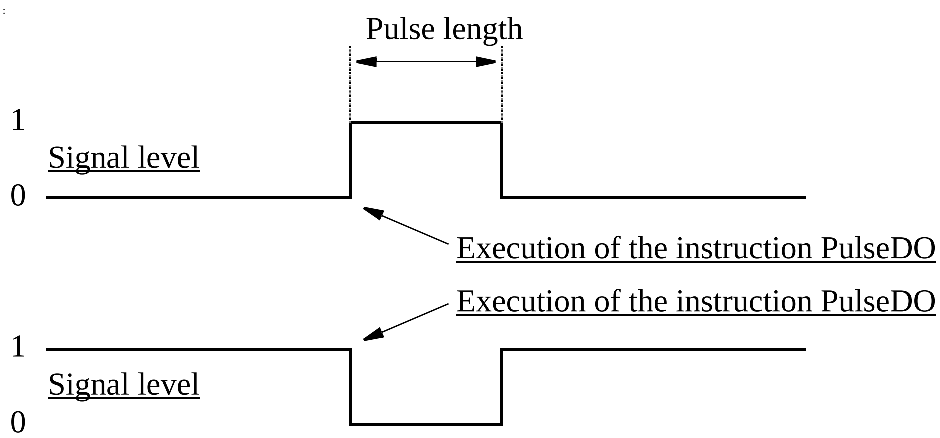


Figure 14 Generation of a pulse on a digital output signal.

The next instruction is executed directly after the pulse starts. The pulse can then be set/reset without affecting the rest of the program execution.

Limitations

The length of the pulse has a resolution of 0.01 seconds. Programmed values that differ from this are rounded off.

Syntax

PulseDO
['\' PLength ':=' < expression (**IN**) of *num* > ',']
[Signal ':='] < variable (**VAR**) of *signaldo* > ';'

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*

RAISE Calls an error handler

RAISE is used to create an error in the program and then to call the error handler of the routine. *RAISE* can also be used in the error handler to propagate the current error to the error handler of the calling routine.

This instruction can, for example, be used to jump back to a higher level in the structure of the program, e.g. to the error handler in the main routine, if an error occurs at a lower level.

Example

```
IF ...
   IF ...
      IF ...
         RAISE escape1;
   .
ERROR
   IF ERRNO=escape1 RAISE;
```

The routine is interrupted to enable it to remove itself from a low level in the program. A jump occurs to the error handler of the called routine.

Arguments

RAISE [Error no.]

Error no. Data type: *errnum*

Error number: Any number between 1 and 90 which the error handler can use to locate the error that has occurred (the *ERRNO* system variable).

It is also possible to book an error number outside the range 1-90 with the instruction *BookErrNo*.

The error number must be specified outside the error handler in a RAISE instruction in order to be able to transfer execution to the error handler of that routine.

If the instruction is present in a routine's error handler, the error number may not be specified. In this case, the error is propagated to the error handler of the calling routine.

Program execution

Program execution continues in the routine's error handler. After the error handler has been executed, program execution can continue with:

- the routine that called the routine in question (RETURN),
- the error handler of the routine that called the routine in question (RAISE).

If the RAISE instruction is present in a routine's error handler, program execution continues in the error handler of the routine that called the routine in question. The same error number remains active.

If the RAISE instruction is present in a trap routine, the error is dealt with by the system's error handler.

Error handling

If the error number is out of range, the system variable ERRNO is set to ERR_ILLRAISE (see "Data types - errnum"). This error can be handled in the error handler.

Syntax

(EBNF)
RAISE [<error number>] ';'

<error number> ::= <expression>

Related information

	Described in:
Error handling	Basic Characteristics - *Error Recovery*
Booking error numbers	Instructions - *BookErrNo*

Reset — Resets a digital output signal

Reset is used to reset the value of a digital output signal to zero.

Examples

Reset do15;

The signal *do15* is set to 0.

Reset weld;

The signal *weld* is set to 0.

Arguments

Reset Signal

Signal Data type: *signaldo*

The name of the signal to be reset to zero.

Program execution

The true value depends on the configuration of the signal. If the signal is inverted in the system parameters, this instruction causes the physical channel to be set to 1.

Syntax

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

Related information

	Described in:
Setting a digital output signal	Instructions - *Set*
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	System Parameters

RestoPath Restores the path after an interrupt

RestoPath is used to restore a path that was stored at a previous stage using the instruction *StorePath*.

Example

```
RestoPath;
```

Restores the path that was stored earlier using *StorePath*.

Program execution

The current movement path of the robot and the external axes is deleted and the path stored earlier using *StorePath* is restored. Nothing moves, however, until the instruction *StartMove* is executed or a return is made using *RETRY* from an error handler.

Example

```
ArcL p100, v100, seam1, weld5, weave1, z10, gun1;
...
ERROR
    IF ERRNO=AW_WELD_ERR THEN
        gun_cleaning;
        RETRY;
    ENDIF
...
PROC gun_cleaning()
    VAR robtarget p1;
    StorePath;
    p1 := CRobT();
    MoveL pclean, v100, fine, gun1;
    ...
    MoveL p1, v100, fine, gun1;
    RestoPath;
ENDPROC
```

In the event of a welding error, program execution continues in the error handler of the routine, which, in turn, calls *gun_cleaning*. The movement path being executed at the time is then stored and the robot moves to the position *pclean* where the error is rectified. When this has been done, the robot returns to the position where the error occurred, *p1*, and stores the original movement once again. The weld then automatically restarts, meaning that the robot is first reversed along the path before welding starts and ordinary program execution can continue.

Limitations

Only the movement path data is stored with the instruction *StorePath*.
If the user wants to order movements on the new path level, the actual stop position must be stored directly after *StorePath* and before *RestoPath* make a movement to the stored stop position on the path.

The movement instruction which precedes this instruction should be terminated with a stop point.

Syntax

RestoPath‘;’

Related information

	Described in:
Storing paths	Instructions - *StorePath*
More examples	Instructions - *StorePath*

RETRY Restarts following an error

RETRY is used to restart program execution after an error has occurred.

Example

```
reg2 := reg3/reg4;
    .
ERROR
    IF ERRNO = ERR_DIVZERO THEN
        reg4 := 1;
        RETRY;
    ENDIF
```

An attempt is made to divide *reg3* by *reg4*. If reg4 is equal to 0 (division by zero), a jump is made to the error handler, which initialises reg4. The *RETRY* instruction is then used to jump from the error handler and another attempt is made to complete the division.

Program execution

Program execution continues with (re-executes) the instruction that caused the error.

Error handling

If the maximum number of retries (4 retries) is exceeded, the program execution stops with an error message and the system variable ERRNO is set to ERR_EXCRTYMAX (see "Data types - errnum").

Limitations

The instruction can only exist in a routine's error handler. If the error was created using a *RAISE* instruction, program execution cannot be restarted with a *RETRY* instruction, then the instruction *TRYNEXT* should be used.

Syntax

RETRY ';'

Related information

	Described in:
Error handlers	Basic Characteristics- *Error Recovery*
Continue with the next instruction	Instructions - *TRYNEXT*

RETURN Finishes execution of a routine

RETURN is used to finish the execution of a routine. If the routine is a function, the function value is also returned.

Examples

```
errormessage;
Set do1;
    .

PROC errormessage()
    TPWrite "ERROR";
    RETURN;
ENDPROC
```

The *errormessage* procedure is called. When the procedure arrives at the RETURN instruction, program execution returns to the instruction following the procedure call, *Set do1*.

```
FUNC num abs_value(num value)
    IF value<0 THEN
        RETURN -value;
    ELSE
        RETURN value;
    ENDIF
ENDFUNC
```

The function returns the absolute value of a number.

Arguments

RETURN [Return value]

Return value Data type: According to the function declaration

The return value of a function.

The return value must be specified in a RETURN instruction present in a function.

If the instruction is present in a procedure or trap routine, a return value may not be specified.

Program execution

The result of the *RETURN* instruction may vary, depending on the type of routine it is used in:

- Main routine: If a program stop has been ordered at the end of the cycle, the program stops. Otherwise, program execution continues with the first instruction of the main routine.
- Procedure: Program execution continues with the instruction following the procedure call.
- Function: Returns the value of the function.
- Trap routine: Program execution continues from where the interrupt occurred.
- Error handler: In a procedure:
  Program execution continues with the routine that called the routine with the error handler (with the instruction following the procedure call).

  In a function:
  The function value is returned.

Syntax

(EBNF)
RETURN [<expression>]';'

Related information

	Described in:
Functions and Procedures	Basic Characteristics - *Routines*
Trap routines	Basic Characteristics - *Interrupts*
Error handlers	Basic Characteristics - *Error Recovery*

Rewind Rewind file position

Rewind sets the file position to the beginning of the file.

Example

Rewind iodev1;

The file referred to by *iodev1* will have the file position set to the beginning of the file.

Arguments

Rewind IODevice

IODevice Data type: *iodev*

Name (reference) of the file to be rewound.

Program execution

The specified file is rewound to the beginning.

Example

```
! IO device and numeric variable for use together with a binary file
VAR iodev dev;
VAR num bindata;

! Open the binary file with \Write switch to erase old contents
Open "flp1:"\File := "bin_file",dev \Write;
Close dev;

! Open the binary file with \Bin switch for binary read and write access
Open "flp1:"\File := "bin_file",dev \Bin;
WriteStrBin dev,"Hello world";

! Rewind the file pointer to the beginning of the binary file
! Read contents of the file and write the binary result on TP
! (gives 72 101 108 108 111 32 119 111 114 108 100 )
Rewind dev;
bindata := ReadBin(dev);
WHILE bindata <> EOF_BIN DO
    TPWrite " " \Num:=bindata;
    bindata := ReadBin(dev);
ENDWHILE

! Close the binary file
Close dev;
```

The instruction *Rewind* is used to rewind a binary file to the beginning so that the contents of the file can be read back with *ReadBin.*

Syntax

Rewind
[IODevice ':='] <variable (**VAR**) of *iodev*>';'

Related information

	Described in:
Opening (etc.) of files	RAPID Summary - *Communication*

Save Save a program module

Save is used to save a program module.

The specified program module in the program memory will be saved with the original (specified in *Load* or *StartLoad*) or specified file path.

It is also possible to save a system module at the specified file path.

Example

```
Load "ram1disk:PART_B.MOD";
...
Save "PART_B";
```

Load the program module with the file name PART_B.MOD from the *ram1disk* into the program memory.

Save the program module PART_B with the original file path *ram1disk* with the original file name PART_B.MOD.

Arguments

Save [\Task] ModuleName [\FilePath] [\File]

[\Task]

Data type: *taskid*

The program task in which the program module should be saved.

If this argument is omitted, the specified program module in the current (executing) program task will be saved.

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 the MAIN task the variable identity will be MAINId, TSK1 - TSK1Id etc.

ModuleName

Data type: *string*

The program module to save.

[\FilePath]

Data type: *string*

The file path and the file name to the place where the program module is to be saved. The file name shall be excluded when the argument *\File* is used.

[\File] Data type: *string*

When the file name is excluded in the argument *FilePath*, it must be specified with this argument.

The argument *FilePath* can only be omitted for program modules loaded with *Load* or *StartLoad-WaitLoad* and the program module will be stored at the same destination as specified in these instructions. To store the program module at another destination, it is also possible to use the argument *FilePath*.

To be able to save a program module that previously was loaded from the teach pendant, external computer, or system configuration, then the argument *FilePath* must be used.

Program execution

Program execution waits for the program module to finish saving before proceeding with the next instruction.

Example

```
Save  "PART_A" \FilePath:="ram1disk:DOORDIR/PART_A.MOD";
```

Save the program module PART_A to the *ram1disk* in the file PART_A.MOD and in the directory DOORDIR.

```
Save  "PART_A" \FilePath:="ram1disk:DOORDIR/" \File:="PART_A.MOD";
```

Same as above but another syntax.

```
Save \Task:=TSK1Id, "PART_A" \FilePath:="ram1disk:DOORDIR/PART_A.MOD";
```

Save program module PART_A in program task TSK1 to the specified destination. This is an example where the instruction *Save* is executing in one program task and the saving is done in another program task.

Limitations

TRAP routines, system I/O events and other program tasks cannot execute during the saving operation. Therefore, any such operations will be delayed.

The save operation can interrupt update of PERS data done step by step from other program tasks. This will result in inconsistent whole PERS data.

A program stop during execution of the *Save* instruction can result in a guard stop with motors off and the error message "20025 Stop order timeout" will be displayed on the Teach Pendant.

Avoid ongoing robot movements during the saving.

Error handling

If the program module cannot be saved because of no module name, unknown, or ambiguous module name, the system variable ERRNO is set to ERR_MODULE.

If the save file cannot be opened because of permission denied, no such directory, or no space left on device, then the system variable ERRNO is set to ERR_IOERROR.

If argument *\FilePath* is not specified for program modules loaded from the Teach Pendant, System Parameters, or an external computer, the system variable ERRNO is set to ERR_PATH.

The errors above can be handled in the error handler.

Syntax

Save
  ['\' Task ':=' <variable (**VAR**) of *taskid*> ',']
  [ModuleName ':='] <expression (**IN**) of *string*>
  ['\' FilePath ':='<expression (**IN**) of *string*>]
  ['\' File ':=' <expression (**IN**) of *string*>] ';'

Related information

	Described in:
Program tasks	Data Types - *taskid*

SearchC Searches circularly using the robot

SearchC (Search Circular) is used to search for a position when moving the tool centre point (TCP) circularly.

During the movement, the robot supervises a digital input signal. When the value of the signal changes to the requested one, the robot immediately reads the current position.

This instruction can typically be used when the tool held by the robot is a probe for surface detection. Using the *SearchC* instruction, the outline coordinates of a work object can be obtained.

Examples

SearchC sen1, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10* at a speed of *v100*. When the value of the signal *sen1* changes to active, the position is stored in *sp*.

SearchC \Stop, sen1, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp* and the robot stops immediately.

Arguments

SearchC [\Stop] | [\PStop] | [\Sup] Signal [\Flanks] SearchPoint CirPoint ToPoint Speed [\V] | [\T] Tool [\WObj] [\Corr]

[\Stop] Data type: *switch*

The robot movement is stopped, as quickly as possible, without keeping the TCP on the path (hard stop), when the value of the search signal changes to active. However, the robot is moved a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\PStop] *(Path Stop)* Data type: *switch*

The robot movement is stopped as quickly as possible, while keeping the TCP on the path (soft stop), when the value of the search signal changes to active. However, the robot is moved a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\Sup] *(Supervision)* Data type: *switch*

The search instruction is sensitive to signal activation during the complete movement (flying search), i.e. even after the first signal change has been reported. If more than one match occurs during a search, program execution stops.

If the argument *\Stop*, *\PStop* or *\Sup* is omitted, the movement continues (flying search) to the position specified in the *ToPoint* argument (same as with argument *\Sup*),

Signal Data type: *signaldi*

The name of the signal to supervise.

[\Flanks] Data type: *switch*

The positive and the negative edge of the signal is valid for a search hit.

If the argument *\Flanks* is omitted, only the positive edge of the signal is valid for a search hit and a signal supervision will be activated at the beginning of a search process. This means that if the signal has a positive value already at the beginning of a search process, the robot movement is stopped as quickly as possible, while keeping the TCP on the path (soft stop). However, the robot is moved a small distance before it stops and is not moved back to the start position. A user recovery error (ERR_SIGSUPSEARCH) will be generated and can be dealt with by the error handler.

SearchPoint Data type: *robtarget*

The position of the TCP and external axes when the search signal has been triggered. The position is specified in the outermost coordinate system, taking the specified tool, work object and active ProgDisp/ExtOffs coordinate system into consideration.

CirPoint Data type: *robtarget*

The circle point of the robot. See the instruction MoveC for a more detailed description of circular movement. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction). *SearchC* always uses a stop point as zone data for the destination.

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the external axes and of the tool reorientation.

[\V] *(Velocity)* Data type: *num*

This argument is used to specify the velocity of the TCP in mm/s directly in the instruction. It is then substituted for the corresponding velocity specified in the speed data.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination position.

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

The work object (coordinate system) to which the robot positions in the instruction are related.

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

[\Corr] *(Correction)* Data type: *switch*

Correction data written to a corrections entry by the instruction *CorrWrite* will be added to the path and destination position, when this argument is present.

Program execution

See the instruction *MoveC* for information about circular movement.

The movement is always ended with a stop point, i.e. the robot is stopped at the destination point.

When a flying search is used, i.e. the *\Sup* argument is specified, the robot movement always continues to the programmed destination point. When a search is made using the switch *\Stop* or *\PStop*, the robot movement stops when the first signal is detected.

The *SearchC* instruction returns the position of the TCP when the value of the digital signal changes to the requested one, as illustrated in Figure 15.

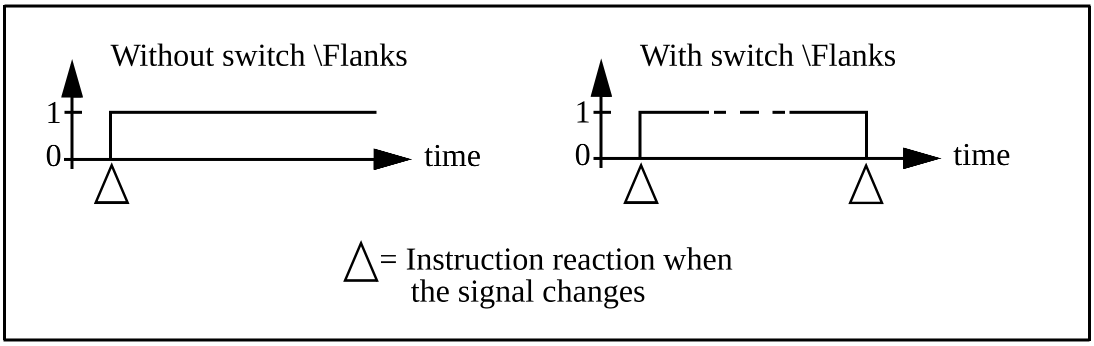


Figure 15 Flank-triggered signal detection (the position is stored when the signal is changed the first time only).

Example

SearchC \Sup, sen1\Flanks, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10*. When the value of the signal *sen1* changes to active or passive, the position is stored in *sp*. If the value of the signal changes twice, program execution stops.

Limitations

Zone data for the positioning instruction that precedes *SearchC* must be used carefully. The start of the search, i.e. when the I/O signal is ready to react, is not, in this case, the programmed destination point of the previous positioning instruction, but a point along the real robot path. Figure 16 illustrates an example of something that may go wrong when zone data other than *fine* is used.

The instruction *SearchC* should never be restarted after the circle point has been passed. Otherwise the robot will not take the programmed path (positioning around the circular path in another direction compared with that programmed).

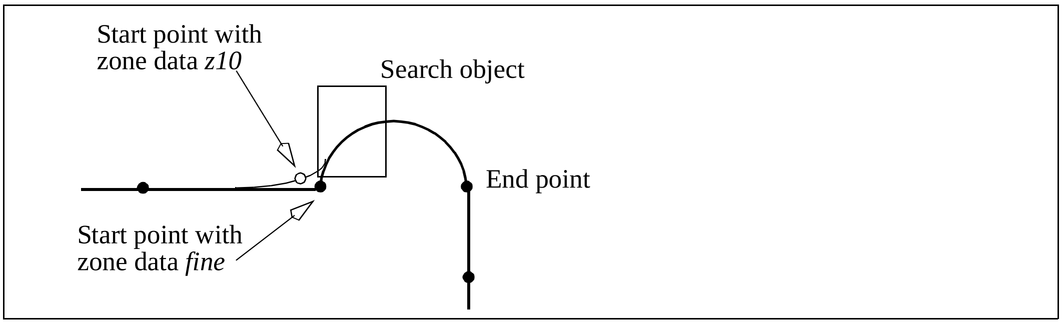


Figure 16 A match is made on the wrong side of the object because the wrong zone data was used.

Typical stop distance using a search velocity of 50 mm/s:

- without TCP on path (switch *Stop*) 1-3 mm
- with TCP on path (switch *PStop*) 12-16 mm

Error handling

An error is reported during a search when:

- no signal detection occurred - this generates the error ERR_WHLSEARCH.
- more than one signal detection occurred – this generates the error ERR_WHLSEARCH only if the *Sup* argument is used.
- the signal has already a positive value at the beginning of the search process - this generates the error ERR_SIGSUPSEARCH only if the *Flanks* argument is omitted.

Errors can be handled in different ways depending on the selected running mode:

Continuous forward / ERR_WHLSEARCH
No position is returned and the movement always continues to the programmed destination point. The system variable ERRNO is set to ERR_WHLSEARCH and the error can be handled in the error handler of the routine.

Continuous forward / Instruction forward / ERR_SIGSUPSEARCH
No position is returned and the movement always stops as quickly as possible at the beginning of the search path. The system variable ERRNO is set to ERR_SIGSUPSEARCH and the error can be handled in the error handler of the routine.

Instruction forward / ERR_WHLSEARCH
No position is returned and the movement always continues to the programmed destination point. Program execution stops with an error message.

Instruction backward
During backward execution, the instruction just carries out the movement without any signal supervision.

Syntax

SearchC
[’\’ Stop’,’] | [’\’ PStop ’,’] | [’\’ Sup ’,’]
[Signal ’:=’] < variable (**VAR**) of *signaldi* >
[‘\’ Flanks]’,’
[SearchPoint ’:=’] < var or pers (**INOUT**) of *robtarget* > ’,’
[CirPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[Speed ’:=’] < expression (**IN**) of *speeddata* >
[’\’ V ’:=’ < expression (**IN**) of *num* >]
| [’\’ T ’:=’ < expression (**IN**) of *num* >] ’,’
[Tool ’:=’] < persistent (**PERS**) of *tooldata* >
[’\’ WObj ’:=’ < persistent (**PERS**) of *wobjdata* >]
[’\’ Corr]’;’

Related information

	Described in:
Linear searches	Instructions - *SearchL*
Writes to a corrections entry	Instructions - *CorrWrite*
Circular movement	Motion and I/O Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Using error handlers	RAPID Summary - *Error Recovery*
Motion in general	Motion and I/O Principles
More searching examples	Instructions - *SearchL*

SearchL Searches linearly using the robot

SearchL (Search Linear) is used to search for a position when moving the tool centre point (TCP) linearly.

During the movement, the robot supervises a digital input signal. When the value of the signal changes to the requested one, the robot immediately reads the current position.

This instruction can typically be used when the tool held by the robot is a probe for surface detection. Using the *SearchL* instruction, the outline coordinates of a work object can be obtained.

Examples

SearchL sen1, sp, p10, v100, probe;

The TCP of the *probe* is moved linearly towards the position *p10* at a speed of *v100*. When the value of the signal *sen1* changes to active, the position is stored in *sp*.

SearchL \Stop, sen1, sp, p10, v100, probe;

The TCP of the *probe* is moved linearly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp* and the robot stops immediately.

Arguments

SearchL [\Stop] | [\PStop] |[\Sup] Signal [\Flanks] SearchPoint ToPoint Speed [\V] | [\T] Tool [\WObj] [\Corr]

[\Stop] Data type: *switch*

The robot movement is stopped as quickly as possible, without keeping the TCP on the path (hard stop), when the value of the search signal changes to active. However, the robot is moved a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\PStop] *(Path Stop)* Data type: *switch*

The robot movement is stopped as quickly as possible, while keeping the TCP on the path (soft stop), when the value of the search signal changes to active. However, the robot is moved a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\Sup] *(Supervision)* Data type: *switch*

The search instruction is sensitive to signal activation during the complete movement (flying search), i.e. even after the first signal change has been reported. If more than one match occurs during a search, program execution stops.

If the argument *\Stop*, *\PStop* or *\Sup* is omitted, the movement continues (flying search) to the position specified in the *ToPoint* argument (same as with argument *\Sup*).

Signal Data type: *signaldi*

The name of the signal to supervise.

[\Flanks] Data type: *switch*

The positive and the negative edge of the signal is valid for a search hit.

If the argument *\Flanks* is omitted, only the positive edge of the signal is valid for a search hit and a signal supervision will be activated at the beginning of a search process. This means that if the signal has the positive value already at the beginning of a search process, the robot movement is stopped as quickly as possible, while keeping the TCP on the path (soft stop). A user recovery error (ERR_SIGSUPSEARCH) will be generated and can be handled in the error handler.

SearchPoint Data type: *robtarget*

The position of the TCP and external axes when the search signal has been triggered. The position is specified in the outermost coordinate system, taking the specified tool, work object and active ProgDisp/ExtOffs coordinate system into consideration.

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction). *SearchL* always uses a stop point as zone data for the destination.

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the external axes and of the tool reorientation.

[\V] *(Velocity)* Data type: *num*

This argument is used to specify the velocity of the TCP in mm/s directly in the instruction. It is then substituted for the corresponding velocity specified in the speed data.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination position.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

[\Corr] *(Correction)* Data type: *switch*

Correction data written to a corrections entry by the instruction *CorrWrite* will be added to the path and destination position, if this argument is present.

Program execution

See the instruction *MoveL* for information about linear movement.

The movement always ends with a stop point, i.e. the robot stops at the destination point.

If a flying search is used, i.e. the *\Sup* argument is specified, the robot movement always continues to the programmed destination point. If a search is made using the switch *\Stop* or *\PStop*, the robot movement stops when the first signal is detected.

The *SearchL* instruction stores the position of the TCP when the value of the digital signal changes to the requested one, as illustrated in Figure 17.

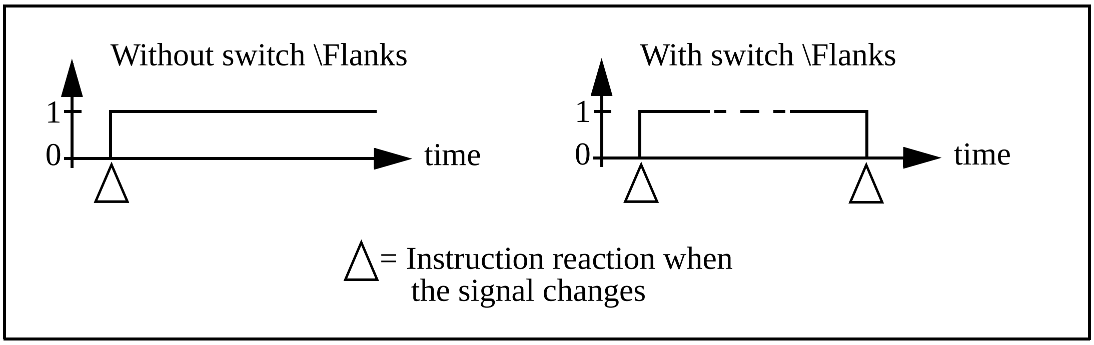


Figure 17 Flank-triggered signal detection (the position is stored when the signal is changed the first time only).

In order to get a fast response, use the interrupt-driven sensor signals *sen1*, *sen2* or *sen3* on the system board.

Examples

SearchL \Sup, sen1\Flanks, sp, p10, v100, probe;

The TCP of the *probe* is moved linearly towards the position *p10*. When the value of the signal *sen1* changes to active or passive, the position is stored in *sp*. If the value of the signal changes twice, program execution stops after the search process is finished.

```
SearchL \Stop, sen1, sp, p10, v100, tool1;
MoveL sp, v100, fine, tool1;
PDispOn *, tool1;
MoveL p100, v100, z10, tool1;
MoveL p110, v100, z10, tool1;
MoveL p120, v100, z10, tool1;
PDispOff;
```

At the beginning of the search process, a check on the signal *sen1* will be done and if the signal already has a positive value, the program execution stops. Otherwise the TCP of *tool1* is moved linearly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp* and the robot is moved back to this point. Using program displacement, the robot then moves relative to the searched position, *sp*.

Limitations

Zone data for the positioning instruction that precedes *SearchL* must be used carefully. The start of the search, i.e. when the I/O signal is ready to react, is not, in this case, the programmed destination point of the previous positioning instruction, but a point along the real robot path. Figure 18 to Figure 20 illustrate examples of things that may go wrong when zone data other than *fine* is used.

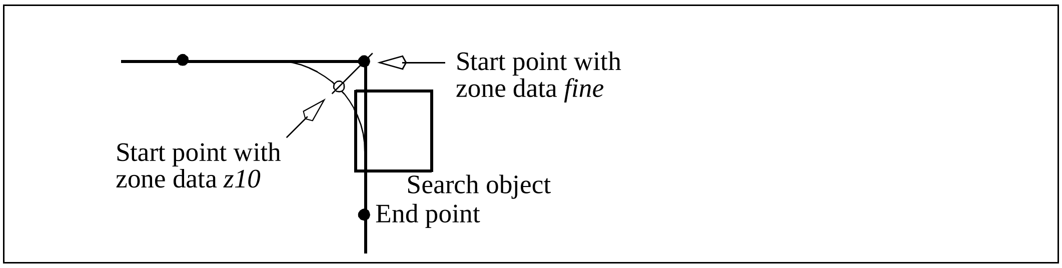


Figure 18 A match is made on the wrong side of the object because the wrong zone data was used.

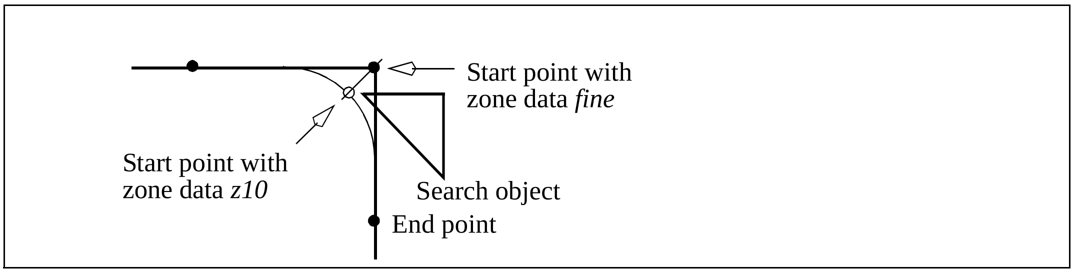


Figure 19 No match detected because the wrong zone data was used.

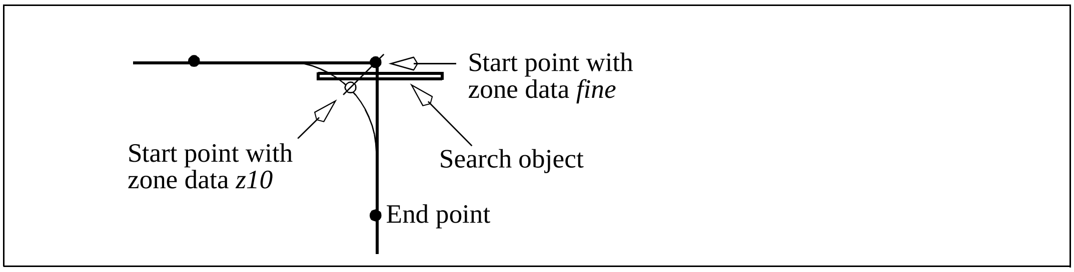


Figure 20 No match detected because the wrong zone data was used.

Typical stop distance using a search velocity of 50 mm/s:

- without TCP on path (switch *Stop*) 1-3 mm
- with TCP on path (switch *PStop*) 12-16 mm

Error handling

An error is reported during a search when:

- no signal detection occurred - this generates the error ERR_WHLSEARCH.
- more than one signal detection occurred – this generates the error ERR_WHLSEARCH only if the *Sup* argument is used.
- the signal already has a positive value at the beginning of the search process - this generates the error ERR_SIGSUPSEARCH only if the *Flanks* argument is omitted.

Errors can be handled in different ways depending on the selected running mode:

Continuous forward / ERR_WHLSEARCH
No position is returned and the movement always continues to the programmed destination point. The system variable ERRNO is set to ERR_WHLSEARCH and the error can be handled in the error handler of the routine.

Continuous forward / Instruction forward / ERR_SIGSUPSEARCH
No position is returned and the movement always stops as quickly as possible at the beginning of the search path.The system variable ERRNO is set to ERR_SIGSUPSEARCH and the error can be handled in the error handler of the routine.

Instruction forward / ERR_WHLSEARCH
No position is returned and the movement continues to the programmed destination point. Program execution stops with an error message.

Instruction backward
During backward execution, the instruction just carries out the movement without any signal supervision.

Example

```
VAR num fk;
.
MoveL p10, v100, fine, tool1;
SearchL \Stop, sen1, sp, p20, v100, tool1;
.
ERROR
   IF ERRNO=ERR_WHLSEARCH THEN
      MoveL p10, v100, fine, tool1;
      RETRY;
   ELSEIF ERRNO=ERR_SIGSUPSEARCH THEN
      TPWrite "The signal of the SearchL instruction is already high!";
      TPReadFK fk,"Try again after manual reset of signal ?","YES","","","","NO";
      IF fk = 1 THEN
         MoveL p10, v100, fine, tool1;
         RETRY;
      ELSE
         Stop;
      ENDIF
   ENDIF
```

If the signal is already active at the beginning of the search process, a user dialog will be activated (TPReadFK ...;). Reset the signal and push YES on the user dialog and the robot moves back to p10 and tries once more. Otherwise program execution will stop.

If the signal is passive at the beginning of the search process, the robot searches from position *p10* to *p20*. If no signal detection occurs, the robot moves back to *p10* and tries once more.

Syntax

SearchL
['\' Stop ','] | ['\' PStop ','] | ['\' Sup ',']
[Signal ':='] < variable (**VAR**) of *signaldi* >
[‘\' Flanks] ','
[SearchPoint ':='] < var or pers (**INOUT**) of *robtarget* > ','
[ToPoint ':='] < expression (**IN**) of *robtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' V ':=' < expression (**IN**) of *num* >]
| ['\' T ':=' < expression (**IN**) of *num* >] ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >]
['\' Corr]';'

Related information

	Described in:
Circular searches	Instructions - *SearchC*
Writes to a corrections entry	Instructions - *CorrWrite*
Linear movement	Motion and I/O Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Using error handlers	RAPID Summary - *Error Recovery*
Motion in general	Motion and I/O Principles

Set — Sets a digital output signal

Set is used to set the value of a digital output signal to one.

Examples

Set do15;

The signal *do15* is set to 1.

Set weldon;

The signal *weldon* is set to 1.

Arguments

Set Signal

Signal Data type: *signaldo*

The name of the signal to be set to one.

Program execution

The true value depends on the configuration of the signal. If the signal is inverted in the system parameters, this instruction causes the physical channel to be set to zero.

Syntax

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

Related information

	Described in:
Setting a digital output signal to zero	Instructions - *Reset*
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	System Parameters

SetAO Changes the value of an analog output signal

SetAO is used to change the value of an analog output signal.

Example

SetAO ao2, 5.5;

The signal *ao2* is set to *5.5*.

Arguments

SetAO Signal Value

Signal Data type: *signalao*

The name of the analog output signal to be changed.

Value Data type: *num*

The desired value of the signal.

Program execution

The programmed value is scaled (in accordance with the system parameters) before it is sent on the physical channel. See Figure 21.

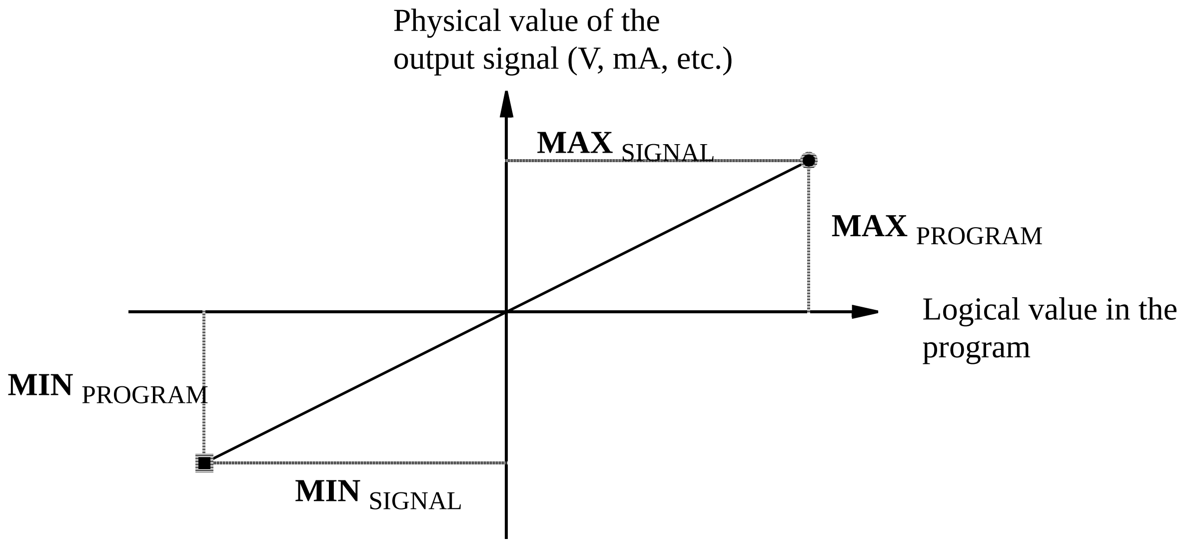


Figure 21 Diagram of how analog signal values are scaled.

Example

SetAO weldcurr, curr_outp;

The signal *weldcurr* is set to the same value as the current value of the variable *curr_outp*.

Syntax

SetAO
[Signal ’:=’] < variable (**VAR**) of *signalao* > ’,’
[Value ’:=’] < expression (**IN**) of *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	System Parameters

SetDO Changes the value of a digital output signal

SetDO is used to change the value of a digital output signal, with or without a time-delay.

Examples

SetDO do15, 1;

The signal *do15* is set to *1*.

SetDO weld, off;

The signal *weld* is set to *off*.

SetDO \SDelay := 0.2, weld, high;

The signal *weld* is set to *high* with a delay of *0.2* s. Program execution, however, continues with the next instruction.

Arguments

SetDO [\SDelay] Signal Value

[\SDelay] *(Signal Delay)* Data type: *num*

Delays the change for the amount of time given in seconds (0.1 - 32s). Program execution continues directly with the next instruction. After the given time-delay, the signal is changed without the rest of the program execution being affected.

If the argument is omitted, the value of the signal is changed directly.

Signal Data type: *signaldo*

The name of the signal to be changed.

Value Data type: *dionum*

The desired value of the signal.

The value is specified as 0 or 1.

Program execution

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

Syntax

SetDO
  ['\' SDelay ':=' < expression (**IN**) of *num* > ',']
  [Signal ':='] < variable (**VAR**) of *signaldo* > ','
  [Value ':='] < expression (**IN**) of *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*

SetGO Changes the value of a group of digital output signals

SetGO is used to change the value of a group of digital output signals, with or without a time delay.

Example

SetGO go2, 12;

The signal *go2* is set to *12*. If *go2* comprises 4 signals, e.g. outputs 6-9, outputs 6 and 7 are set to zero, while outputs 8 and 9 are set to one.

SetGO \SDelay := 0.4, go2, 10;

The signal *go2* is set to *10*. If *go2* comprises 4 signals, e.g. outputs 6-9, outputs 6 and 8 are set to zero, while outputs 7 and 9 are set to one, with a delay of *0.4* s. Program execution, however, continues with the next instruction.

Arguments

SetGO [\SDelay] Signal Value

[\SDelay] *(Signal Delay)* Data type: *num*

Delays the change for the period of time stated in seconds (0.1 - 32s). Program execution continues directly with the next instruction. After the specified time delay, the value of the signals is changed without the rest of the program execution being affected.

If the argument is omitted, the value is changed directly.

Signal Data type: *signalgo*

The name of the signal group to be changed.

Value Data type: *num*

The desired value of the signal group (a positive integer).

The permitted value is dependent on the number of signals in the group:

No. of signals	Permitted value	No. of signals	Permitted 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

Program execution

The programmed value is converted to an unsigned binary number. This binary number is sent on the signal group, with the result that individual signals in the group are set to 0 or 1. Due to internal delays, the value of the signal may be undefined for a short period of time.

Syntax

SetDO
    ['\' SDelay ':=' < expression (**IN**) of *num* > ',']
    [Signal ':='] < variable (**VAR**) of *signalgo* > ','
    [Value ':='] < expression (**IN**) of *num* > ';'

Related information

	Described in:
Other 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 (system parameters)	System Parameters

SingArea Defines interpolation around singular points

SingArea is used to define how the robot is to move in the proximity of singular points.

SingArea is also used to define linear and circular interpolation for robots with less than six axes.

Examples

SingArea \Wrist;

The orientation of the tool may be changed slightly in order to pass a singular point (axes 4 and 6 in line).

Robots with less than six axes may not be able to reach an interpolated tool orientation. By using SingArea \Wrist, the robot can achieve the movement but the orientation of the tool will be slightly changed.

SingArea \Off;

The tool orientation is not allowed to differ from the programmed orientation. If a singular point is passed, one or more axes may perform a sweeping movement, resulting in a reduction in velocity.

Robots with less than six axes may not be able to reach a programmed tool orientation. As a result the robot will stop.

Arguments

SingArea [\Wrist] | [\Off]

[\Wrist] Data type: *switch*

The tool orientation is allowed to differ somewhat in order to avoid wrist singularity. Used when axes 4 and 6 are parallel (axis 5 at 0 degrees). Also used for linear and circular interpolation of robots with less than six axes where the tool orientation is allowed to differ.

[\Off] Data type: *switch*

The tool orientation is not allowed to differ. Used when no singular points are passed, or when the orientation is not permitted to be changed.

If none of the arguments are specified, program execution automatically uses the robot's default argument. For robots with six axes the default argument is *\Off*.

Program execution

If the arguments *\Wrist* is specified, the orientation is joint-interpolated to avoid singular points. In this way, the TCP follows the correct path, but the orientation of the tool deviates somewhat. This will also happen when a singular point is not passed.

The specified interpolation applies to all subsequent movements until a new *SingArea* instruction is executed.

The movement is only affected on execution of linear or circular interpolation.

By default, program execution automatically uses the */Off* argument for robots with six axes. Robots with less than six axes may use either the */Off* argument (IRB640) or the /*Wrist* argument by default. This is automatically set in event routine SYS_RESET.

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Syntax

```
SingArea
   [ '\' Wrist ] | [ '\' Off ] ';'
```

Related information

	Described in:
Singularity	Motion Principles- *Singularity*
Interpolation	Motion Principles - *Positioning during Program Execution*

SoftAct Activating the soft servo

SoftAct (Soft Servo Activate) is used to activate the so called “soft” servo on any axis of the robot or external mechanical unit.

Example

```
SoftAct 3, 20;
```

Activation of soft servo on robot axis *3*, with softness value *20*%.

```
SoftAct 1, 90 \Ramp:=150;
```

Activation of the soft servo on robot axis *1*, with softness value *90*% and ramp factor *150*%.

```
SoftAct \MechUnit:=orbit1, 1, 40 \Ramp:=120;
```

Activation of soft servo on axis *1* for the mechanical unit *orbit1*, with softness value *40*% and ramp factor *120*%.

Arguments

SoftAct [\MechUnit] Axis Softness [\Ramp]

[\MechUnit] *(Mechanical Unit)* Data type: *mecunit*

The name of the mechanical unit. If this argument is omitted, it means activation of the soft servo for specified robot axis.

Axis Data type: *num*

Number of the robot or external axis to work with soft servo.

Softness Data type: *num*

Softness value in percent (0 - 100%). 0% denotes min. softness (max. stiffness), and 100% denotes max. softness.

Ramp Data type: *num*

Ramp factor in percent (>= 100%). The ramp factor is used to control the engagement of the soft servo. A factor 100% denotes the normal value; with greater values the soft servo is engaged more slowly (longer ramp). The default value for ramp factor is 100 %.

Program execution

Softness is activated at the value specified for the current axis. The softness value is valid for all movements, until a new softness value is programmed for the current axis, or until the soft servo is deactivated by an instruction.

Limitations

The same axis must not be activated twice, unless there is a moving instruction in between. Thus, the following program sequence should be avoided, otherwise there will be a jerk in the robot movement:

```
SoftAct n , x ;
SoftAct n , y ;
```

(n = robot axis n, x and y softness values)

Syntax

SoftAct
 [’\’MechUnit ’:=’ < variable (**VAR**) of *mecunit*> ’,’]
 [Axis ’:=’] < expression (**IN**) of *num*> ’,’
 [Softness ’:=’] < expression (**IN**) of *num*>
 [’\’Ramp ’:=’ < expression (**IN**) of *num*>]’;’

Related information

	Described in:
Behaviour with the soft servo engaged	Motion and I/O Principles- *Positioning during program execution*

SoftDeact Deactivating the soft servo

SoftDeact (Soft Servo Deactivate) is used to deactivate the so called “soft” servo on all robot and external axes.

Example

SoftDeact;

Deactivating the soft servo on all axes.

SoftDeact \Ramp:=150;

Deactivating the soft servo on all axes, with ramp factor 150%.

Arguments

SoftDeact [\Ramp]

Ramp Data type: *num*

Ramp factor in percent (>= 100%). The ramp factor is used to control the deactivating of the soft servo. A factor 100% denotes the normal value; with greater values the soft servo is deactivated more slowly (longer ramp). The default value for ramp factor is 100 %.

Program execution

The soft servo is deactivated for all robot and external axes.

Syntax

SoftDeact
[’\’Ramp ’:=’ < expression (**IN**) of *num*>]’;’

Related information

	Described in:
Activating the soft servo	Instructions - *SoftAct*

StartLoad Load a program module during execution

StartLoad is used to start the loading of a program module into the program memory during execution.

When loading is in progress, other instructions can be executed in parallel.
The loaded module must be connected to the program task with the instruction *WaitLoad*, before any of its symbols/routines can be used.

The loaded program module will be added to the modules already existing in the program memory.

Example

```
VAR loadsession load1;

! Start loading of new program module PART_B containing routine routine_b
StartLoad ram1disk \File:="PART_B.MOD", load1;

! Executing in parallel in old module PART_A containing routine_a
%"routine_a"%;

! Unload of old program module PART_A
UnLoad ram1disk \File:="PART_A.MOD";

! Wait until loading and linking of new program module PART_B is ready
WaitLoad load1;

! Execution in new program module PART_B
%"routine_b"%;
```

Start loading of program module *PART_B.MOD* from *ram1disk* into the program memory with instruction *StartLoad*. In parallel with the loading, the program executes *routine_a* in module PART_A.MOD. Then instruction *WaitLoad* waits until the loading and linking is finished.

Variable *load1* holds the identity of the load session, updated by *StartLoad* and referenced by *WaitLoad*.

To save linking time, the instruction *UnLoad* and *WaitLoad* can be combined in the instruction *WaitLoad* by using the option argument *\UnLoadPath*.

Arguments

StartLoad FilePath [\File] LoadNo

FilePath Data type: *string*

The file path and the file name to the file that will be loaded into the program memory. The file name shall be excluded when the argument *\File* is used.

[\File] Data type: *string*

When the file name is excluded in the argument *FilePath*, then it must be defined with this argument.

LoadNo Data type: *loadsession*

This is a reference to the load session that should be used in the instruction *WaitLoad* to connect the loaded program module to the program task.

Program execution

Execution of *StartLoad* will only order the loading and then proceed directly with the next instruction, without waiting for the loading to be completed.

The instruction *WaitLoad* will then wait at first for the loading to be completed, if it is not already finished, and then it will be linked and initialised. The initialisation of the loaded module sets all variables at module level to their init values.

Unsolved references will be accepted if the system parameter for *Tasks/BindRef* is set to NO. However, when the program is started or the teach pendant function *Program Window/File/Check Program* is used, no check for unsolved references will be done if *BindRef* = NO. There will be a run time error on execution of an unsolved reference.

Another way to use references to instructions that are not in the task from the beginning, is to use *Late Binding*. This makes it possible to specify the routine to call with a string expression, quoted between two %%. In this case the *BindRef* parameter could be set to YES (default behaviour). The *Late Binding* way is preferable.

To obtain a good program structure, that is easy to understand and maintain, all loading and unloading of program modules should be done from the main module, which is always present in the program memory during execution.

Examples

StartLoad "ram1disk:DOORDIR/DOOR1.MOD", load1;

Load the program module *DOOR1.MOD* from the *ram1disk* at the directory *DOORDIR* into the program memory.

StartLoad "ram1disk:DOORDIR/" \File:="DOOR1.MOD", load1;

Same as above but with another syntax.

StartLoad "ram1disk:DOORDIR/" \File:="DOOR1.MOD", load1;
...
WaitLoad load1;

is the same as

Load "ram1disk:DOORDIR/" \File:="DOOR1.MOD";

Limitations

It is not allowed to load a system module or a program module that contains a main routine.

Syntax

StartLoad
[FilePath ':='] <expression (**IN**) of *string*>
['\'File ':=' <expression (**IN**) of *string*>] ','
[LoadNo ':='] <variable (**VAR**) of *loadsession*> ';'

Related information

	Described in:
Connect the loaded module to the task	Instructions - *WaitLoad*
Load session	Data Types - *loadsession*
Load a program module	Instructions - *Load*
Unload a program module	Instructions - *UnLoad*
Accept unsolved references	System Parameters - *Controller/Task/ BindRef*

StartMove Restarts robot motion

StartMove is used to resume robot and external axes motion when this has been stopped by the instruction *StopMove*.

Example

```
StopMove;
WaitDI ready_input, 1;
StartMove;
```

The robot starts to move again when the input *ready_input* is set.

Program execution

Any processes associated with the stopped movement are restarted at the same time as motion resumes.

Error handling

If the robot is too far from the path (more than 10 mm or 20 degrees) to perform a start of the interrupted movement, the system variable *ERRNO* is set to ERR_PATHDIST. This error can then be handled in the error handler.

Syntax

StartMove';'

Related information

	Described in:
Stopping movements	Instructions - *StopMove*
More examples	Instructions - *StorePath*

-

Stop — Stops program execution

Stop is used to temporarily stop program execution.

Program execution can also be stopped using the instruction *EXIT*. This, however, should only be done if a task is complete, or if a fatal error occurs, since program execution cannot be restarted with *EXIT*.

Example

```
TPWrite "The line to the host computer is broken";
Stop;
```

Program execution stops after a message has been written on the teach pendant.

Arguments

Stop [\NoRegain]

[\NoRegain] Data type: *switch*

Specifies for the next program start in manual mode, whether or not the robot and external axes should regain to the stop position. In automatic mode the robot and external axes always regain to the stop position.

If the argument *NoRegain* is set, the robot and external axes will not regain to the stop position (if they have been jogged away from it).

If the argument is omitted and if the robot or external axes have been jogged away from the stop position, the robot displays a question on the teach pendant. The user can then answer, whether or not the robot should regain to the stop position.

Program execution

The instruction stops program execution as soon as the robot and external axes reach the programmed destination point for the movement it is performing at the time. Program execution can then be restarted from the next instruction.

If there is a *Stop* instruction in some event routine, the routine will be executed from the beginning in the next event.

Example

```
MoveL p1, v500, fine, tool1;
TPWrite “Jog the robot to the position for pallet corner 1”;
Stop \NoRegain;
p1_read := CRobT();
MoveL p2, v500, z50, tool1;
```

Program execution stops with the robot at *p1*. The operator jogs the robot to *p1_read*. For the next program start, the robot does not regain to *p1,* so the position *p1_read* can be stored in the program.

Limitations

The movement instruction which precedes this instruction should be terminated with a stop point, in order to be able to restart in this instruction following a power failure.

Syntax

```
Stop
[ ’\’ NoRegain ]’;’
```

Related information

	Described in:
Stopping after a fatal error	Instructions - *EXIT*
Terminating program execution	Instructions - *EXIT*
Only stopping robot movements	Instructions - *StopMove*

StopMove — Stops robot motion

StopMove is used to stop robot and external axes movements temporarily. If the instruction *StartMove* is given, movement resumes.

This instruction can, for example, be used in a trap routine to stop the robot temporarily when an interrupt occurs.

Example

```
StopMove;
WaitDI ready_input, 1;
StartMove;
```

The robot movement is stopped until the input, *ready_input*, is set.

Program execution

The movements of the robot and external axes stop without the brakes being engaged. Any processes associated with the movement in progress are stopped at the same time as the movement is stopped.

Program execution continues without waiting for the robot and external axes to stop (standing still).

Examples

```
VAR intnum intno1;
...
CONNECT intno1 WITH go_to_home_pos;
ISignalDI di1,1,intno1;

TRAP go_to_home_pos
    VAR robtarget p10;

    StopMove;
    StorePath;
    p10:=CRobT();
    MoveL home,v500,fine,tool1;
    WaitDI di1,0;
    Move L p10,v500,fine,tool1;
    RestoPath;
    StartMove;
ENDTRAP
```

When the input *di1* is set to 1, an interrupt is activated which in turn activates the interrupt routine *go_to_home_pos*. The current movement is stopped immediately and the robot moves instead to the *home* position. When *di1* is set to 0, the robot returns to the position at which the interrupt occurred and continues to move along the programmed path.

```
VAR intnum intno1;
...
CONNECT intno1 WITH go_to_home_pos;
ISignalDI di1,1,intno1;

TRAP go_to_home_pos ()
    VAR robtarget p10;

    StorePath;
    p10:=CRobT();
    MoveL home,v500,fine,tool1;
    WaitDI di1,0;
    Move L p10,v500,fine,tool1;
    RestoPath;
    StartMove;
ENDTRAP
```

Similar to the previous example, but the robot does not move to the *home* position until the current movement instruction is finished.

Syntax

StopMove';'

Related information

	Described in:
Continuing a movement	Instructions - *StartMove*
Interrupts	RAPID Summary - *Interrupts* Basic Characteristics- *Interrupts*

StorePath — Stores the path when an interrupt occurs

StorePath is used to store the movement path being executed when an error or interrupt occurs. The error handler or trap routine can then start a new movement and, following this, restart the movement that was stored earlier.

This instruction can be used to go to a service position or to clean the gun, for example, when an error occurs.

Example

```
StorePath;
```

The current movement path is stored for later use.

Program execution

The current movement path of the robot and external axes is saved. After this, another movement can be started in a trap routine or an error handler. When the reason for the error or interrupt has been rectified, the saved movement path can be restarted.

Example

```
TRAP machine_ready
    VAR robtarget p1;
    StorePath;
    p1 := CRobT();
    MoveL p100, v100, fine, tool1;
    ...
    MoveL p1, v100, fine, tool1;
    RestoPath;
    StartMove;
ENDTRAP
```

When an interrupt occurs that activates the trap routine *machine_ready*, the movement path which the robot is executing at the time is stopped at the end of the instruction (ToPoint) and stored. After this, the robot remedies the interrupt by, for example, replacing a part in the machine and the normal movement is restarted.

Limitations

Only the movement path data is stored with the instruction *StorePath*.
If the user wants to order movements on the new path level, the actual stop position must be stored directly after *StorePath* and before *RestoPath* make a movement to the stored stop position on the path.

Only one movement path can be stored at a time.

Syntax

StorePath‘;’

Related information

	Described in:
Restoring a path	Instructions - *RestoPath*
More examples	Instructions - *RestoPath*

TEST Depending on the value of an expression ...

TEST is used when different instructions are to be executed depending on the value of an expression or data.

If there are not too many alternatives, the *IF..ELSE* instruction can also be used.

Example

```
TEST reg1
CASE 1,2,3 :
    routine1;
CASE 4 :
    routine2;
DEFAULT :
    TPWrite "Illegal choice";
    Stop;
ENDTEST
```

Different instructions are executed depending on the value of *reg1*. If the value is 1-3 *routine1* is executed. If the value is 4, *routine2* is executed. Otherwise, an error message is printed and execution stops.

Arguments

TEST Test data {CASE Test value {, Test value} : ...}
[DEFAULT: ...] ENDTEST

Test data Data type: All

The data or expression with which the test value will be compared.

Test value Data type: Same as test data

The value which the test data must have for the associated instructions to be executed.

Program execution

The test data is compared with the test values in the first CASE condition. If the comparison is true, the associated instructions are executed. After that, program execution continues with the instruction following ENDTEST.

If the first CASE condition is not satisfied, other CASE conditions are tested, and so on. If none of the conditions are satisfied, the instructions associated with DEFAULT are executed (if this is present).

Syntax

(EBNF)
TEST <expression>
{(**CASE** <test value> { ',' <test value> } ':'
<instruction list>) | **<CSE>** }
[**DEFAULT** ':' <instruction list>]
ENDTEST

<test value> ::= <expression>

Related information

	Described in:
Expressions	Basic Characteristics - *Expressions*

TPErase Erases text printed on the teach pendant

TPErase (Teach Pendant Erase) is used to clear the display of the teach pendant.

Example

```
TPErase;
TPWrite "Execution started";
```

The teach pendant display is cleared before *Execution started* is written.

Program execution

The teach pendant display is completely cleared of all text. The next time text is written, it will be entered on the uppermost line of the display.

Syntax

```
TPErase;
```

Related information

	Described in:
Writing on the teach pendant	RAPID Summary - *Communication*

TPReadFK Reads function keys

TPReadFK (Teach Pendant Read Function Key) is used to write text above the functions keys and to find out which key is depressed.

Example

TPReadFK reg1, “More ?”, stEmpty, stEmpty, stEmpty, “Yes”, “No”;

The text *More ?* is written on the teach pendant display and the function keys 4 and 5 are activated by means of the text strings *Yes* and *No* respectively (see Figure 22). Program execution waits until one of the function keys 4 or 5 is pressed. In other words, *reg1* will be assigned 4 or 5 depending on which of the keys is depressed.

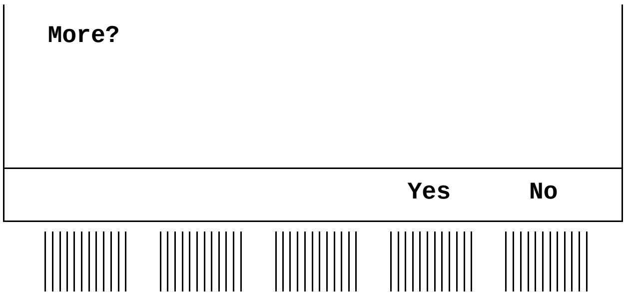


Figure 22 The operator can input information via the function keys.

Arguments

TPReadFK Answer Text FK1 FK2 FK3 FK4 FK5 [\MaxTime] [\DIBreak] [\BreakFlag]

Answer Data type: *num*

The variable for which, depending on which key is pressed, the numeric value 1..5 is returned. If the function key 1 is pressed, 1 is returned, and so on.

Text Data type: *string*

The information text to be written on the display (a maximum of 80 characters).

FKx *(Function key text)* Data type: *string*

The text to be written as a prompt for the appropriate function key (a maximum of 7 characters). FK1 is the left-most key.

Function keys without prompts are specified by the predefined string constant *stEmpty* with value empty string (“”).

[\MaxTime] Data type: *num*

The maximum amount of time [s] that program execution waits. If no function key is depressed within this time, the program continues to execute in the error handler unless the BreakFlag is used (see below). The constant ERR_TP_MAXTIME can be used to test whether or not the maximum time has elapsed.

[\DIBreak] *(Digital Input Break)* Data type: *signaldi*

The digital signal that may interrupt the operator dialog. If no function key is depressed when the signal is set to 1 (or is already 1), the program continues to execute in the error handler, unless the BreakFlag is used (see below). The constant ERR_TP_DIBREAK can be used to test whether or not this has occurred.

[\BreakFlag] Data type: *errnum*

A variable that will hold the error code if maxtime or dibreak is used. If this optional variable is omitted, the error handler will be executed. The constants ERR_TP_MAXTIME and ERR_TP_ DIBREAK can be used to select the reason.

Program execution

The information text is always written on a new line. If the display is full of text, this body of text is moved up one line first. Strings longer than the width of the teach pendant (40 characters) are split into two lines.

Prompts are written above the appropriate function keys. Keys without prompts are deactivated.

Program execution waits until one of the activated function keys is depressed.

Description of concurrent *TPReadFK* or *TPReadNum* request on Teach Pendant (TP request) from same or other program tasks:

- New TP request from other program task will not take focus (new put in queue)
- New TP request from TRAP in the same program task will take focus (old put in queue)
- Program stop take focus (old put in queue)
- New TP request in program stop state takes focus (old put in queue)

Example

```
VAR errnum errvar;
...
TPReadFK reg1, “Go to service position?”, stEmpty, stEmpty, stEmpty, “Yes”, “No”
\MaxTime:= 600
    \DIBreak:= di5\BreakFlag:= errvar;
IF reg1 = 4 or OR errvar = ERR_TP_DIBREAK THEN
    MoveL service, v500, fine, tool1;
```

```
    Stop;
ENDIF
IF errvar = ERR_TP_MAXTIME EXIT;
```

The robot is moved to the service position if the forth function key (“Yes”) is pressed, or if the input 5 is activated. If no answer is given within 10 minutes, the execution is terminated.

Predefined data

CONST string stEmpty := “”;

The predefined constant *stEmpty* should be used for Function Keys without prompts. Using *stEmpty* instead of “”saves about 80 bytes for every Function Key without prompts.

Syntax

TPReadFK
    [Answer’:=’] <var or pers (**INOUT**) of *num*>’,’
    [Text’:=’] <expression (**IN**) of *string*>’,’
    [FK1 ’:=’] <expression (**IN**) of *string*>’,’
    [FK2 ’:=’] <expression (**IN**) of *string*>’,’
    [FK3 ’:=’] <expression (**IN**) of *string*>’,’
    [FK4 ’:=’] <expression (**IN**) of *string*>’,’
    [FK5 ’:=’] <expression (**IN**) of *string*>
    [’\’MaxTime ’:=’ <expression (**IN**) of *num*>]
    [’\’DIBreak ’:=’ <variable (**VAR**) of *signaldi*>]
    [’\’BreakFlag ’:=’ <var or pers (**INOUT**) of *errnum*>]’;’

Related information

	Described in:
Writing to and reading from the teach pendant	RAPID Summary - *Communication*
Replying via the teach pendant	Running Production

TPReadNum Reads a number from the teach pendant

TPReadNum (Teach Pendant Read Numerical) is used to read a number from the teach pendant.

Example

TPReadNum reg1, “How many units should be produced?“;

The text *How many units should be produced?* is written on the teach pendant display. Program execution waits until a number has been input from the numeric keyboard on the teach pendant. That number is stored in *reg1*.

Arguments

TPReadNum Answer String [\MaxTime] [\DIBreak] [\BreakFlag]

Answer Data type: *num*

The variable for which the number input via the teach pendant is returned.

String Data type: *string*

The information text to be written on the teach pendant (a maximum of 80 characters).

[\MaxTime] Data type: *num*

The maximum amount of time that program execution waits. If no number is input within this time, the program continues to execute in the error handler unless the BreakFlag is used (see below). The constant ERR_TP_MAXTIME can be used to test whether or not the maximum time has elapsed.

[\DIBreak] *(Digital Input Break)* Data type: *signaldi*

The digital signal that may interrupt the operator dialog. If no number is input when the signal is set to 1 (or is already 1), the program continues to execute in the error handler unless the BreakFlag is used (see below). The constant ERR_TP_DIBREAK can be used to test whether or not this has occurred.

[\BreakFlag] Data type: *errnum*

A variable that will hold the error code if maxtime or dibreak is used. If this optional variable is omitted, the error handler will be executed.The constants ERR_TP_MAXTIME and ERR_TP_ DIBREAK can be used to select the reason.

Program execution

The information text is always written on a new line. If the display is full of text, this body of text is moved up one line first. Strings longer than the width of the teach pendant (40 characters) are split into two lines.

Program execution waits until a number is typed on the numeric keyboard (followed by Enter or ***OK***).

Reference to *TPReadFK* about description of concurrent *TPReadFK* or *TPReadNum* request on Teach Pendant from same or other program tasks.

Example

```
TPReadNum reg1, “How many units should be produced?“;
FOR i FROM 1 TO reg1 DO
    produce_part;
ENDFOR
```

The text *How many units should be produced?* is written on the teach pendant display. The routine *produce_part* is then repeated the number of times that is input via the teach pendant.

Syntax

TPReadNum
    [Answer’:=’] <var or pers (**INOUT**) of *num*>’,’
    [String’:=’] <expression (**IN**) of *string*>
    [’\’MaxTime ’:=’ <expression (**IN**) of *num*>]
    [’\’DIBreak ’:=’ <variable (**VAR**) of *signaldi*>]
    [’\’BreakFlag ’:=’ <var or pers (**INOUT**) of *errnum*>] ’;’

Related information

	Described in:
Writing to and reading from the teach pendant	RAPID Summary - *Communication*
Entering a number on the teach pendant	Production Running
Examples of how to use the arguments MaxTime, DIBreak and BreakFlag	Instructions - *TPReadFK*

TPShow Switch window on the teach pendant

TPShow (Teach Pendant Show) is used to select Teach Pendant Window from RAPID.

Examples

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.

TPShow TP_LATEST;

The latest used Teach Pendant Window used before the *Operator Input&Output Window* will be active after execution of this instruction.

Arguments

TPShow Window

Window Data type: *tpnum*

The window to show:

TP_PROGRAM	= *Production Window* if in *AUTO* mode. *Program Window* if in *MAN* mode.
TP_LATEST	= Latest used Teach Pendant Window before *Operator Input&Output Window*

Predefined data

```
CONST tpnum TP_PROGRAM := 1;
CONST tpnum TP_LATEST := 2;
```

Program execution

The selected Teach Pendant Window will be activated.

Syntax

TPShow
[Window’:=’] <expression (**IN**) of *tpnum*> ‘;’

Related information

	Described in:
Communicating using the teach pendant	RAPID Summary - *Communication*
Teach Pendant Window number	Data Types - *tpnum*

-

TPWrite Writes on the teach pendant

TPWrite (Teach Pendant Write) is used to write text on the teach pendant. The value of certain data can be written as well as text.

Examples

TPWrite "Execution started";

The text *Execution started* is written on the teach pendant.

TPWrite "No of produced parts="\Num:=reg1;

If, for example, the answer to *No of produced parts=5*, enter 5 instead of *reg1* on the teach pendant.

Arguments

TPWrite String [\Num] | [\Bool] | [\Pos] | [\Orient]

String Data type: *string*

The text string to be written (a maximum of 80 characters).

[\Num] *(Numeric)* Data type: *num*

The data whose numeric value is to be written after the text string.

[\Bool] *(Boolean)* Data type: *bool*

The data whose logical value is to be written after the text string.

[\Pos] *(Position)* Data type: *pos*

The data whose position is to be written after the text string.

[\Orient] *(Orientation)* Data type: *orient*

The data whose orientation is to be written after the text string.

Program execution

Text written on the teach pendant always begins on a new line. When the display is full of text, this text is moved up one line first. Strings that are longer than the width of the teach pendant (40 characters) are divided up into two lines.

If one of the arguments *\Num*, *\Bool*, *\Pos* or *\Orient* is used, its value is first converted to a text string before it is added to the first string. The conversion from value to text string takes place as follows:

Argument	Value	Text string
\Num	23	"23"
\Num	1.141367	"1.14137"
\Bool	TRUE	"TRUE"
\Pos	[1817.3,905.17,879.11]	"[1817.3,905.17,879.11]"
\Orient	[0.96593,0,0.25882,0]	"[0.96593,0,0.25882,0]"

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.

Limitations

The arguments *\Num*, *\Bool*, *\Pos* and *\Orient* are mutually exclusive and thus cannot be used simultaneously in the same instruction.

Syntax

TPWrite
[String':='] <expression (**IN**) of *string*>
['\'Num':=' <expression (**IN**) of *num*>]
| ['\'Bool':=' <expression (**IN**) of *bool*>]
| ['\'Pos':=' <expression (**IN**) of *pos*>]
| ['\'Orient':=' <expression (**IN**) of *orient*>]';'

Related information

	Described in:
Clearing and reading the teach pendant	RAPID Summary - *Communication*

-

TriggC Circular robot movement with events

TriggC (Trigg Circular) is used to set output signals and/or run interrupt routines at fixed positions, at the same time as the robot is moving on a circular path.

One or more (max. 4) events can be defined using the instructions *TriggIO*, *TriggEquip* or *TriggInt* and afterwards these definitions are referred to in the instruction *TriggC*.

Examples

```
VAR triggdata gunon;

TriggIO gunon, 0 \Start \DOp:=gun, on;

MoveL p1, v500, z50, gun1;
TriggC p2, p3, v500, gunon, fine, gun1;
```

The digital output signal *gun* is set when the robot's TCP passes the midpoint of the corner path of the point *p1*.

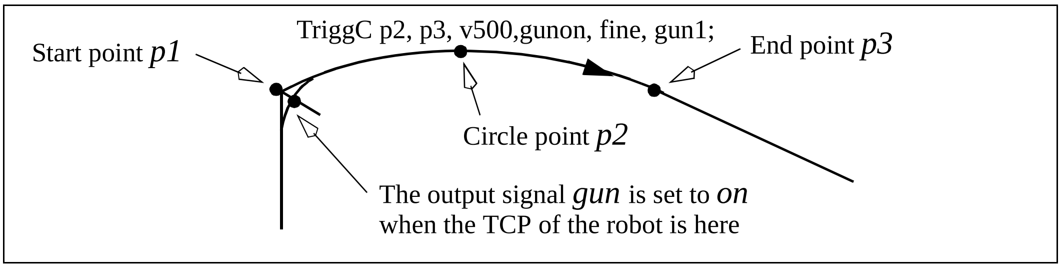


Figure 23 Example of fixed-position IO event.

Arguments

TriggC [\Conc] CirPoint ToPoint Speed [\T]
Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj] [\Corr]

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed at once. This argument is used to shorten the cycle time when, for example, communicating with external equipment, and synchronisation is not required. It can also be used
to tune the execution of the robot path, to avoid warning 50024 Corner path failure, or error 40082 Deceleration limit.

When using the argument \Conc, the number of movement instructions in succession is limited to 5. In a program section that includes StorePath-RestoPath, movement instructions with the argument \Conc are not permitted.

If this argument is omitted and the ToPoint is not a stop point, the subsequent instruction is executed some time before the robot has reached the programmed zone.

CirPoint Data type: *robtarget*

The circle point of the robot. See the instruction *MoveC* for a more detailed description of circular movement. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the external axes and of the tool reorientation.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Trigg_1 Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T2] (*Trigg 2*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T3] (*Trigg 3*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T4] (*Trigg 4)* Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination position.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

[\Corr] *(Correction)* Data type: *switch*

Correction data written to a corrections entry by the instruction *CorrWrite* will be added to the path and destination position, if this argument is present.

Program execution

See the instruction *MoveC* for information about circular movement.

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 0.1 \Time, intno1;
...
TriggC p1, p2, v500, trigg1, fine, gun1;
TriggC p3, p4, v500, trigg1, fine, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the work point is at a position *0.1* s before the point *p2* or *p4* respectively.

Limitations

If the current start point deviates from the usual, so that the total positioning length of the instruction *TriggC* is shorter than usual, it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried out will be undefined. The program logic in the user program may not be based on a normal sequence of trigger activities for an "incomplete movement".

The instruction *TriggC* should never be started from the beginning with the robot in position after the circle point. Otherwise the robot will not take the programmed path (positioning around the circular path in another direction compared with that programmed).

Syntax

TriggC
  ['\' Conc ',']
  [CirPoint ':='] < expression (**IN**) of *robtarget* > ','
  [ToPoint ':='] < expression (**IN**) of *robtarget* > ','
  [Speed ':='] < expression (**IN**) of *speeddata* >
    ['\' T ':=' < expression (**IN**) of *num* >] ','
  [Trigg_1 ':='] < variable (**VAR**) of *triggdata* >
  ['\' T2 ':=' < variable (**VAR**) of *triggdata* >]
  ['\' T3 ':=' < variable (**VAR**) of *triggdata* >]
  ['\' T4 ':=' < variable (**VAR**) of *triggdata* >] ','
  [Zone ':='] < expression (**IN**) of *zonedata* > ','
  [Tool ':='] < persistent (**PERS**) of *tooldata* >
  ['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >]
  ['\' Corr]';'

Related information

	Described in:
Linear movement with triggers	Instructions - *TriggL*
Joint movement with triggers	Instructions - *TriggJ*
Definition of triggers	Instructions - *TriggIO*, *TriggEquip* *TriggInt*
Writes to a corrections entry	Instructions - *CorrWrite*
Circular movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion Principles

TriggEquip Defines a fixed position-time I/O event

TriggEquip (Trigg Equipment) is used to define conditions and actions for setting a digital, a group of digital, or an analog output signal at a fixed position along the robot's movement path with possibility to do time compensation for the lag in the external equipment.

The data defined is used for implementation in one or more subsequent *TriggL*, *TriggC* or *TriggJ* instructions.

Examples

VAR triggdata gunon;

TriggEquip gunon, 10, 0.1 \DOp:=gun, 1;

TriggL p1, v500, gunon, z50, gun1;

The tool *gun1* opens in point p2, when the TCP is *10* mm before the point *p1*. To reach this, the digital output signal *gun* is set to the value *1*, when TCP is *0.1* s before the point p2. The gun is full open when TCP reach point p2.

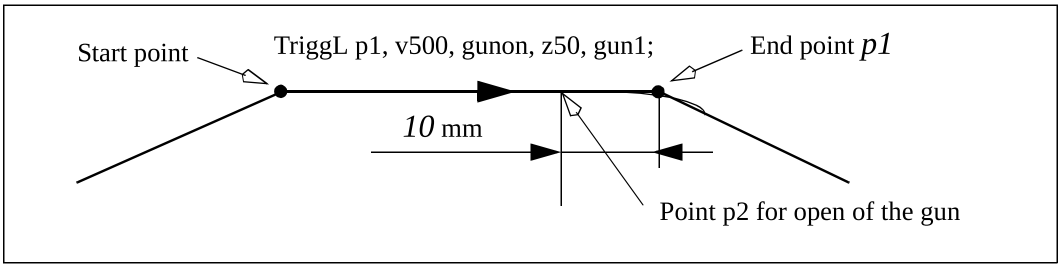


Figure 24 Example of fixed position-time I/O event.

Arguments

TriggEquip TriggData Distance [\Start] EquipLag [\DOp] | [\GOp] | [\AOp] | [\ProcID] SetValue [\Inhib]

TriggData Data type: *triggdata*

Variable for storing the *triggdata* returned from this instruction. These *triggdata* are then used in the subsequent *TriggL, TriggC* or *TriggJ* instructions.

Distance Data type: *num*

Defines the position on the path where the I/O equipment event shall occur.

Specified as the distance in mm (positive value) from the end point of the movement path (applicable if the argument \ *Start* is not set).

See the section entitled Program execution for further details.

[\Start] Data type: *switch*

Used when the distance for the argument *Distance* starts at the movement start point instead of the end point.

EquipLag (*Equipment Lag*) Data type: *num*

Specify the lag for the external equipment in s.

For compensation of external equipment lag, use positive argument value. Positive argument value means that the I/O signal is set by the robot system at specified time before the TCP physical reach the specified distance in relation to the movement start or end point.

Negative argument value means that the I/O signal is set by the robot system at specified time after that the TCP physical has passed the specified distance in relation to the movement start or end point.

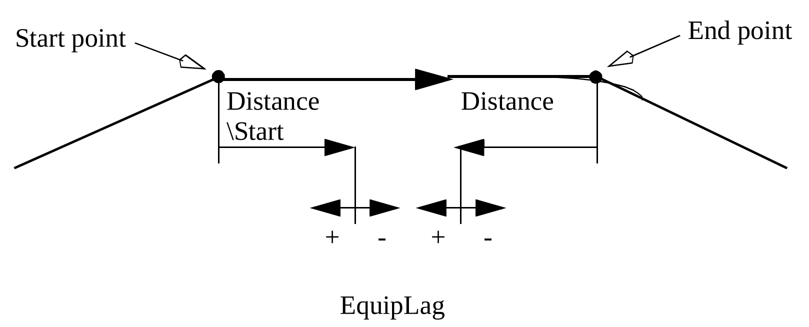


Figure 25 Use of argument EquipLag.

[\DOp] (*Digital OutPut*) Data type: *signaldo*

The name of the signal, when a digital output signal shall be changed.

[\GOp] (*Group OutPut*) Data type: *signalgo*

The name of the signal, when a group of digital output signals shall be changed.

[\AOp] (*Analog Output*) Data type: *signalao*

The name of the signal, when a analog output signal shall be changed.

[\ProcID] (*Process Identity*) Data type: *num*

Not implemented for customer use.

(The identity of the IPM process to receive the event. The selector is specified in the argument *SetValue*.)

SetValue Data type: *num*

Desired value of output signal (within the allowed range for the current signal).

[\Inhib] (*Inhibit*) Data type: *bool*

The name of a persistent variable flag for inhibit the setting of the signal at runtime.

If this optional argument is used and the actual value of the specified flag is TRUE at the position-time for setting of the signal then the specified signal (*DOp*, *GOp* or *AOp*) will be set to 0 in stead of specified value.

Program execution

When running the instruction *TriggEquip*, the trigger condition is stored in the specified variable for the argument *TriggData*.

Afterwards, when one of the instructions *TriggL*, *TriggC* or *TriggJ* is executed, the following are applicable, with regard to the definitions in *TriggEquip*:

<u>The distance specified in the argument *Distance*:</u>

Linear movement	The straight line distance
Circular movement	The circle arc length
Non-linear movement	The approximate arc length along the path (to obtain adequate accuracy, the distance should not exceed one half of the arc length).

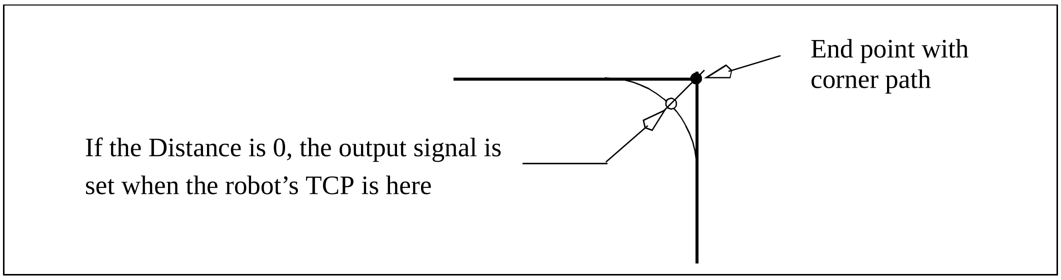


Figure 26 Fixed position-time I/O on a corner path.

The position-time related event will be generated when the start point (end point) is passed, if the specified distance from the end point (start point) is not within the length of movement of the current instruction (*Trigg*...). With use of argument *EquipLag* with negative time (delay), the I/O signal can be set after the end point.

Examples

VAR triggdata glueflow;

```
TriggEquip glueflow, 1 \Start, 0.05 \AOp:=glue, 5.3;

MoveJ p1, v1000, z50, tool1;
TriggL p2, v500, glueflow, z50, tool1;
```

The analog output signal *glue* is set to the value *5.3* when the TCP passes a point located *1* mm after the start point *p1* with compensation for equipment lag *0.05* s.

```
...
TriggL p3, v500, glueflow, z50, tool1;
```

The analog output signal *glue* is set once more to the value *5.3* when the TCP passes a point located *1* mm after the start point *p2*.

Limitations

I/O events with distance (without the argument *Time*) is intended for flying points (corner path). I/O events with distance, using stop points, results in worse accuracy than specified below.

Regarding the accuracy for I/O events with distance and using flying points, the following is applicable when setting a digital output at a specified distance from the start point or end point in the instruction *TriggL* or *TriggC*:

- Accuracy specified below is valid for positive *EquipLag* parameter < 60 ms, equivalent to the lag in the robot servo (without changing the system parameter *Event Preset Time*).
- Accuracy specified below is valid for positive *EquipLag* parameter < configured *Event Preset Time* (system parameter).
- Accuracy specified below is not valid for positive *EquipLag* parameter > configured *Event Preset Time* (system parameter). In this case, an approximate method is used in which the dynamic limitations of the robot are not taken into consideration. *SingArea \Wrist* must be used in order to achieve an acceptable accuracy.
- Accuracy specified below is valid for negative *EquipLag*.

I/O events with time (with the argument *Time*) is intended for stop points. I/O events with time, using flying points, results in worse accuracy than specified below.
I/O events with time can only be specified from the end point of the movement. This time cannot exceed the current braking time of the robot, which is max. approx. 0.5 s (typical values at speed 500 mm/s for IRB2400 150 ms and for IRB6400 250 ms). If the specified time is greater that the current braking time, the event will be generated anyhow, but not until braking is started (later than specified). However, the whole of the movement time for the current movement can be utilised during small and fast movements.

Typical absolute accuracy values for set of digital outputs +/- 5 ms.
Typical repeat accuracy values for set of digital outputs +/- 2 ms.

Used digital output signals (*DOp* or *GOp*) cannot be cross connected to other signals.

Syntax

TriggEquip
[TriggData ':='] < variable (**VAR**) of *triggdata*> ','
[Distance ':='] < expression (**IN**) of *num*>
['\' Start] ','
[EquipLag ':='] < expression (**IN**) of *num*>
['\' DOp ':=' < variable (**VAR**) of *signaldo*>]
| ['\' GOp ':=' < variable (**VAR**) of *signalgo*>]
| ['\' AOp ':=' < variable (**VAR**) of *signalao*>]
| ['\' ProcID ':=' < expression (**IN**) of *num*>] ','
[SetValue ':='] < expression (**IN**) of *num*>
['\' Inhibit ':=' < persistent (**PERS**) of *bool*>] ','

Related information

	Described in:
Use of triggers	Instructions - *TriggL, TriggC, TriggJ*
Definition of other triggs	Instruction - *TriggIO, TriggInt*
More examples	Data Types - *triggdata*
Set of I/O	Instructions - *SetDO, SetGO, SetAO*
Configuration of Event preset time	User's guide System Parameters - *Manipulator*

TriggInt Defines a position related interrupt

TriggInt is used to define conditions and actions for running an interrupt routine at a position on the robot's movement path.

The data defined is used for implementation in one or more subsequent *TriggL*, *TriggC* or *TriggJ* instructions.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 5, intno1;
...
TriggL p1, v500, trigg1, z50, gun1;
TriggL p2, v500, trigg1, z50, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the TCP is at a position *5* mm before the point *p1* or *p2* respectively.

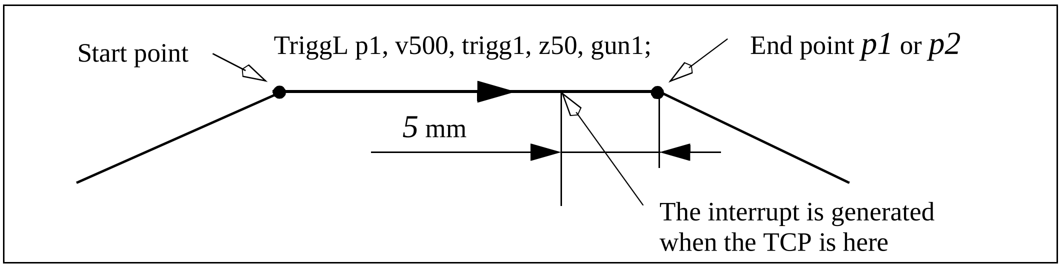


Figure 27 Example position related interrupt.

Arguments

TriggInt TriggData Distance [\Start] | [\Time]
Interrupt

TriggData Data type: *triggdata*

Variable for storing the *triggdata* returned from this instruction. These *triggdata* are then used in the subsequent *TriggL*, *TriggC* or *TriggJ* instructions.

Distance Data type: *num*

Defines the position on the path where the interrupt shall be generated.

Specified as the distance in mm (positive value) from the end point of the movement path (applicable if the argument \ *Start* or *Time* is not set).

See the section entitled Program execution for further details.

[\Start] Data type: *switch*

Used when the distance for the argument *Distance* starts at the movement start point instead of the end point.

[\Time] Data type: *switch*

Used when the value specified for the argument *Distance* is in fact a time in seconds (positive value) instead of a distance.

Position related interrupts in time can only be used for short times (< 0.5 s) before the robot reaches the end point of the instruction. See the section entitled Limitations for more details.

Interrupt Data type: *intnum*

Variable used to identify an interrupt.

Program execution

When running the instruction *TriggInt*, data is stored in a specified variable for the argument *TriggData* and the interrupt that is specified in the variable for the argument *Interrupt* is activated.

Afterwards, when one of the instructions *TriggL*, *TriggC* or *TriggJ* is executed, the following are applicable, with regard to the definitions in *TriggInt*:

The distance specified in the argument *Distance*:

Linear movement	The straight line distance
Circular movement	The circle arc length
Non-linear movement	The approximate arc length along the path (to obtain adequate accuracy, the distance should not exceed one half of the arc length).

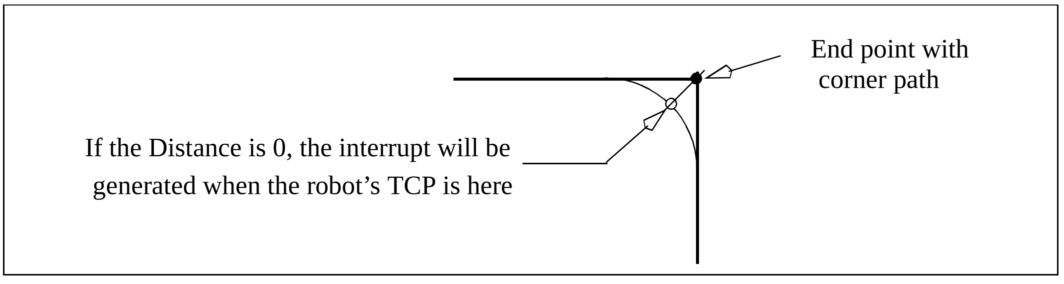


Figure 28 Position related interrupt on a corner path.

The position related interrupt will be generated when the start point (end point) is passed, if the specified distance from the end point (start point) is not within the length of movement of the current instruction (*Trigg...*).

Examples

This example describes programming of the instructions that interact to generate position related interrupts:

VAR intnum intno2;
VAR triggdata trigg2;

- Declaration of the variables *intno2* and *trigg2 (*shall not be initiated).

CONNECT intno2 WITH trap2;

- Allocation of interrupt numbers that are stored in the variable *intno2*
- The interrupt number is coupled to the interrupt routine *trap2*

TriggInt trigg2, 0, intno2;

- The interrupt number in the variable *intno2* is flagged as used
- The interrupt is activated
- Defined trigger conditions and interrupt number are stored in the variable *trigg2*

TriggL p1, v500, trigg2, z50, gun1;

- The robot is moved to the point *p1*.
- When the TCP reaches the point *p1*, an interrupt is generated and the interrupt routine *trap2* is run.

TriggL p2, v500, trigg2, z50, gun1;

- The robot is moved to the point *p2*
- When the TCP reaches the point *p2*, an interrupt is generated and the interrupt routine *trap2* is run once more.

IDelete intno2;

- The interrupt number in the variable *intno2 is* de-allocated.

Limitations

Interrupt events with distance (without the argument *Time*) is intended for flying points (corner path). Interrupt events with distance, using stop points, results in worse accuracy than specified below.

Interrupt events with time (with the argument *Time*) is intended for stop points. Interrupt events with time, using flying points, results in worse accuracy than specified below.
I/O events with time can only be specified from the end point of the movement. This time cannot exceed the current braking time of the robot, which is max. approx. 0.5 s (typical values at speed 500 mm/s for IRB2400 150 ms and for IRB6400 250 ms). If the specified time is greater that the current braking time, the event will be generated anyhow, but not until braking is started (later than specified). However, the whole of the movement time for the current movement can be utilised during small and fast movements.

Typical absolute accuracy values for generation of interrupts +/- 5 ms.
Typical repeat accuracy values for generation of interrupts +/- 2 ms.

Normally there is a delay of 5 to 120 ms between interrupt generation and response, depending on the type of movement being performed at the time of the interrupt. (Ref. to Basic Characteristics RAPID - *Interrupts*).

To obtain the best accuracy when setting an output at a fixed position along the robot's path, use the instructions *TriggIO* or *TriggEquip* in preference to the instructions *TriggInt* with *SetDO/SetGO/SetAO* in an interrupt routine.

Syntax

TriggInt
  [TriggData ':='] < variable (**VAR**) of *triggdata*> ','
  [Distance ':='] < expression (**IN**) of *num*>
  ['\' Start] | ['\' Time] ','
  [Interrupt ':='] < variable (**VAR**) of *intnum*> ';'

Related information

	Described in:
Use of triggers	Instructions - *TriggL, TriggC, TriggJ*
Definition of position fix I/O	Instruction - *TriggIO, TriggEquip*
More examples	Data Types - *triggdata*
Interrupts	Basic Characteristics - *Interrupts*

TriggIO Defines a fixed position I/O event

TriggIO is used to define conditions and actions for setting a digital, a group of digital, or an analog output signal at a fixed position along the robot's movement path.

To obtain a fixed position I/O event, *TriggIO* compensates for the lag in the control system (lag between robot and servo) but not for any lag in the external equipment. For compensation of both lags use *TriggEquip*.

The data defined is used for implementation in one or more subsequent *TriggL*, *TriggC* or *TriggJ* instructions.

Examples

```
VAR triggdata gunon;

TriggIO gunon, 10 \DOp:=gun, 1;

TriggL p1, v500, gunon, z50, gun1;
```

The digital output signal *gun* is set to the value *1* when the TCP is *10* mm before the point *p1*.

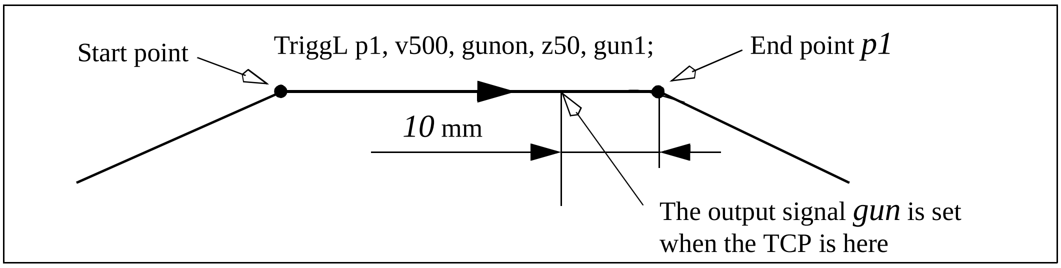


Figure 29 Example of fixed-position IO event.

Arguments

TriggIO TriggData Distance [\Start] | [\Time]
[\DOp] | [\GOp] | [\AOp] | [\ProcID] SetValue
[\DODelay]

TriggData Data type: *triggdata*

Variable for storing the *triggdata* returned from this instruction. These *triggdata* are then used in the subsequent *TriggL*, *TriggC* or *TriggJ* instructions.

Distance Data type: *num*

Defines the position on the path where the I/O event shall occur.

Specified as the distance in mm (positive value) from the end point of the movement path (applicable if the argument \ *Start* or *Time* is not set).

See the section entitled Program execution for further details.

[\Start] Data type: *switch*

Used when the distance for the argument *Distance* starts at the movement start point instead of the end point.

[\Time] Data type: *switch*

Used when the value specified for the argument *Distance* is in fact a time in seconds (positive value) instead of a distance.

Fixed position I/O in time can only be used for short times (< 0.5 s) before the robot reaches the end point of the instruction. See the section entitled Limitations for more details.

[\DOp] (*Digital OutPut*) Data type: *signaldo*

The name of the signal, when a digital output signal shall be changed.

[\GOp] (*Group OutPut*) Data type: *signalgo*

The name of the signal, when a group of digital output signals shall be changed.

[\AOp] (*Analog Output*) Data type: *signalao*

The name of the signal, when a analog output signal shall be changed.

[\ProcID] (*Process Identity*) Data type: *num*

Not implemented for customer use.

(The identity of the IPM process to receive the event. The selector is specified in the argument *SetValue*.)

SetValue Data type: *num*

Desired value of output signal (within the allowed range for the current signal).

[\DODelay] (*Digital Output Delay)* Data type: *num*

Time delay in seconds (positive value) for a digital output signal or group of digital output signals.

Only used to delay setting digital output signals, after the robot has reached the specified position. There will be no delay if the argument is omitted.

The delay is not synchronised with the movement.

Program execution

When running the instruction *TriggIO*, the trigger condition is stored in a specified variable for the argument *TriggData*.

Afterwards, when one of the instructions *TriggL*, *TriggC* or *TriggJ* is executed, the following are applicable, with regard to the definitions in *TriggIO*:

<u>The distance specified in the argument *Distance*:</u>

Linear movement	The straight line distance
Circular movement	The circle arc length
Non-linear movement	The approximate arc length along the path (to obtain adequate accuracy, the distance should not exceed one half of the arc length).

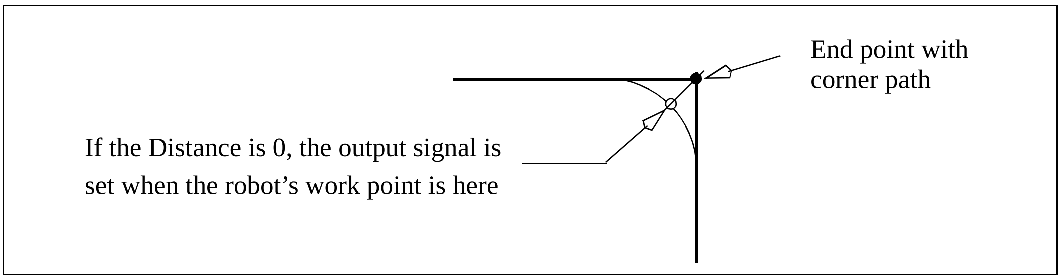


Figure 30 Fixed position I/O on a corner path.

The fixed position I/O will be generated when the start point (end point) is passed, if the specified distance from the end point (start point) is not within the length of movement of the current instruction (*Trigg...*).

Examples

```
VAR triggdata glueflow;

TriggIO glueflow, 1 \Start \AOp:=glue, 5.3;

MoveJ p1, v1000, z50, tool1;
TriggL p2, v500, glueflow, z50, tool1;
```

The analog output signal *glue* is set to the value *5.3* when the work point passes a point located *1* mm after the start point *p1*.

```
...
TriggL p3, v500, glueflow, z50, tool1;
```

The analog output signal *glue* is set once more to the value *5.3* when the work point passes a point located *1* mm after the start point *p2*.

Limitations

I/O events with distance (without the argument *Time*) is intended for flying points (corner path). I/O events with distance, using stop points, results in worse accuracy than specified below.

I/O events with time (with the argument *Time*) is intended for stop points. I/O events with time, using flying points, results in worse accuracy than specified below.
I/O events with time can only be specified from the end point of the movement. This time cannot exceed the current braking time of the robot, which is max. approx. 0.5 s (typical values at speed 500 mm/s for IRB2400 150 ms and for IRB6400 250 ms). If the specified time is greater that the current braking time, the event will be generated anyhow, but not until braking is started (later than specified). However, the whole of the movement time for the current movement can be utilised during small and fast movements.

Typical absolute accuracy values for set of digital outputs +/- 5 ms.
Typical repeat accuracy values for set of digital outputs +/- 2 ms.

Used digital output signals (*DOp* or *GOp*) cannot be cross connected to other signals.

Syntax

TriggIO
  [TriggData ’:=’] < variable (**VAR**) of *triggdata*> ‘,’
  [Distance ’:=’] < expression (**IN**) of *num*>
  [’\’ Start] | [’\’ Time]
  [’\’ DOp ’:=’ < variable (**VAR**) of *signaldo*>]
  | [’\’ GOp ’:=’ < variable (**VAR**) of *signalgo*>]
  | [’\’ AOp ’:=’ < variable (**VAR**) of *signalao*>]
  | [’\’ ProcID ’:=’ < expression (**IN**) of *num*>] ‘,’
  [SetValue ’:=’] < expression (**IN**) of *num*>
  [’\’ DODelay ’:=’ < expression (**IN**) of *num*>] ‘;’

Related information

	Described in:
Use of triggers	Instructions - *TriggL*, *TriggC*, *TriggJ*
Definition of position-time I/O event	Instruction - *TriggEquip*
Definition of position related interrupts	Instruction - *TriggInt*
More examples	Data Types - *triggdata*
Set of I/O	Instructions - *SetDO*, *SetGO*, *SetAO*

TriggJ Axis-wise robot movements with events

TriggJ (Trigg Joint) is used to set output signals and/or run interrupt routines at fixed positions, at the same time as the robot is moving on a circular path.

One or more (max. 4) events can be defined using the instructions *TriggIO*, *TriggEquip* or *TriggInt* and afterwards these definitions are referred to in the instruction *TriggJ*.

Examples

VAR triggdata gunon;

TriggIO gunon, 0 \Start \DOp:=gun, on;

MoveL p1, v500, z50, gun1;
TriggJ p2, v500, gunon, fine, gun1;

The digital output signal *gun* is set when the robot's TCP passes the midpoint of the corner path of the point *p1*.

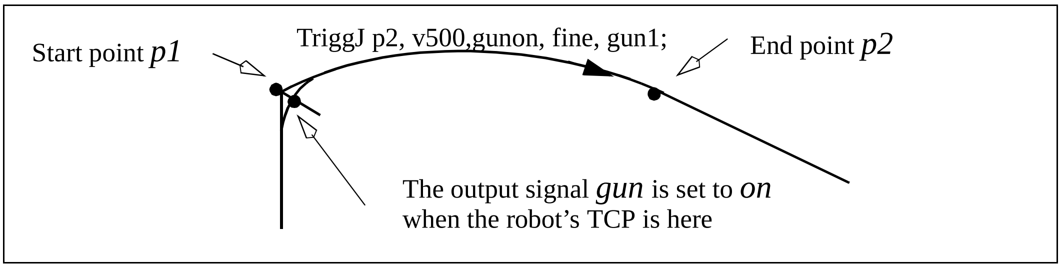


Figure 31 Example of fixed-position IO event.

Arguments

TriggJ [\Conc] ToPoint Speed [\T] Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj]

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed at once. This argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required. It can also be used
to tune the execution of the robot path, to avoid warning 50024 Corner path failure or error 40082 Deceleration limit.

Using the argument \Conc, the number of movement instructions in succession is limited to 5. In a program section that includes StorePath-RestoPath, movement instructions with the argument \Conc are not permitted.

If this argument is omitted and the ToPoint is not a stop point, the subsequent instruction is executed some time before the robot has reached the programmed zone .

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the external axes and of the tool reorientation.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Trigg_1 Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T2] (*Trigg 2*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T3] (*Trigg 3*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T4] (*Trigg 4)* Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination position.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

Program execution

See the instruction *MoveJ* for information about joint movement.

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 0.1 \Time , intno1;
...
TriggJ p1, v500, trigg1, fine, gun1;
TriggJ p2, v500, trigg1, fine, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the work point is at a position *0.1* s before the point *p1* or *p2* respectively.

Limitations

If the current start point deviates from the usual, so that the total positioning length of the instruction *TriggJ* is shorter than usual (e.g. at the start of *TriggJ* with the robot position at the end point), it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried will be undefined. The program logic in the user program may not be based on a normal sequences of trigger activities for an ”incomplete movement”.

Syntax

TriggJ
['\' Conc ',']
[ToPoint ':='] < expression (**IN**) of *robtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' T ':=' < expression (**IN**) of *num* >] ','
[Trigg_1 ':='] < variable (**VAR**) of *triggdata* >
['\' T2 ':=' < variable (**VAR**) of *triggdata* >]
['\' T3 ':=' < variable (**VAR**) of *triggdata* >]
['\' T4 ':=' < variable (**VAR**) of *triggdata* >] ','
[Zone ':='] < expression (**IN**) of *zonedata* > ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ';'

Related information

	Described in:
Linear movement with triggs	Instructions - *TriggL*
Circular movement with triggers	Instructions - *TriggC*
Definition of triggers	Instructions - *TriggIO*, *TriggEquip* or *TriggInt*
Joint movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion Principles

TriggL Linear robot movements with events

TriggL (Trigg Linear) is used to set output signals and/or run interrupt routines at fixed positions, at the same time as the robot is making a linear movement.

One or more (max. 4) events can be defined using the instructions *TriggIO*, *TriggEquip* or *TriggInt* and afterwards these definitions are referred to in the instruction *TriggL*.

Examples

```
VAR triggdata gunon;

TriggIO gunon, 0 \Start \DOp:=gun, on;

MoveJ p1, v500, z50, gun1;
TriggL p2, v500, gunon, fine, gun1;
```

The digital output signal *gun* is set when the robot's TCP passes the midpoint of the corner path of the point *p1*.

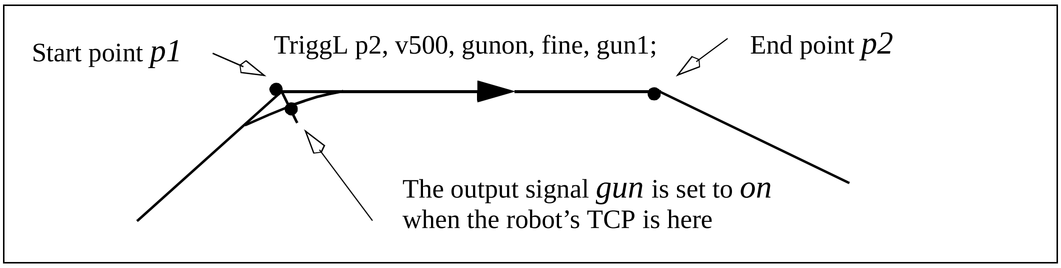


Figure 32 Example of fixed-position IO event.

Arguments

TriggL [\Conc] ToPoint Speed [\T] Trigg_1 [\T2] [\T3] [\T4]
Zone Tool [\WObj] [\Corr]

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed at once. This argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required. It can also be used
to tune the execution of the robot path, to avoid warning 50024 Corner path failure or error 40082 Deceleration limit.

Using the argument \Conc, the number of movement instructions in succession is limited to 5. In a program section that includes StorePath-RestoPath, movement instructions with the argument \Conc are not permitted.

If this argument is omitted and the ToPoint is not a stop point, the subsequent instruction is executed some time before the robot has reached the programmed zone.

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity of the tool centre point, the external axes and of the tool reorientation.

[\T] *(Time)* Data type: *num*

This argument is used to specify the total time in seconds during which the robot moves. It is then substituted for the corresponding speed data.

Trigg_1 Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T2] (*Trigg 2*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T3] (*Trigg 3*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T4] (*Trigg 4)* Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point that is moved to the specified destination position.

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

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

This argument can be omitted, and if it is, the position is related to the world coordinate system. If, on the other hand, a stationary TCP or coordinated external axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

[\Corr] *(Correction)* Data type: *switch*

Correction data written to a corrections entry by the instruction *CorrWrite* will be added to the path and destination position, if this argument is present.

Program execution

See the instruction *MoveL* for information about linear movement.

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 0.1 \Time, intno1;
...
TriggL p1, v500, trigg1, fine, gun1;
TriggL p2, v500, trigg1, fine, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the work point is at a position *0.1* s before the point *p1* or *p2* respectively.

Limitations

If the current start point deviates from the usual, so that the total positioning length of the instruction *TriggL* is shorter than usual (e.g. at the start of *TriggL* with the robot position at the end point), it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried out will be undefined. The program logic in the user program may not be based on a normal sequence of trigger activities for an “incomplete movement”.

Syntax

TriggL
['\' Conc ',']
[ToPoint ':='] < expression (**IN**) of *robtarget* > ','
[Speed ':='] < expression (**IN**) of *speeddata* >
['\' T ':=' < expression (**IN**) of *num* >] ','
[Trigg_1 ':='] < variable (**VAR**) of *triggdata* >
['\' T2 ':=' < variable (**VAR**) of *triggdata* >]
['\' T3 ':=' < variable (**VAR**) of *triggdata* >]
['\' T4 ':=' < variable (**VAR**) of *triggdata* >] ','
[Zone ':='] < expression (**IN**) of *zonedata* > ','
[Tool ':='] < persistent (**PERS**) of *tooldata* >
['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >]
['\' Corr]';'

Related information

	Described in:
Circular movement with triggers	Instructions - *TriggC*
Joint movement with triggers	Instructions - *TriggJ*
Definition of triggers	Instructions - *TriggIO*, *TriggEquip* or *TriggInt*
Writes to a corrections entry	Instructions - *CorrWrite*
Linear movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion Principles

TRYNEXT — Jumps over an instruction which has caused an error

TRYNEXT is used to jump over an instruction which has caused an error. Instead, the next instruction is run.

Example

```
reg2 := reg3/reg4;
    .
ERROR
    IF ERRNO = ERR_DIVZERO THEN
        reg2:=0;
        TRYNEXT;
    ENDIF
```

An attempt is made to divide *reg3* by *reg4*. If reg4 is equal to 0 (division by zero), a jump is made to the error handler, where *reg2* is set to 0. The *TRYNEXT* instruction is then used to continue with the next instruction.

Program execution

Program execution continues with the instruction subsequent to the instruction that caused the error.

Limitations

The instruction can only exist in a routine's error handler.

Syntax

TRYNEXT';'

Related information

	Described in:
Error handlers	Basic Characteristics- *Error Recovery*

TuneReset Resetting servo tuning

TuneReset is used to reset the dynamic behaviour of all robot axes and external mechanical units to their normal values.

Example

TuneReset;

Resetting tuning values for all axes to 100%.

Program execution

The tuning values for all axes are reset to 100%.

The default servo tuning values for all axes are automatically set by executing instruction *TuneReset*

- at a cold start-up
- when a new program is loaded
- when starting program execution from the beginning.

Syntax

TuneReset ';'

Related information

	Described in:
Tuning servos	Instructions - *TuneServo*

UnLoad UnLoad a program module during execution

UnLoad is used to unload a program module from the program memory during execution.

The program module must previously have been loaded into the program memory using the instruction *Load* or *StartLoad* - *WaitLoad*.

Example

```
UnLoad ram1disk \File:="PART_A.MOD";
```

UnLoad the program module PART_A.MOD from the program memory, that previously was loaded into the program memory with *Load*. (See instructions *Load*). (*ram1disk* is a predefined string constant "ram1disk:").

Arguments

UnLoad [\Save] FilePath [\File]

[\Save]

Data type: *switch*

If this argument is used, the program module is saved before the unloading starts. The program module will be saved at original place specified in *Load* or *StartLoad* instruction.

FilePath

Data type: *string*

The file path and the file name to the file that will be unloaded from the program memory. The file path and the file name must be the same as in the previously executed *Load* or *StartLoad* instruction. The file name shall be excluded when the argument *\File* is used.

[\File]

Data type: *string*

When the file name is excluded in the argument *FilePath*, then it must be defined with this argument. The file name must be the same as in the previously executed *Load* or *StartLoad* instruction.

Program execution

To be able to execute a *UnLoad* instruction in the program, a *Load* or *StartLoad - WaitLoad* instruction with the same file path and name must have been executed earlier in the program.

The program execution waits for the program module to be finish unloading before the execution proceeds with the next instruction.

After that the program module is unloaded, the rest of the program modules will be linked.

For more information see the instructions *Load* or *StartLoad-Waitload*.

Examples

UnLoad "ram1disk:DOORDIR/DOOR1.MOD";

UnLoad the program module DOOR1.MOD from the program memory, that previously was loaded into the program memory with *Load*. (See instructions *Load*).

UnLoad "ram1disk:DOORDIR/" \File:="DOOR1.MOD";

Same as above but another syntax.

Unload \Save, "ram1disk:DOORDIR/" \File:="DOOR1.MOD";

Same as above but save the program module before unloading.

Limitations

It is not allowed to unload a program module that is executing.

TRAP routines, system I/O events and other program tasks cannot execute during the unloading.

Avoid ongoing robot movements during the unloading.

Program stop during execution of *UnLoad* instruction results in guard stop with motors off and error message "20025 Stop order timeout" on the Teach Pendant.

Error handling

If the file in the *UnLoad* instruction cannot be unloaded, because of ongoing execution within the module or wrong path (module not loaded with *Load* or *StartLoad*, then the system variable ERRNO is set to ERR_UNLOAD. This error can then be handled in the error handler.

Syntax

UnLoad
[’\’Save ’,’]
[FilePath’:=’]<expression (**IN**) of *string*>
[’\’File’:=’ <expression (**IN**) of *string*>]’;’

Related information

	Described in:
Load a program module	Instructions - *Load* Instructions - *StartLoad-WaitLoad*
Accept unresolved references	System Parameters - *Controller* System Parameters - *Tasks* System Parameters - *BindRef*

WaitDI Waits until a digital input signal is set

WaitDI (Wait Digital Input) is used to wait until a digital input is set.

Example

WaitDI di4, 1;

Program execution continues only after the *di4* input has been set.

WaitDI grip_status, 0;

Program execution continues only after the *grip_status* input has been reset.

Arguments

WaitDI Signal Value [\MaxTime] [\TimeFlag]

Signal Data type: *signaldi*

The name of the signal.

Value Data type: *dionum*

The desired value of the signal.

[\MaxTime] *(Maximum Time)* Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the condition is met, the error handler will be called, if there is one, with the error code ERR_WAIT_MAXTIME. If there is no error handler, the execution will be stopped.

[\TimeFlag] *(Timeout Flag)* Data type: *bool*

The output parameter that contains the value TRUE if the maximum permitted waiting time runs out before the condition is met. If this parameter is included in the instruction, it is not considered to be an error if the max. time runs out. This argument is ignored if the *MaxTime* argument is not included in the instruction.

Program Running

If the value of the signal is correct, when the instruction is executed, the program simply continues with the following instruction.

If the signal value is not correct, the robot enters a waiting state and when the signal changes to the correct value, the program continues. The change is detected with an interrupt, which gives a fast response (not polled).

When the robot is waiting, the time is supervised, and if it exceeds the max time value, the program will continue if a Time Flag is specified, or raise an error if it’s not. If a Time Flag is specified, this will be set to true if the time is exceeded, otherwise it will be set to false.

Syntax

WaitDI
[Signal ’:=’] < variable (**VAR**) of *signaldi* > ’,’
[Value ’:=’] < expression (**IN**) of *dionum* >
[’\’MaxTime ’:=’<expression (**IN**) of *num*>]
[’\’TimeFlag’:=’<variable (**VAR**) of *bool*>] ’;’

Related information

	Described in:
Waiting until a condition is satisfied	Instructions - *WaitUntil*
Waiting for a specified period of time	Instructions - *WaitTime*

WaitDO Waits until a digital output signal is set

WaitDO (Wait Digital Output) is used to wait until a digital output is set.

Example

WaitDO do4, 1;

Program execution continues only after the *do4* output has been set.

WaitDO grip_status, 0;

Program execution continues only after the *grip_status* output has been reset.

Arguments

WaitDO Signal Value [\MaxTime] [\TimeFlag]

Signal Data type: *signaldo*

The name of the signal.

Value Data type: *dionum*

The desired value of the signal.

[\MaxTime] *(Maximum Time)* Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the condition is met, the error handler will be called, if there is one, with the error code ERR_WAIT_MAXTIME. If there is no error handler, the execution will be stopped.

[\TimeFlag] *(Timeout Flag)* Data type: *bool*

The output parameter that contains the value TRUE if the maximum permitted waiting time runs out before the condition is met. If this parameter is included in the instruction, it is not considered to be an error if the max. time runs out. This argument is ignored if the *MaxTime* argument is not included in the instruction.

Program Running

If the value of the signal is correct, when the instruction is executed, the program simply continues with the following instruction.

If the signal value is not correct, the robot enters a waiting state and when the signal changes to the correct value, the program continues. The change is detected with an interrupt, which gives a fast response (not polled).

When the robot is waiting, the time is supervised, and if it exceeds the max time value, the program will continue if a Time Flag is specified, or raise an error if its not. If a Time Flag is specified, this will be set to true if the time is exceeded, otherwise it will be set to false.

Syntax

WaitDO
[Signal ':='] < variable (**VAR**) of *signaldo* > ','
[Value ':='] < expression (**IN**) of *dionum* >
['\'MaxTime ':='<expression (**IN**) of *num*>]
['\'TimeFlag':='<variable (**VAR**) of *bool*>] ';'

Related information

	Described in:
Waiting until a condition is satisfied	Instructions - *WaitUntil*
Waiting for a specified period of time	Instructions - *WaitTime*

WaitLoad Connect the loaded module to the task

WaitLoad is used to connect the module, if loaded with *StartLoad*, to the program task.

The loaded module must be connected to the program task with the instruction *WaitLoad*, before any of its symbol/routines can be used.

The loaded program module will be added to the modules already existing in the program memory.

This instruction can also be combined with the function to unload some other program module, to minimise the number of links (1 instead of 2).

Example

```
VAR loadsession load1;
...
StartLoad "ram1disk:PART_A.MOD", load1;
MoveL p10, v1000, z50, tool1 \WObj:=wobj1;
MoveL p20, v1000, z50, tool1 \WObj:=wobj1;
MoveL p30, v1000, z50, tool1 \WObj:=wobj1;
MoveL p40, v1000, z50, tool1 \WObj:=wobj1;
WaitLoad load1;
%"routine_x"%;
UnLoad "ram1disk:PART_A.MOD";
```

Load the program module PART_A.MOD from the *ram1disk* into the program memory. In parallel, move the robot. Then connect the new program module to the program task and call the routine *routine_x* in the module PART_A.

Arguments

WaitLoad [\UnloadPath] [\UnloadFile] LoadNo

[\UnloadPath] Data type: *string*

The file path and the file name to the file that will be unloaded from the program memory. The file name should be excluded when the argument *\UnloadFile* is used.

[\UnloadFile] Data type: *string*

When the file name is excluded in the argument *\UnloadPath*, then it must be defined with this argument.

LoadNo

Data type: *loadsession*

This is a reference to the load session, fetched by the instruction *StartLoad,* to connect the loaded program module to the program task.

Program execution

The instruction *WaitLoad* will first wait for the loading to be completed, if it is not already done, and then it will be linked and initialised. The initialisation of the loaded module sets all variables at module level to their init values.

Unsolved references will be accepted, if the system parameter for *Tasks/BindRef* is set to NO. However, when the program is started or the teach pendant function *Program Window/File/Check Program* is used, no check for unsolved references will be done if *BindRef* = NO. There will be a run time error on execution of an unsolved reference.

Another way to use references to instructions, that are not in the task from the beginning, is to use *Late Binding.* This makes it possible to specify the routine to call with a string expression, quoted between two %%. In this case the *BindRef* parameter could be set to YES (default behaviour). The *Late Binding* way is preferable.

To obtain a good program structure, that is easy to understand and maintain, all loading and unloading of program modules should be done from the main module, which is always present in the program memory during execution.

Examples

```
StartLoad  "ram1disk:DOORDIR/DOOR2.MOD",  load1;
...
WaitLoad  \UnloadPath:="ram1disk:DOORDIR/DOOR1.MOD", load1;
```

Load the program module *DOOR2.MOD* from the *ram1disk* at the directory *DOORDIR* into the program memory and connect the new module to the task. The program module DOOR1.MOD will be unloaded from the program memory.

```
StartLoad  "ram1disk:DOORDIR/" \File:="DOOR2.MOD",  load1;
! The robot can do some other work
WaitLoad  \UnloadPath:="ram1disk:DOORDIR/" \File:= "DOOR1.MOD",  load1;
```

is the same as the instructions below but the robot can do some other work during the loading time and also faster (only one link).

```
Load  "ram1disk:DOORDIR/"  \File:="DOOR2.MOD";
UnLoad "ram1disk:DOORDIR/" \File:="DOOR1.MOD";
```

Error handling

If the file specified in the *StartLoad* instruction cannot be found, the system variable ERRNO is set to ERR_FILNOTFND at execution of *WaitLoad*.

If argument *LoadNo* refers to an unknown load session, the system variable ERRNO is set to ERR_UNKPROC.

If the module is already loaded into the program memory, the system variable ERRNO is set to ERR_LOADED.

The following errors can only occur when the argument *\UnloadPath* is used in the instruction *WaitLoad*:

- If the program module specified in the argument *\UnloadPath* cannot be unloaded because of ongoing execution within the module, the system variable ERRNO is set to ERR_UNLOAD.
- If the program module specified in the argument *\UnloadPath* cannot be unloaded because the program module is not loaded with *Load* or *StartLoad-WaitLoad* from the RAPID program, the system variable ERRNO is also set to ERR_UNLOAD.

These errors can then be handled in the error handler.

Syntax

WaitLoad
   [[’\’ UnloadPath ’:=’ <expression (**IN**) of *string*>]
     [’\’ UnloadFile ’:=’ <expression (**IN**) of *string*>] ’,’]
   [LoadNo ’:=’] <variable (**VAR**) of *loadsession*> ’;’

Related information

Load a program module during execution	Instructions - *StartLoad*
Load session	Data Types - *loadsession*
Load a program module	Instructions - *Load*
Unload a program module	Instructions - *UnLoad*
Accept unsolved references	System Parameters - *Controller/Task/ BindRef*

VelSet Changes the programmed velocity

VelSet is used to increase or decrease the programmed velocity of all subsequent positioning instructions. This instruction is also used to maximize the velocity.

Example

VelSet 50, 800;

All the programmed velocities are decreased to 50% of the value in the instruction. The TCP velocity is not, however, permitted to exceed 800 mm/s.

Arguments

VelSet Override Max

Override Data type: *num*

Desired velocity as a percentage of programmed velocity. 100% corresponds to the programmed velocity.

Max Data type: *num*

Maximum TCP velocity in mm/s.

Program execution

The programmed velocity of all subsequent positioning instructions is affected until a new *VelSet* instruction is executed.

The argument *Override* affects:

- All velocity components (TCP, orientation, rotating and linear external axes) in *speeddata*.
- The programmed velocity override in the positioning instruction (the argument *\V*).
- Timed movements.

The argument *Override* does not affect:

- The welding speed in *welddata*.
- The heating and filling speed in *seamdata*.

The argument *Max* only affects the velocity of the TCP.

The default values for *Override* and *Max* are 100% and 5000 mm/s respectively. These values are automatically set

- at a cold start-up
- when a new program is loaded
- when starting program executing from the beginning.

Example

```
VelSet 50, 800;
MoveL p1, v1000, z10, tool1;
MoveL p2, v2000, z10, tool1;
MoveL p3, v1000\T:=5, z10, tool1;
```

The speed is *500* mm/s to point *p1* and *800* mm/s to *p2*. It takes *10* seconds to move from p2 to *p3*.

Limitations

The maximum speed is not taken into consideration when the time is specified in the positioning instruction.

Syntax

VelSet
[Override ':='] < expression (**IN**) of *num* > ','
[Max ':='] < expression (**IN**) of *num* > ';'

Related information

	Described in:
Definition of velocity	Data Types - *speeddata*
Positioning instructions	RAPID Summary - *Motion*

WHILE Repeats as long as ...

WHILE is used when a number of instructions are to be repeated as long as a given condition is met.

If it is possible to determine the number of repetitions in advance, the *FOR* instruction can be used.

Example

```
WHILE reg1 < reg2 DO
    ...
    reg1 := reg1 +1;
ENDWHILE
```

Repeats the instructions in the WHILE loop as long as *reg1 < reg2*.

Arguments

WHILE Condition DO ... ENDWHILE

Condition Data type: *bool*

The condition that must be met for the instructions in the WHILE loop to be executed.

Program execution

7. The condition is calculated. If the condition is not met, the WHILE loop terminates and program execution continues with the instruction following ENDWHILE.
8. The instructions in the WHILE loop are executed.
9. The WHILE loop is repeated, starting from point 1.

Syntax

(EBNF)
WHILE <conditional expression> **DO**
    <instruction list>
ENDWHILE

Related information

	Described in:
Expressions	Basic Characteristics - *Expressions*

Write Writes to a character-based file or serial channel

Write is used to write to a character-based file or serial channel. The value of certain data can be written as well as text.

Examples

Write logfile, "Execution started";

The text *Execution started* is written to the file with reference name *logfile*.

Write logfile, "No of produced parts="\Num:=reg1;

The text *No of produced parts=5*, for example, is written to the file with the reference name *logfile* (assuming that the contents of *reg1* is 5).

Arguments

Write IODevice String [\Num] | [\Bool] | [\Pos] | [\Orient] [\NoNewLine]

IODevice Data type: *iodev*

The name (reference) of the current file or serial channel.

String Data type: *string*

The text to be written.

[\Num] *(Numeric)* Data type: *num*

The data whose numeric values are to be written after the text string.

[\Bool] *(Boolean)* Data type: *bool*

The data whose logical values are to be written after the text string.

[\Pos] *(Position)* Data type: *pos*

The data whose position is to be written after the text string.

[\Orient] *(Orientation)* Data type: *orient*

The data whose orientation is to be written after the text string.

[\NoNewLine] Data type: *switch*

Omits the line-feed character that normally indicates the end of the text.

Program execution

The text string is written to a specified file or serial channel. If the argument *\NoNewLine* is not used, a line-feed character (LF) is also written.

If one of the arguments *\Num*, *\Bool*, *\Pos* or *\Orient* is used, its value is first converted to a text string before being added to the first string. The conversion from value to text string takes place as follows:

Argument	Value	Text string
\Num	23	"23"
\Num	1.141367	"1.14137"
\Bool	TRUE	"TRUE"
\Pos	[1817.3,905.17,879.11]	"[1817.3,905.17,879.11]"
\Orient	[0.96593,0,0.25882,0]	"[0.96593,0,0.25882,0]"

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.

Example

```
VAR iodev printer;
.
Open "sio1:", printer\Write;
WHILE DInput(stopprod)=0 DO
    produce_part;
    Write printer, "Produced part="\Num:=reg1\NoNewLine;
    Write printer, "            "\NoNewLine;
    Write printer, CTime();
ENDWHILE
Close printer;
```

A line, including the number of the produced part and the time, is output to a printer each cycle. The printer is connected to serial channel *sio1:*. The printed message could look like this:

```
Produced part=473           09:47:15
```

Limitations

The arguments *\Num*, *\Bool*, *\Pos* and *\Orient* are mutually exclusive and thus cannot be used simultaneously in the same instruction.

This instruction can only be used for files or serial channels that have been opened for writing.

Error handling

If an error occurs during writing, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Syntax

Write
[IODevice’:=’] <variable (**VAR**) of *iodev*>’,’
[String’:=’] <expression (**IN**) of *string*>
[’\’Num’:=’ <expression (**IN**) of *num*>]
| [’\’Bool’:=’ <expression (**IN**) of *bool*>]
| [’\’Pos’:=’ <expression (**IN**) of *pos*>]
| [’\’Orient’:=’ <expression (**IN**) of *orient*>]
[’\’NoNewLine]’;’

Related information

	Described in:
Opening a file or serial channel	RAPID Summary - *Communication*

WriteBin Writes to a binary serial channel

WriteBin is used to write a number of bytes to a binary serial channel.

Example

WriteBin channel2, text_buffer, 10;

10 characters from the *text_buffer* list are written to the channel referred to by *channel2*.

Arguments

WriteBin IODevice Buffer NChar

IODevice Data type: *iodev*

Name (reference) of the current serial channel.

Buffer Data type: *array of num*

The list (array) containing the numbers (characters) to be written.

NChar *(Number of Characters)* Data type*: num*

The number of characters to be written from the *Buffer.*

Program execution

The specified number of numbers (characters) in the list is written to the serial channel.

Limitations

This instruction can only be used for serial channels that have been opened for binary reading and writing.

Error handling

If an error occurs during writing, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Example

```
VAR iodev channel;
VAR num out_buffer{20};
VAR num input;
VAR num nchar;
Open "sio1:", channel\Bin;

out_buffer{1} := 5;                          ( enq )
WriteBin channel, out_buffer, 1;
input := ReadBin (channel \Time:= 0.1);

IF input = 6 THEN                            ( ack )
    out_buffer{1} := 2;                      ( stx )
    out_buffer{2} := 72;                     ( 'H' )
    out_buffer{3} := 101;                    ( 'e' )
    out_buffer{4} := 108;                    ( 'l' )
    out_buffer{5} := 108;                    ( 'l' )
    out_buffer{6} := 111;                    ( 'o' )
    out_buffer{7} := 32;                     ( ' ' )
    out_buffer{8} := StrToByte("w"\Char);    ( 'w' )
    out_buffer{9} := StrToByte("o"\Char);    ( 'o' )
    out_buffer{10} := StrToByte("r"\Char);   ( 'r' )
    out_buffer{11} := StrToByte("l"\Char);   ( 'l' )
    out_buffer{12} := StrToByte("d"\Char);   ( 'd' )
    out_buffer{13} := 3;                     ( etx )
    WriteBin channel, out_buffer, 13;
ENDIF
```

The text string *Hello world* (with associated control characters) is written to a serial channel. The function *StrToByte* is used in the same cases to convert a string into a *byte* (*num*) data.

Syntax

WriteBin
  [IODevice':='] <variable (**VAR**) of *iodev*>','
  [Buffer':='] <array {*} (**IN**) of *num*>','
  [NChar':='] <expression (**IN**) of *num*>';'

Related information

	Described in:
Opening (etc.) of serial channels	RAPID Summary - *Communication*
Convert a string to a byte data	Functions - *StrToByte*
Byte data	Data Types - *byte*

WriteStrBin Writes a string to a binary serial channel

WriteStrBin (Write String Binary) is used to write a string to a binary serial channel or binary file.

Example

WriteStrBin channel2, "Hello World\0A";

The string *"Hello World\0A"* is written to the channel referred to by *channel2*. The string is in this case ended with new line \0A. All characters and hexadecimal values written with *WriteStrBin* will be unchanged by the system.

Arguments

WriteStrBin IODevice Str

IODevice Data type: *iodev*

Name (reference) of the current serial channel.

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

The text to be written.

Program execution

The text string is written to the specified serial channel or file.

Limitations

This instruction can only be used for serial channels or files that have been opened for binary reading and writing.

Error handling

If an error occurs during writing, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Example

```
VAR iodev channel;
VAR num input;
Open "sio1:", channel\Bin;

! Send the control character enq
WriteStrBin channel, "\05";
! Wait for the control character ack
input := ReadBin (channel \Time:= 0.1);
IF input = 6 THEN
    ! Send a text starting with control character stx and ending with etx
    WriteStrBin channel, "\02Hello world\03";
ENDIF

Close channel;
```

The text string *Hello world* (with associated control characters in hexadecimal) is written to a binary serial channel.

Syntax

WriteStrBin
    [IODevice':='] <variable (**VAR**) of *iodev*>','
    [Str':='] <expression (**IN**) of *string*>';'

Related information

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

WaitTime Waits a given amount of time

WaitTime is used to wait a given amount of time. This instruction can also be used to wait until the robot and external axes have come to a standstill.

Example

WaitTime 0.5;

Program execution waits 0.5 seconds.

Arguments

WaitTime [\InPos] Time

[\InPos] Data type: *switch*

If this argument is used, the robot and external axes must have come to a standstill before the waiting time starts to be counted.

Time Data type: *num*

The time, expressed in seconds, that program execution is to wait.

Program execution

Program execution temporarily stops for the given amount of time. Interrupt handling and other similar functions, nevertheless, are still active.

Example

WaitTime \InPos,0;

Program execution waits until the robot and the external axes have come to a standstill.

Limitations

If the argument *Inpos* is used, the movement instruction which precedes this instruction should be terminated with a stop point, in order to be able to restart in this instruction following a power failure.

Argument *Inpos* cannot be used together with SoftServo.

Syntax

```
WaitTime
   ['\'InPos',']
   [Time ':='] <expression (IN) of num>';'
```

Related information

	Described in:
Waiting until a condition is met	Instructions - *WaitUntil*
Waiting until an I/O is set/reset	Instruction - *WaitDI*

WaitUntil — Waits until a condition is met

WaitUntil is used to wait until a logical condition is met; for example, it can wait until one or several inputs have been set.

Example

WaitUntil di4 = 1;

Program execution continues only after the *di4* input has been set.

Arguments

WaitUntil [\InPos] Cond [\MaxTime] [\TimeFlag]

[\InPos] Data type: *switch*

If this argument is used, the robot and external axes must have stopped moving before the condition starts being evaluated.

Cond Data type: *bool*

The logical expression that is to be waited for.

[\MaxTime] Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the condition is set, the error handler will be called, if there is one, with the error code ERR_WAIT_MAXTIME. If there is no error handler, the execution will be stopped.

[\TimeFlag] *(Timeout Flag)* Data type: *bool*

The output parameter that contains the value TRUE if the maximum permitted waiting time runs out before the condition is met. If this parameter is included in the instruction, it is not considered to be an error if the max. time runs out. This argument is ignored if the *MaxTime* argument is not included in the instruction.

Program execution

If the programmed condition is not met on execution of a *WaitUntil* instruction, the condition is checked again every 100 ms.

When the robot is waiting, the time is supervised, and if it exceeds the max time value, the program will continue if a *TimeFlag* is specified, or raise an error if it’s not. If a *TimeFlag* is specified, this will be set to TRUE if the time is exceeded, otherwise it will be set to false.

Examples

```
VAR bool timeout;
WaitUntil start_input = 1 AND grip_status = 1\MaxTime := 60
                \TimeFlag := timeout;
IF timeout THEN
    TPWrite "No start order received within expected time";
ELSE
    start_next_cycle;
ENDIF
```

If the two input conditions are not met within *60 s*econds, an error message will be written on the display of the teach pendant.

```
WaitUntil \Inpos, di4 = 1;
```

Program execution waits until the robot has come to a standstill and the *di4* input has been set.

Limitation

If the argument *Inpos is used*, the movement instruction which precedes this instruction should be terminated with a stop point, in order to be able to restart in this instruction following a power failure.

Syntax

WaitUntil
    ['\'InPos',']
    [Cond ':='] <expression (**IN**) of *bool*>
    ['\'MaxTime ':='<expression (**IN**) of *num*>]
    ['\'TimeFlag':='<variable (**VAR**) of *bool*>] ';'

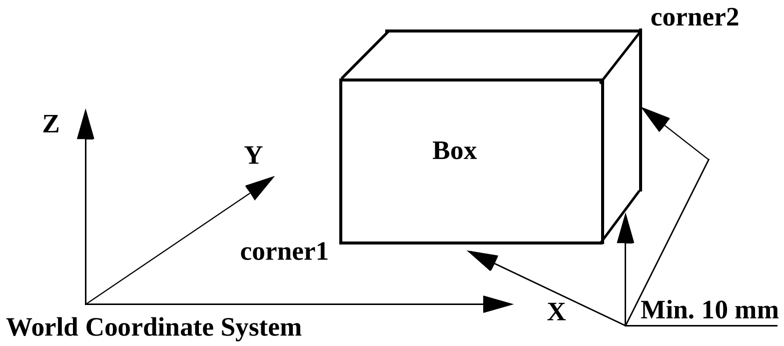
Related information

	Described in:
Waiting until an input is set/reset	Instructions - *WaitDI*
Waiting a given amount of time	Instructions - *WaitTime*
Expressions	Basic Characteristics - *Expressions*

WZBoxDef Define a box-shaped world zone

WZBoxDef (World Zone Box Definition) is used to define a world zone that has the shape of a straight box with all its sides parallel to the axes of the World Coordinate System.

Example




```
VAR shapedata volume;
CONST pos corner1:=[200,100,100];
CONST pos corner2:=[600,400,400];
...
WZBoxDef \Inside, volume, corner1, corner2;
```

Define a straight box with coordinates parallel to the axes of the world coordinate system and defined by the opposite corners *corner1* and *corner2*.

Arguments

WZBoxDef [\Inside] | [\Outside] Shape LowPoint HighPoint

\Inside
Data type: *switch*

Define the volume inside the box.

\Outside
Data type: *switch*

Define the volume outside the box (inverse volume).

One of the arguments *\Inside* or *\Outside* must be specified.

Shape
Data type: *shapedata*

Variable for storage of the defined volume (private data for the system).

LowPoint
Data type: *pos*

Position (x,y,x) in mm defining one lower corner of the box.

HighPoint
Data type: *pos*

Position (x,y,z) in mm defining the corner diagonally opposite to the previous one.

Program execution

The definition of the box is stored in the variable of type *shapedata* (argument *Shape*), for future use in *WZLimSup* or *WZDOSet* instructions.

Limitations

The *LowPoint* and *HighPoint* positions must be valid opposite corners (with different x, y and z coordinate values).

If the robot is used to point out the *LowPoint* or *HighPoint*, work object *wobj0* must be active (use of component *trans* in *robtarget* e.g. p1.trans as argument).

Syntax

```
WZBoxDef
    ['\'Inside] | ['\'Outside] ','
    [Shape':=']<variable (VAR) of shapedata>','
    [LowPoint':=']<expression (IN) of pos>','
    [HighPoint':=']<expression (IN) of pos>';'
```

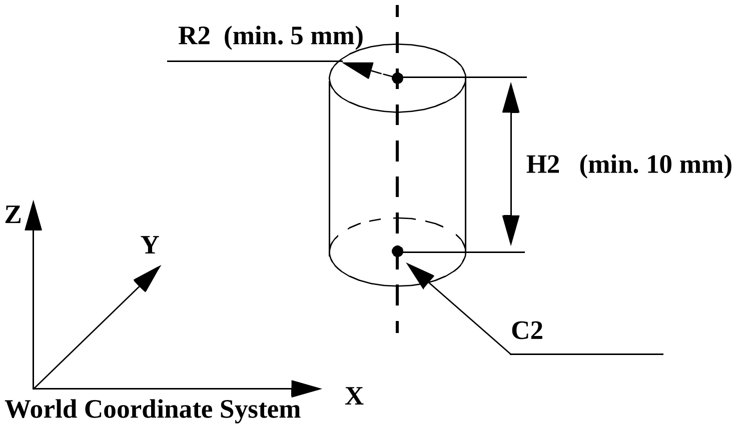
Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Define sphere-shaped world zone	Instructions - *WZSphDef*
Define cylinder-shaped world zone	Instructions - *WZCylDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

WZCylDef Define a cylinder-shaped world zone

WZCylDef (World Zone Cylinder Definition) is used to define a world zone that has the shape of a cylinder with the cylinder axis parallel to the z-axis of the World Coordinate System.

Example




```
VAR shapedata volume;
CONST pos C2:=[300,200,200];
CONST num R2:=100;
CONST num H2:=200;
...
WZCylDef \Inside, volume, C2, R2, H2;
```

Define a cylinder with the centre of the bottom circle in *C2*, radius *R2* and height *H2*.

Arguments

WZCylDef [\Inside] | [\Outside] Shape CentrePoint Radius Height

\Inside Data type: *switch*

Define the volume inside the cylinder.

\Outside Data type: *switch*

Define the volume outside the cylinder (inverse volume).

One of the arguments *\Inside* or *\Outside* must be specified.

Shape Data type: *shapedata*

Variable for storage of the defined volume (private data for the system).

CentrePoint Data type: *pos*

Position (x,y,z) in mm defining the centre of one circular end of the cylinder.

Radius Data type: *num*

The radius of the cylinder in mm.

Height Data type: *num*

The height of the cylinder in mm.
If it is positive (+z direction), the *CentrePoint* argument is the centre of the lower end of the cylinder (as in the above example).
If it is negative (-z direction), the *CentrePoint* argument is the centre of the upper end of the cylinder.

Program execution

The definition of the cylinder is stored in the variable of type *shapedata* (argument *Shape*), for future use in *WZLimSup* or *WZDOSet* instructions.

Limitations

If the robot is used to point out the *CentrePoint*, work object *wobj0* must be active (use of component *trans* in *robtarget* e.g. p1.trans as argument).

Syntax

WZCylDef
  ['\'Inside] | ['\'Outside]','
  [Shape':=']<variable (**VAR**) of *shapedata*>','
  [CentrePoint':=']<expression (**IN**) of *pos*>','
  [Radius':=']<expression (**IN**) of *num*>','
  [Height':=']<expression (**IN**) of *num*>';'

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*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

WZDisable Deactivate temporary world zone supervision

WZDisable (World Zone Disable) is used to deactivate the supervision of a temporary world zone, previously defined either to stop the movement or to set an output.

Example

```
VAR wztemporary wzone;
...
PROC ...
    WZLimSup \Temp, wzone, volume;
    MoveL p_pick, v500, z40, tool1;
    WZDisable wzone;
    MoveL p_place, v200, z30, tool1;
ENDPROC
```

When moving to *p_pick*, the position of the robot's TCP is checked so that it will not go inside the specified volume *wzone*. This supervision is not performed when going to *p_place*.

Arguments

WZDisable WorldZone

WorldZone Data type: *wztemporary*

Variable or persistent variable of type *wztemporary*, which contains the identity of the world zone to be deactivated.

Program execution

The temporary world zone is deactivated. This means that the supervision of the robot's TCP, relative to the corresponding volume, is temporarily stopped. It can be re-activated via the *WZEnable* instruction.

Limitations

Only a temporary world zone can be deactivated. A stationary world zone is always active.

Syntax

WZDisable
[WorldZone':=']<variable or persistent (**INOUT**) of *wztemporary*>';'

Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Temporary world zone data	Data Types - *wztemporary*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone set digital output	Instructions - *WZDOSet*
Activate world zone	Instructions - *WZEnable*
Erase world zone	Instructions - *WZFree*

WZDOSet Activate world zone to set digital output

WZDOSet (World Zone Digital Output Set) is used to define the action and to activate a world zone for supervision of the robot movements.

After this instruction is executed, when the robot's TCP is inside the defined world zone or is approaching close to it, a digital output signal is set to the specified value.

Example

```
VAR wztemporary service;

PROC zone_output()
    VAR shapedata volume;
    CONST pos p_service:=[500,500,700];
    ...
    WZSphDef \Inside, volume, p_service, 50;
    WZDOSet \Temp, service \Inside, volume, do_service, 1;
ENDPROC
```

Definition of temporary world zone *service* in the application program, that sets the signal *do_service*, when the robot's TCP is inside the defined sphere during program execution or when jogging.

Arguments

WZDOSet [\Temp] | [\Stat] WorldZone [\Inside] | [\Before] Shape Signal SetValue

\Temp (*Temporary*) Data type: *switch*

The world zone to define is a temporary world zone.

\Stat (*Stationary*) Data type: *switch*

The world zone to define is a stationary world zone.

One of the arguments *\Temp* or *\Stat* must be specified.

WorldZone Data type: *wztemporary*

Variable or persistent variable, that will be updated with the identity (numeric value) of the world zone.

If use of switch *\Temp*, the data type must be *wztemporary.*
If use of switch *\Stat*, the data type must be *wzstationary.*

\Inside

Data type: *switch*

The digital output signal will be set when the robot's TCP is inside the defined volume.

\Before

Data type: *switch*

The digital output signal will be set before the robot's TCP reaches the defined volume (as soon as possible before the volume).

One of the arguments *\Inside* or *\Outside* must be specified.

Shape

Data type: *shapedata*

The variable that defines the volume of the world zone.

Signal

Data type: *signaldo*

The name of the digital output signal that will be changed.

If a stationary worldzone is used, the signal must be write protected for access from the user (RAPID, TP). Set Access = System for the signal in System Parameters.

SetValue

Data type: *dionum*

Desired value of the signal (0 or 1) when the robot's TCP is inside the volume or just before it enters the volume.

When outside or just outside the volume, the signal is set to the opposite value.

Program execution

The defined world zone is activated. From this moment, the robot's TCP position is supervised and the output will be set, when the robot's TCP position is inside the volume (*\Inside*) or comes close to the border of the volume (*\Before*).

Example

```
VAR wztemporary home;
VAR wztemporary service;
PERS wztemporary equip1:=[0];

PROC main()
  ...
  ! Definition of all temporary world zones
  zone_output;
  ...
  ! equip1 in robot work area
  WZEnable equip1;
```

```
    ...
    ! equip1 out of robot work area
    WZDisable equip1;
    ...
    ! No use for equip1 any more
    WZFree equip1;
    ...
ENDPROC

PROC zone_output()
    VAR shapedata volume;
    CONST pos p_home:=[800,0,800];
    CONST pos p_service:=[800,800,800];
    CONST pos p_equip1:=[-800,-800,0];
    ...
    WZSphDef \Inside, volume, p_home, 50;
    WZDOSet \Temp, home \Inside, volume, do_home, 1;
    WZSphDef \Inside, volume, p_service, 50;
    WZDOSet \Temp, service \Inside, volume, do_service, 1;
    WZCylDef \Inside, volume, p_equip1, 300, 1000;
    WZLimSup \Temp, equip1, volume;
    ! equip1 not in robot work area
    WZDisable equip1;
ENDPROC
```

Definition of temporary world zones *home* and *service* in the application program, that sets the signals *do_home* and *do_service*, when the robot is inside the sphere *home* or *service* respectively during program execution or when jogging.

Also, definition of a temporary world zone *equip1*, which is active only in the part of the robot program when *equip1* is inside the working area for the robot. At that time the robot stops before entering the *equip1* volume, both during program execution and manual jogging. *equip1* can be disabled or enabled from other program tasks by using the persistent variable *equip1* value.

Limitations

A world zone cannot be redefined by using the same variable in the argument *WorldZone*.

A stationary world zone cannot be deactivated, activated again or erased in the RAPID program.

A temporary world zone can be deactivated (*WZDisable*), activated again (*WZEnable*) or erased (*WZFree)* in the RAPID program.

Syntax

WZDOSet
  ('\'Temp) | ('\'Stat) ','
  [WorldZone':=']<variable or persistent (**INOUT**) of *wztemporary*>
  ('\'Inside) | ('\'Before) ','
  [Shape':=']<variable (**VAR**) of *shapedata*>','
  [Signal':=']<variable (**VAR**) of *signaldo*>','
  [SetValue':=']<expression (**IN**) of dio*num*>';'

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*
Stationary world zone	Data Types - *wzstationary*
Define straight box-shaped world zone	Instructions - *WZBoxDef*
Define sphere-shaped world zone	Instructions - *WZSphDef*
Define cylinder-shaped world zone	Instructions - *WZCylDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Signal access mode	User's Guide - System Parameters I/O Signals

WZEnable Activate temporary world zone supervision

WZEnable (World Zone Enable) is used to re-activate the supervision of a temporary world zone, previously defined either to stop the movement or to set an output.

Example

```
VAR wztemporary wzone;
...
PROC ...
    WZLimSup \Temp, wzone, volume;
    MoveL p_pick, v500, z40, tool1;
    WZDisable wzone;
    MoveL p_place, v200, z30, tool1;
    WZEnable wzone;
    MoveL p_home, v200, z30, tool1;
ENDPROC
```

When moving to *p_pick*, the position of the robot's TCP is checked so that it will not go inside the specified volume *wzone*. This supervision is not performed when going to *p_place*, but is reactivated before going to *p_home*

Arguments

WZEnable WorldZone

WorldZone Data type: *wztemporary*

Variable or persistent variable of the type *wztemporary*, which contains the identity of the world zone to be activated.

Program execution

The temporary world zone is re-activated.
Please note that a world zone is automatically activated when it is created. It need only be re-activated when it has previously been deactivated by *WZDisable*.

Limitations

Only a temporary world zone can be deactivated and reactivated. A stationary world zone is always active.

Syntax

WZEnable
[WorldZone':=']<variable or persistent (**INOUT**) of *wztemporary*>';'

Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Temporary world zone data	Data Types - *wztemporary*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone set digital output	Instructions - *WZDOSet*
Deactivate world zone	Instructions - *WZDisable*
Erase world zone	Instructions - *WZFree*

WZFree Erase temporary world zone supervision

WZFree (World Zone Free) is used to erase the definition of a temporary world zone, previously defined either to stop the movement or to set an output.

Example

```
VAR wztemporary wzone;
...
PROC ...
    WZLimSup \Temp, wzone, volume;
    MoveL p_pick, v500, z40, tool1;
    WZDisable wzone;
    MoveL p_place, v200, z30, tool1;
    WZEnable wzone;
    MoveL p_home, v200, z30, tool1;
    WZFree wzone;
ENDPROC
```

When moving to *p_pick*, the position of the robot's TCP is checked so that it will not go inside a specified volume *wzone*. This supervision is not performed when going to *p_place*, but is reactivated before going to *p_home*. When this position is reached, the world zone definition is erased.

Arguments

WZFree WorldZone

WorldZone

Data type: *wztemporary*

Variable or persistent variable of the type *wztemporary*, which contains the identity of the world zone to be erased.

Program execution

The temporary world zone is first deactivated and then its definition is erased.

Once erased, a temporary world zone cannot be either re-activated nor deactivated.

Limitations

Only a temporary world zone can be deactivated, reactivated or erased. A stationary world zone is always active.

Syntax

WZFree
[WorldZone':=']<variable or persistent (**INOUT**) of *wztemporary*>';'

Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Temporary world zone data	Data Types - *wztemporary*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone set digital output	Instructions - *WZDOSet*
Deactivate world zone	Instructions - *WZDisable*
Activate world zone	Instructions - *WZEnable*

WZLimSup Activate world zone limit supervision

WZLimSup (World Zone Limit Supervision) is used to define the action and to activate a world zone for supervision of the working area of the robot.

After this instruction is executed, when the robot's TCP reaches the defined world zone, the movement is stopped both during program execution and when jogging.

Example

```
VAR wzstationary max_workarea;
...
PROC POWER_ON()
   VAR shapedata volume;
   ...
   WZBoxDef \Outside, volume, corner1, corner2;
   WZLimSup \Stat, max_workarea, volume;
ENDPROC
```

Definition and activation of stationary world zone *max_workarea*, with the shape of the area outside a box (temporarily stored in *volume*) and the action work-area supervision. The robot stops with an error message before entering the area outside the box.

Arguments

WZLimSup [\Temp] | [\Stat] WorldZone Shape

\Temp (*Temporary*) Data type: *switch*

The world zone to define is a temporary world zone.

\Stat (*Stationary*) Data type: *switch*

The world zone to define is a stationary world zone.

One of the arguments *\Temp* or *\Stat* must be specified.

WorldZone Data type: *wztemporary*

Variable or persistent variable that will be updated with the identity (numeric value) of the world zone.

If use of switch *\Temp*, the data type must be *wztemporary.*
If use of switch *\Stat*, the data type must be *wzstationary.*

Shape Data type: *shapedata*

The variable that defines the volume of the world zone.

Program execution

The defined world zone is activated. From this moment the robot's TCP position is supervised. If it reaches the defined area the movement is stopped.

Example

```
VAR wzstationary box1_invers;
VAR wzstationary box2;

PROC wzone_power_on()
    VAR shapedata volume;
    CONST pos box1_c1:=[500,-500,0];
    CONST pos box1_c2:=[-500,500,500];
    CONST pos box2_c1:=[500,-500,0];
    CONST pos box2_c2:=[200,-200,300];
    ...
    WZBoxDef \Outside, volume, box1_c1, box1_c2;
    WZLimSup \Stat, box1_invers, volume;
    WZBoxDef \Inside, volume, box2_c1, box2_c2;
    WZLimSup \Stat, box2, volume;
ENDPROC
```

Limitation of work area for the robot with the following stationary world zones:

- Outside working area when outside box1_invers
- Outside working area when inside box2

If this routine is connected to the system event POWER ON, these world zones will always be active in the system, both for program movements and manual jogging.

Limitations

A world zone cannot be redefined using the same variable in argument *WorldZone*.

A stationary world zone cannot be deactivated, activated again or erased in the RAPID program.

A temporary world zone can be deactivated (*WZDisable*), activated again (*WZEnable*) or erased (*WZFree)* in the RAPID program.

Syntax

WZLimSup
['\'Temp] | ['\Stat]','
[WorldZone':=']<variable or persistent (**INOUT**) of *wztemporary*>','
[Shape':='] <variable (**VAR**) of *shapedata*>';'

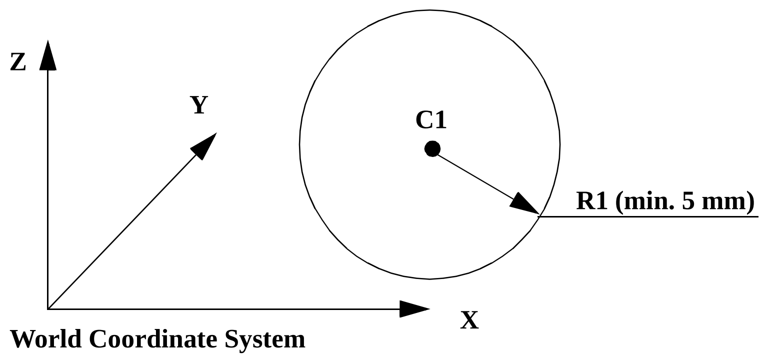
Related information

	Described in:
World Zones	Motion and I/O Principles - *World Zones*
World zone shape	Data Types - *shapedata*
Temporary world zone	Data Types - *wztemporary*
Stationary world zone	Data Types - *wzstationary*
Define straight box-shaped world zone	Instructions - *WZBoxDef*
Define sphere-shaped world zone	Instructions - *WZSphDef*
Define cylinder-shaped world zone	Instructions - *WZCylDef*
Activate world zone digital output set	Instructions - *WZDOSet*

WZSphDef Define a sphere-shaped world zone

WZSphDef (World Zone Sphere Definition) is used to define a world zone that has the shape of a sphere.

Example




```
VAR shapedata volume;
CONST pos C1:=[300,300,200];
CONST num R1:=200;
...
WZSphDef \Inside, volume, C1, R1;
```

Define a sphere named *volume* by its centre *C1* and its radius *R1*.

Arguments

WZSphDef [\Inside] | [\Outside] Shape CentrePoint Radius

\Inside Data type: *switch*

Define the volume inside the sphere.

\Outside Data type: *switch*

Define the volume outside the sphere (inverse volume).

One of the arguments *\Inside* or *\Outside* must be specified.

Shape Data type: *shapedata*

Variable for storage of the defined volume (private data for the system).

CentrePoint
Data type: *pos*

Position (x,y,z) in mm defining the centre of the sphere.

Radius
Data type: *num*

The radius of the sphere in mm.

Program execution

The definition of the sphere is stored in the variable of type *shapedata* (argument *Shape*), for future use in *WZLimSup* or *WZDOSet* instructions.

Limitations

If the robot is used to point out the *CentrePoint*, work object *wobj0* must be active (use of component *trans* in *robtarget* e.g. p1.trans as argument).

Syntax

WZSphDef
[’\’Inside] | [’\’Outside]’,’
[Shape’:=’]<variable (**VAR**) of *shapedata*>’,’
[CentrePoint’:=’]<expression (**IN**) of *pos*>’,’
[Radius’:=’]<expression (**IN**) of *num*>’;’

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 cylinder-shaped world zone	Instructions - *WZCylDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

CONTENTS

Functions

ORobT	Removes a program displacement from a position
PoseInv	Inverts the pose
PoseMult	Multiplies pose data
PoseVect	Applies a transformation to a vector
Pow	Calculates the power of a value
Present	Tests if an optional parameter is used
ReadBin	Reads from a binary serial channel or file
ReadMotor	Reads the current motor angles
ReadNum	Reads a number from a file or the serial channel
ReadStr	Reads a string from a file or serial channel
RelTool	Make a displacement relative to the tool
Round	Round is a numeric value
RunMode	Read the running mode
Sin	Calculates the sine value
Sqrt	Calculates the square root value
StrFind	Searches for a character in a string
StrLen	Gets the string length
StrMap	Maps a string
StrMatch	Search for pattern in string
StrMemb	Checks if a character belongs to a set
StrOrder	Checks if strings are ordered
StrPart	Finds a part of a string
StrToByte	Converts a string to a byte data
StrToVal	Converts a string to a value
Tan	Calculates the tangent value
TestAndSet	Test variable and set if unset
TestDI	Tests if a digital input is set
Trunc	Truncates a numeric value
ValToStr	Converts a value to a string
VectMagn	Magnitude of a pos vector

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

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 33.

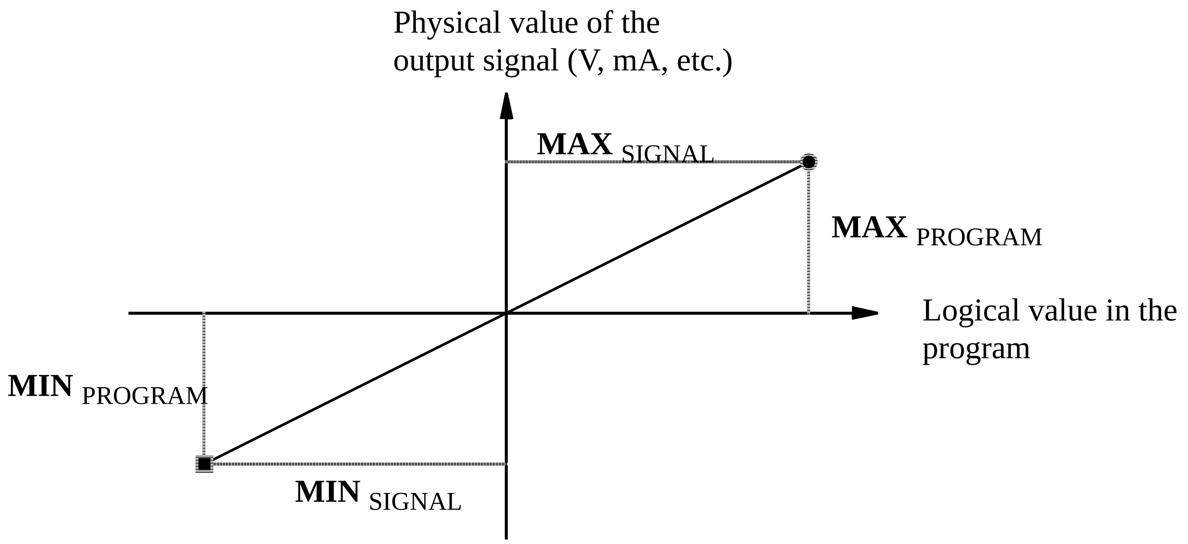


Figure 33 Diagram of how analog signal values are scaled.

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	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	User’s Guide - *System Parameters*

ArgName Gets argument name

ArgName (Argument Name) is used to get the name of the original data object for the current argument or the current data.

Example

```
VAR num abc123 :=5;
...
proc1 abc123;

PROC proc1 (num par1)
    VAR string parstring;
    ...
    parstring:=ArgName(par1);
    TPWrite "Argument name "+parstring+" with value "\Num:=par1;
ENDPROC
```

The variable *parstring* is assigned the string value "*abc123*". On TP the following string is written: "Argument name abc123 with value 5".

Return value

Data type: *string*

The original data object name.

Arguments

ArgName (Parameter)

Parameter Data type: *anytype*

The formal parameter identifier (for the routine in which *ArgName* is located) or the data identity.

Program execution

The function returns the original data object name for an entire object of the type constant, variable or persistent. The original data object can be global, local in the program module or local in a routine (normal RAPID scope rules).

If it is a part of a data object, the name of the whole data object is returned.

Example

Convert from identifier to string

This function can also be used to convert from identifier to string, by stating the identifier in the argument *Parameter* for any data object with global, local in module or local in routine scope:

```
VAR num chales :=5;
...
proc1;

PROC proc1 ()
   VAR string name;
   ...
   name:=ArgName(chales);
   TPWrite "Global data object "+name+" has value "\Num:=chales;
ENDPROC
```

The variable *name* is assigned the string value "*chales*" and on TP the following string is written: "Global data object chales has value 5".

Routine call in several steps

Note that the function returns the **origina**l data object name:

```
VAR num chales :=5;
...
proc1 chales;
...
PROC proc1 (num parameter1)
   ...
   proc2 parameter1;
   ...
ENDPROC

PROC proc2 (num par1)
   VAR string name;
   ...
   name:=ArgName(par1);
   TPWrite "Original data object name "+name+" with value "\Num:=par1;
ENDPROC
```

The variable *name* is assigned the string value "*chales*" and on TP the following string is written: "Original data object name charles with value 5".

Error handling

If one of the following errors occurs, the system variable ERRNO is set to ERR_ARGNAME:

- Argument is expression value
- Argument is not present
- Argument is of type switch

This error can then be handled in the error handler.

Syntax

ArgName '('
[Parameter':='] < reference (**REF**) of any type> ')'

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*

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

	Described in:
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

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 ....:	Writable ASCII char	1	ASCII table (*)

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

Arguments

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

ByteData Data type: *byte*

The byte 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’(’
    [ByteData ’:=’] <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

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.

If the clock has overflowed, program execution is stopped with an error message.

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;

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.

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.

When programming, it is very sensible to always specify arguments \Tool and \WObj. The function will always then return the wanted position, although 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;

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.

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.

When programming, it is very sensible to always specify arguments \Tool and \WObj. The function will always then return the wanted position, although 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*

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;

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*.

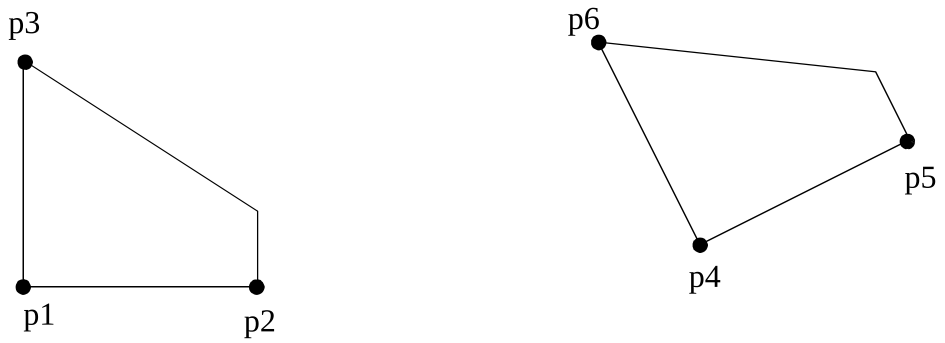
Related information

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

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



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

```
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;
```

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. This position must be measured and determined with great accuracy.

NewP2 Data type: *robtarget*

The second displaced position. It should be noted that this position can be measured and determined with less accuracy in one direction, e.g. this position must be placed on a line describing the new direction of *p1* to *p2*.

NewP3 Data type: *robtarget*

The third displaced position. This position can be measured and determined with less accuracy in two directions, e.g. it has to be placed in a plane describing the new plane of *p1*, *p2* and *p3*.

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*.

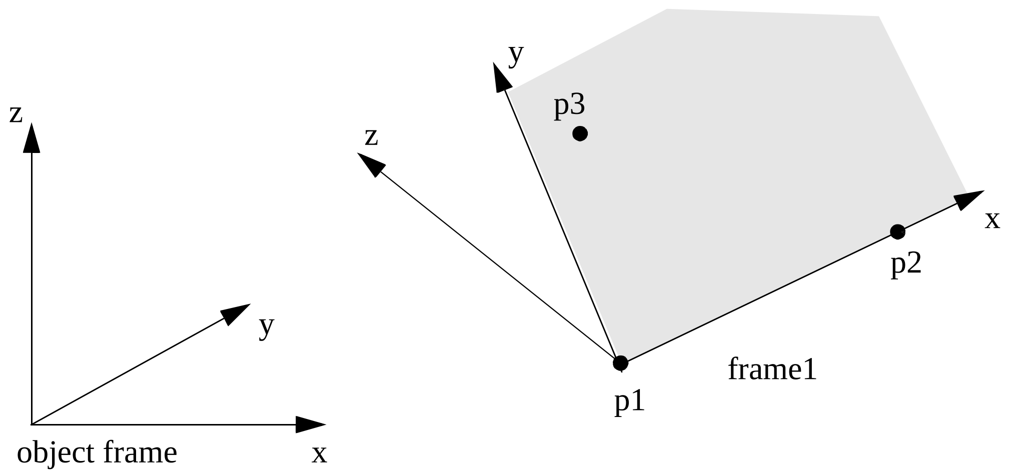
Related information

	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*

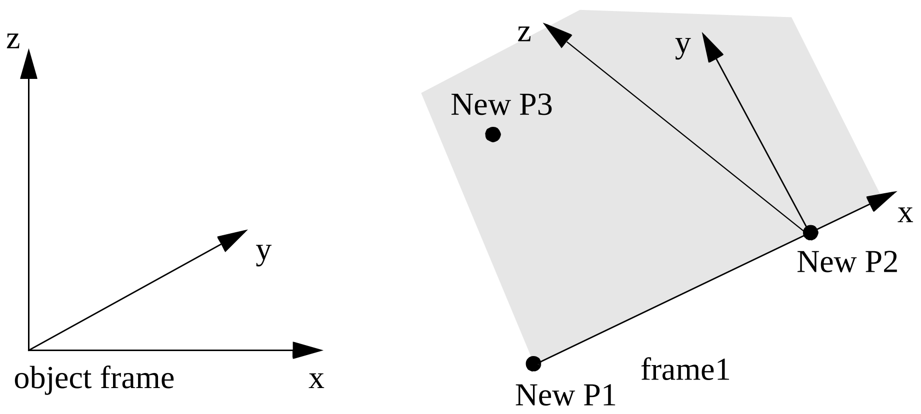
The second position, which will define the direction of the x-axis of the new frame.

NewP3 Data type: *robtarget*

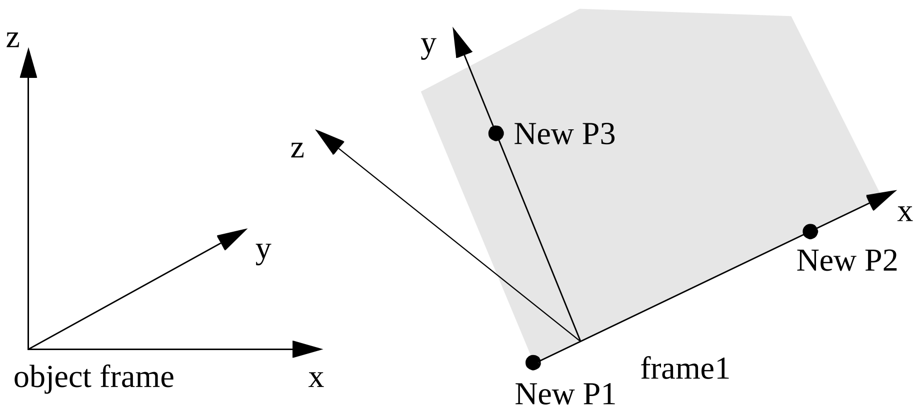
The third position, which will define the xy-plane of the new frame. The position of point 3 will be on the positive y side, see the figure above.

[**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.



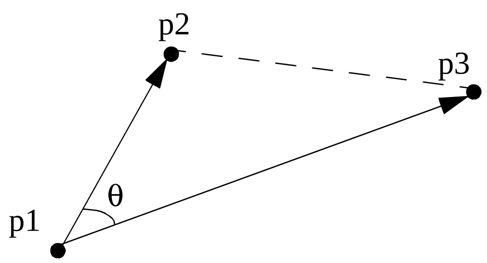
Origin = 3 means that the origin is placed on the line going through NewP1 and NewP2 and so that NewP3 will be placed on the y axis, see the figure below.



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* cannot be too small i.e. the positions cannot be too close. The distances between the positions *p1 - p2* and *p1 - p3* cannot be shorter then 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*.

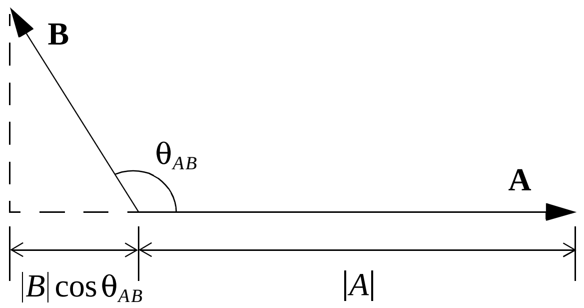
Related information

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

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* > ’)’

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 := GetEuler(\X, object.rot);
angley := GetEuler(\Y, object.rot);
anglez := GetEuler(\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.

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, e^x.

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 "ram1disk:notmymod.mod";
WHILE TRUE DO
    ! Call some routine in notmymod
    notmymodrout;
    IF FileTime("ram1disk:notmymod.mod" \ModifyTime)
        > ModTime("notmymod") THEN
        UnLoad "ram1disk:notmymod.mod";
        Load "ram1disk: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*

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*

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*

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*

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 "ram1disk:mymod.mod";
      WHILE TRUE DO
         ! Call some routine in mymod
         mymodrout;
         IF FileTime("ram1disk:mymod.mod" \ModifyTime)
            > ModTime("mymod") THEN
            UnLoad "ram1disk:mymod.mod";
            Load "ram1disk: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 34). 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.

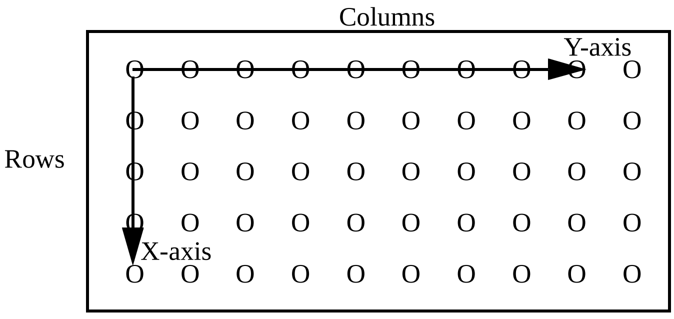


Figure 34 The position and orientation of the pallet is specified by defining a work object.

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 <u>new</u> y axis
- -rotation around the <u>new</u> 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 <u>new</u> y axis
- -rotation around the <u>new</u> 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*.

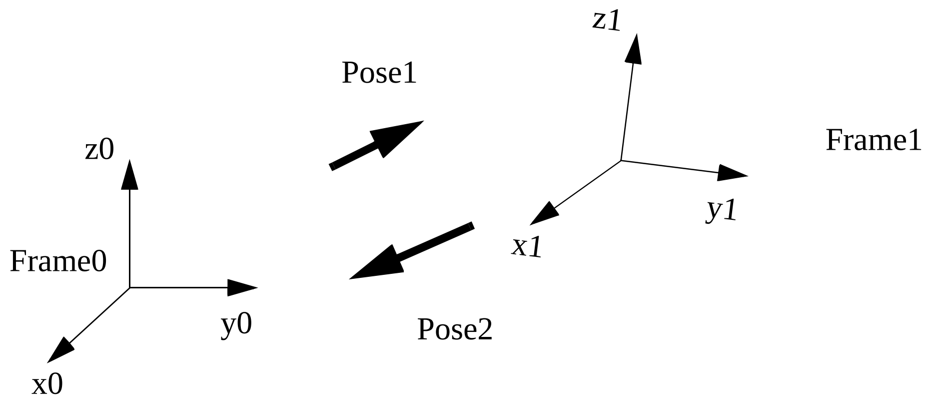
Related information

	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*.

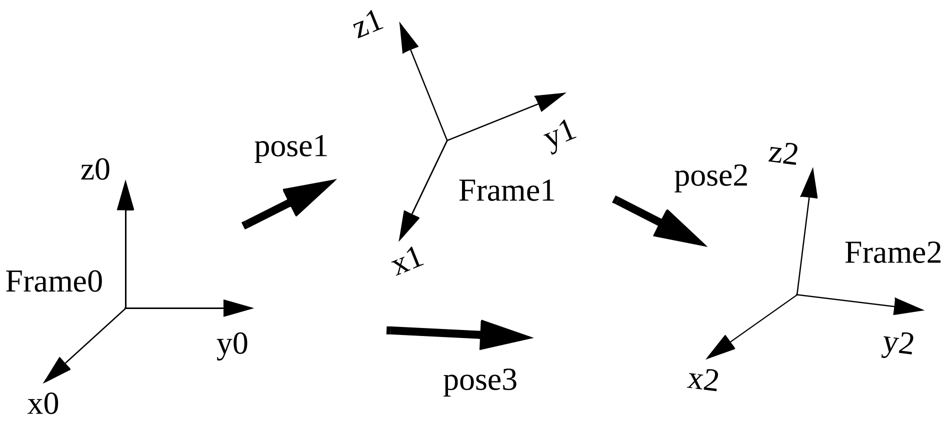
Related information

	Described in:
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*.

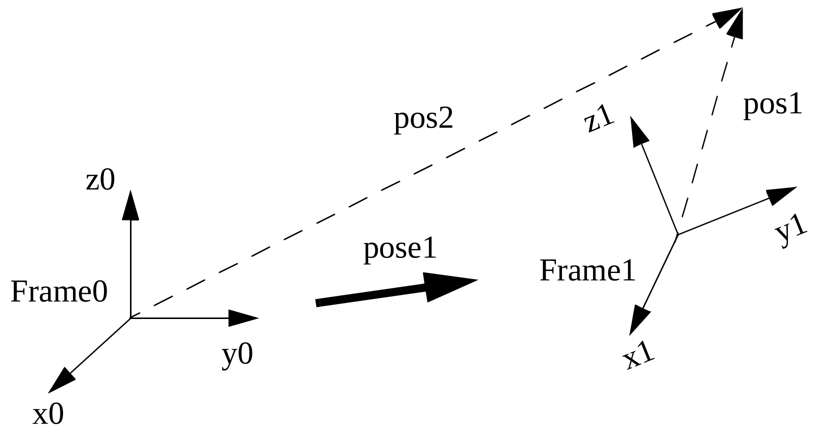
Related information

	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 (x^y).

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 $y \leq 0$.

Syntax

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

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 from a binary serial channel or file

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

Example

```
VAR iodev inchannel;
.
Open "sio1:", inchannel\Bin;
character := ReadBin(inchannel);
```

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

Return value

Data type: *num*

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

Arguments

ReadBin (IODevice [\Time])

IODevice Data type: *iodev*

The name (reference) of the current 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 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 RAPID program at program start.

Program execution

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

Example

```
Open “flp1:myfile.bin”, file\Bin;
.
Rewind file;
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 of the file and display the binary data received on the teach pendant, converted to ASCII characters (one char on each line).

Limitations

The function can only be used for channels and files that have been opened for binary reading and writing.

Error handling

If an error occurs during reading, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in 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.) serial channels	RAPID Summary - *Communication*
Convert a byte to a string data	Functions - *ByteToStr*
Byte data	Data Types - *byte*

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 the serial channel

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

Example

```
VAR iodev infile;
.
Open "flp1:file.doc", infile\Read;
reg1 := ReadNum(infile);
```

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

Return value

Data type: *num*

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

Arguments

ReadNum (IODevice [\Time])

IODevice Data type: *iodev*

The name (reference) of the file 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 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 RAPID program at program start.

Program execution

The function reads a line from a file, i.e. reads everything up to and including the next line-feed character (LF), but not more than 80 characters. If the line 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. If all the characters read are not digits, 0 is returned.

Example

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

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 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. 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*>
    [’\’Time’:=’ <expression (**IN**) of *num*>]’)’

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

Related information

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

ReadStr Reads a string from a file or serial channel

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

Example

```
VAR iodev infile;
.
Open "flp1:file.doc", infile\Read;
text := ReadStr(infile);
```

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

Return value

Data type: *string*

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

Arguments

ReadStr (IODevice [\Time])

IODevice Data type: *iodev*

The name (reference) of the file 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 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.

Program execution

The function reads a line from a file, i.e. reads everything up to and including the next line-feed character (LF), but not more than 80 characters. If the line exceeds 80 characters, the remainder of the characters will be read on the next reading.

Example

```
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.

Limitations

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

Error handling

If an error occurs during reading, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in 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*>
    ['\'Time':=' <expression (**IN**) of *num*>]')'

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

Related information

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

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*

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

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	ASCII table

RAPID character codes (e.g. "\07" for BEL control character) cannot 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*

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*

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*

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*.

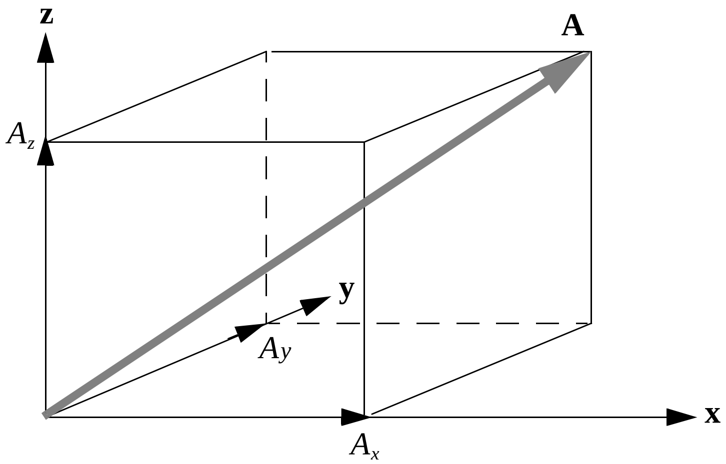
Related information

	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*

INDEX