The heart of Robotics

RAPID reference manual

RAPID reference manual - part1b, Instructions S-Z

Controller software IRC5
RobotWare 5.0

ABB

RAPID reference manual
3HAC 16581-1
Revision B

Controller software IRC5
RAPID reference manual - part 1b, Instructions S-Z

RobotWare 5.0

Table of contents

Instructions S-Z

Index

ABB Automation Technologies AB
Robotics
SE-721 68 Västerås
Sweden

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 "HOME:/PART_B.MOD";
...
Save "PART_B";
```

Load the program module with the file name *PART_B.MOD* from *HOME:* into the program memory.

Save the program module *PART_B* with the original file path *HOME:* and with the original file name *PART_B.MOD*.

Arguments

Save [\TaskRef] ModuleName [\FilePath] [\File]

[\TaskRef] **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 T_ROB1 task the variable identity will be T_ROB1Id, 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 FlexPendant, external computer, or system configuration, 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:="HOME:/DOORDIR/PART_A.MOD";
```

Save the program module *PART_A* to *HOME:* in the file *PART_A.MOD* and in the directory *DOORDIR*.

```
Save  "PART_A" \FilePath:="HOME:” \File:=”DOORDIR/PART_A.MOD";
```

Same as above but another syntax.

```
Save \TaskRef:=TSK1Id, "PART_A" \FilePath:="HOME:/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 FlexPendant.

Avoid ongoing robot movements during the saving.

Error handling

If the program module cannot be saved because there is 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, the system variable ERRNO is set to ERR_IOERROR.

If argument *\FilePath* is not specified for program modules loaded from the FlexPendant, 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
  [’\’ TaskRef ’:=’ <variable (**VAR**) of *taskid*> ’,’]
  [ModuleName ’:=’] <expression (**IN**) of *string*>
  [’\’ FilePath ’:=’<expression (**IN**) of *string*>]
  [’\’ File ’:=’ <expression (**IN**) of *string*>] ’;’

Related information

Table 1

	Described in:
Program tasks	Data Types - *taskid*

SCWrite - Send variable data to a client application

SCWrite (Superior Computer Write) is used to send the name, type, dimension and value of a persistent variable to a client application. It is possible to send both single variables and arrays of variables.

Examples

```
PERS num cycle_done;

PERS num numarr{2}:=[1,2];

SCWrite cycle_done;
```

The name, type and value of the persistent variable *cycle_done* is sent to all client applications.

```
SCWrite \ToNode := "138.221.228.4", cycle_done;
```

The name, type and value of the persistent variable *cycle_done* is sent to all client applications. The argument ToNode will be ignored.

```
SCWrite numarr;
```

The name, type, dim and value of the persistent variable *numarr* is sent to all client applications.

```
SCWrite \ToNode := "138.221.228.4", numarr;
```

The name, type, dim and value of the persistent variable *numarr* is sent to all client applications. The argument ToNode will be ignored.

Arguments

SCWrite [\ToNode] Variable

[\ToNode] **Data type:** *string*

The node name does not have any effect, the node name can still be used.

Variable **Data type:** *anytype*

The name of a persistent variable.

Program execution

The name, type, dim and value of the persistent variable is sent to all client applications. ‘dim’ is the dimension of the variable and is only sent if the variable is an array.

Syntax

SCWrite
  [’\’ ToNode ’:=’ < expression (**IN**) of *string*> ’,’]
  [Variable’:=’] < persistent (**PERS**) of *anytype*>’;’

Error handling

The SCWrite instruction will return an error in the following cases:

- The variable could not be sent to the client. This can have the following cause:

  - The SCWrite messages comes so close so that they cannot be sent to the client.
    Solution: Put in a *WaitTime* instruction between the *SCWrite* instructions.
  - The variable value is to large, decrease the size of the ARRAY or RECORD
  - The error message will be:
    ```
    41473 System access error
    Failed to send YYYYYY
    ```
    Where YYYY is the name of the variable.

When an error occurs the program halts and must be restarted. The *ERRNO* system variable will contain the value *ERR_SC_WRITE*.

The SCWrite instruction will not return an error if the client application may for example be closed down or the communication is down. The program will continue executing.

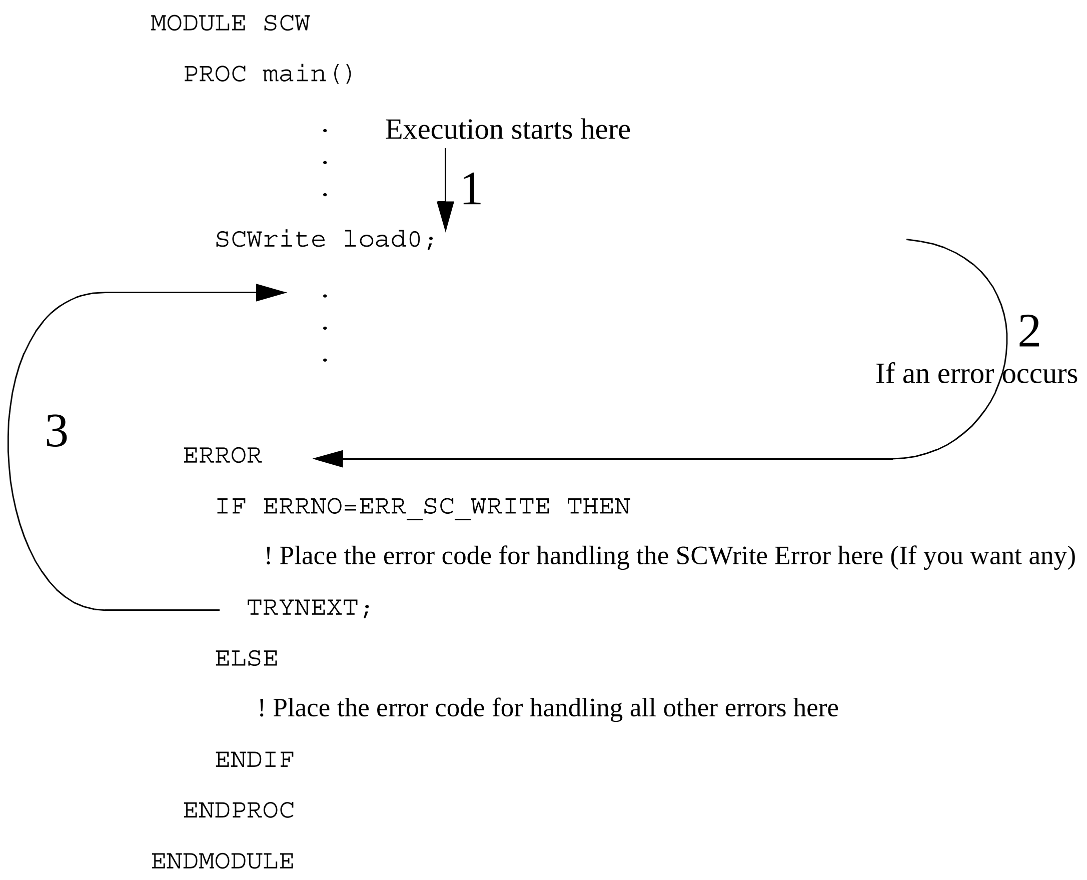
SCWrite error recovery

To avoid stopping the program when a error occurs in a SCWrite instruction it have to be handled by an *error handler.* The error will then only be reported to the log and the program will continue running.

Remember that the error handling will make it more difficult to find errors in the client communication since the error is never reported to the display on the FlexPendant (but it can be found in the log).

Using RobotWare 5.0 or later

The RAPID program looks as follows:.



```
MODULE SCW
  PROC main()
            .
            .
            .
    SCWrite load0;
            .
            .
            .

  ERROR
    IF ERRNO=ERR_SC_WRITE THEN
       ! Place the error code for handling the SCWrite Error here (If you want any)
      TRYNEXT;
    ELSE
       ! Place the error code for handling all other errors here
    ENDIF
  ENDPROC
ENDMODULE
```

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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

When using search instructions, it is important to configure the I/O system to have very short time from setting the physical signal to the system get the information about the setting (use I/O unit with interrupt control, not poll control). How to do this can differ between fieldbuses. If using DeviceNet, the ABB units DSQC 327A (AD Combi I/O) and DSQC 328A (Digital IO) will give short times, since they are using connection type Change of State. If using other fieldbuses make sure to configure the network in a proper way to get right conditions.

Examples

SearchC di1, 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 *di1* changes to active, the position is stored in *sp*.

SearchC \Stop, di2, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10*. When the value of the signal *di2* changes to active, the position is stored in *sp* and the robot stops immediately.

Arguments

SearchC [\Stop] | [\PStop] | [\SStop] | [\Sup] Signal [\Flanks] SearchPoint CirPoint ToPoint [\ID] Speed [\V] | [\T] Tool [\WObj] [\Corr]

[\Stop] *(Stiff 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.

⚠ To stop the searching with stiff stop (switch \Stop) is only allowed if the TCP-speed is lower than 100 mm/s. At stiff stop with higher speed, some axes can move in unpredictable direction.

[\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 distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\SStop] *(Soft Stop)* **Data type:** *switch*

The robot movement is stopped as quickly as possible, while keeping the TCP close to or on the path (soft stop), when the value of the search signal changes to active. However, the robot is moved only a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed. *SStop* is faster then *PStop*. But when the robot is running faster than 100 mm/s, it stops in the direction of the tangent of the movement which causes it to marginally slide of the path.

[\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*, *\SStop 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.

[\ID] *(Synchronization id)* **Data type:** *identno*

This argument must be used in a MultiMove System, if coordinated synchronized movement, and is not allowed in any other cases.

The specified id number must be the same in all cooperating program tasks. The id number gives a guarantee that the movements are not mixed up at runtime.

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*, *\PStop* or *\SStop*, 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 1.

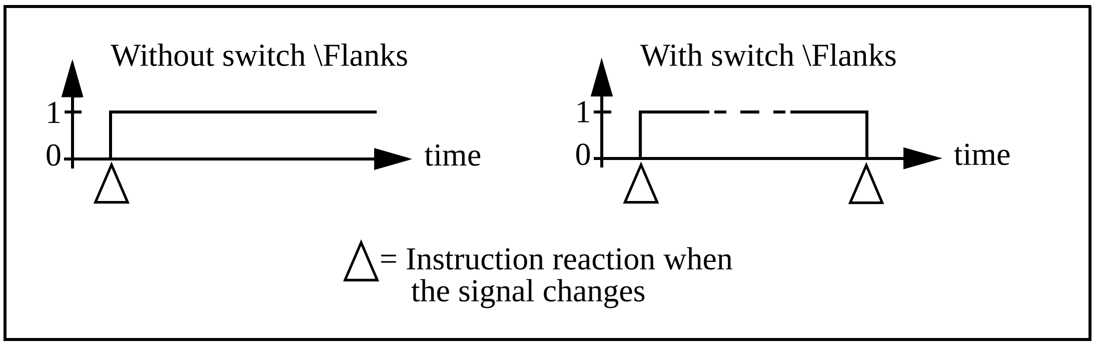


Figure 1 Flank-triggered signal detection (the position is stored when the signal is changed the first time only).

Example

SearchC \Sup, di1\Flanks, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10*. When the value of the signal *di1* changes to active or passive, the position is stored in *sp*. If the value of the signal changes twice, program execution stops.

Limitations

General limitations according to instruction MoveC.

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 2 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).

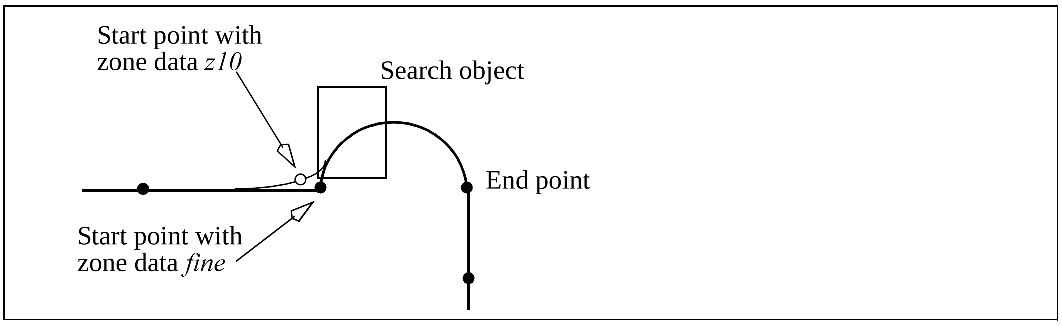


Figure 2 A match is made on the wrong side of the object because the wrong zone data was used.

Repetition accuracy for search hit position with TCP speed 20 - 1000 mm/s
0.1 - 0.3 mm.

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) 15-25 mm
- with TCP near path (switch \SStop) 4-8 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 / Instruction 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 backward
During backward execution, the instruction just carries out the movement without any signal supervision.

Syntax

SearchC
  [’\’ Stop’,’] | [’\’ PStop ’,’] | [’\’ SStop ’,’] | [’\’ Sup ’,’]
  [Signal ’:=’] < variable (**VAR**) of *signaldi* >
  [‘\’ Flanks]’,’
  [SearchPoint ’:=’] < var or pers (**INOUT**) of *robtarget* > ’,’
  [CirPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
  [ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
  [’\’ ID ’:=’ < expression (**IN**) of *identno* >]’,’
  [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

Table 2

	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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

When using search instructions, it is important to configure the I/O system to have very short time from setting the physical signal to the system get the information about the setting (use I/O unit with interrupt control, not poll control). How to do this can differ between fieldbuses. If using DeviceNet, the ABB units DSQC 327A (AD Combi I/O) and DSQC 328A (Digital IO) will give short times, since they are using connection type Change of State. If using other fieldbuses make sure to configure the network in a proper way to get right conditions.

Examples

```
SearchL di1, 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 *di1* changes to active, the position is stored in *sp*.

```
SearchL \Stop, di2, sp, p10, v100, probe;
```

The TCP of the *probe* is moved linearly towards the position *p10*. When the value of the signal *di2* changes to active, the position is stored in *sp* and the robot stops immediately.

Arguments

SearchL [\Stop] | [\PStop] | [\SStop] | [\Sup] Signal [\Flanks] SearchPoint ToPoint [\ID] Speed [\V] | [\T] Tool [\WObj] [\Corr]

[\Stop] *(Stiff 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.

⚠ To stop the searching with stiff stop (switch \Stop) is only allowed if the TCP-speed is lower than 100 mm/s. At stiff stop with higher speed, some axes can move in unpredictable direction.

[\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 distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\SStop] *(Soft Stop)* **Data type:** *switch*

The robot movement is stopped as quickly as possible, while keeping the TCP close to or on the path (soft stop), when the value of the search signal changes to active. However, the robot is moved only a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed. *SStop* is faster then *PStop*. But when the robot is running faster than 100 mm/s it stops in the direction of the tangent of the movement which causes it to marginally slide off the path.

[\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*, *\SStop* 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.

[\ID] *(Synchronization id)* **Data type:** *identno*

This argument must be used in a MultiMove System, if coordinated synchronized movement, and is not allowed in any other cases.

The specified id number must be the same in all cooperating program tasks. The id number gives a guarantee that the movements are not mixed up at runtime.

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, \PStop or \SStop*, 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 3.

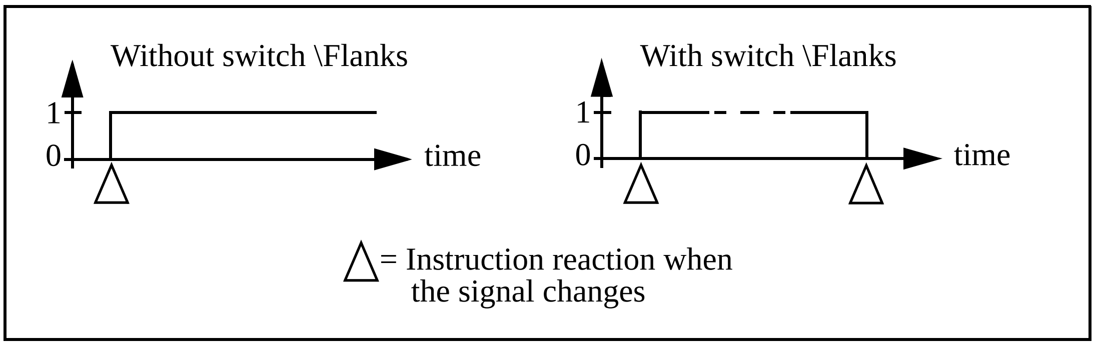


Figure 3 Flank-triggered signal detection (the position is stored when the signal is changed the first time only).

Examples

SearchL \Sup, di1\Flanks, sp, p10, v100, probe;

The TCP of the *probe* is moved linearly towards the position *p10*. When the value of the signal *di1* 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, di1, sp, p10, v100, tool1;
MoveL sp, v100, fine \Inpos := inpos50, 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 *di1* 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 *di1* changes to active, the position is stored in *sp.* The robot is moved back to this point using an accurately defined stop 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 4 to Figure 6 illustrate examples of things that may go wrong when zone data other than *fine* is used.

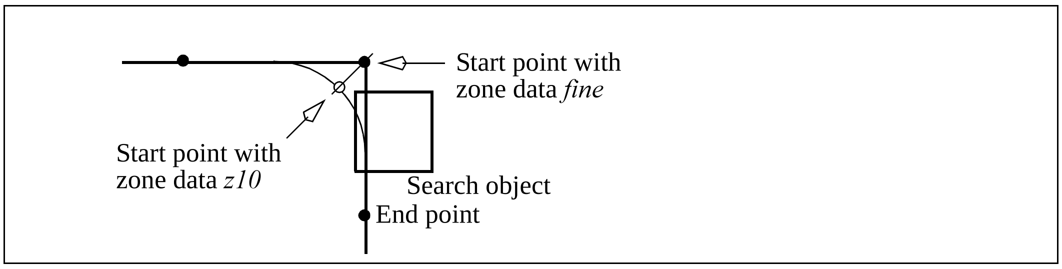


Figure 4 A match is made on the wrong side of the object because the wrong zone data was used.

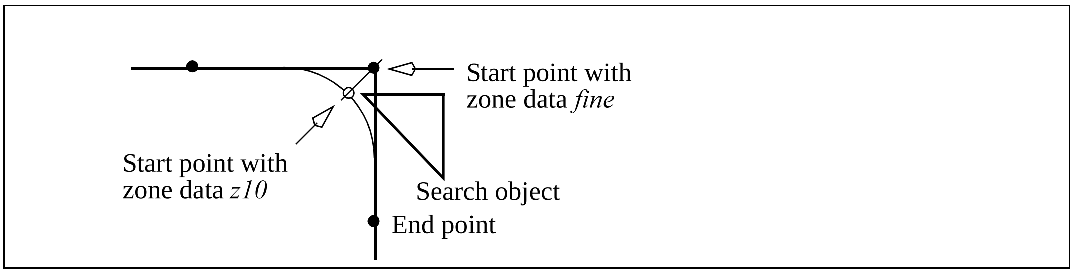


Figure 5 No match detected because the wrong zone data was used.

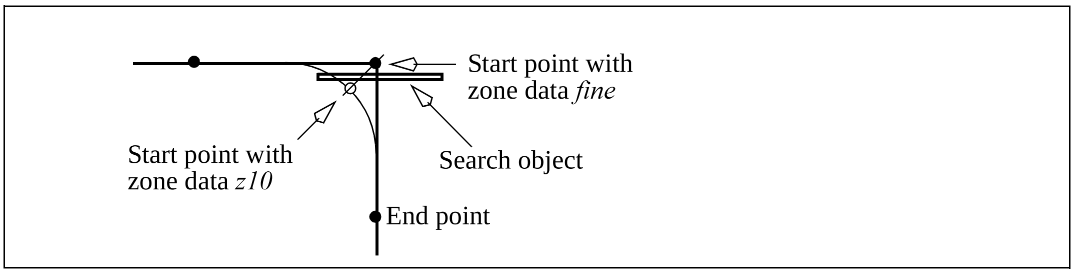


Figure 6 No match detected because the wrong zone data was used.

Repetition accuracy for search hit position with TCP speed 20 - 1000 mm/s
0.1 - 0.3 mm.

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*) 15-25 mm
- with TCP near path (switch *SStop*) 4-8 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 / Instruction 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 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, di1, 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 ’,’] | [’\’ SStop ’,’] | [’\’ Sup ’,’]
  [Signal ’:=’] < variable (**VAR**) of *signaldi* >
  [‘\’ Flanks] ’,’
  [SearchPoint ’:=’] < var or pers (**INOUT**) of *robtarget* > ’,’
  [ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
  [’\’ ID ’:=’ < expression (**IN**) of *identno* >]’,’
  [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

Table 3

	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

There is a short delay before the signal physically gets its new value. If you do not want the program execution to continue until the signal has got its new value, you can use the instruction *SetDO* with the optional parameter *\Sync*.

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.

Error handling

Following recoverable error can be generated. The error can be handled in an error handler. The system variable ERRNO will be set to:

ERR_NORUNUNIT	if there is no contact with the unit

Syntax

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

Related information

Table 4

	Described in:
Setting a digital output signal to zero	Instructions - *Reset*
Change the value of a digital output signal	Instruction - *SetDO*
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

SetAllDataVal - Set a value to all data objects in a defined set

SetAllDataVal (Set All Data Value) make it possible to set a new value to all data objects of a certain type that match the given grammar.

Example

```
VAR mydata mydata0:=0;
...
SetAllDataVal "mydata"\TypeMod:="mytypes"\Hidden,mydata0;
```

This will set all data objects of data type *mydata* in the system to the same value as the variable *mydata0* has (in the example to *0*). The user defined data type *mydata* is defined in the module *mytypes*.

Arguments

SetAllDataVal Type [\TypeMod] [\Object] [\Hidden] Value

Type **Data type:** *string*

The type name of the data objects to be set.

[\TypeMod] *(Type Module)* **Data type:** *string*

The module name where the data type is defined, if using user defined data types.

[\Object] **Data type:** *string*

The default behaviour is to set all data object of the data type above, but this option make it possible to name one or several objects with a regular expression. (see also *SetDataSearch*)

[\Hidden] **Data type:** *switch*

This match also data objects that are in routines (routine data or parameters) hidden by some routine in the call chain.

Value **Data type:** *anytype*

Variable which holds the new value to be set. The data type must be same as the data type for the object to be set.

Program running

The instruction will fail if the specification for *Type* or *TypeMod* is wrong.

If the matching data object is an array, all elements of the array will be set to the specified value.

If the matching data object is read-only data, the value will not be changed.

If the system don’t have any matching data objects the instruction will accept it and return successfully.

Limitations

For a semivalue data type, it is not possible to search for the associated value data type. E.g. if searching for *dionum* no search hit for signals *signaldi* and if searching for num no search hit for signal *signalgi* or *signalai*.

It is not possible to set a value to a variable declared as LOCAL in a built in RAPID module.

Syntax

SetAllDataVal
    [Type ’:=’] < expression (**IN**) of *string* >
    [’\’TypeMod ’:=’<expression (**IN**) of *string*>]
    [’\’Object ’:=’<expression (**IN**) of *string*>]
    [’\’Hidden] ’,’
    [’\’Value ’:=’] <variable (**VAR**) of *anytype*>’;’

Related information

Table 5

	Described in:
Define a symbol set in a search session	Instructions - *SetDataSearch*
Get next matching symbol	Functions - *GetNextSym*
Get the value of a data object	Instructions - *GetDataVal*
Set the value of a data object	Instructions - *SetDataVal*
The related data type datapos	Data Types - *datapos*

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 7.

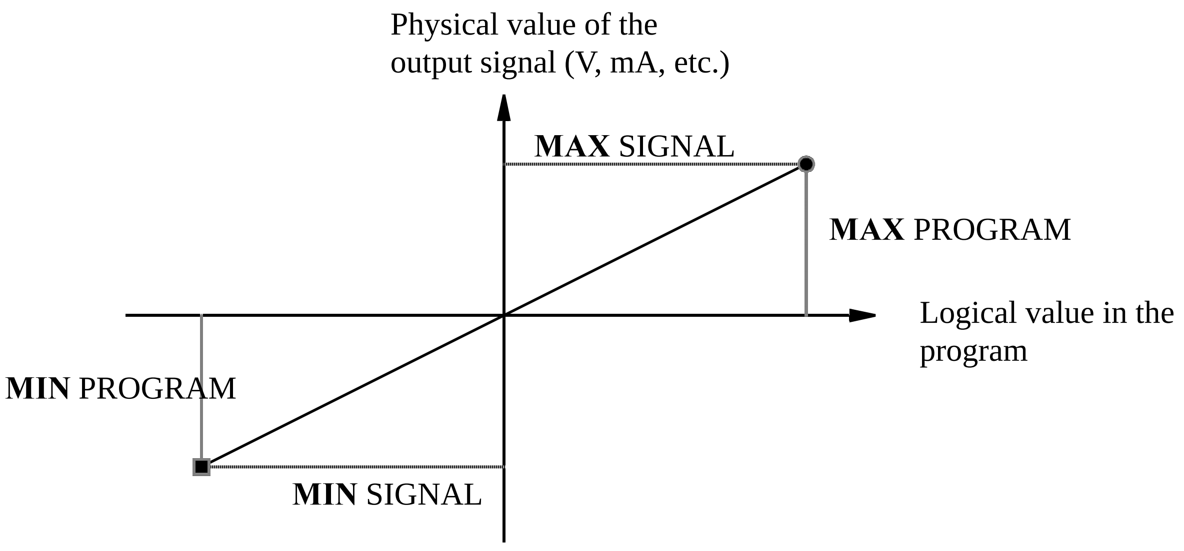

Physical value of the output signal (V, mA, etc.)
MAX SIGNAL
MAX PROGRAM
Logical value in the program
MIN PROGRAM
MIN SIGNAL

Figure 7 Diagram of how analog signal values are scaled.

Error handling

Following recoverable error can be generated. The error can be handled in an error handler. The system variable ERRNO will be set to:

ERR_NORUNUNIT	if there is no contact with the unit

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

Table 6

	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

SetDataSearch - Define the symbol set in a search sequence

SetDataSearch is used together with *GetNextSym* to retrieve data objects from the system.

Example

```
VAR datapos block;
VAR string name;
...
SetDataSearch “robtarget”\InTask;
WHILE GetNextSym(name,block \Recursive) DO
```

This session will find all *robtarget*’s object in the task.

Arguments

SetDataSearch **Type [\TypeMod] [\Object] [\PersSym] [\VarSym][\ConstSym] [\InTask] | [\InMod] [\InRout][\GlobalSym] | [\LocalSym]**

Type **Data type:** *string*

The data type name of the data objects to be retrieved.

[\TypeMod] *(Type Module)* **Data type:** *string*

The module name where the data type is defined, if using user defined data types.

[\Object] **Data type:** *string*

The default behaviour is to set all data object of the data type above, but this option makes it possible to name one or several data objects with a regular expression.

A regular expression is a powerful mechanism to specify a grammar to match the data object names. The string could consist of either ordinary characters and metacharacters. A metacharacter is a special operator used to represent one or more ordinary characters in the string, with the purpose to extend the search. It is possible to see if a string matches a specified pattern as a whole, or search within a string for a substring matching a specified pattern.

Within a regular expression, all alphanumeric characters match themselves, that is to say, the pattern “abc” will only match a data object named “abc”. To match all data object names containing the character sequence “abc”, it is necessary to add some metacharacters. The regular expression for this is “.*abc.*”.

The available metacharacter set is shown below.

Expression	Meaning
.	Any single character.
[s]	Any single character in the non-empty set s, where s is a sequence of characters. Ranges may be specified as c-c.
[^s]	Any singlecharacter not in the set s.
r*	Zero or more occurrences of the regular expression r.
r+	One or more occurrences of the regular expression r.
r?	Zero or one occurrence of the regular expression r.
(r)	The regular expression r. Used for separate that regular expression from another.
r | r'	The regular expressions r or r'.
.*	Any character sequence (zero, one or several characters).

The default behaviour is to accept any symbols, but if one or several of following *PersSym*, *VarSym* or *ConstSym* is specified, only symbols that match the specification are accepted:

[\PersSym] *(Persistent Symbols)* **Data type:** *switch*

Accept persistent variable (PERS) symbols.

[\VarSym] *(Variable Symbols)* **Data type:** *switch*

Accept variable (VAR) symbols.

[\ConstSym] *(Constant Symbols)* **Data type:** *switch*

Accept constant (CONST) symbols.

If no one of the flags *\InTask* or *\InMod* are specified, the search is started at system level. The system level is the root to all other symbol definitions in the symbol tree. At the system level all built in symbols are located (built in symbols declared LOCAL will NOT be found) plus the handle to the task level. At the task level all loaded global symbols are located plus the handle to the modules level.

If the *\Recursive* flag is set in *GetNextSym,* the search session will enter all loaded modules and routines below the system level.

[\InTask] *(In Task)* **Data type:** *switch*

Start the search at the task level. At the task level all loaded global symbols are located plus the handle to the modules level.

If the *\Recursive* flag is set in *GetNextSym,* the search session will enter all loaded modules and routines below the task level.

[\InMod] *(In Module)* **Data type:** *string*

Start the search at the specified module level. At the module level all loaded global and local symbols declared in the specified module are located plus the handle to the routines level.

If the *\Recursive* flag is set in *GetNextSym,* the search session will enter all loaded routines below the specified module level (declared in the specified module).

[\InRout] *(In Routine)* **Data type:** *string*

Search only at the specified routine level.

The module name for the routine must be specified in the argument *\InMod.*

The default behaviour is to match both local and global module symbols, but if one of following *\GlobalSym* or *\LocalSym* is specified, only symbols that match the specification is accepted:

[\GlobalSym] *(Global Symbols)* **Data type:** *switch*

Skip local module symbols.

Limitation: Global symbols that are built in will NOT be given.

[\LocalSym] *(Local Symbols)* **Data type:** *switch*

Skip global module symbols.

Limitation: Global symbols that are built in will be given, but local symbols that are built in will NOT be given.

Program running

The instruction will fail if the specification for one of *Type*, *TypeMod*, *InMod* or *InRout* is wrong.

If the system doesn’t have any matching objects the instruction will accept it and return successfully, but the first *GetNextSym* will return FALSE.

Limitations

Array data objects can not be defined in the symbol search set and can not be find in a search sequence.

For a semivalue data type, it is not possible to search for the associated value data type. E.g. if searching for *dionum* no search hit for signals *signaldi* and if searching for num no search hit for signal *signalgi* or *signalai*.

Built in symbols declared as LOCAL will not be found and built in symbols declared global will be found as local.

Syntax

SetDataSearch
    [Type ':='] < expression (**IN**) of *string* >
    ['\'TypeMod ':='<expression (**IN**) of *string*>]
    ['\'Object ':='<expression (**IN**) of *string*>]
    ['\'PersSym]
    ['\'VarSym]
    ['\'ConstSym]
    ['\'InTask] | ['\'InMod ':='<expression (**IN**) of *string*>]
    ['\'InRout ':='<expression (**IN**) of *string*>]
    ['\'GlobalSym] | ['\'LocalSym] ';'

Related information

Table 7

	Described in:
Get next matching symbol	Functions - *GetNextSym*
Get the value of a data object	Instructions - G*etDataVal*
Set the value of a data object	Instructions - *SetDataVal*
Set the value of many data objects	Instructions - *SetAllDataVal*
The related data type datapos	Data Types - *datapos*

SetDataVal - Set the value of a data object

SetDataVal (Set Data Value) makes it possible to set a value for a data object that is specified with a string variable.

Example

```
VAR num value:=3;
...
SetDataVal “reg”+ValToStr(ReadNum(mycom)),value;
```

This will set the value *3* to a register, the number of which is received from the serial channel *mycom*.

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

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

Arguments

SetDataVal Object [\Block] Value

Object **Data type:** *string*

The name of the data object.

[\Block] **Data type:** *datapos*

The enclosed block to the data object. This can only be fetched with the *GetNextSym* function.

If this argument is omitted, the value of the visible data object in the current program execution scope will be set. No array data objects will be found.

Value **Data type:** *anytype*

Variable which holds the new value to be set. The data type must be the same as the data type for the data object to be set. The set value must be fetched from a variable, but can be stored in a constant, variable or persistent.

Error handling

The system variable ERRNO is set to ERR_SYM_ACCESS if:

- the data object is non-existent
- the data object is read-only data
- the data object is routine data or routine parameter and not located in the current active routine

The error can be handled in the error handler of the routine.

Limitations

Array data objects cannot be defined in the symbol search set and cannot be found in a search sequence.

For a semivalue data type, it is not possible to search for the associated value data type. E.g. if searching for *dionum*, no search hit for signals *signaldi* will be obtained and if searching for num, no search hit for signals *signalgi* or *signalai* will be obtained.

It is not possible to set a value to a variable declared as LOCAL in a built in RAPID module.

Syntax

SetDataVal
    [Object ':='] < expression (**IN**) of *string* >
    ['\'Block ':='<variable (**VAR**) of *datapos*>] ','
    [Value ':='] <variable (**VAR**) of *anytype*>]';'

Related information

Table 8

	Described in:
Define a symbol set in a search session	Instructions - *SetDataSearch*
Get next matching symbol	Functions - *GetNextSym*
Get the value of a data object	Instructions - *GetDataVal*
Set the value of many data objects	Instructions - *SetAllDataVal*
The related data type datapos	Data Types - *datapos*

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 or synchronisation.

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.

SetDO \Sync ,do1, 0;

The signal *do1* is set to *0*. Program execution waits until the signal is physically set to the specified value.

Arguments

SetDO [\SDelay]|[\Sync] Signal Value

[\SDelay] *(Signal Delay)* **Data type:** *num*

Delays the change for the amount of time given in seconds (max. 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.

[\Sync] *(Synchronisation)* **Data type:** *switch*

If this argument is used, the program execution will wait until the signal is physically set to the specified value.

If neither of the arguments *\SDelay* or *\Sync* are used, the signal will be set as fast as possible and the next instruction will be executed at once, without waiting for the signal to be physically set.

Signal **Data type:** *signaldo*

The name of the signal to be changed.

Value **Data type:** *dionum*

The desired value of the signal 0 or 1.

Table9 System interpretation of specified Value

Specified *Value*	Set digital output to
0	0
Any value except 0	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.

Error handling

Following recoverable error can be generated. The error can be handled in an error handler. The system variable ERRNO will be set to:

ERR_NORUNUNIT	if there is no contact with the unit

Syntax

SetDO
    [’\’ SDelay ’:=’ < expression (**IN**) of *num* > ’,’] |[’\’ Sync ’,’]
    [Signal ’:=’] < variable (**VAR**) of *signaldo* > ’,’
    [Value ’:=’] < expression (**IN**) of *dionum* > ’;’

Related information

Table 10

	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*

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 (max. 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
1	0 - 1
2	0 - 3
3	0 - 7
4	0 - 15
5	0 - 31
6	0 - 63
7	0 - 127
8	0 - 255
9	0-511
10	0-1023
11	0-2047
12	0 - 4095
13	0 - 8191
14	0 - 16383
15	0 - 32767
16	0 - 65535
17	0 - 131071
18	0 - 262143
19	0 - 524287
20	0 - 1048575
21	0 - 2097151
22	0 - 4194303
23	0 - 8388607

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.

Limitations

Maximum number of signals that can be used for a group is 23. This limitation is valid for all instructions and functions using group signals.

Error handling

Following recoverable error can be generated. The error can be handled in an error handler. The system variable ERRNO will be set to:

ERR_NORUNUNIT	if there is no contact with the unit

Syntax

SetDO
  ['\' SDelay ':=' < expression (**IN**) of *num* > ',']
  [Signal ':='] < variable (**VAR**) of *signalgo* > ','
  [Value ':='] < expression (**IN**) of *num* > ';'

Related information

Table 11

	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

SetSysData - Set system data

SetSysData activates the specified system data name for the specified data type.

With this instruction it is possible to change the current active Tool, Work Object or PayLoad (for robot).

Example

SetSysData tool5;

The tool *tool5* is activated.

SetSysData tool0 \ObjectName := "tool6";

The tool *tool6* is activated.

SetSysData anytool \ObjectName := "tool2";

The tool *tool2* is activated.

Arguments

SetSysData SourceObject [\ObjectName]

SourceObject **Data type:** *anytype*

Persistent, which name should be active as current system data name.

The data type of this argument also specifies the type of system data (Tool, Work Object or PayLoad) to be activated.

The value of this argument is not affected.
The value of the current system data is not affected.

[\ObjectName] **Data type:** *string*

If this optional argument is specified, it specifies the name of the data object to be active (overrides name specified in argument *SourceObject*). The data type of the data object to be active is always fetched from the argument *SourceObject*.

Program execution

The current active system data object for the Tool, Work Object or PayLoad is set according to the arguments.

Note that this instruction only activates a new data object (or the same as before) and never changes the value of any data object.

Syntax

```
SetSysData
    [ SourceObject':='] < persistent(PERS) of anytype>
    ['\'ObjectName':=' < expression(IN) of string> ] ';'
```

Related information

Table 12

	Described in:
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Get system data	Instructions - *GetSysData*

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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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

Table 13

	Described in:
Singularity	Motion Principles- *Singularity*
Interpolation	Motion Principles - *Positioning during Program Execution*

SkipWarn - Skip the latest warning

SkipWarn (Skip Warning) is used to skip the latest requested warning message to be stored in the Service Log during execution in running mode continuously or cycle (no warnings skipped in FWD or BWD step).

With *SkipWarn* it is possible to repeatedly do error recovery in RAPID without filling the Service Log with only warning messages.

Example

```
    %"notexistingproc"%;
    nextinstruction;
ERROR
    IF ERRNO = ERR_REFUNKPRC THEN
        SkipWarn;
        TRYNEXT;
    ENDIF
ENDPROC
```

The program will execute the *nextinstruction* and no warning message will be stored in the Service Log.

Syntax

```
SkipWarn ';'
```

Related information

Table 14

	Described in:
Error recovery	RAPID Summary - *Error Recovery* Basic Characteristics - *Error Recovery*
Error number	Data Types - *errnum*

SocketAccept - Accept an incoming connection

SocketAccept is used to accept incoming connection requests.
SocketAccept can only used for a server applications.

Examples

```
VAR socketdev server_socket;
VAR socketdev client_socket;
...
SocketCreate server_socket;
SocketBind server_socket, “192.168.0.1”, 1025;
SocketListen server_socket;
SocketAccept server_socket, client_socket;
```

A server socket is created and bound to port *1025* on the controller network address *192.168.0.1*. After execution of *SocketListen,* the server socket start to listen for incoming connections on this port and address. *SocketAccept* waits for any incoming connections, accept the connection request and return a client socket for the established connection.

Arguments

SocketAccept Socket ClientSocket [\ClientAddress] [\Time]

Socket **Data type:** *socketdev*

The server socket that are waiting for incoming connections. The socket must already be created, bounded and ready for listen.

ClientSocket **Data type:** *socketdev*

The returned new client socket, that will be updated with the accepted incoming connection request.

[\ClientAddress] **Data type:** *string*

The variable that will be updated with the IP-address of the accepted incoming connection request.

[\Time] **Data type:** *num*

The maximum amount of time [s] that program execution waits for incoming connections. If this time runs out before any incoming connection, the error handler will be called, if there is one, with the error code ERR_SOCK_TIMEOUT. If there is no error handler, the execution will be stopped.

If parameter *\Time* is not used, the waiting time is 60 s.

Program execution

The server socket will wait for any incoming connection requests. When accepted the incoming connection request, the instruction is ready and the returned client socket is by default connected and can be used in *SocketSend* and *SocketReceive* instructions.

Examples

```
VAR socketdev server_socket;
VAR socketdev client_socket;
VAR string receive_string;
VAR string client_ip;
...
SocketCreate server_socket;
SocketBind server_socket, “192.168.0.1”, 1025;
SocketListen server_socket;
WHILE TRUE DO
    SocketAccept server_socket, client_socket \ClientAddress:=client_ip;
    SocketReceive client_socket \Str := receive_string;
    SocketSend client_socket \Str := “Hello client with ip-address ”+client_ip;
    SocketClose client_socket;
ENDWHILE
ERROR
    RETRY;
UNDO
    SocketClose server_socket;
    SocketClose client_socket;
```

A server socket is created and bound to port *1025* on the controller network address *192.168.0.1*. After execution of *SocketListen,* the server socket start to listen for incoming connections on this port and address.*SocketAccept* will accept the incoming connection from some client and store the client address in the string *client_ip*. Then the server receive a string message from the client and store the message in *receive_string*. Then the server respond with the message “Hello client with ip-address xxx.xxx.x.x” and close the client connection.

After that, the server is ready for a connection from the same or some other client in the WHILE loop. If PP is moved to main in the program, all open sockets are closed (*SocketClose* can always be done, even if the socket is not created).

Error handling

Following recoverable errors can be generated. The errors can be handled in an ERROR handler. The system variable ERRNO will be set to:

ERR_SOCK_TIMEOUT	The connection was not established within the time out time.

Syntax

SocketAccept
[Socket ’:=’] < variable (**VAR**) of *socketdev* > ’,’
[ClientSocket ’:=’] < variable (**VAR**) of *socketdev* >
[’\’ ClientAddress ’:=’ < variable (**VAR**) of *string*>]
[’\’ Time ’:=’ < expression (**IN**) of *num* >] ’;’

Related information

Table 15

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Create a new socket	Instructions - *SocketCreate*
Connect to remote computer (only client)	Instructions - *SocketConnect*
Send data to remote computer	Instructions - *SocketSend*
Receive data from remote computer	Instructions - *SocketReceive*
Close the socket	Instructions - *SocketClose*
Bind a socket (only server)	Instructions - *SocketBind*
Listening connections (only server)	Instructions - *SocketListen*
Get current socket state	Functions - *SocketGetStatus*
Example client socket application	Instructions - *SocketSend*

SocketBind - Bind a socket to a port number

SocketBind is used to bind a socket to the specified server port number and IP-address. *SocketBind* can only used for server applications.

Examples

```
VAR socketdev server_socket;

SocketCreate server_socket;
SocketBind server_socket, “192.168.0.1”, 1025;
```

A server socket is created and bound to port *1025* on the controller network address *192.168.0.1*. The server socket can now be used in an *SocketListen* instruction to listen for incoming connections on this port and address.

Arguments

SocketBind Socket LocalAddress LocalPort

Socket **Data type:** *socketdev*

The server socket to bind. The socket must be created but not already bounded.

LocalAddress **Data type:** *string*

The server network address to bind the socket to. The only valid addresses are any public LAN addresses or the controller service port address, 192.168.125.1.

LocalPort **Data type:** *num*

The server port number to bind the socket to. Generally ports 1025-5000 are free to use. Ports below 1025 can already be taken.

Program execution

The server socked is bounded to the specified server port and IP-address.

An error is generated, if the specified port is already in use.

Syntax

```
SocketBind
    [ Socket ':=' ] < variable (VAR) of socketdev > ','
    [ LocalAddress ':=' ] < expression (IN) of string > ','
    [ LocalPort ':=' ] < expression (IN) of num > ';'
```

Related information

Table 16

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Create a new socket	Instructions - *SocketCreate*
Connect to remote computer (only client)	Instructions - *SocketConnect*
Send data to remote computer	Instructions - *SocketSend*
Receive data from remote computer	Instructions - *SocketReceive*
Close the socket	Instructions - *SocketClose*
Listening connections (only server)	Instructions - *SocketListen*
Accept connections (only server)	Instructions - *SocketAccept*
Get current socket state	Functions - *SocketGetStatus*
Example client socket application	Instructions - *SocketSend*
Example server socket application	Instructions - *SocketAccept*

SocketClose - Close a socket

SocketClose is used when a socket connection is no longer going to be used.

After that a socket has been closed, it can not be used in any socket call except *SocketCreate*.

Examples

SocketClose socket1;

The socket is closed and can not be used anymore.

Arguments

SocketClose Socket

Socket **Data type:** *socketdev*

The socket to be closed.

Program execution

The socket will be closed and it’s allocateed resources will be released.

Any socket can be closed at any time. The socket can not be used after closing. It can however be reused for a new connection after a call to *SocketCreate*.

Syntax

SocketClose
    [Socket ’:=’] < variable (**VAR**) of *socketdev* > ’;’

Related information

Table 17

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Create a new socket	Instructions - *SocketCreate*
Send data to remote computer	Instructions - *SocketSend*
Receive data from remote computer	Instructions - *SocketReceive*
Close the socket	Instructions - *SocketClose*
Bind a socket (only server)	Instructions - *SocketBind*
Listening connections (only server)	Instructions - *SocketListen*
Accept connections (only server)	Instructions - *SocketAccept*
Get current socket state	Functions - *SocketGetStatus*
Example client socket application	Instructions - *SocketSend*
Example server socket application	Instructions - *SocketAccept*

SocketConnect - Connect to a remote computer

SocketConnect is used to connect the socket to a remote computer in a client application.

Examples

```
SocketConnect socket, “192.168.0.1“, 1025;
```

Trying to connect to a remote computer at ip-address 192.168.0.1 and port 1025.

Arguments

SocketConnect Socket Address Port [\Time]

Socket **Data type:** *socketdev*

The client socket to connect. The socket must be created but not already connected.

Address **Data type:** *string*

The address of the remote computer. The remote computer must be specified as an IP address. It is not possible to use the name of the remote computer.

Port **Data type:** *num*

The port on the remote computer. Must be an integer between 1 and 65535, e.g. 80 for an web server connection.

[\Time] **Data type:** *num*

The maximum amount of time [s] that program execution waits for the connection to be accepted or denied. If this time runs out before that condition is met, the error handler will be called, if there is one, with the error code ERR_SOCK_TIMEOUT. If there is no error handler, the execution will be stopped.

If parameter *\Time* is not used, the waiting time is 60 s.

Program execution

The socket tries to connect to the remote computer on the specified address and port. The program execution will wait until the connection is established, failed or an timeout occur.

Examples

```
VAR num retry_no := 0;
VAR socketdev my_socket;
...
SocketCreate my_socket;
SocketConnect my_socket, “192.168.0.1”, 1025;
...
ERROR
    IF ERRNO = ERR_SOCK_TIMEOUT THEN
        IF retry_no < 5 THEN
            WaitTime 1;
            retry_no := retry_no + 1;
            RETRY;
        ELSE
            RAISE;
        ENDIF
    ENDIF
```

A socket is created and tries to connect to a remote computer. If the connection is not established within the default time out time, i.e. 60 seconds, the error handler retries to connect. Four retries are done then the error is reported to the user.

Error handling

Following recoverable errors can be generated. The errors can be handled in an ERROR handler. The system variable ERRNO will be set to:

ERR_SOCK_CLOSED	The socket is closed (has been closed or is not created). Use *SocketCreate* to create a new socket.
ERR_SOCK_ISCON	The socket is already connected and can be used for communication. To change the connection, use *SocketClose* to close the current connection, then create a new socket with *SocketCreate* and try to connect again.
ERR_SOCK_CONNREF	The connection was refused by the remote computer.
ERR_SOCK_TIMEOUT	The connection was not established within the time out time.

Syntax

```
SocketConnect
   [ Socket ’:=’ ] < variable (VAR) of socketdev > ’,’
   [ Address ’:=’ ] < expression (IN) of string > ’,’
   [ Port ’:=’ ] < expression (IN) of num >
   [ ’\’ Time ‘:=’ < expression (IN) of num > ] ’;’
```

Related information

Table 18

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Create a new socket	Instructions - *SocketCreate*
Connect to remote computer (only client)	Instructions - *SocketConnect*
Send data to remote computer	Instructions - *SocketSend*
Receive data from remote computer	Instructions - *SocketReceive*
Bind a socket (only server)	Instructions - *SocketBind*
Listening connections (only server)	Instructions - *SocketListen*
Accept connections (only server)	Instructions - *SocketAccept*
Get current socket state	Functions - *SocketGetStatus*
Example client socket application	Instructions - *SocketSend*
Example server socket application	Instructions - *SocketAccept*

SocketCreate - Create a new socket

SocketCreate is used to create a new socket for connection based communication.

The socket messaging is of type stream protocol TCP/IP with delivery guarantee. Both server and client application can be developed. Datagram protocol UDP/IP with broadcast is not supported.

Examples

```
VAR socketdev socket1;
...
SocketCreate socket1;
```

A new socket device is created and assigned into the variable *socket1*.

Arguments

SocketCreate Socket

Socket **Data type:** *socketdev*

The variable for storage of system internal socket data.

Program execution

The instruction create a new socket device.

The socket must not already be in use.
The socket is in use between *SocketCreate* and *SocketClose*.

Limitations

Any number of sockets can be declared but it is only possible to use 8 sockets at the same time.

Syntax

SocketCreate
    [Socket ':='] < variable (**VAR**) of *socketdev* > ';'

Related information

Table 19

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Connect to remote computer (only client)	Instructions - *SocketConnect*
Send data to remote computer	Instructions - *SocketSend*
Receive data from remote computer	Instructions - *SocketReceive*
Close the socket	Instructions - *SocketClose*
Bind a socket (only server)	Instructions - *SocketBind*
Listening connections (only server)	Instructions - *SocketListen*
Accept connections (only server)	Instructions - *SocketAccept*
Get current socket state	Functions - *SocketGetStatus*
Example client socket application	Instructions - *SocketSend*
Example server socket application	Instructions - *SocketAccept*

SocketListen - Listen for incoming connections

SocketListen is used to start listen for incoming connections, i.e. start act as a server. *SocketListen* can only used for a server applications.

Examples

```
VAR socketdev server_socket;
...
SocketCreate server_socket;
SocketBind server_socket, “192.168.0.1”, 1025;
SocketListen server_socket;
```

A server socket is created and bound to port *1025* on the controller network address *192.168.0.1*. After execution of *SocketListen,* the server socket start to listen for incoming connections on this port and address.

Arguments

SocketListen Socket

Socket **Data type:** *socketdev*

The server socket that should start listen for incoming connections. The socket must already be created and bounded.

Program execution

The server socket start listen for incoming connections. When the instruction is ready, the socket is ready to accept an incoming connection.

Syntax

SocketListen
  [Socket ’:=’] < variable (**VAR**) of *socketdev* > ’;’

Related information

Table 20

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Create a new socket	Instructions - *SocketCreate*
Connect to remote computer (only client)	Instructions - *SocketConnect*
Send data to remote computer	Instructions - *SocketSend*
Receive data from remote computer	Instructions - *SocketReceive*
Close the socket	Instructions - *SocketClose*
Bind a socket (only server)	Instructions - *SocketBind*
Accept connections (only server)	Instructions - *SocketAccept*
Get current socket state	Functions - *SocketGetStatus*
Example client socket application	Instructions - *SocketSend*
Example server socket application	Instructions - *SocketAccept*

SocketReceive - Receive data from remote computer

SocketReceive is used for receiving data from a remote computer.
SocketReceive can be used both for client and server applications.

Examples

```
VAR string str_data;

SocketReceive socket \Str := str_data;
```

Receive data from a remote computer and store it in the string variable *str_data*.

Arguments

SocketReceive Socket [\Str] | [\RawData] | [\Data] [\Time]

Socket **Data type:** *socketdev*

In client application the socket which receive the data, the socket must already be created and connected.

In server application the socket which receive the data, the socket must already be accepted.

[\Str] **Data type:** *string*

The variable, in which the received *string* data should be stored.
Max. number of characters 80 can be handled.

[\RawData] **Data type:** *rawbytes*

The variable, in which the received *rawbytes* data should be stored.
Max. number of *rawbytes* 1024 can be handled.

[\Data] **Data type:** *array of byte*

The variable, in which the received *byte* data should be stored.
Max. number of *byte* 1024 can be handled.

Only one of the option parameters *\Str*, *\RawData* and *\Data* can be used at the same time.

[\Time] **Data type:** *num*

The maximum amount of time [s] that program execution waits for the data to be received. If this time runs out before the data is transferred, the error handler will be called, if there is one, with the error code ERR_SOCK_TIMEOUT. If there is no error handler, the execution will be stopped.

If parameter *\Time* is not used, the waiting time is 60 s.

Program execution

The execution of *SocketReceive* will wait until the data is available or fail with a time-out error.

Examples

```
VAR socketdev client_socket;
VAR string receive_string;
...
SocketCreate client_socket;
SocketConnect client_socket, “192.168.0.2”, 1025;
SocketSend client_socket \Str := “Hello server”;
SocketReceive client_socket \Str := receive_string;
...
SocketClose client_socket;
```

This is an example of a client program.
A client socket is created and connected to a remote computer server with IP-address *192.168.0.2* on port *1025*.
Then the client send “*Hello server*” to the server and the server respond with “Hello client” to the client, which is stored in the variable *receive_string*.

Error handling

Following recoverable errors can be generated. The errors can be handled in an ERROR handler. The system variable ERRNO will be set to:

ERR_SOCK_CLOSED	The socket is closed.
ERR_SOCK_TIMEOUT	No data was received within the time out time.

Limitations

The maximum size of the data that can be received in one call is limited to 1024 bytes.

Syntax

```
SocketReceive
    [ Socket ':=' ] < variable (VAR) of socketdev >
    [ \Str ':=' < variable (VAR) of string > ]
    | [ \RawData ':=' < variable (VAR) of rawbytes > ]
    | [ \Data ':=' < array {*} (VAR) of byte > ]
    [ '\' Time ':=' < expression (IN) of num > ] ';'
```

Related information

Table 21

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Create a new socket	Instructions - *SocketCreate*
Connect to remote computer (only client)	Instructions - *SocketConnect*
Send data to remote computer	Instructions - *SocketSend*
Close the socket	Instructions - *SocketClose*
Bind a socket (only server)	Instructions - *SocketBind*
Listening connections (only server)	Instructions - *SocketListen*
Accept connections (only server)	Instructions - *SocketAccept*
Get current socket state	Functions - *SocketGetStatus*
Example server socket application	Instructions - *SocketAccept*

SocketSend - Send data to remote computer

SocketSend is used to send data to a remote computer.
SocketSend can be used both for client and server applications.

Examples

```
SocketSend socket1 \Str := “Hello world”;
```

Sends the message “*Hello world*” to the remote computer.

Arguments

SocketSend Socket [\Str] | [\RawData] | [\Data] [\NoOfBytes]

Socket **Data type:** *socketdev*

In client application the socket to send from, the socket must already be created and connected.

In server application the socket to send upon, the socket must already be accepted.

[\Str] **Data type:** *string*

The *string* to send to the remote computer.

[\RawData] **Data type:** *rawbytes*

The *rawbytes* data to send to the remote computer.

[\Data] **Data type:** *array of byte*

The data in the *byte* array to send to the remote computer.

Only one of the option parameters *\Str, \RawData* and *\Data* can be used at the same time.

[\NoOfBytes] **Data type:** *num*

If this argument is specified, only this number of bytes will be sent to the remote computer. The call to *SocketSend* will fail if *\NoOfBytes* is larger the actual number of bytes in the data structure to send.

Program execution

The specified data is sent to the remote computer. If the connection is broken an error is generated.

Examples

```
VAR socketdev client_socket;
VAR string receive_string;
...
SocketCreate client_socket;
SocketConnect client_socket, “192.168.0.2”, 1025;
SocketSend client_socket \Str := “Hello server”;
SocketReceive client_socket \Str := receive_string;
...
SocketClose client_socket;
```

This is an example of a client program.
A client socket is created and connected to a remote computer server with IP-address *192.168.0.2* on port *1025*.
Then the client send “*Hello server*” to the server and the server respond with “Hello client” to the client, which is stored in the variable *receive_string*.

Error handling

Following recoverable errors can be generated. The errors can be handled in an ERROR handler. The system variable ERRNO will be set to:

ERR_SOCK_CLOSED	The socket is closed.

Limitations

The size of the data to send is limited to 1024 bytes.

Syntax

SocketSend
  [Socket ’:=’] < variable (**VAR**) of *socketdev* >
  [\Str ’:=’ < expression (**IN**) of *string* >]
  | [\RawData ’:=’ < variable (**VAR**) of *rawdata* >]
  | [\Data ’:=’ < array {*} (**IN**) of *byte* >]
  [’\’ NoOfBytes ‘:=’ < expression (**IN**) of *num* >] ’;’

Related information

Table 22

	Described in:
Socket communication in general	Application manual - Robot communication and I/O control
Create a new socket	Instructions - *SocketCreate*
Connect to remote computer (only client)	Instructions - *SocketConnect*
Receive data from remote computer	Instructions - *SocketReceive*
Close the socket	Instructions - *SocketClose*
Bind a socket (only server)	Instructions - *SocketBind*
Listening connections (only server)	Instructions - *SocketListen*
Accept connections (only server)	Instructions - *SocketAccept*
Get current socket state	Functions - *SocketGetStatus*
Example server socket application	Instructions - *SocketAccept*

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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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

Soft servo for any robot or external axis is always deactivated when there is a power failure. This limitation can be handled in the user program when restarting after a power failure.

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

Table 23

	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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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

Table 24

	Described in:
Activating the soft servo	Instructions - *SoftAct*

SpcCon - Connects to a statistical process controller

SpcCon is used to allocate an SPC entry before starting supervision limit calculation and process supervision.

Example

```
VAR spcdescr id;
VAR spcstat status;
...
SpcCon id, status\Header:="voltage";
```

The statistical process controller is allocating one entry named "voltage" and returns a descriptor for this entry (parameter *id*). The descriptor are then used by other SPC instructions to operate on the entry. *SpcCon* will also return the status of the connection operation in the variable *status*.

Arguments

SpcCon **Descr Status [\GrpSize] [\Teach] [\Strict] [\Header][\BackupFile]**

Descr **Data type:** *spcdescr*

The descriptor of the connected SPC entry.

Status **Data type:** *spcstat*

Status of the connection operation.

[\GrpSize] **Data type:** *num*

The number of parameter samples in each subgroup (min = 1, max = 100, default = 1).

[\Teach] **Data type:** *num*

The number of subgroups that has to be collected before the calculation of the supervision limits occur and the process supervision starts (default = 50).

[\Strict] **Data type:** *switch*

Normally the statistical process controller indicates an error if one subgroup average value exceeds the +/-3 sigma (standard deviation) limit. If *strict* is activated the statistical process controller will also indicate an error if two consecutive subgroup average values exceeds the +/-1.5 sigma limit.

[\Header] **Data type:** *string*

The name (identifier) of the connected SPC entry.

[\BackupFile] **Data type:** *string*

The backup file contains the supervision limits. If the backup file defined in the instruction does not exist, new limits will be calculated and stored in the file. If the backup file does exist, the limits stored in the file will be used and the supervision limit calculation will be omitted.

Example

```
VAR spcdescr id;
VAR spcstat status;
...
SpcCon id, status\GrpSize:=3\Teach:=2\Strict\Header:="voltage";
```

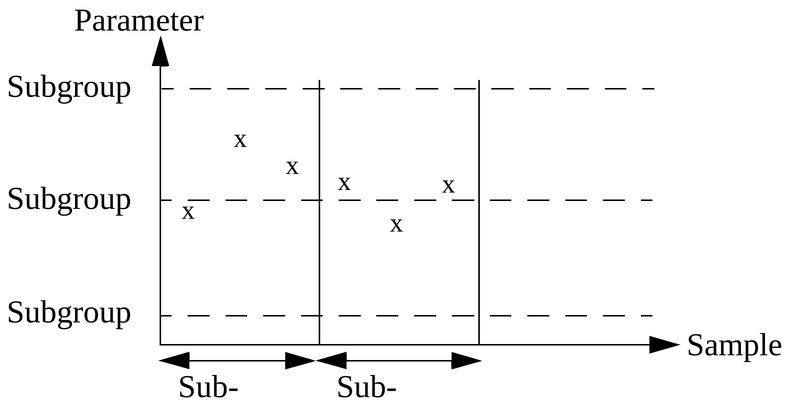


Figure 8 Statistical process control chart.

Before the parameter supervision starts, some limits has to be calculated. The calculation is based on subgroups of parameter samples, where the number of samples in each subgroup is 3 (defined by *GrpSize*) and the number of subgroups are 2 (defined by *Teach*).
The calculation will emerge in the forced limit +/-3 sigma. If the switch argument *Strict* is active, the optional limit +/-1.5 sigma will be included in the parameter supervision.

Syntax

```
SpcCon
   [ Descr ':=' ] < variable (VAR) of spcdescr > ','
   [ Status ':=' ] < var or pers (INOUT) of spcstat >
   [ '\' GrpSize ':=' < expression (IN) of num > ]
   [ '\' Teach ':=' < expression (IN) of num > ]
   [ '\' Strict ]
   [ '\' Header ':=' < expression (IN) of string > ] ';'
   [ '\' BackupFile ':=' < expression (IN) of string > ] ';'
```

Related information

Table 25

	Described in:
Disconnects from a statistical process controller	Instructions - *SpcDiscon - Disconnects from a statistical process controller*
Writes to a statistical process controller	Instructions - *SpcWrite - Writes to a statistical process controller*
Reads the current process status	Instructions - *SpcRead - Reads the current process status*
Dumps the process information on a file or a serial channel	Instructions - *SpcDump - Dump statistical process control information*
Statistical process control data	Data types - *spcdata - Statistical process control data*
Statistical process control descriptor	Data types - *spcdescr - Statistical process controller descriptor*
Statistical process control status	Data types - *spcstat - Statistical process control status*

SpcDiscon - Disconnects from a statistical process controller

SpcDiscon is used to deallocate a previously allocated SPC entry.

Example

```
VAR spcdescr id;
...
SpcDiscon id;
```

The instruction deallocates the SPC entry *id*.

Arguments

SpcDiscon Descr

Descr **Data type:** *spcdescr*

The name descriptor of the connected SPC entry.

Example

```
VAR spcdescr id;
VAR spcstat status;
...
SpcCon id, status\Header:="voltage";
...
SpcDiscon id;
```

The statistical process controller is allocating one entry named "voltage" and returns the descriptor *id*. For deallocation of the SPC entry the same descriptor is used in *SpcDiscon*.

Syntax

SpcDiscon
    [Descr ’:=’] < variable **(VAR)** of *spcdescr* > ’;’

Related information

Table 26

	Described in:
Connects to a statistical process controller	Instructions - *SpcCon - Connects to a statistical process controller*
Writes to a statistical process controller	Instructions - *SpcWrite - Writes to a statistical process controller*
Reads the current process status	Instructions - *SpcRead - Reads the current process status*
Dumps the process information on a file a serial channel	Instructions - *SpcDump - Dump statistical process control information*
Statistical process control data	Data types - *spcdata - Statistical process control data*
Statistical process control descriptor	Data types - *spcdescr - Statistical process controller descriptor*
Statistical process control status	Data types - *spcstat - Statistical process control status*

SpcDump - Dump statistical process control information

SpcDump is used to dump statistical process control information on a file or a serial channel.

Example

```
VAR spcdescr id;
...
SpcDump id, "flp1:spc.file", "Parameter voltage";
```

SpcDump will dump the statistical process control information on the file spc.file on flp1.

Arguments

SpcDump Descr SpcFile Header

Descr **Data type:** *spcdescr*

The descriptor of the connected SPC entry.

SpcFile **Data type:** *string*

The name and path of the file or serial channel where the statistical process control information should be dumped.

Header **Data type:** *string*

The header of the dump (a text that can mark up a specific dump).

Example

```
VAR spcdescr id;
VAR spcstat status;
SpcCon id, status\GrpSize:=3\Teach:=2\Header:="voltage";
SpcDump id, "flp1:spc.file", "Parameter voltage";
```

The statistical process controller is allocating one entry with *SpcCon* and is then dumping the information on the file "flp1:spc.file" to the same entry in *SpcDump*.
It is possible to add a header in the file, in this case "Parameter voltage".

Statistical process control information includes:

- Subgroup size.
- Mean values for subgroup average values and standard deviations.
- Supervision limits for the subgroup average values and standard deviations (the +/-3 sigma limit and the +/-1.5 sigma limit if it is activated).
- At most the 100 latest charted subgroup values.
- Values that has exceeded the limits among the latest 100 charted subgroup values.

Dump file example:

```
spcobj1  spc_info                 /* Process name and chart name
*/
2                                 /* Subgroup size */
60                                /* Chart length (num. of sub-
group samples, max. 100 latest) */
1                                 /* Strict rule (+/-1.5 sigma), 0 =
not active, 1 = active */
0                                 /* Assymetric limits, 0 = not
active, 1 = active */

1.014                             /* Average mean value */
0.00989746                        /* Average standard deviation */

1.040314                          /* Upper mean value limit */
0.987686                          /* Lower mean value limit */
0.0323292                         /* Upper standard dev. limit */
0                                 /* Lower standard dev. limit */
1.027157                          /* IF STRICT RULE... Upper
mean value strict limit */
1.00084317                        /* Lower mean value strict limit
*/
0.0211133                         /* Upper standard dev. strict
limit */
0                                 /* Lower standard dev. strict
limit ...ENDIF STRICT RULE */

1.015                             /* START Subgroup mean val-
ues...
```

Figure 9 SPC dump file.

Syntax

SpcDump
  [Descr ’:=’] < variable **(VAR)** of *spcdescr* > ’,’
  [SpcFile ’:=’] < expression **(IN)** of *string* > ’,’
  [Header ’:=’] < expression **(IN)** of *string* > ’;’

Related information

Table 27

	Described in:
Connects to a statistical process control-ler	Instructions - *SpcCon - Connects to a statistical process controller*
Disconnects from a statistical process controller	Instructions - *SpcDiscon - Disconnects from a statistical process controller*
Writes to a statistical process controller	Instructions - *SpcWrite - Writes to a statistical process controller*
Reads the current process status	Instructions - *SpcRead - Reads the current pro-cess status*
Statistical process control data	Data types - *spcdata - Statistical process control data*
Statistical process control descriptor	Data types - *spcdescr - Statistical process con-troller descriptor*
Statistical process control status	Data types - *spcstat - Statistical process control status*

SpcRead - Reads the current process status

SpcRead is used to read some essencial SPC information, such as mean values and limits for subgroup average values and standard deviations.

Example

```
VAR spcdescr id;
VAR spcdata info;
...
CorrRead id info;
```

The current process information are available in the variable *info*;

Arguments

SpcRead Descr Data

Descr **Data type:** *spcdescr*

The descriptor of the connected SPC entry.

Data **Data type:** *spcdata*

SPC information.

Example

```
VAR spcdescr id;
VAR spcstat status;
VAR spcdata info;
SpcCon id, status\Header:=”voltage”;
SpcRead id, info;
IF info.ok = TRUE THEN
    ! Continue execution
    ...
ENDIF
```

Process information includes:

- mean values and limits for subgroup average values and standard deviations.
- information whether the latest measured subgroup has exceeded the limits or not.

In the example, variable *info* are used to check if both the latest subgroup average value and standard deviation are within the limits.

Syntax

```
SpcRead
   [ Descr ’:=’ ] < variable (VAR) of spcdescr > ’,’
   [ Data ’:=’ ] < variable (VAR) of spcdata > ’;’
```

Related information

Table 28

	Described in:
Connects to a statistical process controller	Instructions - *SpcCon - Connects to a statistical process controller*
Disconnects from a statistical process controller	Instructions - *SpcDiscon - Disconnects from a statistical process controller*
Writes to a statistical process controller	Instructions - *SpcWrite - Writes to a statistical process controller*
Dumps the process information on a file or a serial channel	Instructions - *SpcDump - Dump statistical process control information*
Statistical process control data	Data types - *spcdata - Statistical process control data*
Statistical process control descriptor	Data types - *spcdescr - Statistical process controller descriptor*
Statistical process control status	Data types - *spcstat - Statistical process control status*

SpcWrite - Writes to a statistical process controller

SpcWrite provides the statistical process controller with parameter sample values.

Example

```
VAR spcdescr id;
VAR spcstat status;
VAR num value;
...
GetProcVal value;
SpcWrite id, value, status;
```

A parameter sample value (variable *value*), are written to the statistical process controller. The variable *value* represents the current measured process value and are in the example recieved from the userdefined procedure *GetProcVal*.

Arguments

SpcWrite Descr Value Status

Descr **Data type:** *spcdescr*

The descriptor of the connected SPC entry.

Value **Data type:** *num*

The parameter value.

Status **Data type:** *spcstat*

Status of the write operation.

Example

```
VAR spcdescr id;
VAR spcstat status;
VAR num value;
...
SpcCon id, status\GrpSize:=3\Teach:=2;
SpcWrite id, value, status;
```

The statistical process controller is allocating one entry with *SpcCon* and is then delivering the parameter value within the variable *value* to the same entry in *SpcWrite*.

The instruction *SpcWrite* are returning the statistical process status in the variable *status*. The value can be evaluated and proper actions taken.

SpcWrite has major functions:

- The instruction must be used in the beginning of the parameter measurement to provide the statistical process controller with values for calculation of supervision limits. The example shows that *SpcWrite* must be used 6 times before the actual supervision starts (2 subgroups with 3 samples in each subgroup).
- When the supervision limits are calculated, *SpcWrite* provides the statistical process controller with samples dedicated for supervision. The example shows that each subgroup of 3 samples will be evaluated (the supervision limits must not be exceeded).

Syntax

SpcWrite
[Descr ':='] < variable **(VAR)** of *spcdescr* > ','
[Value ':='] < expression **(VAR)** of *num* > ','
[Status':='] < var or pers **(INOUT)** of *spcstat* > ';'

Related information

Table 29

	Described in:
Connects to a statistical process controller	Instructions - *SpcCon - Connects to a statistical process controller*
Disconnects from a statistical process controller	Instructions - *SpcDiscon - Disconnects from a statistical process controller*
Reads the current process status	Instructions - *SpcRead - Reads the current process status*
Dumps the process information on a file or a serial channel	Instructions - *SpcDump - Dump statistical process control information*
Statistical process control data	Data types - *spcdata - Statistical process control data*
Statistical process control descriptor	Data types - *spcdescr - Statistical process controller descriptor*
Statistical process control status	Data types - *spcstat - Statistical process control status*

SpyStart - Start recording of execution time data

SpyStart is used to start the recording of instruction and time data during execution.

The execution data will be stored in a file for later analysis.

The stored data is intended for debugging RAPID programs, specifically for multi-tasking systems (only necessary to have *SpyStart* - *SpyStop* in one program task).

Example

```
SpyStart “HOME:/spy.log”;
```

Starts recording the execution time data in the file *spy.log* on the *HOME:* disk.

Arguments

SpyStart File

File **Data type:** *string*

The file path and the file name to the file that will contain the execution data.

Program execution

The specified file is opened for writing and the execution time data begins to be recorded in the file.

Recording of execution time data is active until:

- execution of instruction SpyStop
- starting program execution from the beginning
- loading a new program
- next warm start-up

Limitations

Avoid using the floppy disk (option) for recording since writing to the floppy is very time consuming.

Never use the spy function in production programs because the function increases the cycle time and consumes memory on the mass memory device in use.

Error handling

If the file in the *SpyStart* instruction can’t be opened then the system variable ERRNO is set to ERR_FILEOPEN (see “Data types - errnum”). This error can then be handled in the error handler.

File format

```
TASK            INSTR       IN        CODE          OUT

MAIN            FOR i FROM 1 TO 3 DO 0:READY :0
MAIN            mynum := mynum+i; 1:READY :  1
MAIN            ENDFOR    2:          READY :  2
MAIN            mynum := mynum+i; 2:READY :  2
MAIN            ENDFOR    2:          READY :  2
MAIN            mynum := mynum+i; 2:READY :  2
MAIN            ENDFOR    2:          READY :  3
MAIN            SetDO do1,1; 3:       READY :  3
MAIN            IF di1=0 THEN 3:      READY :  4
MAIN            MoveL p1, v1000,fine,tool0;  4:WAIT :14
----- SYSTEM TRAP-----
MAIN            MoveL p1, v1000, fine, tool0;111:READY :111
MAIN            ENDIF     108:        READY :  108
MAIN            MoveL p2, v1000,fine,tool0; 111:WAIT :118
----- SYSTEM TRAP-----
MAIN            MoveL p2, v1000, fine, tool0;326:READY :326
MAIN             SpyStop;   326:
```

TASK column shows executed program task
INSTR column shows executed instruction in specified program task
IN column shows the time in ms at enter of the executed instruction
CODE column shows if the instruction is READY or
if the instruction WAIT for completion at **OU**T time
OUT column shows the time in ms at leave of the executed instruction

All times are given in ms (relative values).

----- SYSTEM TRAP----- means that the system is doing something else than execution of RAPID instructions.

If procedure call to some NOSTEPIN procedure (module) the output list shows only the name of the called procedure. This is repeated for every executed instruction in the NOSTEPIN routine.

Syntax

```
SpyStart
    [File':=']<expression (IN) of string>';'
```

Related information

Table 30

	Described in:
Stop recording of execution data	Instructions - *SpyStop*

SpyStop - Stop recording of time execution data

SpyStop is used to stop the recording of time data during execution.

The data, which can be useful for optimising the execution cycle time, is stored in a file for later analysis.

Example

```
SpyStop;
```

Stops recording the execution time data in the file specified by the previous *SpyStart* instruction.

Program execution

The execution data recording is stopped and the file specified by the previous *SpyStart* instruction is closed.
If no *SpyStart* instruction has been executed before, the *SpyStop* instruction is ignored.

Examples

```
IF debug = TRUE SpyStart "HOME:/spy.log";
produce_sheets;
IF debug = TRUE SpyStop;
```

If the debug flag is true, start recording execution data in the file *spy.log* on the *HOME:* disk, perform actual production; stop recording, and close the file *spy.log*.

Limitations

Avoid using the floppy disk (option) for recording since writing to the floppy is very time consuming.

Never use the spy function in production programs because the function increases the cycle time and consumes memory on the mass memory device in use.

Syntax

```
SpyStop';'
```

Related information

Table 31

	Described in:
Start recording of execution data	Instructions - *SpyStart*

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.

A program or system module can be loaded in static (default) or dynamic mode:

Static mode

Table 32 How different operations affect a static loaded program or system modules

	Set PP to main from TP	Open new RAPID program
Program Module	Not affected	Unloaded
System Module	Not affected	Not affected

Dynamic mode

Table 33 How different operations affect a dynamic loaded program or system modules

	Set PP to main from TP	Open new RAPID program
Program Module	Unloaded	Unloaded
System Module	Unloaded	Unloaded

Both static and dynamic loaded modules can be unloaded by the instruction *UnLoad*.

Example

```
VAR loadsession load1;

! Start loading of new program module PART_B containing routine routine_b
! in dynamic mode
StartLoad \Dynamic, diskhome \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 diskhome \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”%;
```

Starts the loading of program module *PART_B.MOD* from *diskhome* 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. The module is loaded in dynamic mode.

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 [\Dynamic] FilePath [\File] LoadNo

[\Dynamic] **Data type:** *switch*

The switch enables loading of a program module in dynamic mode. Otherwise the loading is in static mode.

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 at the linking time, if the system parameter for *Controller/Task/Check unsolved references* is set to 0.

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 parameter *Check unsolved references* could be set to 1 (default behaviour). The *Late Binding* way is preferable.

There will always be a run time error if trying to execute an unsolved reference.

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.

For loading of program that contains a *main* procedure to a main program (with another *main* procedure), see instruction *Load.*

Examples

```
StartLoad \Dynamic, “HOME:/DOORDIR/DOOR1.MOD”, load1;
```

Loads the program module *DOOR1.MOD* from the *HOME:* at the directory *DOORDIR* into the program memory. The program module is loaded in dynamic mode.

```
StartLoad \Dynamic, "HOME:" \File:="/DOORDIR/DOOR1.MOD", load1;
```

Same as above but with another syntax.

```
StartLoad "HOME:" \File:="/DOORDIR/DOOR1.MOD", load1;
```

Same as the two examples above but the module is loaded in static mode.

```
StartLoad \Dynamic, "HOME:" \File:="/DOORDIR/DOOR1.MOD", load1;
...
WaitLoad load1;
```

is the same as

```
Load \Dynamic, "HOME:" \File:="/DOORDIR/DOOR1.MOD";
```

Error handling

If the variable specified in argument *LoadNo* is already in use, the system variable ERRNO is set to ERR_LOADNO_INUSE. This error can then be handled in the error handler.

Syntax

StartLoad
  [‘\’Dynamic ‘,’]
  [FilePath ’:=’] <expression (**IN**) of *string*>
  [’\’File ’:=’ <expression (**IN**) of *string*>] ’,’
  [LoadNo ’:=’] <variable (**VAR**) of *loadsession*> ’;’

Related information

Table 34

	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*
Cancel loading of a program module	Instructions - *CancelLoad*
Accept unsolved references	System Parameters - *Controller/Task/Check unsolved references*

StartMove - Restarts robot movement

StartMove is used to resume robot and external axes movement and belonging process when this has been stopped by the instruction *StopMove or* by some recoverable error.

For base system, it’s possible to use this instruction in following type of program tasks:

- main task, for restart of the movement in that task
- any other task, for restart of the movements in the main task

For MultiMove System, it’s possible to use this instruction in following type of program tasks:

- motion task, for restart of the movement in that task
- non motion task, for restart of the movement in the connected motion task. Besides that, if movement is restarted in one connected motion task belonging to a coordinated synchronized task group, the movement is restarted in all the cooperated tasks

Example

```
StopMove;
WaitDI ready_input, 1;
StartMove;
```

The robot starts to move again when the input *ready_input* is set.

Arguments

StartMove [\AllMotionTasks]

[\AllMotionTasks] **Data type:** *switch*

Restart the movement of all mechanical units in the system.
The switch [\AllMotionTasks] can only be used from a non-motion program task.

Program execution

Any processes associated with the stopped movement are restarted at the same time as motion resumes.

With the switch \AllMotionTasks, (only allowed from non-motion program task), the movements for all mechanical units in the system are restarted.

In a base system without the switch \AllMotionTasks, the movements for following mechanical units are restarted:

- always the mechanical units in the main task, independent of which task executes the *StartMove* instruction

In a MultiMove system without the switch \AllMotionTasks, the movements for following mechanical units are restarted:

- the mechanical units in the motion task executing *StartMove*
- the mechanical units in the motion task that are connected to the non motion task executing *StartMove.* Besides that, if mechanical units are restarted in one connected motion task belonging to a coordinated synchronized task group, the mechanical units are restarted in all the cooperated tasks

Error handling

If the robot is too far from the path (more than 10 mm or 20 degrees) to perform a restart of the interrupted movement, the system variable *ERRNO* is set to ERR_PATHDIST.

If the robot is in hold state at the time *StartMove* is executed, the system variable ERRNO is set to ERR_STARTMOVE

If the program execution is stopped several times during the regain to path movement with *StartMove*, the system variable ERRNO is set to ERR_PROGSTOP

If the robot is moving at the time *StartMove* is executed, the system variable ERRNO is set to ERR_ALRDY_MOVING.

These errors can then be handled in the error handler:

- at ERR_PATHDIST, move the robot closer to the path before doing RETRY
- at ERR_STARTMOVE, ERR_PROGSTOP or ERR_ALRDY_MOVING wait some time before trying to do RETRY

Not possible to do any error recovery if StartMove is executed in any error handler.

Syntax

```
StartMove
    ['\'AllMotionTasks]';'
```

Related information

Table 35

	Described in:
Stopping movements	Instructions - *StopMove*
Restart the robot movement	Instructions - *StartMoveRetry with RETRY*
More examples	Instructions - *StorePath, RestoPath*

StartMoveRetry - Restarts robot movement and RETRY execution

StartMoveRetry is used to resume robot and external axes movement and belonging process and also retry the execution from an ERROR handler.

This instruction can be used in an ERROR handler in following type of program tasks:

- main task in a base system
- any motion task in a MultiMove System

Example

```
VAR robtarget p_err;
...
MoveL p1\ID:=50 , v1000, z30, tool1 \WObj:=stn1;
...
ERROR
   IF ERRNO = ERR_PATH_STOP THEN
      StorePath;
      p_err := CRobT(\Tool:= tool1 \WObj:=wobj0);
      ! Fix the problem
      MoveL p_err, v100, fine, tool1;
      RestoPath;
      StartMoveRetry;
   ENDIF
ENDPROC
```

This is an example from a MultiMove System with coordinated synchronized movements (two robots working on some rotated work object).

During the movement to position *p1*, the other cooperated robot get some process error so that the coordinated synchronized movements stops.
This robots then the get the error ERR_PATH_STOP and the execution is transfered to the ERROR handler.

In the ERROR handler we do following:

- *StorePath* stores the original path, goes the a new path level and set the MultiMove System in independent mode
- if there is some problem also with this robot, we can do some movements on the new path level
- before *RestoPath* we must go back to the error position
- *RestoPath* goes back to the original path level and set the MultiMove System back to synchronized mode again
- *StartMoveRetry* restarts the interrupted movement and any process and also transfer the execution back for resume of the normal execution

Program execution

StartMoveRetry do following sequence:

- regain to path
- restart any processes associated with the stopped movement
- restart the interrupted movement
- do RETRY of the program execution

StartMoveRetry do the same as *StartMove* and RETRY together in one indivisible operation.

Only the mechanical units in the program task that execute *StartMoveRetry* are restarted.

Limitations

Can only be used in an ERROR handler in a motion task.

In a MultiMove System executing coordinated synchronized movements following programming rules must be followed in the ERROR handler:

- *StartMoveRetry* must be used in all cooperated program tasks
- if need for movement in any ERROR handler, the instructions *StorePath ... RestoPath* must be used in all cooperated program tasks
- the program must move the robot back to the error position before *RestoPath* is executed , if the robot was moved on the *StorePath* level

Error handling

If the robot is too far from the path (more than 10 mm or 20 degrees) to perform a restart of the interrupted movement, the system variable *ERRNO* is set to ERR_PATHDIST.

If the robot is in hold state at the time *StartMoveRetry* is executed, the system variable ERRNO is set to ERR_STARTMOVE

If the program execution is stopped several times during the regain to path movement with *StartMoveRetry*, the system variable ERRNO is set to ERR_PROGSTOP

If the robot is moving at the time *StartMoveRetry* is executed, the system variable ERRNO is set to ERR_ALRDY_MOVING.

Not possible to do any error recovery from these errors, because *StartMoveRetry* can only be executed in some error handler.

Syntax

```
StartMoveRetry ';'
```

Related information

Table 36

	Described in:
Stopping movements	Instructions - *StopMove*
Continuing a movement	Instructions - *StartMove*
More examples	Instructions - *StorePath, RestoPath*

STCalib - Calibrate a Servo Tool

STCalib is used to calibrate the distance between the tool tips. This is necessary after tip change or tool change and it is recommended after performing a tip dress or after using the tool for a while.

NB The tool performs two close/open movements during the calibration. The first close movement will detect the tip contact position.

Example

```
VAR num curr_tip_wear;
VAR num retval;
CONST num max_adjustment := 20;
```

```
STCalib gun1 \ToolChg;
```

Calibrate a servo gun after a toolchange.

```
STCalib gun1 \TipChg;
```

Calibrate a servo gun after a tipchange.

```
STCalib gun1 \TipWear \RetTipWear := curr_tip_wear;
```

Calibrate a servo gun after tip wear. Save the tip wear in variable curr_tip_wear.

```
STCalib gun1 \TipChg \RetPosAdj:=retval;
IF retval > max_adjustment THEN
TPWrite “The tips are lost!“;
...
```

Calibrate a servo gun after a tipchange. Check if the tips are missing.

```
STCalib gun1 \TipChg \PrePos:=10;
```

Calibrate a servo gun after a tipchange. Move fast to position 10 mm, then start to search for contact position with slower speed.

Arguments

STCalib ToolName [\ToolChg] | [\TipChg] | [\TipWear] [\RetTip-Wear] [\RetPosAdj] [\PrePos]

ToolName **Data type:** *string*

The name of the mechanical unit.

[\ToolChg] **Data type:** *switch*

Calibration after a tool change.

[\TipChg] **Data type:** *switch*

Calibration after a tip change.

[\TipWear] **Data type:** *switch*

Calibration after tip wear.

[\RetTipWear] **Data type:** *num*

The achieved tip wear[mm].

[\RetPosAdj] **Data type:** *num*

The positional adjustment since the last calibration [mm].

[\PrePos] **Data type:** *num*

The position to move with high speed to before search for contact position with slower speed is started [mm].

Program execution

Calibration modes

If the mechanical unit exists the servo tool is ordered to calibrate. The calibration is done according to the switches, see below. If the RetTipWear parameter is used then the tip wear is updated.

Calibration after toolchange:

The tool will close with slow speed waiting for tips in contact, open fast, close fast to a low force and open again in one sequence. The tip wear will remain unchanged.

Calibration after tipchange:

The tool will close with slow speed waiting for tips in contact, open fast, close fast to a low force and open again in one sequence. The tip wear will be reset.

Calibration after tipwear:

The tool will close with high speed to the contact position, open fast, close fast to a low force and open again in one sequence. The tip wear will be updated.

Positional adjustment

The optional argument RetPosAdj can be used to detect if for example the tips are lost after a tip change. The parameter will hold the value of the positional adjustment since the last calibration. The value can be negative or positive.

Using a pre position

In order to speed up the calibration, it is possible to define a pre position.When the calibration starts, the gun arm will be run fast to the pre position, stop and then continue slowly*) forward in order to detect the tip contact position. If a pre position is used, select it carefully! It is important that the tips do not get in contact until *after* the pre position is reached! Otherwise the accuracy of the calibration will become poor and motion supervision errors may possibly occur. A pre position will be ignored if it is larger than the current gun position (in order not to slow down the calibration).

*) The second movement will also be fast if the \TipWear option is used.

Error handling

If the specified servo tool name is not a configured servo tool, the system variable ERRNO is set to ERR_NO_SGUN.

If the gun is not open when STCalib is invoked, the system variable ERRNO is set to ERR_SGUN_NOTOPEN.

If the servo tool mechanical unit is not activated, the system variable ERRNO is set to ERR_SGUN_NOTACT. Use instruction ActUnit to activate the servo tool.

If the servo tool position is not initialized, the system variable ERRNO is set to ERR_SGUN_NOTINIT. The servo tool position must be initialized the first time the gun is installed, or after a fine calibration is made. Use the service routine ManService-Calib, or perform a tip change calibration. The tip wear will be reset.

If the servo tool tips are not synchronized, the system variable ERRNO is set to ERR_SGUN_NOTSYNC. The servo tool tips must be synchronized if the revolution counter has been lost and/or updated. No process data such as tip wear will be lost.

If the instruction is invoked from a background task, and there is an emergency stop, the instruction will be finished and the system variable ERRNO set to ERR_SGUN_ESTOP. Note that if the instruction is invoked from the main task, the program pointer will be stopped at the instruction, and the instruction will be restarted from the beginning at program restart.

If the argument PrePos is specified with a value less than zero, the system variable ERRNO is set to ERR_SGUN_NEGVAL.

If the instruction is invoked from a background task, and the system is in motors off state, the sytem variable ERRNO will be set to ERR_SGUN_MOTOFF.

All errors above can be handled in a Rapid error handler.

Syntax

```
STCalib
    [ ’ToolName ’:=’ ] < expression (IN) of string > ‘,’
    [ ’\’ToolChg] | [’\’TipChg] | [ ’\’TipWear]
    [ ’\’RetTipWear ’:=’ < variable or persistent(INOUT) of num > ]’;’
    [ ’\’RetPosAdj ’:=’ < variable or persistent(INOUT) of num > ]’;’
    [ ’\’PrePos ’:=’ < expression (IN) of num > ]’;’
```

Related information

Table 37

	Described in:
Open a servo tool	Instructions - *STOpen*
Close a servo tool	Instruction - *STClose*

STClose - Close a Servo Tool

STClose is used to close the Servo Tool.

Example

VAR num curr_thickness;

STClose gun1, 1000, 5;

Close the servo gun with tip force 1000N and plate thickness 5 mm.

STClose gun1, 2000, 3\RetThickness:=curr_thickness;

Close the servo gun with tip force 2000N and plate thickness 3mm.Get the measured thickness in variable curr_thickness.

Arguments

STClose ToolName TipForce Thickness [\RetThickness]

ToolName **Data type:** *string*

The name of the mechanical unit.

TipForce **Data type:** *num*

The desired tip force [N].

Thickness **Data type:** *num*

The expected contact position for the servo tool [mm].

[\RetThickness] **Data type:** *num*

The achieved thickness [mm].

Program execution

If the mechanical unit exists the servo tool is ordered to close to the expected thickness and force.

The closing will start to move the tool arm to the expected contact position (thickness). In this position the movement is stopped and a switch from position control mode to force control mode is done.

The tool arm is moved with max speed and acceleration as it is defined in the system parameters for corresponding external axis. As for other axes movements, the speed is reduced in manual mode.

When the desired tip force is achieved the instruction is ready and the achieved thickness is returned if the optional argument RetThickness is specified.

It is possible to close the tool during a programmed robot movement as long as the robot movement not includes a movement of the tool arm.

For more details, see Servo tool motion control.

Error handling

If the specified servo tool name is not a configured servo tool, the system variable ERRNO is set to ERR_NO_SGUN.

If the gun is not open when STClose is invoked, the system variable ERRNO is set to ERR_SGUN_NOTOPEN.

If the servo tool mechanical unit is not activated, the system variable ERRNO is set to ERR_SGUN_NOTACT. Use instruction ActUnit to activate the servo tool.

If the servo tool position is not initialized, the system variable ERRNO is set to ERR_SGUN_NOTINIT. The servo tool position must be initialized the first time the gun is installed, or after a fine calibration is made. Use the service routine ManService-Calib, or perform a tip change calibration. The tip wear will be reset.

If the servo tool tips are not synchronized, the system variable ERRNO is set to ERR_SGUN_NOTSYNC. The servo tool tips must be synchronized if the revolution counter has been lost and/or updated. No process data such as tip wear will be lost.

If the instruction is invoked from a background task, and there is an emergency stop, the instruction will be finished and the system variable ERRNO set to ERR_SGUN_ESTOP. Note that if the instruction is invoked from the main task, the program pointer will be stopped at the instruction, and the instruction will be restarted from the beginning at program restart.

If the instruction is invoked from a background task, and the system is in motors off state, the sytem variable ERRNO will be set to ERR_SGUN_MOTOFF.

All errors above can be handled in a Rapid error handler.

Syntax

STClose
[’ToolName ’:=’] < expression (**IN**) of *string* > ‘,’
[’Tipforce ’:=’] < expression (**IN**) of *num* > ‘,’
[’Thickness ’:=’] < expression (**IN)** of *num* >]
[‘\’ ’RetThickness ’:=’ < variable or persistent(**INOUT)** of *num* >]’;’

Related information

Table 38

	Described in:
Open a servo tool	Instructions - *STOpen*

StepBwdPath - Move backwards one step on path

StepBwdPath is used to move the TCP backwards on the robot path from a RESTART event routine.

It's up to the user, to introduce a restart process flag, so *StepBwdPath* in the RESTART event routine is **only** executed at process restart and **not** at all program restart.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

Example

StepBwdPath 30, 1;

Move backwards *30* mm in *1* second.

Arguments

StepBwdPath StepLength StepTime

StepLength **Data type:** *num*

Specifies the distance, in millimetres, to move backwards during this step. This argument must be a positive value.

StepTime **Data type:** *num*

Specifies the time, in seconds, the movement will take. This argument must have a positive value.

Program execution

The robot moves back on its path, for the specified distance. The path is exactly the same, in the reverse way, as it was before the stop occurred. In the case of a quick stop or emergency stop, the RESTART event routine is called after the regain phase has completed, so the robot will already be back on its path when this instruction is executed.

The actual speed for this movement is the lowest of:
- *StepLength* / *StepTime*
- The programmed speed on the segment
- 250 mm/s

Limitations

After the program has been stopped, it is possible to step backwards on the path with the following limits:
- The 1st step backward will be reduced to the current segment for the robot
- Further backward steps will be limited to the previous segment

If an attempt is made to move beyond these limit, the error handler will be called with ERRNO set to ERR_BWDLIMIT.

Syntax

StepBwdPath
    [StepLength ’:=’] < expression (**IN**) of *num* >’,’
    [StepTime ’:=’] < expression (**IN**) of *num* > ’;’

Related information

Table 39

	Described in:
Motion in general	Motion and I/O Principles
Positioning instructions	RAPID Summary- *Motion*

STIndGun - Sets the gun in independent mode

STIndGun (Servo Tool independent gun) is used to set the gun in independent mode and thereafter move the gun to a specified independent position. The gun will stay in independent mode until the instruction *STIndGunReset* is executed.

During independent mode, the control of the gun is separated from the robot. The gun can be closed, opened, calibrated or moved to a new independent position, but it will not follow coordinated robot movements.

Independent mode is useful if the gun performs a task that is independent of the robot's task, e.g. tip dressing of a stationary gun.

Example

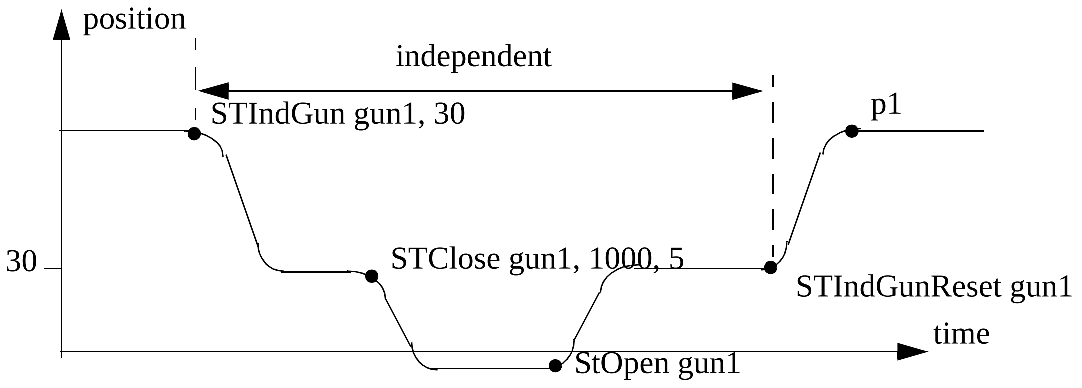
This procedure could be run from a background task while the robot in the main task can continue with e.g. moveinstructions.

```
PROC tipdress()

    ! Note that the gun will move to current robtarget position, if already in
    ! independent mode.
    STIndGunReset gun1;
    .......
    STIndGun gun1, 30;
    StClose gun1, 1000, 5;
    WaitTime 10;
    STOpen gun1;
    ......
    STIndGunReset gun1;

ENDPROC
```

Independent mode is activated and the gun is moved to an independent position (30 mm). During independent mode the instructions *StClose*, *WaitTime* and *STOpen* are executed, without interfering with robot motion. The instruction *StIndGunReset* will take the gun out of independent mode and move the gun to current robtarget position.



The position p1 depends on the position of the gun given in the robtarget just performed by the robot.

Arguments

STIndGun ToolName GunPos

ToolName **Data type:** *string*

The name of the mechanical unit.

GunPos **Data type:** *num*

The position (stroke) of the servo gun in mm.

Syntax

STIndGun
[ToolName ':='] < expression (**IN**) of *string* > ','
[GunPos ':=' < expression (**IN**) of *num* >]';'

STIndGunReset - Resets the gun from independent mode

STIndGunReset (Servo Tool independent gun reset) is used to reset the gun from independent mode and thereafter move the gun to current robtarget position.

Example

```
STIndGunReset gun1;
```

Arguments

STIndGunReset ToolName

ToolName **Data type:** *string*

The name of the mechanical unit.

Program execution

The instruction will reset the gun from independent mode and move the gun to current robtarget position. During this movement the coordinated speed of the gun must be zero, otherwise an error will occur. The coordinated speed will be zero if the robot is standing still or if the current robot movement includes a “zero movement” of the gun.

Syntax

```
STIndGunReset
    [ ToolName ':=' ] < expression (IN) of string > ';'
```

SToolRotCalib - Calibration of TCP and rotation for stationary tool

SToolRotCalib (Stationary Tool Rotation Calibration) is used to calibrate the TCP and rotation of a stationary tool.

The position of the robot and its movements are always related to its tool coordinate system, i.e. the TCP and tool orientation. To get the best accuracy, it is important to define the tool coordinate system as correctly as possible.

The calibration can also be done with a manual method using the FlexPendant (described in User's Manual - *Calibration*).

Description

To define the TCP and rotation of a stationary tool, you need a movable pointing tool mounted on the end effector of the robot.

Before using the instruction *SToolRotCalib*, some preconditions must be fulfilled:

- The stationary tool that is to be calibrated must be stationary mounted and defined with the correct component *robhold* (*FALSE*).
- The pointing tool (*robhold TRUE*) must be defined and calibrated with the correct TCP values.
- If using the robot with absolute accuracy, the load and centre of gravity for the pointing tool should be defined.
  LoadIdentify can be used for the load definition.
- The pointing tool, *wobj0* and *PDispOff* must be activated before jogging the robot.
- Jog the TCP of the pointing tool as close as possible to the TCP of the stationary tool (origin of the tool coordinate system) and define a *robtarget* for the reference point *RefTip*.
- Jog the robot without changing the tool orientation so the TCP of the pointing tool is pointing at some point on the positive z-axis of the tool coordinate system and define a *robtarget* for point *ZPos*.
- Jog the robot without changing the tool orientation so the TCP of the pointing tool is pointing at some point on the positive x-axis of the tool coordinate system and define a *robtarget* for point *XPos*.

As a help for pointing out the positive z-axis and x-axis, some type of elongator tool can be used.

Notice that you must not modify the positions RefTip, ZPos and XPos in the instruction *SToolRotCalib*, while the tool used in the creation of the points is not the same as the tool being calibrated.

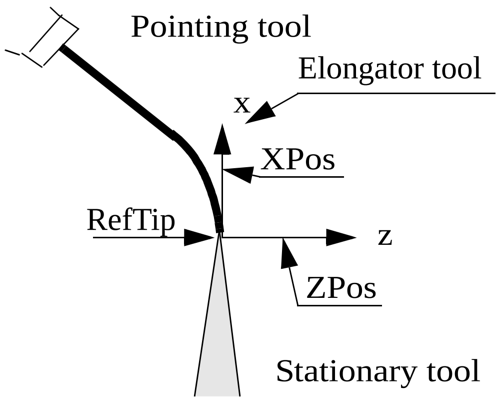


Figure 10 Definition of robtargets RefTip, ZPos and XPos

Example

```
! Created with pointing TCP pointing at the stationary tool coordinate system
CONST robtarget pos_tip := [...];
CONST robtarget pos_z := [...];
CONST robtarget pos_x := [...];

PERS tooldata tool1:= [ FALSE, [[0, 0, 0], [1, 0, 0 ,0]],
                    [0, [0, 0, 0], [1, 0, 0, 0], 0, 0, 0]];

! Instructions for creating or ModPos of pos_tip, pos_z and pos_x
MoveJ pos_tip, v10, fine, point_tool;
MoveJ pos_z, v10, fine, point_tool;
MoveJ pos_x, v10, fine, point_tool;

SToolRotCalib pos_tip, pos_z, pos_x, tool1;
```

The position of the TCP (*tframe.trans*) and the tool orientation (*tframe.rot*) of *tool1* in the world coordinate system is calculated and updated.

Arguments

SToolRotCalib RefTip ZPos XPos Tool

RefTip **Data type:** *robtarget*

The reference tip point.

ZPos **Data type:** *robtarget*

The elongator point that defines the positive z direction.

XPos **Data type:** *robtarget*

The elongator point that defines the positive x direction.

Tool **Data type:** *tooldata*

The name of the tool that is to be calibrated.

Program execution

The system calculates and updates the TCP (*tframe.trans*) and the tool orientation (*tfame.rot*) in the specified *tooldata.* The calculation is based on the specified 3 *robtarget*. The remaining data in *tooldata* is not changed.

Syntax

SToolRotCalib
  [RefTip ’:=’] < expression (**IN**) of *robtarget* > ’,’
  [ZPos ’:=’] < expression (**IN**) of *robtarget* > ’,’
  [XPos ’:=’] < expression (**IN**) of *robtarget* > ’,’
  [Tool ’:=’] < persistent (**PERS**) of *tooldata* > ’;’

Related information

Table 40

	Described in:
Calibration of TCP for a moving tool	Instructions - *MToolTCPCalib*
Calibration of rotation for a moving tool	Instructions - *MToolRotCalib*
Calibration of TCP for a stationary tool	Instructions - *SToolTCPCalib*

SToolTCPCalib - Calibration of TCP for stationary tool

SToolTCPCalib (Stationary Tool TCP Calibration) is used to calibrate the Tool Centre Point - TCP for a stationary tool.

The position of the robot and its movements are always related to its tool coordinate system, i.e. the TCP and tool orientation. To get the best accuracy, it is important to define the tool coordinate system as correctly as possible.

The calibration can also be done with a manual method using the FlexPendant (described in User's Manual - *Calibration*).

Description

To define the TCP of a stationary tool, you need a movable pointing tool mounted on the end effector of the robot.

Before using the instruction *SToolTCPCalib*, some preconditions must be fulfilled:

- The stationary tool that is to be calibrated must be stationary mounted and defined with the correct component *robhold* (*FALSE*).
- The pointing tool (*robhold TRUE*) must be defined and calibrated with the correct TCP values.
- If using the robot with absolute accuracy, the load and centre of gravity for the pointing tool should be defined.
  LoadIdentify can be used for the load definition.
- The pointing tool, *wobj0* and *PDispOff* must be activated before jogging the robot.
- Jog the TCP of the pointing tool as close as possible to the TCP of the stationary tool and define a *robtarget* for the first point p1.
- Define a further three positions p2, p3, and p4, all with different orientations.
- It is recommended that the TCP is pointed out with different orientations to obtain a reliable statistical result, although it is not necessary.

Notice that you must not modify the positions Pos1 to Pos4 in the instruction *SToolTCPCalib*, while the tool used in the creation of the points is not the same as the tool being calibrated.

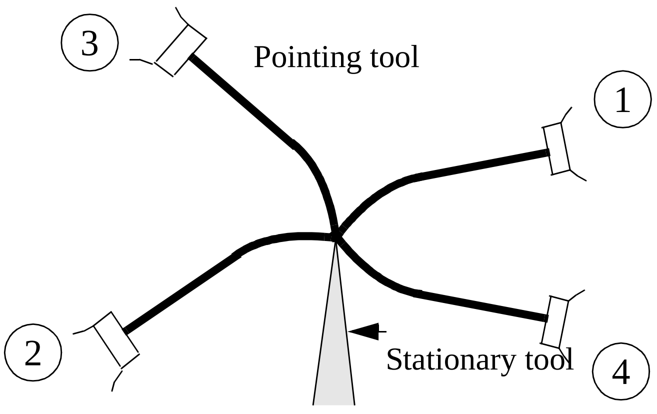


Figure 11 Definition of 4 robtargets p1...p4

Example

```
! Created with pointing TCP pointing at the stationary TCP
CONST robtarget p1 := [...];
CONST robtarget p2 := [...];
CONST robtarget p3 := [...];
CONST robtarget p4 := [...];

PERS tooldata tool1:= [ FALSE, [[0, 0, 0], [1, 0, 0 ,0]],
                [0.001, [0, 0, 0.001], [1, 0, 0, 0], 0, 0, 0]];
VAR num max_err;
VAR num mean_err;

! Instructions for creating or ModPos of p1 - p4
MoveJ p1, v10, fine, point_tool;
MoveJ p2, v10, fine, point_tool;
MoveJ p3, v10, fine, point_tool;
MoveJ p4, v10, fine, point_tool;

SToolTCPCalib p1, p2, p3, p4, tool1, max_err, mean_err;
```

The TCP value (*tframe.trans*) of *tool1* will be calibrated and updated. *max_err* and *mean_err* will hold the max error in mm from the calculated TCP and the mean error in mm from the calculated TCP, respectively.

Arguments

SToolTCPCalib Pos1 Pos2 Pos3 Pos4 Tool MaxErr MeanErr

Pos1 **Data type:** *robtarget*

The first approach point.

Pos2 **Data type:** *robtarget*

The second approach point.

Pos3 **Data type:** *robtarget*

The third approach point.

Pos4 **Data type:** *robtarget*

The fourth approach point.

Tool **Data type:** *tooldata*

The name of the tool that is to be calibrated.

MaxErr **Data type:** *num*

The maximum error in mm for one approach point.

MeanErr **Data type:** *num*

The average distance that the approach points are from the calculated TCP, i.e. how accurately the robot was positioned relative to the stationary TCP.

Program execution

The system calculates and updates the TCP value in the world coordinate system (*tfame.trans*) in the specified *tooldata.* The calculation is based on the specified 4 *robtarget*. The remaining data in tooldata, such as tool orientation (*tframe.rot*), is not changed.

Syntax

SToolTCPCalib
    [Pos1 ’:=’] < expression (**IN**) of *robtarget* > ’,’
    [Pos2 ’:=’] < expression (**IN**) of *robtarget* > ’,’
    [Pos3 ’:=’] < expression (**IN**) of *robtarget* > ’,’
    [Pos4 ’:=’] < expression (**IN**) of *robtarget* > ’,’
    [Tool ’:=’] < persistent (**PERS**) of *tooldata* > ’,’
    [MaxErr ’:=’] < variable (**VAR**) of *num* > ’,’
    [MeanErr ’:=’] < variable (**VAR**) of *num* > ’;’

Related information

Table 41

	Described in:
Calibration of TCP for a moving tool	Instructions - *MToolTCPCalib*
Calibration of rotation for a moving tool	Instructions - *MToolRotCalib*
Calibration of TCP and rotation for a stationary tool	Instructions - *SToolRotCalib*

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 FlexPendant.

Arguments

Stop [\NoRegain]

[\NoRegain] **Data type:** *switch*

Specifies for the next program start, whether or not the robot and external axes should 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 FlexPendant. The user can then answer, whether or not the robot should regain to the stop position.

Program execution

The instruction stops program execution when the robot and external axes has reached zero speed for the movement it is performing at the time, and stand still. Program execution can then be restarted from the next instruction.

If the instruction is used in a task declared Static or Semistatic, the behaviour depends on the value of the system parameter *TrustLevel*. See documentation for System Parameters.

If the instruction is used in a MultiMove system, the behaviour depends on the system parameter *MultiStop*. See documentation for System Parameters.

Example

```
MoveL p1, v500, fine, tool1;
TPWrite “Jog the robot to the position for pallet corner 1”;
Stop \NoRegain;
p1_read := CRobT(\Tool:=tool1 \WObj:=wobj0);
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.

Syntax

```
Stop
   [ ’\’ NoRegain ]’;’
```

Related information

Table 42

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

STOpen - Open a Servo Tool

STOpen is used to open the Servo Tool.

Example

```
STOpen gun1;
```

Open the servo tool gun1.

Arguments

STOpen ToolName

ToolName **Data type:** *string*

The name of the mechanical unit.

Program execution

If the mechanical unit exists the servo tool is ordered to open. The tip force is reduced to zero and the tool arm is moved back to the pre_close position.

The tool arm is moved with max speed and acceleration as it is defined in the system parameters for corresponding external axis. As for other axes movements, the speed is reduced in manual mode.

It is possible to open the tool during a programmed robot movement as long as the robot movement not includes a movement of the tool arm.

For more details, see Servo tool motion control.

Error handling

If the specified servo tool name is not a configured servo tool, the system variable ERRNO is set to ERR_NO_SGUN.

If the servo tool mechanical unit is not activated, the system variable ERRNO is set to ERR_SGUN_NOTACT. Use instruction ActUnit to activate the servo tool.

If the servo tool position is not initialized, the system variable ERRNO is set to ERR_SGUN_NOTINIT. The servo tool position must be initialized the first time the gun is installed, or after a fine calibration is made. Use the service routine ManService-Calib, or perform a tip change calibration. The tip wear will be reset.

If the servo tool tips are not synchronized, the system variable ERRNO is set to ERR_SGUN_NOTSYNC. The servo tool tips must be synchronized if the revolution counter has been lost and/or updated. No process data such as tip wear will be lost.

All errors above can be handled in a Rapid error handler.

NOTE:

If the instruction is invoked from a background task, and there is an emergency stop, the instruction will be finished without an error.

If the instruction is invoked from a background task, and the system is in motors off state,the instruction will be finished without an error.

Syntax

STOpen
    ['ToolName ':='] < expression (**IN**) of *string* > ','

Related information

Table 43

	Described in:
Close a servo tool	Instructions - *STClose*

StopMove - Stops robot movement

StopMove is used to stop robot and external axes movements and any belonging process temporarily. If the instruction *StartMove* is given, movement and process resumes.

This instruction can, for example, be used in a trap routine to stop the robot temporarily when an interrupt occurs.

For base system, it's possible to use this instruction in following type of program tasks:

- main task, for stop of the movement in that task
- any other task, for stop of the movements in the main task

For MultiMove System, it's possible to use this instruction in following type of program tasks:

- motion task, for stop of the movement in that task
- non motion task, for stop of the movement in the connected motion task. Besides that, if movement is stopped in one motion task belonging to a coordinated synchronized task group, the movement is stopped in all the cooperated tasks

Example

```
StopMove;
WaitDI ready_input, 1;
StartMove;
```

The robot movement is stopped until the input, *ready_input*, is set.

Arguments

StopMove [\Quick] [\AllMotionTasks]

[\Quick] **Data type:** *switch*

Stops the robot on the path as fast as possible.

Without the optional parameter *\Quick*, the robot stops on the path, but the braking distance is longer (same as for normal Program Stop).

[\AllMotionTasks] **Data type:** *switch*

Stop the movement of all mechanical units in the system.
The switch [\AllMotionTasks] can only be used from a non-motion program task.

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 after waiting for the robot and external axes to stop (standing still).

With the switch \AllMotionTasks (only allowed from non-motion program task), the movements for all mechanical units in the system are stopped.

In a base system without the switch \AllMotionTasks, the movements for following mechanical units are stopped:

- always the mechanical units in the main task, independent of which task executes the *StopMove* instruction

In a MultiMove system without the switch \AllMotionTasks, the movements for following mechanical units are stopped:

- the mechanical units in the motion task executing *StopMove*
- the mechanical units in the motion task that are connected to the non motion task executing *StopMove.* Besides that, if mechanical units are stopped in one connected motion task belonging to a coordinated synchronized task group, the mechanical units are stopped in all the cooperated tasks.

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(\Tool:=tool1 \WObj:=wobj0);
    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 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(\Tool:=tool1 \WObj:=wobj0);
    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
    ['\'Quick]
    ['\'AllMotionTasks]';'
```

Related information

Table 44

	Described in:
Continuing a movement	Instructions - *StartMove, StartMoveRetry*
Store - restore path	Instructions - *StorePath - RestoPath*

StopMoveReset - Reset the system stop move flag

StopMoveReset is used to reset the system stop move flag, without starting any movements.

Asynchronously raised movements errors, such as ERR_PATH_STOP or specific process error during the movements, can be handled in the ERROR handler. When such error occurs, the movements is stopped at once and the system stop move flag is set for actual program tasks. This means that the movement is not restarted, if doing any ProgStart while program pointer is inside the ERROR handler.

Restart of the movements after such movement error will be done after one of these action:

- Execute *StartMove* or *StartMoveRetry*

- Execute *StopMoveReset* and the movement will restart at next ProgStart

Example

```
...
ArcL p101, v100, seam1, weld1, weave1, z10, gun1;
...
ERROR
    IF ERRNO=AW_WELD_ERR OR ERRNO=ERR_PATH_STOP THEN
        ! Execute something but without any restart of the movement
        ! ProgStop - ProgStart must be allowed
        ...
        ! No idea to try to recover from this error, so let the error stop the program
        ! Reset the move stop flag, so it's possible to manual restart the program
        ! and the movement after that the program has stopped
        StopMoveReset;
    ENDIF
ENDPROC
```

After that above ERROR handler has executed the ENDPROC, the program execution stops and the pointer is at the beginning of the *ArcL* instruction. Next ProgStart restart the program and movement from the position where the original movement error occurred.

Syntax

```
StopMoveReset ';'
```

Related information

Table 45

	Described in
Stop the movement	Instructions - *StopMove*
Continuing a movement	Instructions - *StartMove, StartMoveRetry*
Store - restore path	Instructions - *StorePath - RestoPath*

StorePath - Stores the path when an interrupt occurs

StorePath is used to store the movement path being executed, e.g. 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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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

Table 46

	Described in:
Restoring a path	Instructions - *RestoPath*
More examples	Instructions - *RestoPath*
More examples	Instructions - *PathRecStart*

STTune - Tuning Servo Tool

STTune is used to tune/change a servo tool parameter. The parameter is changed temporarily from the original value, which is set up in the system parameters. The new tune value will be active immediately after executing the instruction.

STTune is useful in tuning procedures. A tuning procedure is typically used to find an optimal value for a parameter. An experiment (i.e. a program execution with a servo tool movement) is repeated when using different parameter tune values.

STTune shall not be used during calibration or tool closure.

Example

```
STTune SEOLO_RG, 0.050, CloseTimeAdjust;
```

The servo tool parameter CloseTimeAdjust is temporarily set to 0.050 seconds.

Arguments

STTune MecUnit TuneValue Type

MecUnit **Data type:** *mecunit*

The name of the mechanical unit.

TuneValue **Data type:** *num*

New tuning value.

Type **Data type:** *tunegtype*

Parameter type. Servo tool parameters available for tuning are RampTorqRefOpen, RampTorqRefClose, KV, SpeedLimit, CollAlarmTorq, CollContactPos, CollisionSpeed, CloseTimeAdjust, ForceReadyDelayT, PostSyncTime, CalibTime, CalibForceLow, CalibForceHigh. These types are predefined in the system parameters and defines the original values.

Description

RampTorqRefOpen

Tunes the system parameter “Ramp when decrease force”, which decides how fast force is released while opening the tool. The unit is Nm/s and a typical value 200.

Corresponding system parameter: Topics *Manipulator*, Type *Force master*, parameter *ramp_torque_ref_opening.*

RampTorqRefClose

Tunes the system parameter “Ramp when increase force”, which decides how fast force is built up while opening the tool. The unit is Nm/s and a typical value 80.

Corresponding system parameter: Topics *Manipulator*, Type *Force master*, parameter *ramp_torque_ref_closing.*

KV

Tunes the system parameter “KV”, which is used for speed limitation. The unit is Nms/rad and a typical value 1. For more details, see the external axis documentation.

Corresponding system parameter: Topics *Manipulator*, Type *Force master*, parameter *Kv.*

SpeedLimit

Tunes the system parameter “Speed limit”, which is used for speed limitation. The unit is rad/s (motor speed) and a typical value 60. For more details, see the external axis documentation.

Corresponding system parameter: Topics *Manipulator*, Type *Force master*, parameter *speed_limit.*

CollAlarmTorq

Tunes the system parameter “Collision alarm torque”, which is used for the automatic calibration of new tips. The unit is Nm (motor torque) and a typical value 1. For more details, see the external axis documentation.

Corresponding system parameter: Topics *Manipulator*, Type *Force master*, parameter *alarm_torque.*

CollContactPos

Tunes the system parameter “Collision delta pos”, which is used for automatic calibration of new tips. The unit is m and a typical value 0,002. For more details, see the external axis documentation.

Corresponding system parameter: Topics *Manipulator*, Type *Force master*, parameter *distance_to_contact_position.*

CollisionSpeed

Tunes the system parameter “Collision speed”, which is used for automatic calibration of new tips. The unit is m/s and a typical value 0,02. For more details, see the external axis documentation.

Corresponding system parameter: Topics *Manipulator*, Type *Force master*, parameter *col_speed.*

CloseTimeAdjust

Constant time adjustment (s), positive or negative, of the moment when the tool tips reaches contact during a tool closure. May be used to delay the closing slightly when the synchronized pre closing is used for welding.

Corresponding system parameter: Topics *Manipulator*, Type *SG process*, parameter *min_close_time_adjust*.

ForceReadyDelayT

Constant time delay (s) before sending the weld ready signal after reaching the programmed force.

Corresponding system parameter: Topics *Manipulator*, Type *SG process*, parameter *pre_sync_delay_time.*

PostSyncTime

Release time anticipation (s) of the next robot movement after a weld. This tune type can be tuned to synchronize the gun opening with the next robot movement. The synchronization may fail if the parameters is set too high.

Corresponding system parameter: Topics *Manipulator*, Type *SG process*, parameter *post_sync_time.*

CalibTime

The wait time (s) during a calibration before the positional tool tip correction is done. For best result, do not use a too low value, for instance 0.5 s.

Corresponding system parameter: Topics *Manipulator*, Type *SG process*, parameter *calib_time.*

CalibForceLow

The minimum tip force (N) used during a TipWear calibration. For best result of the thickness detection, it is recommended to use the minimum programmed weld force.

Corresponding system parameter: Topics *Manipulator,* Type *SG process*, parameter *calib_force_low.*

CalibForceHigh

The maximum tip force (N) used during a TipWear calibration. For best result of the thickness detection, it is recommended to use the max programmed weld force.

Corresponding system parameter: Topics *Manipulator*, Type *SG process*, parameter *calib_force_high.*

Program execution

The specified tuning type and tuning value are activated for the specified mechanical unit. This value is applicable for all movements until a new value is programmed for the current mechanical unit, or until the tuning types and values are reset using the instruction *STTuneReset*.

The original tune values may be permanently changed in the system parameters.

The default servo tool tuning values are automatically set

- by executing instruction *STTuneReset*
- at a cold start-up
- when a new program is loaded
- when starting program execution from the beginning.

Error handling

If the specified servo tool name is not a configured servo tool, the system variable ERRNO is set to ERR_NO_SGUN.

The error can be handled in a Rapid error handler.

Syntax

STTune
    [MecUnit ':='] < variable (**VAR**) of *mecunit* > ','
    [TuneValue ':='] < expression (**IN**) of *num* > ','
    ['Type ':='] < expression (**IN**) of *tunegtype* >]';'

Related information

Table 47

	Described in:
Restore of servo tool parameters	Instructions - *TuneReset*
Tuning of servo tool	External axes manual

STTuneReset - Resetting Servo tool tuning

STTuneReset is used to restore original values of servo tool parameters if they have been changed by the STTune instruction.

Example

STTuneReset SEOLO_RG;

Restore *original values of servo tool parameters* for the mechanical unit SEOLO_RG.

Arguments

STTuneReset MecUnit

MecUnit **Data type:** *mecunit*

The name of the mechanical unit.

Program execution

The original servo tool parameters are restored.

This is also achieved

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

Error handling

If the specified servo tool name is not a configured servo tool, the system variable ERRNO is set to ERR_NO_SGUN.

The error can be handled in a Rapid error handler.

Syntax

STTuneReset
  [MecUnit ':='] < variable (**VAR**) of *mecunit* > ','

Related information

Table 48

	Described in:
Tuning of servo tool parameters	Instructions - *STTune*
Tuning of servo tool parameters	External axes manual

SyncMoveOff - End coordinated synchronized movements

SyncMoveOff is used to end a sequence of synchronized movements, in most cases also coordinated movements. First all involved program tasks will wait to synchronize in a stop point and then the motion planner for the involved program tasks are set to independent mode.

The instruction *SyncMoveOff* can only be used in a *MultiMove System* with option *Coordinated Robots* and only in program tasks defined as *Motion Task*.

To reach safe synchronization functionality, the meeting point (parameter SyncID) must have an unique name in each program task. The name must also be the same for the program tasks that should meet in the meeting point.

Example

Program example in task T_ROB1

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;
VAR syncident sync2;

...
SyncMoveOn sync1, task_list;
...
SyncMoveOff sync2;
...
```

Program example in task T_ROB2

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;
VAR syncident sync2;

...
SyncMoveOn sync1, task_list;
...
SyncMoveOff sync2;
...
```

The program task, that first reach *SyncMoveOff* with identity *sync2*, waits until the other task reach it’s *SyncMoveOff* with the same identity *sync2.*
At that synchronization point *sync2*, the motion planner for the involved program tasks is set to independent mode. After that both task T_ROB1 and T_ROB2 continue it’s execution.

Arguments

SyncMoveOff SyncID [\TimeOut]

SyncID **Data type:** *syncident*

Variable that specify the name of the unsynchronization (meeting) point.
Data type *syncident* is a non-value type, only used as an identifier for naming the unsynchronization point.

The variable must be defined and have equal name in all cooperated program tasks. It's recommended to always define the variable global in each task (VAR *syncident* ...).

[\TimeOut] **Data type:** *num*

The max. time for waiting for the other program tasks to reach the unsynchronization point. Time-out in seconds (resolution 0,001s). If this argument is not specified, the program task will wait for ever.

If this time runs out before all program tasks has reach the unsynchronization point, the error handler will be called, if there is one, with the error code ERR_SYNCMOVEOFF. If there is no error handler, the execution will be stopped.

Program execution

The program task, that first reach *SyncMoveOff* , waits until all other specified tasks reach it's *SyncMoveOff* with the same *SyncID* identity. At that *SyncID* unsynchronization point, the motion planner for the involved program tasks is set to independent mode. After that involved program tasks continue it's execution.

The motion planner for the involved program tasks is set to unsynchronized mode means following:

- All RAPID program tasks and all movements from these tasks are working independently of each other again
- Any Move instruction must **not** be marked with any ID number.
  See instruction *MoveL*

It is possible to exclude program task for testing purpose from FlexPendant - Task Selection Panel. The instructions *SyncMoveOn* and *SyncMoveOff* will still works with the reduced number of program tasks, even for only one program task.

Example

Program example in task T_ROB1

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;

PROC main()
    ...
    MoveL p_zone, vmax, z50, tcp1;
    WaitSyncTask sync1, task_list;
    MoveL p_fine, v1000, fine, tcp1;
    syncmove;
    ...
ENDPROC

PROC syncmove()
    SyncMoveOn sync2, task_list;
    MoveL * \ID:=10, v100, z10, tcp1 \WOBJ:= rob2_obj;
    MoveL * \ID:=20, v100, fine, tcp1 \WOBJ:= rob2_obj;
    SyncMoveOff sync3;
    UNDO
        SyncMoveUndo;
ENDPROC
```

Program example in task T_ROB2

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;

PROC main()
    ...
    MoveL p_zone, vmax, z50, obj2;
    WaitSyncTask sync1, task_list;
    MoveL p_fine, v1000, fine, obj2;
    syncmove;
    ...
ENDPROC

PROC syncmove()
    SyncMoveOn sync2, task_list;
    MoveL * \ID:=10, v100, z10, obj2;
    MoveL * \ID:=20, v100, fine, obj2 ;
    SyncMoveOff sync3;
    UNDO
        SyncMoveUndo;
```

```
ENDPROC
```

First program tasks T_ROB1 and T_ROB2 are waiting at *WaitSyncTask* with identity *sync1* for each other, programmed with corner path for the preceding movements for saving cycle time.

Then the program tasks are waiting at *SyncMoveOn* with identity *sync2* for each other, programmed with a necessary stop point for the preceding movements. After that the motion planner for the involved program tasks is set to synchronized mode.

After that T_ROB2 are moving the *obj2* to *ID* point *10* and *20* in world coordinate system while T_ROB1 are moving the *tcp1* to *ID* point *10* and *20* on the moving object *obj2*.

Then the program tasks are waiting at *SyncMoveOff* with identity *sync3* for each other, programmed with a necessary stop point for the preceding movements. After that the motion planner for the involved program tasks is set to independent mode.

Program example with use of time-out function

```
VAR syncident sync3;

...
SyncMoveOff sync3 \TimeOut := 60;
...
ERROR
    IF ERRNO = ERR_SYNCMOVEOFF THEN
        RETRY;
    ENDIF
```

The program task waits in instruction *SyncMoveOff* for some other program task to reach the same synchronization point *sync3*. After waiting in *60* s, the error handler is called with ERRNO equal to ERR_SYNCMOVEOFF.
Then the instruction *SyncMoveOff* is called again for additional wait in 60 s.

Error handling

If time-out because *SyncMoveOff* not ready in time, the system variable ERRNO is set to ERR_SYNCMOVEOFF.

This error can be handled in the ERROR handler.

Limitations

The *SyncMoveOff* instruction can only be executed if all involved robots stand still in a stop point.

Syntax

SyncMoveOff
  [SyncID ’:=’] < variable (**VAR**) of *syncident*>
  [’\’ TimeOut ’:=’ < expression (**IN**) of *num* >] ’;’

Related information

Table 49

	Described in:
Specify cooperated program tasks	Data Types - *tasks*
Identity for synchronization point	Data Types - *syncident*
Start coordinated synchronized move-ments	Instruction - *SyncMoveOn*
Set independent movements	Instruction - *SyncMoveUndo*
Test if in synchronized mode	Function - *IsSyncModeOn*

SyncMoveOn - Start coordinated synchronized movements

SyncMoveOn is used to start a sequence of synchronized movements, in most cases also coordinated movements. First all involved program tasks will wait to synchronize in a stop point and then the motion planner for the involved program tasks is set to synchronized mode.

The instruction *SyncMoveOn* can only be used in a *MultiMove System* with option *Coordinated Robots* and only in program tasks defined as *Motion Task.*

To reach safe synchronization functionality, the meeting point (parameter SyncID) must have an unique name in each program task. The name must also be the same for the program tasks that should meet in the meeting point.

Example

Program example in task T_ROB1

```
PERS tasks task_list{2} := [ ["T_ROB1"], ["T_ROB2"] ];
VAR syncident sync1;
VAR syncident sync2;

...
SyncMoveOn sync1, task_list;
...
SyncMoveOff sync2;
...
```

Program example in task T_ROB2

```
PERS tasks task_list{2} := [ ["T_ROB1"], ["T_ROB2"] ];
VAR syncident sync1;
VAR syncident sync2;

...
SyncMoveOn sync1, task_list;
...
SyncMoveOff sync2;
...
```

The program task, that first reaches *SyncMoveOn* with identity *sync1*, waits until the other task reaches it's *SyncMoveOn* with the same identity *sync1.*
At that synchronization point *sync1*, the motion planner for the involved program tasks is set to synchronized mode. After that both task T_ROB1 and T_ROB2 continue their execution.

Arguments

SyncMoveOn SyncID TaskList [\TimeOut]

SyncID **Data type:** *syncident*

Variable that specifies the name of the synchronization (meeting) point.
Data type *syncident* is a non-value type, only used as an identifier for naming the synchronization point.

The variable must be defined and have equal name in all cooperated program tasks. It's recommended to always define the variable global in each task (VAR *syncident* ...).

TaskList **Data type:** *tasks*

Persistent variable, that in a task list (array) specifies the name (*string*) of the program tasks, that should meet in the synchronization point with name according argument *SyncID*.

The persistent variable must be defined and have equal name and equal contents in all cooperated program tasks. It's recommended to always define the variable global in the system (PERS *tasks* ...).

[\TimeOut] **Data type:** *num*

The max. time for waiting for the other program tasks to reach the synchronization point. Time-out in seconds (resolution 0,001s). If this argument is not specified, the program task will wait for ever.

If this time runs out before all program tasks have reached the synchronization point, the error handler will be called, if there is one, with the error code ERR_SYNCMOVEON. If there is no error handler, the execution will be stopped.

Program execution

The program task, that first reaches *SyncMoveOn* , waits until all other specified tasks reach their *SyncMoveOn* with the same *SyncID* identity. At that *SyncID* synchronization point, the motion planner for the involved program tasks is set to synchronized mode. After that involved program tasks continue their execution.

The motion planner for the involved program tasks is set to synchronized mode means following:

- Each movement instruction in any program task in the *TaskList,* is working synchronous with 1, 2 or 3 movement instructions in other program tasks in the *TaskList*
- All cooperated movement instructions are planned and interpolated in the same Motion Planner

- All movements start and end at the same time. The movement that takes the longest time will be the speed master, with reduced speed in relation to the work object for the other movements
- All cooperated Move instruction must be marked with the same ID number. See instruction *MoveL*

It is possible to exclude program tasks for testing purpose from FlexPendant - Task Selection Panel. The instruction *SyncMoveOn* will still work with the reduced number of program tasks, even for only one program task.

Example

Program example in task T_ROB1

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;

PROC main()
    ...
    MoveL p_zone, vmax, z50, tcp1;
    WaitSyncTask sync1, task_list;
    MoveL p_fine, v1000, fine, tcp1;
    syncmove;
    ...
ENDPROC

PROC syncmove()
    SyncMoveOn sync2, task_list;
    MoveL * \ID:=10, v100, z10, tcp1 \WOBJ:= rob2_obj;
    MoveL * \ID:=20, v100, fine, tcp1 \WOBJ:= rob2_obj;
    SyncMoveOff sync3;
    UNDO
        SyncMoveUndo;
ENDPROC
```

Program example in task T_ROB2

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;
```

```
PROC main()
    ...
    MoveL p_zone, vmax, z50, obj2;
    WaitSyncTask sync1, task_list;
    MoveL p_fine, v1000, fine, obj2;
    syncmove;
    ...
ENDPROC

PROC syncmove()
    SyncMoveOn sync2, task_list;
    MoveL * \ID:=10, v100, z10, obj2;
    MoveL * \ID:=20, v100, fine, obj2 ;
    SyncMoveOff sync3;
    UNDO
        SyncMoveUndo;
ENDPROC
```

First program tasks T_ROB1 and T_ROB2 are waiting at *WaitSyncTask* with identity *sync1* for each other, programmed with corner path for the preceding movements for saving cycle time.

Then the program tasks are waiting at *SyncMoveOn* with identity *sync2* for each other, programmed with a necessary stop point for the preceding movements. After that the motion planner for the involved program tasks is set to synchronized mode.

After that T_ROB2 are moving the *obj2* to *ID* point *10* and *20* in world coordinate system while T_ROB1 are moving the *tcp1* to *ID* point *10* and *20* on the moving object *obj2*.

Program example with use of time-out function

```
PERS tasks task_list{2} := [ ["T_ROB1"], ["T_ROB2"] ];
VAR syncident sync1;

...
SyncMoveOn sync3, task_list \TimeOut := 60;
...
ERROR
    IF ERRNO = ERR_SYNCMOVEON THEN
        RETRY;
    ENDIF
```

The program task T_ROB1 waits in instruction *SyncMoveOn* for the program task T_ROB2 to reach the same synchronization point *sync3*. After waiting in *60* s, the error handler is called with ERRNO equal to ERR_SYNCMOVEON. Then the instruction *SyncMoveOn* is called again for additional wait in 60 s.

Program example with three tasks

Program example in task T_ROB1

```
PERS tasks task_list1{2} := [ [“T_ROB1”], [“T_ROB2”]];
PERS tasks task_list2{3} := [ [“T_ROB1”], [“T_ROB2”], [“T_ROB3”]];
VAR syncident sync1;
...
VAR syncident sync5;

...
    SyncMoveOn sync1, task_list1;
    ...
    SyncMoveOff sync2;
    WaitSyncTask sync3,task_list2;
    SyncMoveOn sync4, task_list2;
    ...
    SyncMoveOff sync5;
...
```

Program example in task T_ROB2

```
PERS tasks task_list1{2} := [ [“T_ROB1”], [“T_ROB2”] ];
PERS tasks task_list2{3} := [ [“T_ROB1”], [“T_ROB2”], [“T_ROB3”]];
VAR syncident sync1;
...
VAR syncident sync5;

...
    SyncMoveOn sync1, task_list1;
    ...
    SyncMoveOff sync2;
    WaitSyncTask sync3,task_list2;
    SyncMoveOn sync4, task_list2;
    ...
    SyncMoveOff sync5;
...
```

Program example in task T_ROB3

```
PERS tasks task_list2{3} := [ [“T_ROB1”], [“T_ROB2”], [“T_ROB3”]];
VAR syncident sync3;
VAR syncident sync4;
VAR syncident sync5;

...
    WaitSyncTask sync3,task_list2;
    SyncMoveOn sync4, task_list2;
    ...
    SyncMoveOff sync5;
...
```

In this example, at first program tasks T_ROB1 and T_ROB2 are moving synchronized and T_ROB3 is moving independent. Further on in the program all three tasks are moving synchronized. To prevent the instruction *SyncMoveOn* to be executed in T_ROB3 before the first synchronization of T_ROB1 and T_ROB2 is ended, the instruction *WaitSyncTask* is used.

Error handling

If time-out because *SyncMoveOn* not ready in time, the system variable ERRNO is set to ERR_SYNCMOVEON.

This error can be handled in the ERROR handler.

Limitations

The *SyncMoveOn* instruction can only be executed if all involved robots stand still in a stop point.

Only one coordinated synchronized movement group can be active at the same time.

Syntax

SyncMoveOn
  [SyncID ’:=’] < variable (**VAR**) of *syncident*> ’,’
  [TaskList ’:=’] < persistent array {*} (**PERS**) of *tasks*> ’,’
  [’\’ TimeOut ’:=’ < expression (**IN**) of *num* >] ’;’

Related information

Table 50

	Described in:
Specify cooperated program tasks	Data Types - *tasks*
Identity for synchronization point	Data Types - *syncident*
End coordinated synchronized movements	Instruction - *SyncMoveOff*
Set independent movements	Instruction - *SyncMoveUndo*
Test if in synchronized mode	Function - *IsSyncModeOn*

SyncMoveUndo - Set independent movements

SyncMoveUndo is used to force a reset of synchronized coordinated movements and set the system to independent movement mode.

The instruction *SyncMoveUndo* can only be used in a *MultiMove System* with option *Coordinated Robots* and only in program tasks defined as *Motion Task*.

Example

Program example in task T_ROB1

```
PERS tasks task_list{2} := [ ["T_ROB1"], ["T_ROB2"] ];
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;

PROC main()
    ...
    MoveL p_zone, vmax, z50, tcp1;
    WaitSyncTask sync1, task_list;
    MoveL p_fine, v1000, fine, tcp1;
    syncmove;
    ...
ENDPROC

PROC syncmove()
    SyncMoveOn sync2, task_list;
    MoveL * \ID:=10, v100, z10, tcp1 \WOBJ:= rob2_obj;
    MoveL * \ID:=20, v100, fine, tcp1 \WOBJ:= rob2_obj;
    SyncMoveOff sync3;
    UNDO
        SyncMoveUndo;
ENDPROC
```

If the program is stopped while the execution is inside the procedure *syncmove* and the program pointer is moved out of the procedure *syncmove,* then all instruction inside the UNDO handler is executed. In this example the instruction *SyncMoveUndo* is executed and the system is set to independent movement mode.

Program execution

Force reset of synchronized coordinated movements and set the system to independent movement mode.

It's enough to execute *SyncMoveUndo* in one program task to set the whole system to the independent movement mode. The instruction can be executed several times without any error if the system is already in independent movement mode.

The system is set to the default independent movement mode also

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

Syntax

```
SyncMoveUndo ';'
```

Related information

Table 51

	Described in:
Specify cooperated program tasks	Data Types - *tasks*
Identity for synchronization point	Data Types - *syncident*
Start coordinated synchronized movements	Instruction - *SyncMoveOn*
End coordinated synchronized movements	Instruction - *SyncMoveOff*
Test if in synchronized mode	Function - *IsSyncModeOn*

SyncToSensor - Sync to sensor

SyncToSensor (Sync To Sensor) is used to start or stop synchronization of robot movement to sensor movement.

Example

```
WaitSensor Ssync1;
MoveL *, v1000, z10, tool, \WObj:=wobj0;
SyncToSensor Ssync1\On;
MoveL *, v1000, z20, tool, \WObj:=wobj0;
MoveL *, v1000, z20, tool, \WObj:=wobj0;
SyncToSensor Ssync1\Off;
```

Arguments

SyncToSensor Mecunt [\On] | [\Off]

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

The moving mechanical unit to which the robot position in the instruction is related.

Program execution

SyncToSensor SSYNC1 /On means that the robot starts to move synchronized with sensor SSYNC1 .So the robot passes at the teached robtarget at the same time as the sensor passes the external position stored in the robtarget .

SyncToSensor SSYNC1 /Off means that the robot stops moving synchronized with the sensor .

Limitations

If the instruction SyncToSensor SSYNC1 /On is issued while the sensor has not been connected via WaitSensor then the robot will stop.

Syntax

SyncToSensor
[Mecunt':='] < persistent (PERS) of *mecunit*>
['\' On] | ['\' Off] ';'

Related information

Table 52

	Described in:
Wait for connection on sensor	Instructions - *WaitSensor*
Drop object on sensor	Instructions - *DropSensor*

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

Table 53

	Described in:
Expressions	Basic Characteristics - *Expressions*

TestSignDefine - Define test signal

TestSignDefine is used to define one test signal for the robot motion system.

A test signal continuously mirrors some specified motion data stream, for example, torque reference for some specified axis. The actual value at a certain time can be read in RAPID with the function *TestSignRead*.

Only test signals for external robot axes can be reached.
For use of the test signal for the master robot axes or the need for use of not predefined test signals, please contact the nearest ABB Flexible Automation centre.

Example

TestSignDefine 1, resolver_angle, Orbit, 2, 0,1;

Test signal *resolver_angle* connected to channel *1,* will give the value of the resolver angle for external robot *Orbit* axis *2,* sampled at 100 ms rate.

Arguments

TestSignDefine Channel SignalId MechUnit Axis SampleTime

Channel **Data type:** *num*

The channel number 1-12 to be used for the test signal.
The same number must be used in the function *TestSignRead* for reading the actual value of the test signal.

SignalId **Data type:** *testsignal*

The name or number of the test signal.
Refer to predefined constants described in data type *testsignal.*

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

The name of the mechanical unit.

Axis **Data type:** *num*

The axis number within the mechanical unit.

SampleTime **Data type:** *num*

Sample time in seconds.

For sample time < 0.004 s, the function *TestSignRead* returns the mean value of the latest available internal samples as shown in the table below.

Table 54 Specification of sample time

Sample Time in seconds	Result from TestSignRead
0	Mean value of the latest 8 samples generated each 0.5 ms
0.001	Mean value of the latest 4 samples generated each 1 ms
0.002	Mean value of the latest 2 samples generated each 2 ms
Greater or equal to 0.004	Momentary value generated at specified sample time
0.1	Momentary value generated at specified sample time 100 ms

Program execution

The definition of test signal is activated and the robot system starts the sampling of the test signal.

The sampling of the test signal is active until:

- A new *TestSignDefine* instruction for the actual channel is executed
- All test signals are deactivated with execution of instruction *TestSignReset*
- All test signals are deactivated with a warm start of the system

Error handling

If there is an error in the parameter *MechUnit*, the system parameter ERRNO is set to ERR_UNIT_PAR. If there is an error in the parameter *Axis*, ERRNO is set to ERR_AXIS_PAR.

Syntax

```
TestSignDefine
    [ Channel ':=' ] < expression (IN) of num> ','
    [ SignalId ':=' ] < expression (IN) of testsignal> ','
    [ MechUnit ':=' ] < variable (VAR) of mecunit> ','
    [Axis ':=' ] < expression (IN) of num> ','
    [ SampleTime ':=' ] < expression (IN) of num > ';'
```

Related information

Table 55

	Described in:
Test signal	Data Types - *testsignal*
Read test signal	Functions - *TestSignRead*
Reset test signals	Instructions - *TestSignReset*

TestSignReset - Reset all test signal definitions

TestSignReset is used to deactivate all previously defined test signals.

Example

```
TestSignReset;
```

Deactivate all previously defined test signals.

Program execution

The definitions of all test signals are deactivated and the robot system stops the sampling of any test signals.

The sampling of defined test signals is active until:

- A warm start of the system
- Execution of this instruction *TestSignReset*

Syntax

```
TestSignReset';'
```

Related information

Table 56

	Described in:
Define test signal	Instructions - *TestSignDefine*
Read test signal	Functions - *TestSignRead*

TextTabInstall - Installing a text table

TextTabInstall is used to install a text table in the system.

Example

```
! System Module with Event Routine to be executed at event
! POWER ON, RESET or START

PROC install_text()
    IF TextTabFreeToUse("text_table_name") THEN
        TextTabInstall "HOME:/text_file.eng";
    ENDIF
ENDPROC
```

The first time the event routine *install_text* is executed, the function *TextTabFreeToUse* returns TRUE and the text file *text_file.eng* is installed in the system. After that the installed text strings can be fetched from the system to RAPID by the functions *TextTabGet* and *TextGet*.

Next time the event routine *install_text* is executed, the function *TextTabFreeToUse* returns FALSE and the installation is **not** repeated.

Arguments

TextTabInstall File

File **Data type:** *string*

The file path and the file name to the file that contains text strings to be installed in the system.

Limitations

Limitations for installation of text tables (text resources) in the system:

- It is not possible to install the same text table more than once in the system
- It is not possible to uninstall (free) a single text table from the system.
  The only way to uninstall text tables from the system is to cold start the system. All text tables (both system and user defined) will then be uninstalled.

Error handling

If the file in the *TextTabInstall* instruction cannot be opened then the system variable ERRNO is set to ERR_FILEOPEN. This error can then be handled in the error handler.

Syntax

```
TextTabInstall
    [ File ':=' ] < expression (IN) of string > ';'
```

Related information

Table 57

	Described in:
Test whether text table free	Functions - *TextTabFreeToUse*
Format of text files	RAPID Kernel Reference Manual - *Text files*
Get text table number	Functions - *TextTabGet*
Get text from system text tables	Functions - *TextGet*
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*

TPErase - Erases text printed on the FlexPendant

TPErase (FlexPendant Erase) is used to clear the display of the FlexPendant.

Example

```
TPErase;
TPWrite "Execution started";
```

The FlexPendant display is cleared before *Execution started* is written.

Program execution

The FlexPendant 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

Table 58

	Described in:
Writing on the FlexPendant	RAPID Summary - *Communication*

TPReadFK - Reads function keys

TPReadFK (FlexPendant Read Function Key) is used to write text on 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 FlexPendant display and the function keys 4 and 5 are activated by means of the text strings *Yes* and *No* respectively (see Figure 12). 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.

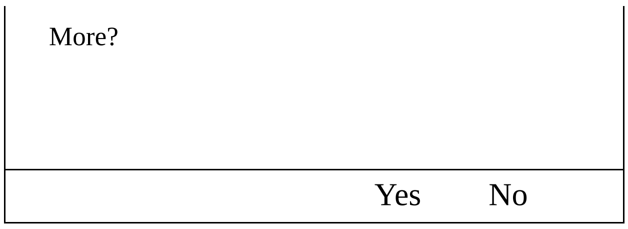


Figure 12 The operator can input information via the function keys.

Arguments

TPReadFK TPAnswer TPText TPFK1 TPFK2 TPFK3 TPFK4 TPFK5 [\MaxTime] [\DIBreak] [\DOBreak] [\BreakFlag]

TPAnswer **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.

TPText **Data type:** *string*

The information text to be written on the display (a maximum of 80 characters).

TPFKx *(Function key text)* **Data type:** *string*

The text to be written on the appropriate function key (a maximum of 22 characters). TPFK1 is the left-most key.

Function keys without text 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.

[\DOBreak] *(Digital Output Break)* **Data type:** *signaldo*

The digital signal that support termination request from other tasks. If no button is selected 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_DOBREAK 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, ERR_TP_ DIBREAKand ERR_TP_DOBREAK 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. There can be up to 7 lines above the new text written.

Text is written on the appropriate function keys.

Program execution waits until one of the activated function keys is depressed.

Description of concurrent *TPReadFK* or *TPReadNum* request on FlexPendant (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 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.

Error handling

If there is a timeout (parameter *\MaxTime*) before an input from the operator, the system variable ERRNO is set to ERR_TP_MAXTIME and the execution continues in the error handler.

If digital input is set (parameter *\DIBreak*) before an input from the operator, the system variable ERRNO is set to ERR_TP_DIBREAK and the execution continues in the error handler.

If a digital output occurred (parameter *\DOBreak*) before an input from the operator, the system variable ERRNO is set to ERR_TP_DOBREAK and the execution continues in the error handler.

If there is no client, e.g. a Flex Pendant, to take care of the instruction, the system variable ERRNO is set to ERR_TP_NO_CLIENT and the execution continues in the error handler.

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

Limitations

Avoid using a too small value for the timeout parameter \MaxTime when TPReadFK is frequently executed, for example in a loop. It can result in an unpredictable behaviour of the system performance, like slow TPU response.

Predefined data

CONST string stEmpty := “”;

The predefined constant *stEmpty* should be used for Function Keys without text. Using *stEmpty* instead of “”saves about 80 bytes for every Function Key without text.

Syntax

TPReadFK
    [TPAnswer’:=’] <var or pers (**INOUT**) of *num*>’,’
    [TPText’:=’] <expression (**IN**) of *string*>’,’
    [TPFK1 ’:=’] <expression (**IN**) of *string*>’,’
    [TPFK2 ’:=’] <expression (**IN**) of *string*>’,’
    [TPFK3 ’:=’] <expression (**IN**) of *string*>’,’
    [TPFK4 ’:=’] <expression (**IN**) of *string*>’,’
    [TPFK5 ’:=’] <expression (**IN**) of *string*>
    [’\’MaxTime ’:=’ <expression (**IN**) of *num*>]
    [’\’DIBreak ’:=’ <variable (**VAR**) of *signaldi*>]
    [’\’DOBreak ’:=’ <variable (**VAR**) of *signaldo*>]
    [’\’BreakFlag ’:=’ <var or pers (**INOUT**) of *errnum*>]’;’

Related information

Table 59

	Described in:
Writing to and reading from the Flex-Pendan	RAPID Summary - *Communication*t
Replying via the FlexPendant	Running Production

TPReadNum - Reads a number from the FlexPendant

TPReadNum (FlexPendant Read Numerical) is used to read a number from the FlexPendant.

Example

TPReadNum reg1, “How many units should be produced?“;

The text *How many units should be produced?* is written on the FlexPendant display. Program execution waits until a number has been input from the numeric keyboard on the FlexPendant. That number is stored in *reg1*.

Arguments

TPReadNum TPAnswer TPText [\MaxTime] [\DIBreak] [\DOBreak] [\BreakFlag]

TPAnswer **Data type:** *num*

The variable for which the number input via the FlexPendant is returned.

TPText **Data type:** *string*

The information text to be written on the FlexPendant (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.

[\DOBreak] *(Digital Output Break)* **Data type:** *signaldo*

The digital signal that support termination request from other tasks. If no button is selected 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_DOBREAK 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, ERR_TP_ DIBREAK and ERR_TP_DOBREAK 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. There can be up to 7 lines above the new text written.

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 FlexPendant 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 FlexPendant display. The routine *produce_part* is then repeated the number of times that is input via the FlexPendant.

Error handling

If time out (parameter *\MaxTime*) before input from the operator, the system variable ERRNO is set to ERR_TP_MAXTIME and the execution continues in the error handler.

If digital input set (parameter *\DIBreak*) before input from the operator, the system variable ERRNO is set to ERR_TP_DIBREAK and the execution continues in the error handler.

If a digital output occurred (parameter *\DOBreak*) before an input from the operator, the system variable ERRNO is set to ERR_TP_DOBREAK and the execution continues in the error handler.

If there is no client, e.g. a Flex Pendant, to take care of the instruction, the system variable ERRNO is set to ERR_TP_NO_CLIENT and the execution continues in the error handler.

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

Syntax

TPReadNum
[TPAnswer':='] <var or pers (**INOUT**) of *num*>','
[TPText':='] <expression (**IN**) of *string*>
['\'MaxTime ':=' <expression (**IN**) of *num*>]
['\'DIBreak ':=' <variable (**VAR**) of *signaldi*>]
['\'DOBreak ':=' <variable (**VAR**) of *signaldo*>]
['\'BreakFlag ':=' <var or pers (**INOUT**) of *errnum*>] ';'

Related information

Table 60

	Described in:
Writing to and reading from the FlexPendant	RAPID Summary - *Communication*
Entering a number on the FlexPendant	Production Running
Examples of how to use the arguments MaxTime, DIBreak and BreakFlag	Instructions - *TPReadFK*

TPShow - Switch window on the FlexPendant

TPShow (FlexPendant Show) is used to select FlexPendant 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 FlexPendant Window before the current FlexPendant 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 before current	=	Latest used FlexPendant Window FlexPendant Window.
TP_SCREENVIEWER Viewer	=	*Screen Viewer Window,* if the Screen option is active.

Predefined data

```
CONST tpnum TP_PROGRAM := 1;
CONST tpnum TP_LATEST := 2;
CONST tpnum TP_SCREENVIEWER := 3;
```

Program execution

The selected FlexPendant Window will be activated.

Syntax

TPShow
[Window':='] <expression (**IN**) of *tpnum*> ';'

Related information

Table 61

	Described in:
Communicating using the FlexPendant	RAPID Summary - *Communication*
FlexPendant Window number	Data Types - *tpnum*

TPWrite - Writes on the FlexPendant

TPWrite (FlexPendant Write) is used to write text on the FlexPendant. The value of certain data can be written as well as text.

Examples

TPWrite "Execution started";

The text *Execution started* is written on the FlexPendant.

TPWrite "No of produced parts="\Num:=reg1;

If, for example, *reg1 holds the value 5, the text No of produced parts=5*, is written on the FlexPendant.

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 FlexPendant always begins on a new line. When the display is full of text (11 lines), this text is moved up one line first.

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
    [TPText':='] <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

Table 62

	Described in:
Clearing and reading the FlexPendant	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. 6) events can be defined using the instructions *TriggIO*, *TriggEquip,* or *TriggInt,* and afterwards these definitions are referred to in the instruction *TriggC*.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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*.

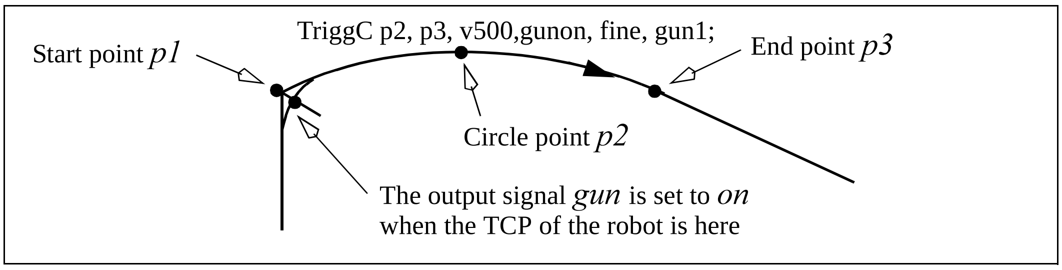

Start point *p1*
TriggC p2, p3, v500,gunon, fine, gun1;
End point *p3*
Circle point *p2*
The output signal *gun* is set to *on*
when the TCP of the robot is here

Figure 13 Example of fixed-position IO event.

Arguments

TriggC **[\Conc] CirPoint ToPoint [\ID] Speed [\T] Trigg_1 [\T2] [\T3] [\T4] [\T5] [\T6] Zone [\Inpos] Tool [\WObj] [\Corr]**

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

Subsequent instructions are executed while the robot is moving. The argument can be used to avoid unwanted stops, caused by overloaded CPU, when using fly-by points, and in this way shorten cycle time.This is useful when the programmed points are very close together at high speeds.The argument is also useful when, for example, communicating with external equipment and synchronisation between the external equipment and robot movement 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.

This argument can not be used in coordinated synchronized movement in a MultiMove System.

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).

[\ID] *(Synchronization id)* **Data type:** *identno*

This argument must be used in a MultiMove System, if coordinated synchronized movement, and is not allowed in any other cases.

The specified id number must be the same in all cooperating program tasks. The id number gives a guarantee that the movements are not mixed up at runtime.

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*.

[\T5] **(***Trigg 5)* **Data type:** *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T6] **(***Trigg 6)* **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.

[\Inpos] *(In position)* **Data type:** *stoppointdata*

This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the *Zone* parameter.

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.

Error handling

If the programmed *ScaleValue* argument for the specified analog output signal *AOp* in some of the connected *TriggSpeed* instructions, results is out of limit for the analog signal together with the programmed *Speed* in this instruction, the system variable ERRNO is set to ERR_AO_LIM.

If the programmed *DipLag* argument in some of the connected *TriggSpeed* instructions, is too big in relation to the used Event Preset Time in System Parameters, the system variable ERRNO is set to ERR_DIPLAG_LIM.

These errors can be handled in the error handler.

Limitations

General limitations according to instruction MoveC.

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 > ’,’
    [ ’\’ ID ’:=’ < expression (IN) of identno >]’,’
    [ 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 > ]
    [ ’\’ T5 ’:=’ < variable (VAR) of triggdata > ]
    [ ’\’ T6 ’:=’ < variable (VAR) of triggdata > ] ‘,’
    [Zone ’:=’ ] < expression (IN) of zonedata >
    [ ’\’ Inpos ’:=’ < expression (IN) of stoppointdata > ]‘,’
    [ Tool ’:=’ ] < persistent (PERS) of tooldata >
    [ ’\’ WObj ’:=’ < persistent (PERS) of wobjdata > ]
    [ ’\’ Corr ]’;’
```

Related information

Table 63

	Described in:
Linear movement with triggers	Instructions - *TriggL*
Joint movement with triggers	Instructions - *TriggJ*
Definition of triggers	Instructions - *TriggIO, TriggEquip, TriggInt or TriggCheckIO*
Writes to a corrections entry	Instructions - *CorrWrite*
Circular movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of stop point data	Data Types - *stoppointdata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in genera	Motion Principles

TriggCheckIO - Defines IO check at a fixed position

TriggCheckIO is used to define conditions for testing the value of a digital, a group of digital, or an analog input or output signal at a fixed position along the robot's movement path. If the condition is fulfilled there will be no specific action, but if it is not, an interrupt routine will be run after the robot has optionally stopped on path as fast as possible.

To obtain a fixed position I/O check, *TriggCheckIO* compensates for the lag in the control system (lag between servo and robot).

The data defined is used for implementation in one or more subsequent *TriggL*, *TriggC* or *TriggJ* instructions.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

Examples

```
VAR triggdata checkgrip;
VAR intnum intno1;

CONNECT intno1 WITH trap1;
TriggCheckIO checkgrip, 100, airok, EQ, 1, intno1;

TriggL p1, v500, checkgrip, z50, grip1;
```

The digital input signal *airok* is checked to have the value *1* when the TCP is *100* mm before the point *p1*. If it is set, normal execution of the program continues; if it is not set, the interrupt routine *trap1* is run.

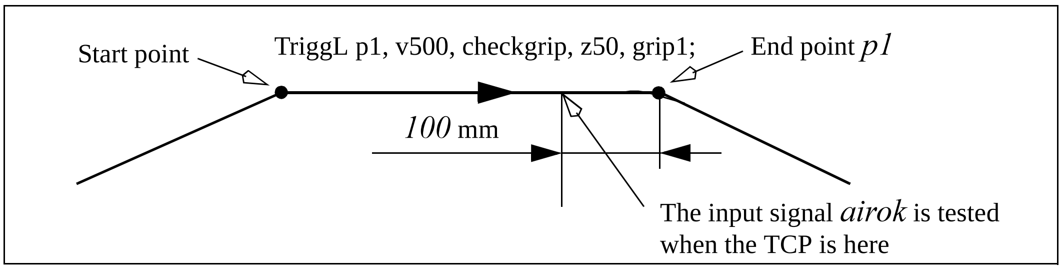

Start point
TriggL p1, v500, checkgrip, z50, grip1;
End point *p1*
100 mm
The input signal *airok* is tested when the TCP is here

Figure 14 Example of fixed-position IO check.

Arguments

TriggCheckIO TriggData Distance [\Start] | [\Time] Signal Relation CheckValue [\StopMove] 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 I/O check 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.

Signal **Data type:** *signalxx*

The name of the signal that will be tested. May be anytype of IO signal.

Relation **Data type:** *opnum*

Defines how to compare the actual value of the signal with the one defined by the argument *CheckValue*. Refer to the *opnum* data type for the list of the predefined constants to be used.

CheckValue **Data type:** *num*

Value to which the actual value of the input or output signal is to be compared (within the allowed range for the current signal).

[\StopMove] **Data type:** *switch*

Specifies that, if the condition is **not** fulfilled, the robot will stop on path as quickly as possible before the interrupt routine is run.

Interrupt **Data type:** *intnum*

Variable used to identify the interrupt routine to run.

Program execution

When running the instruction *TriggCheckIO*, 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 *TriggCheckIO*:

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).

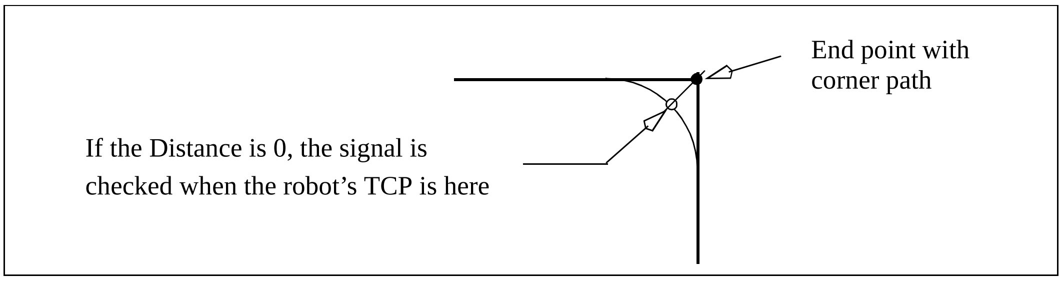

End point with
corner path
If the Distance is 0, the signal is
checked when the robot's TCP is here

Figure 15 Fixed position I/O check on a corner path.

The fixed position I/O check will be done 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...*).

When the TCP of the robot is at specified place on the path, following I/O check will be done by the system:

- Read the value of the I/O signal
- Compare the read value with *CheckValue* according specified *Relation*
- If the comparision is TRUE, nothing more is done
- If the comparison is FALSE following is done:
- If optional parameter *StopMove* is present, the robot is stopped on the path as quick as possible
- Generate and execute the specified TRAP routine

Examples

```
VAR triggdata checkgate;
VAR intnum gateclosed;

CONNECT gateclosed WITH waitgate;
TriggCheckIO checkgate, 150, gatedi, EQ, 1 \StopMove, gateclosed;
TriggL p1, v600, checkgate, z50, grip1;
                        ....



TRAP waitgate
                        ! log some information
                        ...
                        WaitDI gatedi,1;
                        StartMove;
ENDTRAP
```

The gate for the next workpiece operation is checked to be open (digital input signal *gatedi* is checked to have the value *1*) when the TCP is *150* mm before the point *p1*. If it is open, the robot will move on to *p1* and continue; if it is not open, the robot is stopped on path and the interrupt routine *waitgate* is run. This interrupt routine logs some information and typically waits for the conditions to be OK to execute a *StartMove* instruction in order to restart the interrupted path.

Limitations

I/O checks with distance (without the argument *Time*) is intended for flying points (corner path). I/O checks with distance, using stop points, results in worse accuracy than specified below.

I/O checks with time (with the argument *Time*) is intended for stop points. I/O checks with time, using flying points, results in worse accuracy than specified below.

I/O checks 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 IO check 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 test of digital inputs +/- 5 ms.
Typical repeat accuracy values for test of digital inputs +/- 2 ms.

Syntax

TriggCheckIO
    [TriggData ':='] < variable (**VAR**) of *triggdata*> ','
    [Distance ':='] < expression (**IN**) of *num*>
    ['\' Start] | ['\' Time] ','
    [Signal ':='] < variable (**VAR**) of *anytype*> ','
    [Relation ':='] < expression (**IN**) of *opnum*> ','
    [CheckValue ':='] < expression (**IN**) of *num*>
    ['\' StopMove] ','
    [Interrupt ':='] < variable(**VAR**) of *intnum*> ';'

Related information

Table 64

	Described in:
Use of triggers	Instructions - *TriggL*, *TriggC*, *TriggJ*
Definition of position-time I/O event	Instruction - *TriggIO,TriggEquip*
Definition of position related interrupts	Instruction - *TriggInt*
More examples	Data Types - *triggdata*
Definition of comparison operators	Data Types - *opnum*

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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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.

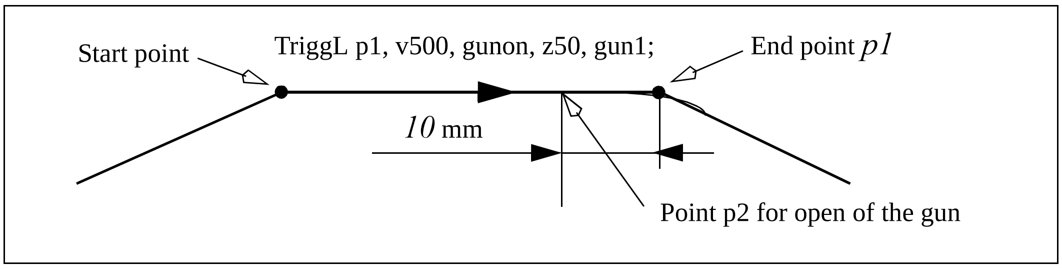

Start point
TriggL p1, v500, gunon, z50, gun1;
End point *p1*
10 mm
Point p2 for open of the gun

Figure 16 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.

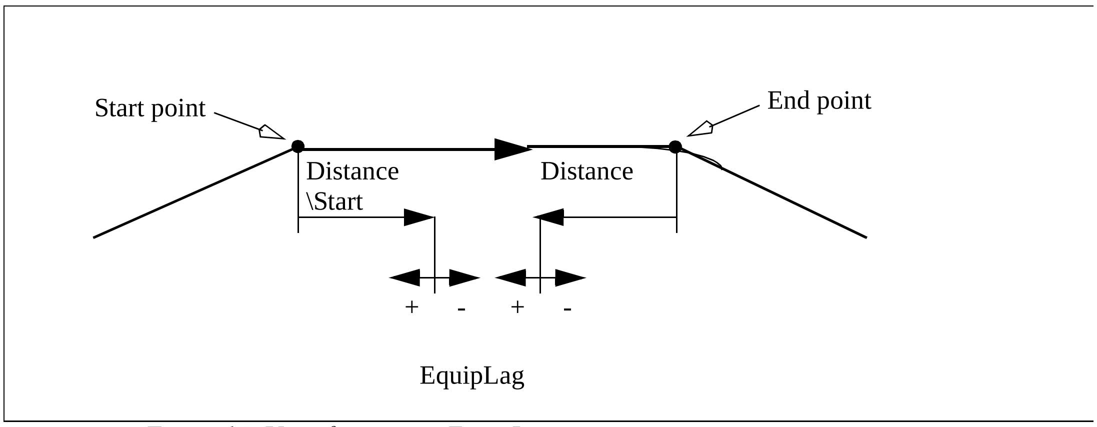


Figure 17 Use of argument EquipLag.

[\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*:

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).

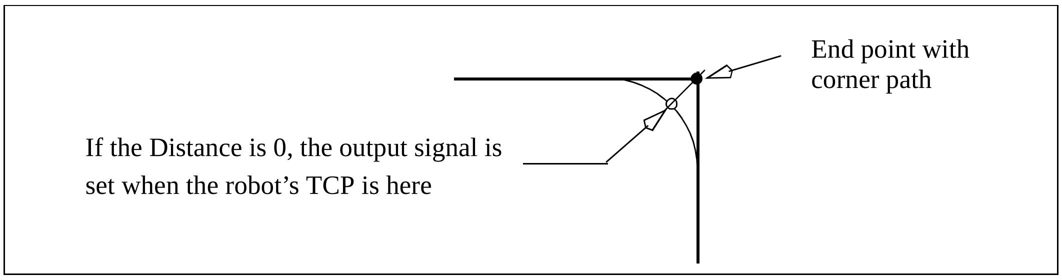

End point with
corner path
If the Distance is 0, the output signal is
set when the robot's TCP is here

Figure 18 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*.

Error handling

If the programmed *SetValue* argument for the specified analog output signal *AOp* is out of limit, the system variable ERRNO is set to ERR_AO_LIM. This error can be handled in the error handler.

Limitations

I/O events with distance (with the argument *EquipLag* = 0) 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*). The lag can vary between different robot types, for example it is lower for IRB140.
- 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 *EquipLag* != 0) 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.

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*>
    ['\' Inhib ':=' < persistent (**PERS**) of *bool*>] ','

Related information

Table 65

	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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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.

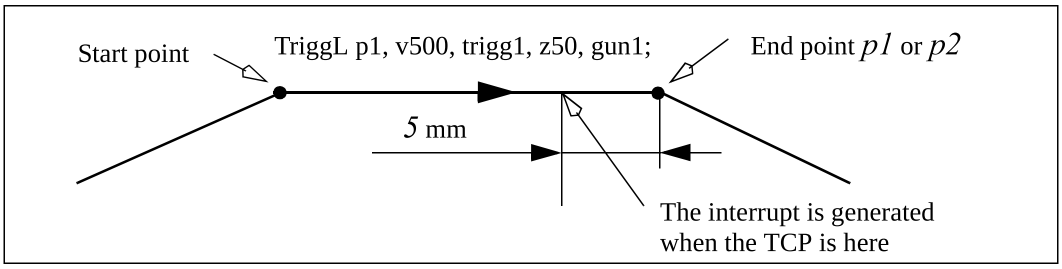

Start point
TriggL p1, v500, trigg1, z50, gun1;
End point p1 or p2
5 mm
The interrupt is generated when the TCP is here

Figure 19 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).

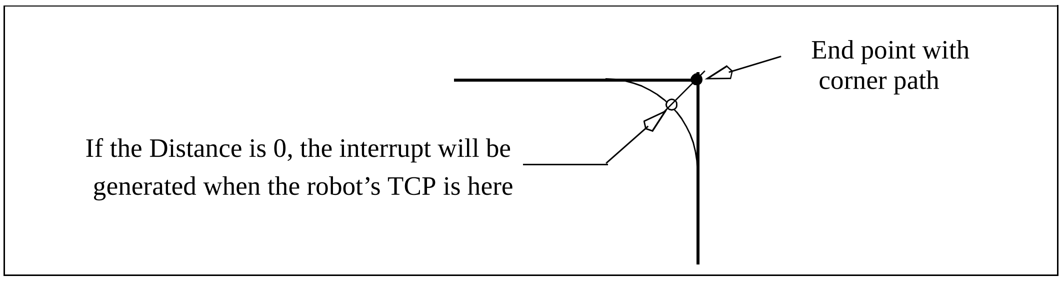

End point with corner path
If the Distance is 0, the interrupt will be generated when the robot’s TCP is here

Figure 20 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

Table 66

	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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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*.

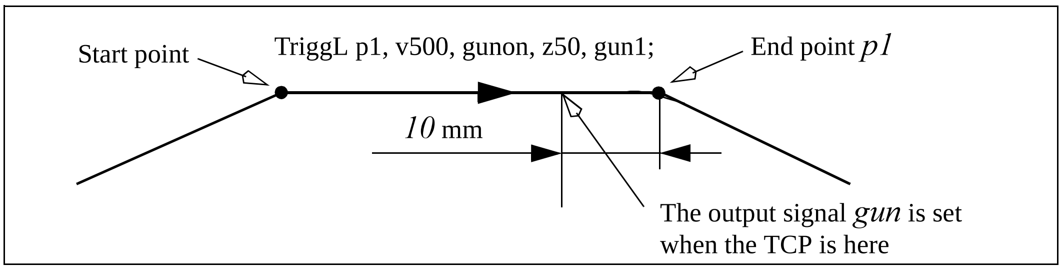

Start point
TriggL p1, v500, gunon, z50, gun1;
End point p1
10 mm
The output signal gun is set when the TCP is here

Figure 21 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, group, or analog output signal.

Only used to delay setting of 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*:

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).

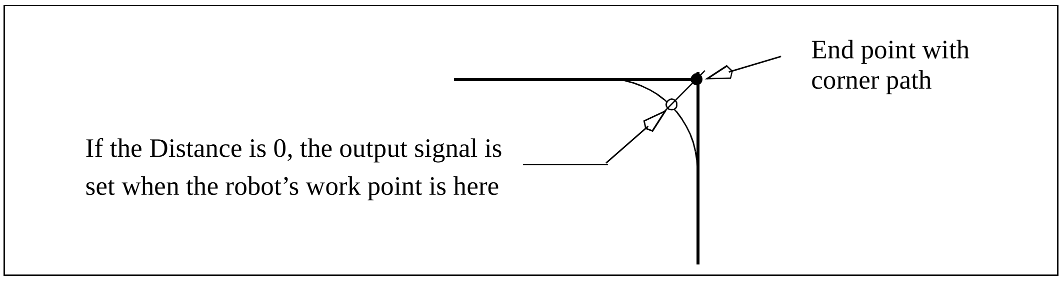

End point with corner path
If the Distance is 0, the output signal is set when the robot's work point is here

Figure 22 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*.

Error handling

If the programmed *SetValue* argument for the specified analog output signal *AOp* is out of limit, the system variable ERRNO is set to ERR_AO_LIM. This error can be handled in the error handler.

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.

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

Table 67

	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 (TriggJoint) is used to set output signals and/or run interrupt routines at fixed positions, at the same time as the robot is moving quickly from one point to another when that movement does not have be in a straight line.

One or more (max. 6) events can be defined using the instructions *TriggIO, TriggEquip,* or *TriggInt,* and afterwards these definitions are referred to in the instruction *TriggJ.*

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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*.

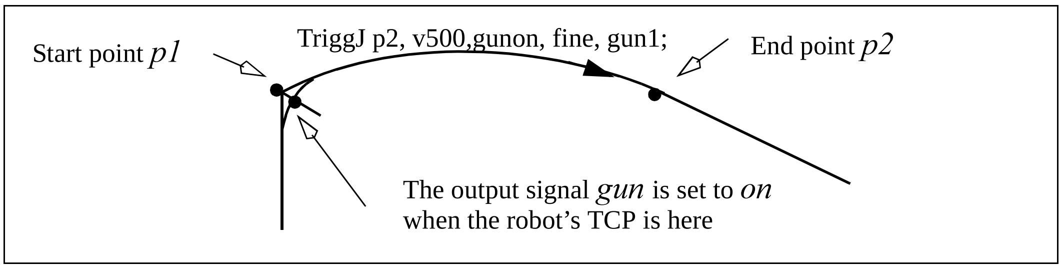

Start point *p1*
TriggJ p2, v500,gunon, fine, gun1;
End point *p2*
The output signal *gun* is set to *on* when the robot's TCP is here

Figure 23 Example of fixed-position IO event.

Arguments

TriggJ **[\Conc] ToPoint [\ID] Speed [\T] Trigg_1 [\T2] [\T3] [\T4] [\T5] [\T6] Zone [\Inpos] Tool [\WObj]**

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

Subsequent instructions are executed while the robot is moving. The argument can be used to avoid unwanted stops, caused by overloaded CPU, when using fly-by points, and in this way shorten cycle time.This is useful when the programmed points are very close together at high speeds.

The argument is also useful when, for example, communicating with external equipment and synchronisation between the external equipment and robot movement 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, the subsequent instruction is only executed after the robot has reached the specified stop point or 100 ms before the specified zone.

This argument can not be used in coordinated synchronized movement in a MultiMove System.

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).

[\ID] *(Synchronization id)* **Data type:** *identno*

This argument must be used in a MultiMove System, if coordinated synchronized movement, and is not allowed in any other cases.

The specified id number must be the same in all cooperating program tasks. The id number gives a guarantee that the movements are not mixed up at runtime.

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*.

[\T5] (*Trigg 5)* **Data type:** *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T6] (*Trigg 6)* **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.

[\Inpos] *(In position)* **Data type:** *stoppointdata*

This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the *Zone* parameter.

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.

Error handling

If the programmed *ScaleValue* argument for the specified analog output signal *AOp* in some of the connected *TriggSpeed* instructions, results in out of limit for the analog signal together with the programmed *Speed* in this instruction, the system variable ERRNO is set to ERR_AO_LIM.

If the programmed *DipLag* argument in some of the connected *TriggSpeed* instructions, is too big in relation to the Event Preset Time used in System Parameters, the system variable ERRNO is set to ERR_DIPLAG_LIM.

These errors can be handled in the error handler.

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 sequence of trigger activities for an “incomplete movement”.

Syntax

TriggJ
  ['\' Conc ',']
  [ToPoint ':='] < expression (**IN**) of *robtarget* > ','
  ['\' ID ':=' < expression (**IN**) of *identno* >]','
  [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* >]
  ['\' T5 ':=' < variable (**VAR**) of *triggdata* >]
  ['\' T6 ':=' < variable (**VAR**) of *triggdata* >] ','
  [Zone ':='] < expression (**IN**) of *zonedata* >
  ['\' Inpos ':=' < expression (**IN**) of *stoppointdata* >]','
  [Tool ':='] < persistent (**PERS**) of *tooldata* >
  ['\' WObj ':=' < persistent (**PERS**) of *wobjdata* >] ';'

Related information

Table 68

	Described in:
Linear movement with triggs	Instructions - *TriggL*
Circular movement with triggers	Instructions - *TriggC*
Definition of triggers	Instructions - *TriggIO, TriggEquip, TriggInt or TriggCheckIO*
Joint movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of stop point data	Data Types - *stoppointdata*
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. 6) events can be defined using the instructions *TriggIO, TriggEquip,* or *TriggInt,* and afterwards these definitions are referred to in the instruction *TriggL*.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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*.

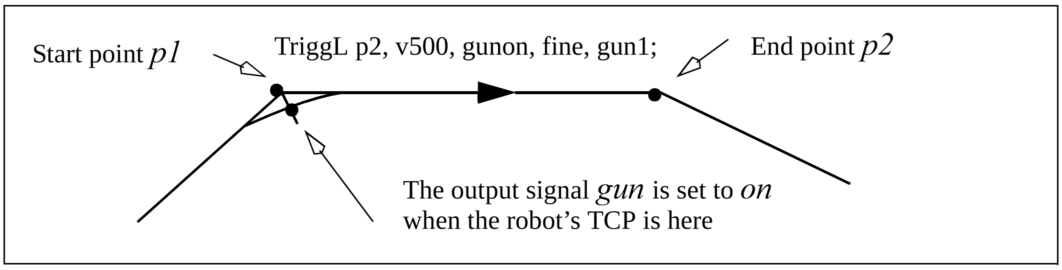

Start point *p1*
TriggL p2, v500, gunon, fine, gun1;
End point *p2*
The output signal *gun* is set to *on*
when the robot's TCP is here

Figure 24 Example of fixed-position IO event.

Arguments

TriggL **[\Conc] ToPoint [\ID] Speed [\T] Trigg_1 [\T2] [\T3] [\T4] [\T5] [\T6] Zone [\Inpos] Tool [\WObj] [\Corr]**

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

Subsequent instructions are executed while the robot is moving. The argument can be used to avoid unwanted stops, caused by overloaded CPU, when using fly-by points, and in this way shorten cycle time.This is useful when the programmed points are very close together at high speeds.

The argument is also useful when, for example, communicating with external equipment and synchronisation between the external equipment and robot movement 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.

This argument can not be used in coordinated synchronized movement in a MultiMove System.

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).

[\ID] *(Synchronization id)* **Data type:** *identno*

This argument must be used in a MultiMove System, if coordinated synchronized movement, and is not allowed in any other cases.

The specified id number must be the same in all cooperating program tasks. The id number gives a guarantee that the movements are not mixed up at runtime.

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*.

[\T5] **(***Trigg 5)* **Data type:** *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T6] **(***Trigg 6)* **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.

[\Inpos] *(In position)* **Data type:** *stoppointdata*

This argument is used to specify the convergence criteria for the position of the robot's TCP in the stop point. The stop point data substitutes the zone specified in the *Zone* parameter.

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.

Error handling

If the programmed *ScaleValue* argument for the specified analog output signal *AOp* in some of the connected *TriggSpeed* instructions, results in out of limit for the analog signal together with the programmed *Speed* in this instruction, the system variable ERRNO is set to ERR_AO_LIM.

If the programmed *DipLag* argument in some of the connected *TriggSpeed* instructions, is too big in relation to the Event Preset Time used in System Parameters, the system variable ERRNO is set to ERR_DIPLAG_LIM.

These errors can be handled in the error handler.

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 > ’,’
    [ ’\’ ID ’:=’ < expression (IN) of identno >]’,’
    [ 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 > ]
    [ ’\’ T5 ’:=’ < variable (VAR) of triggdata > ]
    [ ’\’ T6 ’:=’ < variable (VAR) of triggdata > ] ‘,’
    [Zone ’:=’ ] < expression (IN) of zonedata >
    [ ’\’ Inpos ’:=’ < expression (IN) of stoppointdata > ]‘,’
    [ Tool ’:=’ ] < persistent (PERS) of tooldata >
    [ ’\’ WObj ’:=’ < persistent (PERS) of wobjdata > ]
    [ ’\’ Corr ]’;’
```

Related information

Table 69

	Described in:
Circular movement with triggers	Instructions - *TriggC*
Joint movement with triggers	Instructions - *TriggJ*
Definition of triggers	Instructions - *TriggIO, TriggEquip, TriggInt or TriggCheckIO*
Writes to a corrections entry	Instructions - *CorrWrite*
Linear movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of stop point data	Data Types - *stoppointdata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion Principles

TriggSpeed - Defines TCP speed proportional analog output with fixed position-time scale event

TriggSpeed is used to define conditions and actions for control of an analog output signal with output value proportional to the actual TCP speed. The beginning, scaling, and ending of the analog output can be specified at a fixed position-time along the robot's movement path. It is possible to use time compensation for the lag in the external equipment for the beginning, scaling, and ending of the analog output and also for speed dips of the robot.

The data defined is used in one or more subsequent *TriggL*, *TriggC*, or *TriggJ* instructions.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

Examples

```
VAR triggdata glueflow;

TriggSpeed glueflow, 0, 0.05, glue_ao, 0.8\DipLag=:0.04 \ErrDO:=glue_err;
TriggL p1, v500, glueflow, z50, gun1;

TriggSpeed glueflow, 10, 0.05, glue_ao, 1;
TriggL p2, v500, glueflow, z10, gun1;

TriggSpeed glueflow, 0, 0.05, glue_ao, 0;
TriggL p3, v500, glueflow, z50, gun1;
```

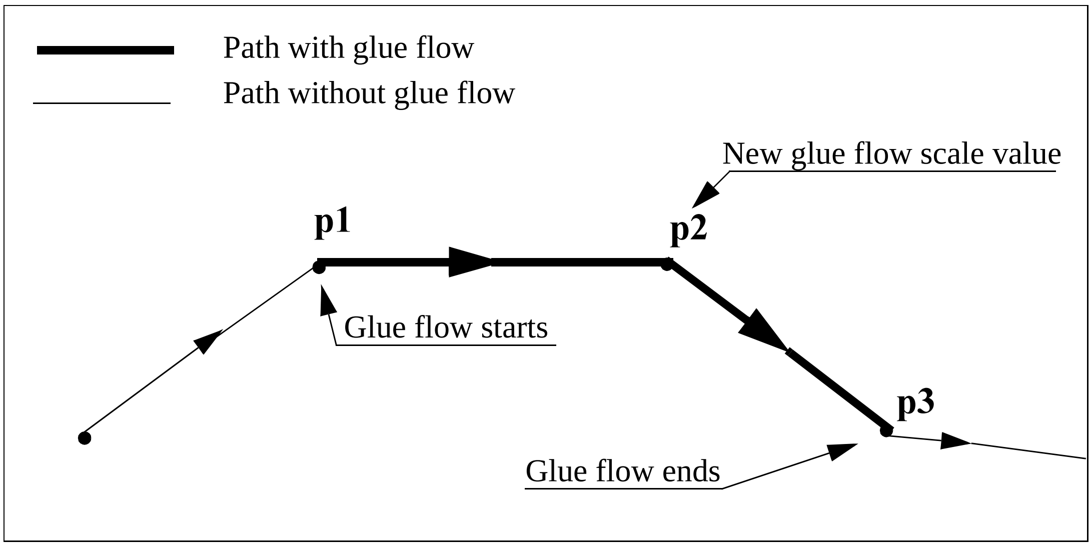


Figure 25 Example of TriggSpeed sequence.

The glue flow (analog output *glue_ao*) with scale value *0.8* start when TCP is *0.05* s before point *p1*, new glue flow scale value *1* when TCP is *10* mm plus *0.05* s before point *p2* and the glue flow ends (scale value *0*) when TCP is *0.05* s before point *p3*.

Any speed dip by the robot is time compensated in such a way that the analog output signal *glue_ao* is affected *0.04* s before the TCP speed dip occurs.

If overflow of the calculated logical analog output value in *glue_ao*, the digital output signal *glue_err* is set. If no overflow any more, *glue_err* is reset.

Arguments

TriggSpeed **TriggData Distance [\Start] ScaleLag AOp ScaleValue [\DipLag] [\ErrDO] [\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 for change of the analog output value.

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.

ScaleLag **Data type:** *num*

Specify the lag as time in s (positive value) in the external equipment for change of the analog output value (starting, scaling and ending).

For compensation of external equipment lag, this argument value means that the analog output signal is set by the robot at specified time before the TCP physically reaches the specified distance in relation to the movement start or end point.

The argument can also be used to extend the analog output beyond the end point. Set the time in seconds that the robot shall keep the analog output. Set the time with a negative sign. The limit is -0.10 seconds.

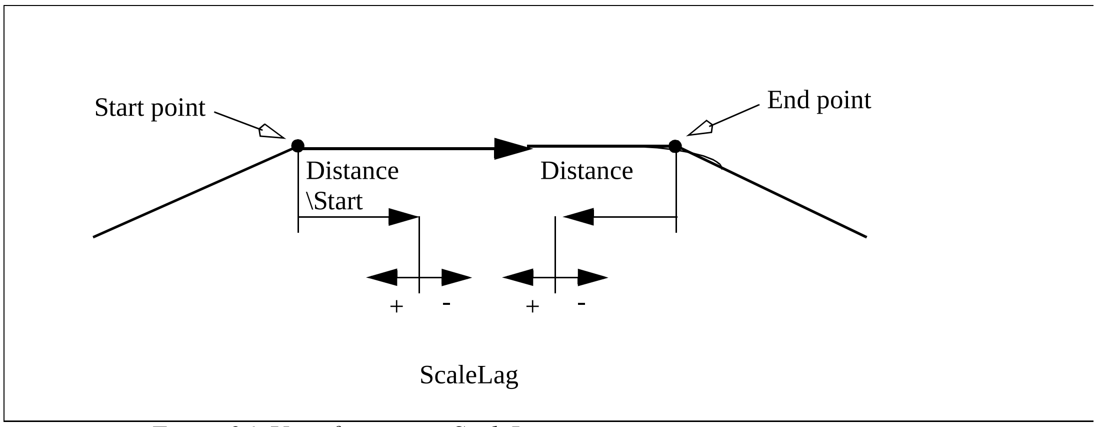


Figure 26 Use of argument ScaleLag.

AOp (*Analog Output*) **Data type:** *signalao*

The name of the analog output signal.

ScaleValue **Data type:** *num*

The scale value for the analog output signal.

The physical output value for the analog signal is calculated by the robot:

- Logical output value = Scale value * Actual TCP speed in mm/s
- Physical output value = According definition in configuration for actual analog output signal with above Logical output value as input

[\DipLag] **Data type:** *num*

Specify the lag as time in s (positive value) for the external equipment when changing of the analog output value due to robot speed dips.

For compensation of external equipment lag, this argument value means that the analog output signal is set by the robot at a specified time before the TCP speed dip occurs.

This argument can only be used by the robot for the first *TriggSpeed* (in combination with one of *TriggL*, *TriggC,* or *TriggJ*) in a sequence of several *TriggSpeed* instructions. The first specified argument value is valid for all the following *TriggSpeed* in the sequence.

[\ErrDO] (*Error Digital Output*) **Data type:** *signaldo*

The name of the digital output signal for reporting analog value overflow.

If during movement the calculation of the logical analog output value for signal in argument *AOp* result in overflow due to overspeed, this signal is set and the physical analog output value is reduced to the maximum value. If no overflow any more, the signal is reset.

This argument can only be used by the robot for the 1:st *TriggSpeed* (in combination with one of *TriggL*, *TriggC* or *TriggJ*) in a sequence of several *TriggSpeed* instructions. The 1:st given argument value is valid for all the following *TriggSpeed* in the sequence.

[\Inhib] (*Inhibit*) **Data type:** *bool*

The name of a persistent variable flag for inhibiting the setting of the analog signal at runtime.

If this optional argument is used and the actual value of the specified flag is TRUE at the time for setting the analog signal, then the specified signal *AOp* will be set to 0 instead of a calculated value.

This argument can only be used by the robot for the 1st *TriggSpeed* (in combination with one of *TriggL*, *TriggC*, or *TriggJ*) in a sequence of several *TriggSpeed* instructions. The 1st given argument value is valid for all the following *TriggSpeed* in the sequence.

Program execution

When running the instruction *TriggSpeed*, 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 *TriggSpeed*:

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).

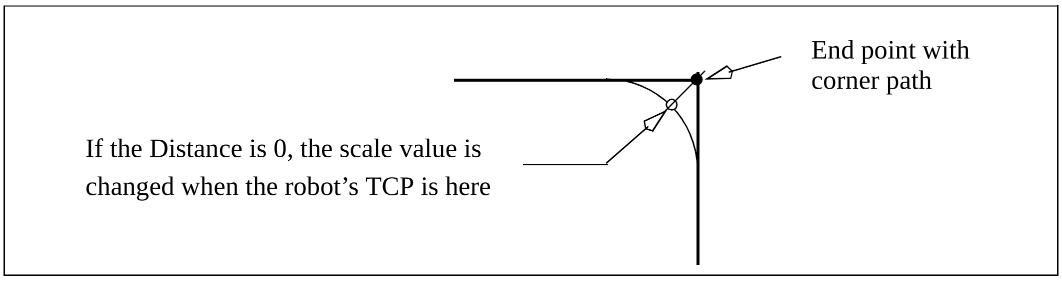
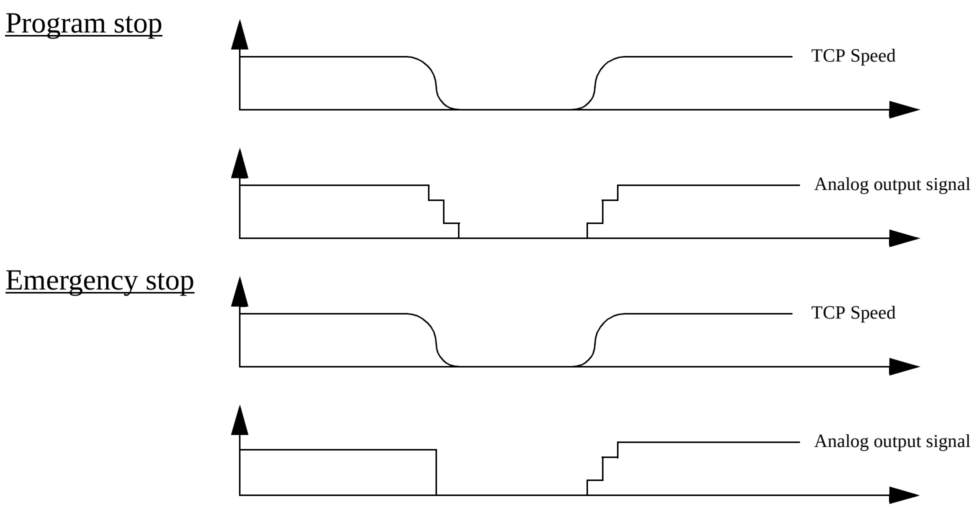

End point with corner path
If the Distance is 0, the scale value is changed when the robot's TCP is here

Figure 27 Fixed position-time scale value event on a corner path.

The position-time related scale value 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...*).

The 1:st *TriggSpeed* used by one of *TriggL*, *TriggC* or *TriggJ* instruction will internally in the system create a process with the same name as the analog output signal. The same process will be used by all succeeding *Trigg...,* which refer to same signal name and setup by a *TriggSpeed* instruction.

The process will immediately set the analog output to 0, in the event of a program emergency stop. In the event of a program stop, the analog output signal will stay TCP-speed proportional until the robot stands still. The process keeps “alive” ready for a restart. When the robot restarts, the signal is TCP-speed proportional directly from the start.



The process will “die” after handling a scale event with value 0, if no succeeding *Trigg...* is in the queue at the time.

Examples

```
VAR triggdata flow;

TriggSpeed flow, 10 \Start, 0.05, flowsignal, 0.5 \DipLag:=0.03;

MoveJ p1, v1000, z50, tool1;
TriggL p2, v500, flow, z50, tool1;
```

The analog output signal *flowsignal* is set to a logical value = (*0.5* * actual TCP speed in mm/s) *0.05* s before the TCP passes a point located *10* mm after the start point *p.* The output value is adjusted to be proportional to the actual TCP speed during the movement to *p2*.

```
...
TriggL p3, v500, flow, z10, tool1;
```

The robot moves from *p2* to *p3* still with the analog output value proportional to the actual TCP speed. The analog output value will be decreased at time *0.03* s before the robot reduce the TCP speed during the passage of the corner path *z10*.

Limitations

Accuracy of position-time related scale value event:

Typical absolute accuracy values for scale value events +/- 5 ms.
Typical repeat accuracy values for scale value events +/- 2 ms.

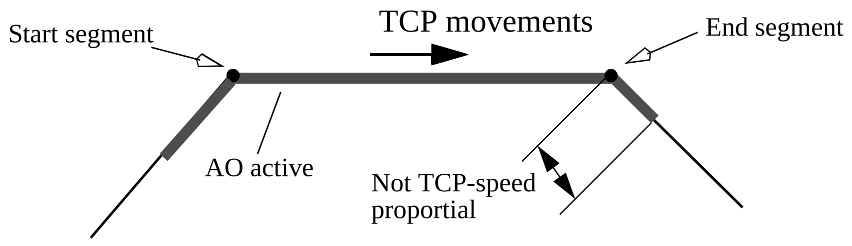
Accuracy of TCP speed dips adaptation (deceleration - acceleration phases):

Typical absolute accuracy values for TCP speed dips adaptation +/- 5 ms.
Typical repeat accuracy values for TCP speed dips adaptation +/- 2ms
(the value depends of the configured *Path resolution*).

Negative ScaleLag

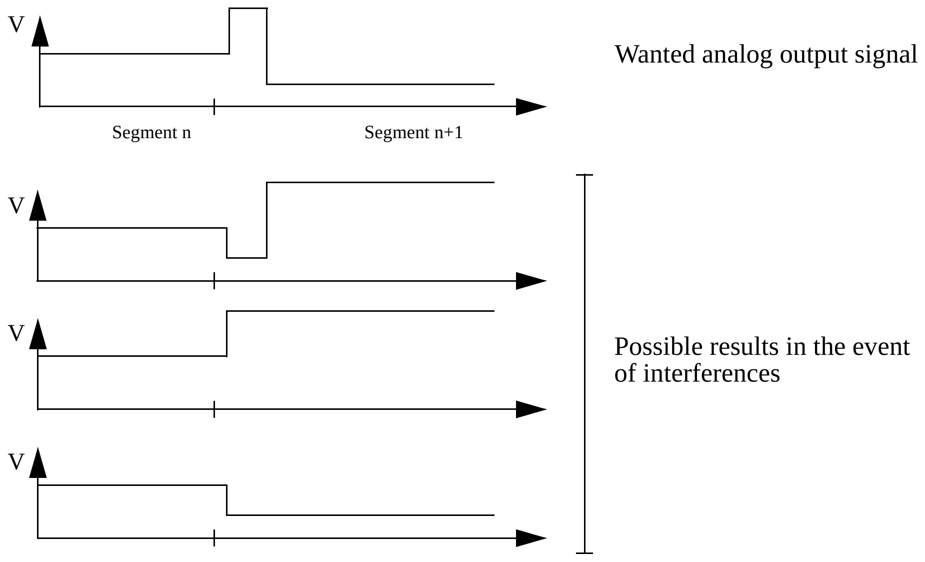
If a negative value on parameter *ScaleLag* is used to move the zero scaling over to the next segment, the analog output signal will not be reset if a program stop occurs. An emergency stop will always reset the analog signal.

The analog signal is no longer TCP-speed proportional after the end point on the segment.



Error handling

Given two consecutive segments with *TriggL/TriggSpeed* instructions. A negative value in parameter *ScaleLag* makes it possible to move the scale event from the first segment to the beginning of the second segment. If the second segment scales at the beginning, there is no control if the two scalings interfere.


V
Segment n
Segment n+1
Wanted analog output signal
V
V
V
Possible results in the event of interferences

Related system parameters

The servo parameter *Event Preset Time* is used to delay the robot to make it possible to activate/control the external equipment before the robot runs through the position.

Table 70 Recommendation for setup of system parameter Event Preset Time 1)

ScaleLag	*DipLag*	Required *Event Preset Time* to avoid runtime execution error	Recommended *Event Preset Time* to obtain best accuracy
ScaleLag > *DipLag*	Always	*DipLag*, if *DipLag* > Servo Lag	*ScaleLag* in s plus 0.090 s
ScaleLag < *DipLag*	*DipLag* < Servo Lag	---------------- “ -------------	0.090 s
---------- “ ---------	*DipLag* > Servo Lag	--------------- “ --------------	*DipLag* in s plus 0.030 s

1) Typical Servo Lag is 0.056 seconds

Syntax

TriggSpeed
  [TriggData ’:=’] < variable (**VAR**) of *triggdata*> ‘,’
  [Distance ’:=’] < expression (**IN**) of *num*>
  [’\’ Start] ‘,’
  [ScaleLag ’:=’] < expression (**IN**) of *num*> ‘,’
  [AOp ’:=’] < variable (**VAR**) of *signalao*> ‘,’
  [ScaleValue ’:=’] < expression (**IN**) of *num*>
  [’\’ DipLag ’:=’ < expression (**IN**) of *num*>]
  [’\’ ErrDO ’:=’ < variable (**VAR**) of *signaldo*>]
  [’\’ Inhib ’:=’ < persistent (**PERS**) of *bool* >] ‘;’

Related information

Table 71

	Described in:
Use of triggers	Instructions - *TriggL*, *TriggC*, *TriggJ*
Definition of other triggs	Instruction - *TriggIO, TriggInt, TriggEquip*
More examples	Data Types - *triggdata*
Configuration of Event preset time	System Parameters - *Manipulator*

TriggStopProc - Generate restart data for trigg signals at stop

The instruction *TriggStopProc* creates an internal supervision process in the system for zero setting of specified process signals and the generation of restart data in a specified persistent variable at every program stop (STOP) or emergency stop (QSTOP) in the system.

TriggStopProc and the data type *restartdata* are intended to be used for restart after program stop (STOP) or emergency stop (QSTOP) of own process instructions defined in RAPID (NOSTEPIN routines).

It is possible in a user defined *RESTART* event routine, to analyse the current restart data, step backwards on the path with instruction *StepBwdPath* and activate suitable process signals before the movement restarts.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

Arguments

TriggStopProc RestartRef [\DO] [\GO1] [\GO2] [\GO3] [\GO4] ShadowDO

RestartRef *(Restart Reference)* **Data type:** *restartdata*

The persistent variable in which restart data will be available after every stop of program execution.

[\DO1] *(Digital Output 1)* **Data type:** *signaldo*

The signal variable for a digital process signal to be zero set and supervised in restart data when program execution is stopped.

[\GO1] *(Group Output 1)* **Data type:** *signalgo*

The signal variable for a digital group process signal to be zero set and supervised in restart data when program execution is stopped.

[\GO2] *(Group Output 2)* **Data type:** *signalgo*

The signal variable for a digital group process signal to be zero set and supervised in restart data when program execution is stopped.

[\GO3] *(Group Output 3)* **Data type:** *signalgo*

The signal variable for a digital group process signal to be zero set and supervised in restart data when program execution is stopped.

[\GO4] *(Group Output 4)* **Data type:** *signalgo*

The signal variable for a digital group process signal to be zero set and supervised in restart data when program execution is stopped.

At least one of the option parameters DO1, GO1 ... GO4 must be used.

ShadowDO *(Shadow Digital Output)* **Data type:** *signaldo*

The signal variable for the digital signal, which must mirror whether or not the process is active along the robot path.

This signal will not be zero set by the process *TriggStopProc* at STOP or QSTOP, but its values will be mirrored in *restartdata*.

Program execution

Setup and execution of TriggStopProc

TriggStopProc must be called from both:

- the START event routine or in the init part of the program
  (set PP to main kill the internal process for *TriggStopProc*)
- the POWERON event routine
  (power off kill the internal process for *TriggStopProc*)

The internal name of the process for *TriggStopProc* is the same as the signal name in the argument *ShadowDO*. If *TriggStopProc,* with same the signal name in argument *ShadowDO*, is executed twice, only the last executed *TriggStopProc* will be active.

Execution of *TriggStopProc* only starts the supervision of I/O signals at STOP and QSTOP.

Program stop STOP

The process *TriggStopProc* comprises the following steps:

- Wait until the robot stands still on the path.
- Store the current value (prevalue according to *restartdata*) of all used
  process signals.
  Zero set all used process signals except *ShadowDO.*
- Do the following during the next time slot, about 500 ms:
  - If some process signals change its value during this time:
    - Store its current value again (postvalue according to *restatdata*)
    - Zero set that signal, except *ShadowDO*
  - Count the number of value transitions (flanks) of the signal *ShadowDO*
- Update the specified persistent variable with restart data.

Emergency stop (QSTOP)

The process *TriggStopProc* comprises the following steps:

- Do the next step as soon as possible.
- Store the current value (prevalue according to *restartdata*) of all used process signals.
  Zero set all used process signals except *ShadowDO.*
- Do the following during the next time slot, about 500 ms:
  - If some process signal changes its value during this time:
    - Store its current value again (postvalue according to *restatdata*)
    - Zero set that signal, except *ShadowDO*
  - Count the number of value transitions (flanks) of the signal *ShadowDO*
- Update the specified persistent variable with restart data.

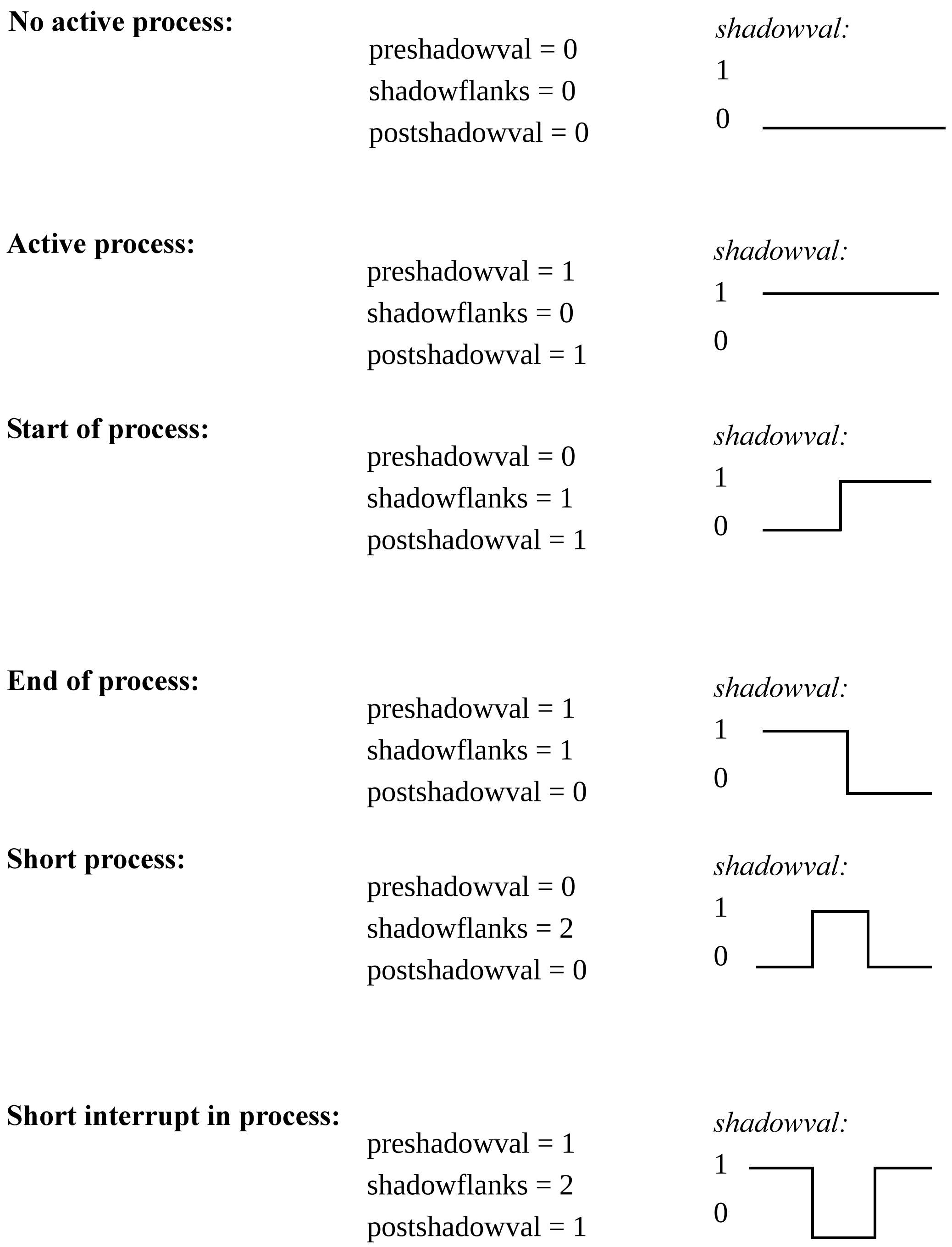
Critical area for process restart

Both the robot servo and the external equipment have some lags. All the instructions in the *Trigg* family are designed so that all signals will be set at suitable places on the robot path, independently of different lags in external equipment, to obtain process results that are as good as possible. Because of this, the settings of I/O signals can be delayed between 0 - 80 ms internally in the system, after the robot stands still at program stop (STOP) or after registration of an emergency stop (QSTOP). Because of this disadvantage for the restart functionality, both the prevalue and postvalue and also the shadow flanks are introduced in restart data.

If this critical timeslot of 0 - 80 ms coincides with following process cases, it is difficult to perform a good process restart:

- At the start of the process
- At the end of the process
- During a short process
- During a short interrupt in the process

Figure 28 Process phases at STOP or QSTOP within critical time slot 0-80 ms



Performing a restart

A restart of own process instructions (NOSTEPIN routines) along the robot path must be done in a RESTART event routine.

The RESTART event routine can consist of the following steps:

- After QSTOP, the regain to path is done at program start
- Analyse the restart data from the latest STOP or QSTOP
- Determine the strategy for process restart from the result of the analyse such as:
  - Process active, do process restart
  - Process inactive, do no process restart
  - Do suitable actions depending of which type of process application if:
    - Start of process
    - End of process
    - Short process
    - Short interrupt in process
-
  - Process start-up or process end-up take suitable actions depending on which type of process application
- Step backwards on the path
- Activate suitable process signals with values according to restart data
- Continue the program results in restart of the movement.

Limitation

No support for restart of own process instructions after a power failure.

Syntax

TriggStopProc
    [RestartRef ’:=’] < persistent (**PERS**) of *restartdata*>
    [’\’ DO1 ’:=’ < variable (**VAR**) of *signaldo*>]
    [’\’ GO1 ’:=’ < variable (**VAR**) of *signalgo*>]
    [’\’ GO2 ’:=’ < variable (**VAR**) of *signalgo*>]
    [’\’ GO3 ’:=’ < variable (**VAR**) of *signalgo*>]
    [’\’ GO4 ’:=’ < variable (**VAR**) of *signalgo*>] ’,’
    [ShadowDO ’:=’] < variable (**VAR**) of *signaldo*> ’;’

Related information

Table 72

	Described in:
Predefined process instructions	Instructions - *TriggL, TriggC*
Restart data	Data Types - *restartdata*
Step backward on path	Instructions - *StepBwdPath*

TRYNEXT - Jumps over an instruction which has caused an error

The TRYNEXT instruction is used to resume execution after an error, starting with the instruction following the instruction that caused the error.

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 assigned 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

Table 73

	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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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

Table 74

	Described in:
Tuning servos	Instructions - *TuneServo*

TuneServo - Tuning servos

TuneServo is used to tune the dynamic behavior of separate axes on the robot. It is not necessary to use *TuneServo* under normal circumstances, but sometimes tuning can be optimised depending on the robot configuration and the load characteristics. For external axes *TuneServo* can be used for load adaptation.

Incorrect use of the *TuneServo* can cause oscillating movements or torques that can damage the robot. You must bear this in mind and be careful when using the *TuneServo*.

Avoid doing TuneServo commands at the same time as the robot is moving. It can result in momentary high CPU loads causing error indication and stops.

Note. To obtain optimal tuning it is essential that the correct load data is used. Check on this before using *TuneServo*.

Generally, optimal tuning values often differ between different robots. Optimal tuning may also change with time.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

Improving path accuracy

For robots running at lower speeds, TuneServo can be used to improve the path accuracy by:

- Tuning tune_kv and tune_ti (see the tune types description below).
- Tuning friction compensation parameters (see below).

These two methods can be combined.

Other possibilities to improve the path accuracy:

- Decreasing path resolution can improve the path. Note: a value of path resolution which is too low will cause CPU load problems.
- The accuracy of straight lines can be improved by decreasing acceleration using AccSet. Example: AccSet 20, 10.

Description

Tune_df

Tune_df is used for reducing overshoots or oscillations along the path.

There is always an optimum tuning value that can vary depending on position and movement length. This optimum value can be found by changing the tuning in small steps (1 - 2%) on the axes that are involved in this unwanted behavior. Normally the optimal tuning will be found in the range 70% - 130%. Too low or too high tuning values have a negative effect and will impair movements considerably.

When the tuning value at the start point of a long movement differs considerably from the tuning value at the end point, it can be advantageous in some cases to use an intermediate point with a corner zone to define where the tuning value will change.

Some examples of the use of *TuneServo* to optimise tuning follow below:

IRB 6400, in a press service application (extended and flexible load), **axes 4 - 6**: Reduce the tuning value for the current wrist axis until the movement is acceptable. A change in the movement will not be noticeable until the optimum value is approached. A low value will impair the movement considerably. Typical tuning value 25%.

IRB 6400, upper parts of working area. Axis 1 can often be optimised with a tuning value of 85% - 95%.

IRB 6400, short movement (< 80 mm). Axis 1 can often be optimised with a tuning value of 94% - 98%.

IRB 2400, with track motion. In some cases axes 2 - 3 can be optimised with a tuning value of 110% - 130%. The movement along the track can require a different tuning value compared with movement at right angles to the track.

Overshoots and oscillations can be reduced by decreasing the acceleration or the acceleration ramp (*AccSet*), which will however increase the cycle time. This is an alternative method to the use of *TuneServo*.

Tune_dg

Tune_dg can reduce overshoots on rare occasions. Normally it should not be used.

Tune_df should always be tried first in cases of overshoots.

Tuning of tune_dg can be performed with large steps in tune value (e.g. 50%, 100%, 200%, 400%).

Never use tune_dg when the robot is moving.

Tune_dh

Tune_dh can be used for reducing vibrations and overshoots (e.g. large flexible load).

Tune value must always be lower than 100. Tune_dh increases path deviation and normally also increases cycle time.

Example:

IRB6400 with large flexible load which vibrates when the robot has stopped. Use tune_dh with tune value 15.

Tune_dh should only be executed for one axis. All axes in the same mechanical unit automatically get the same tune_value.

Never use tune_dh when the robot is moving.

Tune_di

Tune_di can be used for reducing path deviation at high speeds.

A tune value in the range 50 - 80 is recommended for reducing path deviation. Overshoots can increase (lower tune value means larger overshoot).

A higher tune value than 100 can reduce overshoot (but increases path deviation at high speed).

Tune_di should only be executed for one axis. All axes in the same mechanical unit automatically get the same tune_value.

Tune_dk, Tune_dl

Only for ABB internal use. Do not use these tune types. Incorrect use can cause oscillating movements or torques that can damage the robot.

Tune_kp, tune_kv, tune_ti external axes

These tune types affect position control gain (kp), speed control gain (kv) and speed control integration time (ti) for external axes. These are used for adapting external axes to different load inertias. Basic tuning of external axes can also be simplified by using these tune types.

Tune_kp, tune_kv, tune_ti robot axes

For robot axes, these tune types have another significance and can be used for reducing path errors at low speeds (< 500 mm/s).

Recommended values: tune_kv 100 - 180%, tune_ti 50 - 100%. Tune_kp should not be used for robot axes. Values of tune_kv/tune_ti which are too high or too low will cause vibrations or oscillations. Be careful if trying to exceed these recommended values. Make changes in small steps and avoid oscillating motors.

Always tune one axis at a time. Change the tuning values in small steps. Try to improve the path where this specific axis changes its direction of movement or where it accelerates or decelerates.

Never use these tune types at high speeds or when the required path accuracy is fulfilled.

Friction compensation: tune_fric_lev and tune_fric_ramp

These tune types can be used to reduce robot path errors caused by friction and backlash at low speeds (10 - 200 mm/s). These path errors appear when a robot axis changes direction of movement. Activate friction compensation for an axis by setting the system parameter *Friction ffw on* to TRUE (topic: Manipulator, type: Control parameters).

The friction model is a constant level with opposite sign of the axis speed direction. *Friction ffw level (Nm)* is the absolute friction level at (low) speeds and is greater than *Friction ffw ramp (rad/s) (see figure)*.

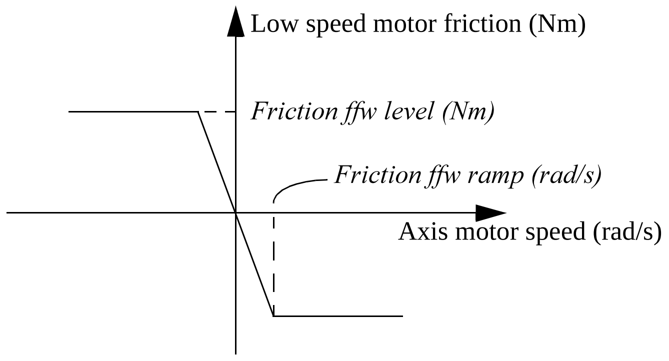


Figure 29 Friction model

Tune_fric_lev overrides the value of the system parameter *Friction ffw level*.

Tuning *Friction ffw level* (using tune_fric_lev) for each robot axis can improve the robots path accuracy considerably in the speed range 20 - 100 mm/s. For larger robots (especially the IRB6400 family) the effect will however be minimal as other sources of tracking errors dominate these robots.

Tune_fric_ramp overrides the value of the system parameter *Friction ffw ramp*. In most cases there is no need to tune the *Friction ffw ramp*. The default setting will be appropriate.

Tune one axis at a time. Change the tuning value in small steps and find the level that minimises the robot path error at positions on the path where this specific axis changes direction of movement. Repeat the same procedure for the next axis etc.

The final tuning values can be transferred to the system parameters. Example:

Friction ffw level = 1. Final tune value (tune_fric_lev) = 150%.

Set Friction ffw level = 1.5 and tune value = 100% (default value) which is equivalent.

Arguments

TuneServo MecUnit Axis TuneValue [\Type]

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

The name of the mechanical unit.

Axis **Data type:** *num*

The number of the current axis for the mechanical unit (1 - 6).

TuneValue **Data type:** *num*

Tuning value in percent (1 - 500). 100% is the normal value.

[\Type] **Data type:** *tunetype*

Type of servo tuning. Available types are *TUNE_DF*, *TUNE_KP*, *TUNE_KV*, *TUNE_TI*, *TUNE_FRIC_LEV*, *TUNE_FRIC_RAMP*, *TUNE_DG*, *TUNE_DH*, *TUNE_DI*. Type *TUNE_DK* and *TUNE_DL* only for ABB internal use. These types are predefined in the system with constants.

This argument can be omitted when using tuning type *TUNE_DF*.

Example

```
TuneServo MHA160R1, 1, 110 \Type:= TUNE_KP;
```

Activating of tuning type *TUNE_KP* with the tuning value *110*% on axis *1* in the mechanical unit *MHA160R1*.

Program execution

The specified tuning type and tuning value are activated for the specified axis. This value is applicable for all movements until a new value is programmed for the current axis, or until the tuning types and values for all axes are reset using the instruction *TuneReset*.

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.

Limitations

Any active servo tuning are always set to default values at power fail.
This limitation can be handled in the user program at restart after power failure.

Syntax

TuneServo
    [MecUnit ’:=’] < variable (VAR) of *mecunit*> ’,’
    [Axis ’:=’] < expression (IN) of num> ’,’
    [TuneValue ’:=’] < expression (IN) of num>
    [’\’ Type ’:=’ <expression (IN) of *tunetype*>]’;’

Related information

Table 75

	Described in:
Other motion settings	Summary Rapid - *Motion Settings*
Types of servo tuning	Data Types - *tunetype*
Reset of all servo tunings	Instructions - *TuneReset*
Tuning of external axes	System parameters - *Manipulator*
Friction compensation	System parameters - *Manipulator*

UIMsgBox - User Message Dialog Box type basic

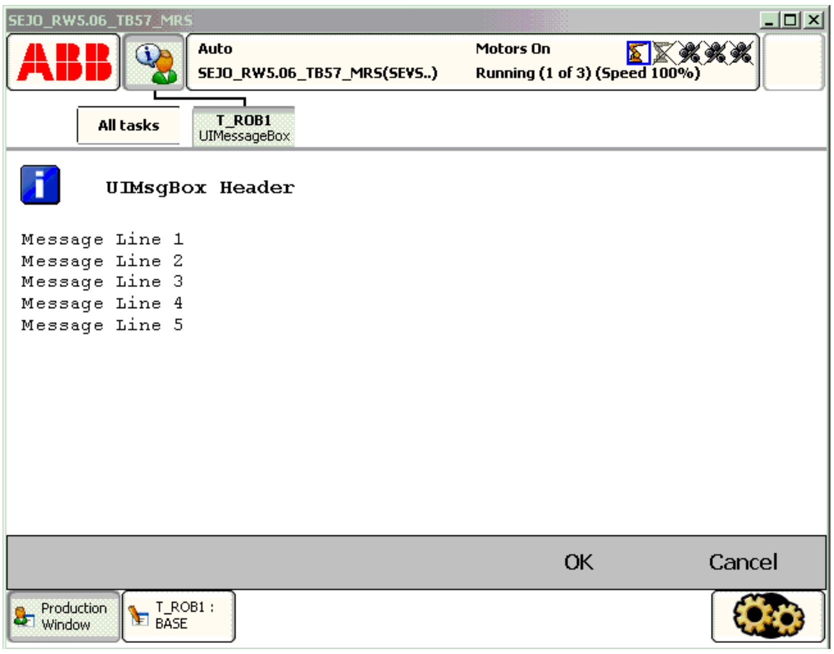
UIMsgBox (User Interaction Message Box) is used to communicate with the user of the robot system on available User Device such as the FlexPendant.
After that the output information has been written, the user selection of the displayed push buttons is transferred back to the program.

Example

```
UIMsgBox ”Continue the program ?”;
```

The message “Continue the program ?“ is displayed. The program proceeds, when the user press the default button OK.

```
VAR btnres answer;
...
UIMsgBox
    \Header:=”UIMsgBox Header”,
    ”Message Line 1”
    \MsgLine2:=”Message Line 2”
    \MsgLine3:=”Message Line 3”
    \MsgLine4:=”Message Line 4”
    \MsgLine5:=”Message Line 5”
    \Buttons:=btnOKCancel
    \Icon:=iconInfo
    \Result:=answer;
IF answer = resOK my_proc;
```



```
UIMsgBox Header

Message Line 1
Message Line 2
Message Line 3
Message Line 4
Message Line 5
```

Above message box is with icon, header, message line 1 to 5 and push buttons are written on the FlexPendant display. Program execution waits until OK or Cancel is pressed. In other words, *answer* will be assigned 1 (OK) or 5 (Cancel) depending on which of the buttons is depressed. If answer is OK, *my_proc* will be called.

Note that Message Line 1 ... Message Line 5 are displayed on separate lines 1 to 5 (the switch *\Wrap* is not used).

Arguments

UIMsgBox [\Header] MsgLine1 [\MsgLine2] [\MsgLine3] [\MsgLine4] [\MsgLine5] [\Wrap] [\Buttons] [\Icon] [\Result] [\MaxTime] [\DIBreak] [\DOBreak] [\BreakFlag]

[\Header] **Data type:** *string*

Header text to be written at the top of the message box.
Max. 32 characters.

MsgLine1 *(Message Line 1)* Data type: *string*

Text line 1 to be written on the display.

[\MsgLine2]	*(Message Line 2)*	**Data type**: *string*
[\MsgLine3]	*(Message Line 3)*	**Data type:** *string*
[\MsgLine4]	*(Message Line 4)*	**Data type:** *string*
[\MsgLine5]	*(Message Line 5)*	**Data type:** *string*

Additional text lines 2 ... 5 to be written on the display.

Max. layout space 9 lines with 40 characters.

[\Wrap] **Data type:** *switch*

If selected, all the strings *MsgLine1* ... *MsgLine5* will be concatenated to one string with single space between each individual strings and spread out on as few lines as possible.

Default, each message string *MsgLine1 ... MsgLine5* will be on separate line on the display.

[\Buttons] **Data type:** *buttondata*

Defines the push buttons to be displayed. Only one of the predefined buttons combination of type *buttondata* can be used.

Default, the system display the OK button.

[\Icon] **Data type:** *icondata*

Defines the icon to be displayed. Only one of the predefined icons of type *icondata* can be used.

Default no icon.

[\Result] **Data type:** *btnres*

The variable for which, depending on which button is pressed, the numeric value 0..7 is returned. Only one of the predefined constants of type *btnres* can be used to test the user selection.

If any type of system break such as *\MaxTime, \DIBreak or \DOBreak* or if *\Buttons:=btnNone*, *resUnkwn* equal to 0 is returned.

[\MaxTime] **Data type:** *num*

The maximum amount of time [s] that program execution waits. If no button is selected 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 input signal that may interrupt the operator dialog. If no button is selected 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.

[\DOBreak] *(Digital Output Break)* **Data type:** *signaldo*

The digital output signal that may interrupt the operator dialog. If no button is selected 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_DOBREAK can be used to test whether or not this has occurred.

[\BreakFlag] **Data type:** *errnum*

A variable (before used set to 0 by the system) that will hold the error code if *\MaxTime, \DIBreak or \DOBreak* is used. The constants ERR_TP_MAXTIME, ERR_TP_ DIBREAK and ERR_TP_ DOBREAK can be used to select the reason. If this optional variable is omitted, the error handler will be executed.

Program execution

The message box with icon, header, message lines and buttons are displayed according to the programmed arguments. Program execution waits until the user select one button or the message box is interrupted by time-out or signal action.
The user selection and interrupt reason are transfer back to the program.

New message box on TRAP level take focus from message box on basic level.

Predefined data

```
Icons:
CONST icondata iconNone := 0;
CONST icondata iconInfo := 1;
CONST icondata iconWarning := 2;
CONST icondata iconError := 3;

Buttons:
CONST buttondata btnNone := -1;
CONST buttondata btnOK := 0;
CONST buttondata btnAbrtRtryIgn := 1;
CONST buttondata btnOKCancel := 2;
CONST buttondata btnRetryCancel := 3;
CONST buttondata btnYesNo := 4;
CONST buttondata btnYesNoCancel := 5;

Results:
CONST btnres resUnkwn := 0;
CONST btnres resOK := 1;
CONST btnres resAbort := 2;
CONST btnres resRetry := 3;
CONST btnres resIgnore := 4;
CONST btnres resCancel := 5;
CONST btnres resYes := 6;
CONST btnres resNo := 7;
```

Example

```
VAR errnum err_var;
...
UIMsgBox \Header:= “Cycle step 4”, “Robot moving to load position”
    \Buttons:=btnNone \Icon:=iconInfo \MaxTime:=60 \DIBreak:=di5
    \BreakFlag:=err_var;

TEST err_var
CASE ERR_TP_MAXTIME:
    ! Time out error
CASE ERR_TP_DIBREAK:
    ! Robot in load position
DEFAULT:
    ! Not such case defined
ENDTEST
```

The message box is displayed while the robot is moving to the it’s load position. The operator can not answer or remove the message box.
The message box is removed when the robot is in position (*di1* set to 1) or at time-out (after *60* seconds).

Error handling

If parameter *BreakFlag* is not used, these situations can then be dealt with by the error handler:

If there is a time-out (parameter *MaxTime*) before an input from the operator, the system variable ERRNO is set to ERR_TP_MAXTIME and the execution continues in the error handler.

If digital input is set (parameter *DIBreak*) before an input from the operator, the system variable ERRNO is set to ERR_TP_DIBREAK and the execution continues in the error handler.

If a digital output is set (parameter *DOBreak*) before an input from the operator, the system variable ERRNO is set to ERR_TP_DOBREAK and the execution continues in the error handler.

This situation can only be dealt with by the error handler:

If there is no client, e.g. a Flex Pendant, to take care of the instruction, the system variable ERRNO is set to ERR_TP_NO_CLIENT and the execution continues in the error handler.

Limitations

Avoid using a too small value for the time-out parameter *MaxTime* when *UIMsgBox* is frequently executed, for example in a loop. It can result in an unpredictable behavior of the system performance, like slow TPU response.

Syntax

UIMsgBox
    [‘\’Header’:=’ <expression (**IN**) of *string*> ‘,’]
    [MsgLine1’:=’] <expression (**IN**) of *string*>
    [‘\’MsgLine2’:=’<expression (**IN**) of *string*>]
    [‘\’MsgLine3’:=’<expression (**IN**) of *string*>]
    [‘\’MsgLine4’:=’<expression (**IN**) of *string*>]
    [‘\’MsgLine5’:=’<expression (**IN**) of *string*>]
    [‘\’Wrap]
    [‘\’Buttons’:=’ <expression (**IN**) of *buttondata*>]
    [‘\’Icon’:=’ <expression (**IN**) of *icondata*>]
    [‘\’Result’:=’< var or pers (**INOUT**) of *btnres*>]
    [’\’MaxTime ’:=’ <expression (**IN**) of *num*>]
    [’\’DIBreak ’:=’ <variable (**VAR**) of *signaldi*>]
    [’\’DOBreak ’:=’ <variable (**VAR**) of *signaldo*>]
    [’\’BreakFlag ’:=’ <var or pers (**INOUT**) of *errnum*>]’;’

Related information

Table 76

	Described in:
Icon display data	Data Types - *icondata*
Push button data	Data Types - *buttondata*
Push button result data	Data Types - *btnres*
User Interaction Message Box type advanced	Functions - *UIMessageBox*
User Interaction Number Entry	Functions - *UINumEntry*
User Interaction Number Tune	Functions - *UINumTune*
User Interaction Alpha Entry	Functions - *UIAlphaEntry*
User Interaction List View	Functions - *UIListView*
System connected to FlexPendant etc.	Functions - *UIClientExist*

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 diskhome \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*). *diskhome* is a predefined string constant "HOME:".

Arguments

UnLoad [\ErrIfChanged] | [\Save] FilePath [\File]

[\ErrIfChanged] **Data type:** *switch*

If this argument is used, and the module has been changed since it was loaded into the system, the instruction will throw the error code ERR_NOTSAVED to the error handler if any.

[\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 the original place specified in the *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 an *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 finish unloading before the execution proceeds with the next instruction.

After that the program module is unloaded and the rest of the program modules will be linked.

For more information see the instructions *Load* or *StartLoad-Waitload*.

Examples

```
UnLoad "HOME:/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 "HOME:" \File:="DOORDIR/DOOR1.MOD";
```

Same as above but another syntax.

```
Unload \Save, "HOME:" \File:="DOORDIR/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 FlexPendant.

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*), the system variable ERRNO is set to ERR_UNLOAD.

If the argument ErrIfChanged is used and the module has been changed, the execution of this routine will set the system variable ERRNO to ERR_NOTSAVED.

Those errors can then be handled in the error handler.

Syntax

UnLoad
    ['\'ErrIfChanged ','] | ['\'Save ',']
    [FilePath':=']<expression (**IN**) of *string*>
    ['\'File':=' <expression (**IN**) of *string*>]';'

Related information

Table 77

	Described in:
Load a program module	Instructions - *Load* Instructions - *StartLoad-WaitLoad*
Accept unresolved references	System Parameters - *Controller* System Parameters - *Tasks* System Parameters - *BindRef*

UnpackRawBytes - Unpack data from rawbytes data

UnpackRawBytes is used to unpack the contents of a ‘container’ of type rawbytes to variables of type byte, num or string.

Example

```
VAR iodev io_device;
VAR rawbytes raw_data_out;
VAR rawbytes raw_data_in;
VAR num integer;
VAR num float;
VAR string string1;
VAR byte byte1;
VAR byte data1;

! Data packed in raw_data_out according to the protocol
...

Open “chan1:”, io_device\Bin;
WriteRawBytes io_device, raw_data_out;
ReadRawBytes io_device, raw_data_in, 27 \Time := 1;
Close io_device;
```

According to the protocol, that is known to the programmer, the message is sent to device *‘chan1:’*. Then the answer is read from the device.

The answer contains as an example the following:

Table 78

byte number:	contents:
1-4	integer ‘5’
5-8	float ‘234.6’
9-25	string “This is real fun!”
26	hex value ‘4D’
27	ASCII code 122, i.e. ‘z’

```
UnpackRawBytes raw_data_in, 1, integer \IntX := DINT;
```

The contents of *integer* will be 5 integer.

```
UnpackRawBytes raw_data_in, 5, float \Float4;
```

The contents of *float* will be 234.6 decimal.

UnpackRawBytes raw_data_in, 9, string1 \ASCII:=17;

The contents of *string1* will be "This is real fun!".

UnpackRawBytes raw_data_in, 26, byte1 \Hex1;

The contents of *byte1* will be '4D' hexadecimal.

UnpackRawBytes raw_data_in, 27, data1 \ASCII:=1;

The contents of *data1* will be 122, the ASCII code for "z".

Arguments

UnpackRawBytes RawData [\Network] StartIndex Value [\Hex1] | [\IntX] | [\Float4] | [\ASCII]

RawData **Data type:** *rawbytes*

Variable container to unpack data from.

[\Network] **Data type:** *switch*

Indicates that integer and float shall be unpacked from big-endian (network order) representation in *RawData*. ProfiBus and InterBus use big-endian.

Without this switch, integer and float will be unpacked in little-endian (not network order) representation from *RawData*. DeviceNet use little-endian.

Only relevant together with option parameter *\IntX - UINT, UDINT, INT, DINT* and *\Float4.*

StartIndex **Data type:** *num*

StartIndex, between 1 and 1024, indicates where to start unpacking data from *RawData*.

Value **Data type:** *anytype*

Variable containing the data that were unpacked from *RawData*.

Allowed data types are: *byte, num* or *string*.

[\Hex1] **Data type:** *switch*

The data to be unpacked and placed in *Value* has hexadecimal format in 1 *byte* and will be converted to decimal format in a *byte* variable.

[\IntX] **Data type:** *inttypes*

The data to be unpacked has the format according to the specified constant of data type *inttypes*. The data will be converted to a *num* variable containing an integer and stored in *Value*.

See predefined data below.

[\Float4] **Data type:** *switch*

The data to be unpacked and placed in *Value* has float, 4 bytes, format and will be converted to a *num* variable containing a float.

[\ASCII] **Data type:** *num*

The data to be unpacked and placed in *Value* has *byte* or *string* format.

If *Value* is of type *byte*, the data will be interpreted as ASCII code and converted to *byte* format (1 character).

If *Value* is of type *string*, the data will be stored as *string* (1...80 characters). String data is not NULL terminated in data of type *rawbytes*.

One of argument *\Hex1*, *\IntX*, *\Float4* or *\ASCII* must be programmed.

The following combinations are allowed:

Table 79

Data type of *Value*:	Allowed option parameters:
num	\IntX
num	\Float4
string	\ASCII:=n with n between 1 and 80
byte	\Hex1 \ASCII:=1

Program execution

During program execution data is unpacked from the ‘container’ of type *rawbytes* into a variable of type *anytype*.

Predefined data

The following symbolic constants of the data type *inttypes* are predefined and can be used to specify the integer type stored in *RawData* with parameter *\IntX.*

Table 80

Symbolic constant	Constant value	Integer format	Integer value range
USINT	1	Unsigned 1 byte integer	0 ... 255
UINT	2	Unsigned 2 byte integer	0 ... 65 535
UDINT	4	Unsigned 4 byte integer	0 - 8 388 608 *)
SINT	- 1	Signed 1 byte integer	- 128 ... 127
INT	- 2	Signed 2 byte integer	- 32 768 ... 32 767
DINT	- 4	Signed 4 byte integer	- 8 388 607 ... 8 388 608 *)

*) RAPID limitation for storage of integer in data type *num*.

Syntax

UnpackRawBytes
  [RawData ’:=’] < variable (**VAR**) of *rawbytes*>
  [’\’ Network] ’,’
  [StartIndex ’:=’] < expression (**IN**) of *num*> ’,’
  [Value ’:=’] < variable (**VAR**) of *anytype*>
  [’\’ Hex1] | [’\’ IntX ’:=’ < expression (**IN**) of *inttypes*>] | [’\’ Float4]
  | [’\’ ASCII ‘:=’ < expression (**IN**) of *num*>] ’;’

Related information

Table 81

	Described in:
rawbytes data	Data Types - *rawbytes*
Get the length of *rawbytes* data	Functions - *RawBytesLen*
Clear the contents of *rawbytes* data	Instructions - *ClearRawBytes*
Copy the contents of *rawbytes* data	Instructions - *CopyRawBytes*
Pack DeviceNet header into *rawbytes* data	Instructions - *PackDNHeader*
Pack data into *rawbytes* data	Instructions - *PackRawBytes*
Write *rawbytes* data	Instructions - *WriteRawBytes*
Read *rawbytes* data	Instructions - *ReadRawBytes*
Unpack data from *rawbytes* data	Instructions - *UnpackRawBytes*
Bit/Byte Functions	RAPID Summary - Bit Functions
String functions	RAPID Summary - String Functions

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.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

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 *vmax.v_tcp* 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.

*) Max. TCP speed for the used robot type and normal pratical TCP values.
The RAPID function *MaxRobSpeed* returns the same value.

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

Table 82

	Described in:
Definition of velocity	Data Types - *speeddata*
Max. TCP speed for this robot	Function - *MaxRobSpeed*
Positioning instructions	RAPID Summary - *Motion*

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 execution

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.

In manual mode, if the argument *\Inpos* is used and *Time* is greater than 3 s, an alert box will pop up asking if you want to simulate the instruction. If you don´t want the alert box to appear you can set system parameter SimMenu to NO (System Parameters, Topics:Communication, Types:System misc).

Syntax

WaitDI
  [Signal ':='] < variable (**VAR**) of *signaldi* > ','
  [Value ':='] < expression (**IN**) of *dionum* >
  ['\'MaxTime ':='<expression (**IN**) of *num*>]
  ['\'TimeFlag':='<variable (**VAR**) of *bool*>] ';'

Related information

Table 83

	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.

In manual mode, if the argument *\Inpos* is used and *Time* is greater than 3 s, an alert box will pop up asking if you want to simulate the instruction. If you don´t want the alert box to appear you can set system parameter SimMenu to NO (System Parameters, Topics:Communication, Types:System misc).

Syntax

WaitDO
    [Signal ':='] < variable (**VAR**) of *signald*o > ','
    [Value ':='] < expression (**IN**) of *dionum* >
    ['\'MaxTime ':='<expression (**IN**) of *num*>]
    ['\'TimeFlag':='<variable (**VAR**) of *bool*>] ';'

Related information

Table 84

	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 symbols/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, in order to minimise the number of links (1 instead of 2).

Example

```
VAR loadsession load1;
...
StartLoad "HOME:/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 "HOME:/PART_A.MOD";
```

Load the program module *PART_A.MOD* from *HOME:* 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 at the linking time, if the system parameter for *Controller/Task/Check unsolved references* is set to 0.

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 parameter *Check unsolved references* could be set to 1 (default behaviour). The *Late Binding* way is preferable.

There will always be a run time error if trying to execute an unsolved reference.

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.

For loading of program that contains a *main* procedure to a main program (with another *main* procedure), see instruction *Load.*

Examples

```
StartLoad  "HOME:/DOORDIR/DOOR2.MOD",  load1;
...
WaitLoad  \UnloadPath:="HOME:/DOORDIR/DOOR1.MOD", load1;
```

Load the program module *DOOR2.MOD* from *HOME:* in 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  "HOME:" \File:="DOORDIR/DOOR2.MOD",  load1;
! The robot can do some other work
WaitLoad  \UnloadPath:="HOME:" \File:= "DOORDIR/DOOR1.MOD",  load1;
```

Is the same as the instructions below but the robot can do some other work during the loading time and also do it faster (only one link).

```
Load  "HOME:"  \File:="DOORDIR/DOOR2.MOD";
UnLoad "HOME:" \File:="DOORDIR/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.

If the module cannot be loaded because the program memory is full, the system variable ERRNO is set to ERR_PRGMEMFULL.

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.

Note that RETRY cannot be used for error recovery for any errors from *WaitLoad*.

Syntax

WaitLoad
    [['\' UnloadPath ':=' <expression (**IN**) of *string*>]
      ['\' UnloadFile ':=' <expression (**IN**) of *string*>] ',']
    [LoadNo ':='] <variable (**VAR**) of *loadsession*> ';'

Related information

Table 85

	Described in:
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*
Cancel loading of a program module	Instructions - *CancelLoad*
Accept unsolved references	System Parameters - *Controller/Task/Check unsolved references*

WaitSensor - Wait for connection on sensor

WaitSensor (Wait Sensor) connects to an object in the start window on the sensor mechanical unit.

Example

```
WaitSensor Ssync1;
```

The program connects to the first object in the object queue that is within the start window on the sensor. If there is no object in the start window then execution stops and waits for an object.

Arguments

WaitSensor Mecunt[\RelDist][\PredTime][\MaxTime][\TimeFlag]

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

The moving mechanical unit to which the robot position in the instruction is related.

[\RelDist] (Relative Distance) **Data type:** num

Waits for an object to enter the start window and go beyond the distance specified by the argument. If the work object is already connected, then execution stops until the object passes the given distance. If the object has already gone past the *Relative Distance* then execution continues.

[\PredTime] (Prediction Time) **Data type:** num

Waits for an object to enter the start window and go beyond the distance specified by the argument. If the work object is already connected, then execution stops until the object passes the given distance. If the object has already gone past the *Prediction Time* then execution continues.

[\MaxTime] *(Maximum Time)* **Data type:** *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the sensor connection or reldist reached, 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 sensor connection or reldist reached. 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 there is no object in the start window then program execution stops. If an object is present, then the object is connected to the sensor and execution continues.

If a second *WaitSensor* instruction is issued while connected then an error is returned unless the *\RelDist* optional argument is used.

Examples

```
WaitSensor SSYNC1\RelDist:=500.0;
```

If not connected, then wait for the object to enter the start window and then wait for the object to pass the 500 mm point on the sensor.

If already connected to the object, then wait for the object to pass 500 mm.

```
WaitSensor SSYNC1\RelDist:=0.0;
```

If not connected, then wait for an object in the start window.

If already connected, then continue execution as the object has already gone past 0.0 mm.

```
WaitSensor Ssync1;
WaitSensor Ssync1\RelDist:=0.0;
```

The first *WaitSensor* connects to the object in the start window. The second *WaitSensor* will return immediately if the object is still connected, but will wait for the next object if the previous object had moved past the Maximum Distance or was dropped.

```
WaitSensor Ssync1\RelDist:=0.5\PredTime:=0.1;
```

The *WaitSensor* will return immediately if the object thas passed 0.5 meter but otherwise will wait for an object will reach =Reldist - C1speed * Predtime . The goal here is to anticipate delays before starting a new move instruction .

```
WaitSensor Ssync1\RelDist:=0.5\MaxTime:=0.1\Timeflag:=flag1;
```

The *WaitSensor* will return immediately if the object thas passed 0.5 meter but otherwise will wait 0.1 sec for an object .If no object passes 0.5 meter during this 0.1 sec the instruction will return with flag1 =TRUE.

Limitations

It requires 50 ms to connect to the first object in the start window. Once connected, a second *WaitSensor* with *\RelDist* optional argument will take only normal RAPID instruction execution time.

Error handling

If following errors occurs during execution of the *WaitSensor* instruction, the system variable ERRNO will be set. These errors can then be handled in the error handler.

ERR_CNV_NOT_ACT	The sensor is not activated.
ERR_CNV_CONNECT	The *WaitSensor* instruction is already connected.
ERR_CNV_DROPPED	The object that the instruction *WaitSensor* was waiting for has been dropped by another task. (DSQC 354Revision 2: an object had passed the start window)

ERR_WAIT_MAXTIME The object did not come in time and there is no Timeflag

Syntax

WaitSensor
    [Mecunt':=']< persistent (PERS) of *mecunit*> ';'
    ['\' RelDist ':=' < expression (IN) of *num* >]
    ['\' PredTime':=' < expression (IN) of *num* >]
    ['\'MaxTime ':='<expression (IN) of *num*>]
    ['\'TimeFlag':='<variable (VAR) of *bool*>] ';'

Related information

Table 86

	Described in:
Drop object on sensor	Instructions - *DropSensor*
Sync to sensor	Instructions - *SyncToSensor*

WaitSyncTask - Wait for synchronization point with other program tasks

WaitSyncTask is used to synchronize several program tasks at a special point in each programs. Each program task waits until all program tasks have reach the named synchronization point.

Note that WaitSyncTask only synchronize the program execution. To reach synchronization of both the program exection and the robot movements, the move instruction before the WaitSyncTask must be a stop-point in all involved program tasks.

To reach safe synchronization functionality, the meeting point (parameter SyncID) must have an unique name in each program task. The name must also be the same for the program tasks that should meet in the meeting point.

Example

Program example in task T_ROB1

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;

...
WaitSyncTask sync1, task_list;
...
```

Program example in task T_ROB2

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;

...
WaitSyncTask sync1, task_list;
...
```

The program task, that first reach *WaitSyncTask* with identity *sync1*, waits until the other program task reach it’s *WaitSyncTask* with the same identity *sync1*. Then both program task T_ROB1 and T_ROB2 continue it’s execution.

Arguments

WaitSyncTask SyncID TaskList [\TimeOut]

SyncID **Data type:** *syncident*

Variable that specify the name of the synchronization (meeting) point.
Data type *syncident* is a non-value type, only used as an identifier for naming the

synchronization point.

The variable must be defined and have equal name in all cooperated program tasks. It's recommended to always define the variable global in each program task (VAR *syncident* ...).

TaskList **Data type:** *tasks*

Persistent variable, that in a task list (array) specifies the name (*string*) of the program tasks, that should meet in the synchronization point with name according argument *SyncID*.

The persistent variable must be defined and have equal name and equal contents in all cooperated program tasks. It's recommended to always define the variable global in the system (PERS *tasks* ...).

[\TimeOut] **Data type:** *num*

The max. time for waiting for the other program tasks to reach the synchronization point. Time-out in seconds (resolution 0,001s). If this argument is not specified, the program task will wait for ever.

If this time runs out before all program tasks has reach the synchronization point, the error handler will be called, if there is one, with the error code ERR_WAITSYNCTASK. If there is no error handler, the execution will be stopped.

Program execution

The actual program task will wait at *WaitSyncTask,* until the other program tasks in the *TaskList* has reached the same *SyncID* point*.* At that time, respective program task will continue to execute its next instruction.

WaitSyncTask can be programmed between move instructions with corner zone in between. Depending on the timing balance between the program tasks at execution time, the system can:

- At best timing, keep all corner zones
- at worst timing, only keep the corner zone for the program task that reach the *WaitSyncTask* last. For the other program tasks it will results in stop points.

It is possible to exclude program task for testing purpose from FlexPendant - Task Selection Panel. The instruction *WaitSyncTask* will still works with the reduced number of program tasks, even for only one program task.

Example

Program example in task T_ROB1

```
PERS tasks task_list{2} := [ [“T_ROB1”], [“T_ROB2”] ];
VAR syncident sync1;

...
WaitSyncTask sync1, task_list \TimeOut := 60;
...
ERROR
    IF ERRNO = ERR_WAITSYNCTASK THEN
        RETRY;
    ENDIF
```

The program task T_ROB1 waits in instruction *WaitSyncTask* for the program task T_ROB2 to reach the same synchronization point. After waiting in *60* s, the error handler is called with ERRNO equal to ERR_WAITSYNCTASK. Then the instruction *WaitSyncTask* is called again for additional wait in 60 s.

Error handling

If time-out because *WaitSyncTask* not ready in time, the system variable ERRNO is set to ERR_WAITSYNCTASK.

This error can be handled in the ERROR handler.

Syntax

WaitSyncTask
    [SyncID ’:=’] < variable (**VAR**) of *syncident*> ’,’
    [TaskList ’:=’] < persistent array {*} (**PERS**) of *tasks*> ’,’
    [’\’ TimeOut ’:=’ < expression (**IN**) of *num* >] ’;’

Related information

Table 87

	Described in:
Specify cooperated program tasks	Data Types- *tasks*
Identity for synchronization point	Data Types- *syncident*

WaitTestAndSet - Wait until variable unset - then set

WaitTestAndSet instruction waits for a specified *boolean* persistent variable value to become false. When the variable value becomes false, the instruction will set value to true and continue the execution. The persistent variable can be used as a binary semaphore for synchronization and mutual exclusion.

This instruction have the same underlying functionality as the *TestAndSet* function, but the *WaitTestAndSet* is waiting as long as the boolean is false while the *TestAndSet* instruction terminates immedeately.

It is not recommended to use *WaitTestAndSet* instruction in a TRAP-routines, UNDO-handler or event routines.

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 FlexPendant - Operator Output & Input

Examples

MAIN program task:

```
PERS bool tproutine_inuse := FALSE;
....
WaitTestAndSet(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;
....
WaitTestAndSet(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 *WaitTestAndSet* function guarantees that all three lines from each task are not separated.

If program task MAIN takes the semaphore *WaitTestAndSet(tproutine_inuse)* first, then program task BACK1 must wait until the program task MAIN has left the semaphore.

Arguments

WaitTestAndSet Object

Object **Data type:** *bool*

User defined data object to be used as semaphore. The data object must be a PERS object. If *WaitTestAndSet* are used between different program tasks, the object must be a global PERS.

Program execution

This instruction will in one indivisible step check the user defined persistent variable:

- if it has the value false, set it to true

- if it has the value true, wait until it become false and then set it to true

```
IF Object = FALSE THEN
    Object := TRUE;
ELSE
    ! Wait until it become FALSE
    Object := TRUE;
ENDIF
```

After that the instruction is ready.

Examples

```
PERS bool semPers:= FALSE;
...
PROC doit(...)
    WaitTestAndSet semPers;
    ....
    semPers := FALSE;
ENDPROC
```

Note in this case: If program execution is stopped in the routine *doit* and the program pointer is moved to *main*, the variable *semPers* will not be reset. To avoid this, reset the variable *semPers* to FALSE in the START event routine.

Syntax

WaitTestAndSet [Object ':='] < persistent (**PERS**) of *bool*> ';'

Related information

Table 88

	Described in:
Test variable and set if unset (type polled with WaitTime)	Functions - *TestAndSet*

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. This argument can only be used if the task controls mechanical units.

Time **Data type:** *num*

The time, expressed in seconds, that program execution is to wait.
Min. value 0 s. Max. value no limit. Resolution 0.001 s.

Program execution

Program execution temporarily stops for the given amount of time. Interrupt handling and other similar functions, nevertheless, are still active.

In manual mode, if the argument *\Inpos* is used and *Time* is greater than 3 s, an alert box will pop up asking if you want to simulate the instruction. If you don´t want the alert box to appear you can set system parameter SimMenu to NO (System Parameters, Topics:Communication, Types:System misc).

Example

```
WaitTime \InPos,0;
```

Program execution waits until the robot and the external axes have come to a standstill.

Limitations

Argument *Inpos* cannot be used together with SoftServo.

Syntax

```
WaitTime
    ['\'InPos',']
    [Time ':='] <expression (IN) of num>';'
```

Related information

Table 89

	Described in:
Waiting until a condition is met	Instructions - *WaitUntil*
Waiting until an I/O is set/reset	Instructions - *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. This argument can only be used if the task controls mechanical units.

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.

In manual mode, if the argument *\Inpos* is used and *Time* is greater than 3 s, an alert box will pop up asking if you want to simulate the instruction. If you don´t want the alert box to appear you can set system parameter SimMenu to NO (System Parameters, Topics:Communication, Types:System misc).

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 FlexPendant.

```
WaitUntil \Inpos, di4 = 1;
```

Program execution waits until the robot has come to a standstill and the *di4* input has been set.

Limitation

Argument *\Inpos* can't be used together with SoftServo.

Syntax

WaitUntil
  ['\'InPos',']
  [Cond ':='] <expression (**IN**) of *bool*>
  ['\'MaxTime ':='<expression (**IN**) of *num*>]
  ['\'TimeFlag':='<variable (**VAR**) of *bool*>] ';'

Related information

Table 90

	Described in:
Waiting until an input is set/reset	Instructions - *WaitDI*
Waiting a given amount of time	Instructions - *WaitTime*
Expressions	Basic Characteristics - *Expressions*

WaitWObj - Wait for work object on conveyor

WaitWObj (Wait Work Object) connects to a work object in the start window on the conveyor mechanical unit.

Example

```
WaitWObj wobj_on_cnv1;
```

The program connects to the first object in the object queue that is within the start window on the conveyor. If there is no object in the start window then execution stops and waits for an object.

Arguments

WaitWObj WObj [\RelDist][\MaxTime][\TimeFlag]

WObj	*(Work Object)*	**Data type:** *wobjdata*

The moving work object (coordinate system) to which the robot position in the instruction is related. The mechanical unit conveyor is to be specified by the *ufmec* in the work object.

[\RelDist]	(Relative Distance)	**Data type:** num

Waits for an object to enter the start window and go beyond the distance specified by the argument. If the work object is already connected, then execution stops until the object passes the given distance. If the object has already gone past the *Relative Distance* then execution continues.

[\MaxTime]	*(Maximum Time)*	**Data type:** *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the sensor connection or reldist reached, 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 sensor connection or reldist reached. 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 there is no object in the start window then program execution stops. If an object is present, then the work object is connected to the conveyor and execution continues.

If a second **WaitWObj** instruction is issued while connected then an error is returned unless the **\RelDist** optional argument is used.

Examples

WaitWObj wobj_on_cnv1\RelDist:=500.0;

If not connected, then wait for the object to enter the start window and then wait for the object to pass the 500 mm point on the conveyor.

If already connected to the object, then wait for the object to pass 500 mm.

WaitWObj wobj_on_cnv1\RelDist:=0.0;

If not connected, then wait for an object in the start window.

If already connected, then continue execution as the object has already gone past 0.0 mm.

WaitWObj wobj_on_cnv1;

WaitWObj wobj_on_cnv1\RelDist:=0.0;

The first WaitWObj connects to the object in the start window. The second Wait-WObj will return immediately if the object is still connected, but will wait for the next object if the previous object had moved past the Maximum Distance or was dropped.

WaitWObj wobj_on_cnv1\RelDist:=500.0\MaxTime:=0.1\Timeflag:=flag1;

The **WaitWobj** will return immediately if the object thas passed 500 mm but otherwise will wait 0.1 sec for an object .If no object passes 500 mm
during this 0.1 sec the instruction will return with flag1 =TRUE.

Limitations

It requires 50 ms to connect to the first object in the start window. Once connected, a second **WaitWObj** with **\RelDist** optional argument will take only normal RAPID instruction execution time.

Error handling

If following errors occurs during execution of the *WaitWobj* instruction, the system variable ERRNO will be set. These errors can then be handled in the error handler.

ERR_CNV_NOT_ACT	The conveyor is not activated.
ERR_CNV_CONNECT	The *WaitWobj* instruction is already connected.
ERR_CNV_DROPPED	The object that the instruction *WaitWobj* was waiting for has been dropped by another task. (DSQC 354Revision 2: an object had passed the start window)
ERR_WAIT_MAXTIME	The object did not come in time and there is no Timeflag

Syntax

WaitWObj
    [WObj ':=']< persistent (PERS) of *wobjdata*> ';'
    ['\' RelDist ':=' < expression (IN) of *num* >]
    ['\'MaxTime ':='<expression (IN) of *num*>]
    ['\'TimeFlag':='<variable (VAR) of *bool*>] ';'

WarmStart - Restart the controller

WarmStart is used to restart the controller.

The system parameters can be changed from RAPID with the instruction *WriteCfg-Data.* You must restart the controller in order for a change to have effect on some of the system parameters. The restart can be done with this instruction *WarmStart*.

Examples

```
WriteCfgData “/MOC/MOTOR_CALIB/ROB_1”,”cal_offset”,offset1;
WarmStart;
```

Writes the value of the num variable *offset1* as calibration offset for axis ROB_1 and generates a restart of the controller.

Program execution

Warmstart takes effect at once and the program pointer is set to the next instruction.

Syntax

```
WarmStart ‘;’
```

Related information

Table 91

	Described in:
Write attribute of a system parameter	Instructions - *WriteCfgData*
Configuration	System Parameters

WHILE - Repeats as long as ...

WHILE is used when a number of instructions are to be repeated as long as a given condition expression evaluates to a *TRUE* value.

Example

```
WHILE reg1 < reg2 DO
    ...
    reg1 := reg1 + 1;
ENDWHILE
```

Repeats the instructions in the *WHILE*-block as long as *reg1 < reg2*.

Arguments

WHILE Condition DO ... ENDWHILE

Condition **Data type:** *bool*

The condition that must be evaluated to a *TRUE* value for the instructions in the *WHILE*-block to be executed.

Program execution

1. The condition expression is evaluated. If the expression evaluates to a *TRUE* value, the instructions in the *WHILE*-block are executed.

2. The condition expression is then evaluated again and if the result of this evaluation is *TRUE*, the instructions in the *WHILE*-block are executed again.

3. This process continues until the result of the expression evaluation becomes *FALSE*.
The iteration is then terminated and the program execution continues from the instruction after the *WHILE*-block.
If the result of the expression evaluation is *FALSE* at the very outset, the instructions in the *WHILE*-block are not executed at all and the program control transfers immediately to the instruction that follows after the *WHILE*-block.

Remarks

If it is possible to determine the number of repetitions, the *FOR* instruction can be used.

Syntax

```
(EBNF)
WHILE <conditional expression> DO
    <instruction list>
ENDWHILE
```

Related information

Table 92

	Described in:
Expressions	Basic Characteristics - *Expressions*
Repeats a given number of times	Instructions - *FOR*

WorldAccLim - Control acceleration in world coordinate system

WorldAccLim (World Acceleration Limitation) is used to limit the acceleration/deceleration of the tool (and gripload) in the world coordinate system.

Only implemented for robot type IRB5400-04, IRB6600 and IRB7600 with track motion.

The limitation will be achieved in the gravity centre point of the actual tool, actual gripload (if present) and the mounting flange of the robot, all together.

This instruction can only be used in the *Main* task or, if in a MultiMove System, in Motion tasks.

Example

WorldAccLim \On := 3.5;

Acceleration is limited to $3.5 m/s^2$.

WorldAccLim \Off;

The acceleration is reset to maximum (default).

Arguments

WorldAccLim [\On] | [\Off]

[\On] **Data type:** *num*

The absolute value of the acceleration limitation in m/s^2.

[\Off] **Data type:** *switch*

Maximum acceleration (default).

Program execution

The acceleration limitations applies for the next executed robot segment and is valid until a new *WorldAccLim* instruction is executed.

The maximum acceleration (*WorldAccLim \Off*) is automatically set

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

It is recommended to use just one type of limitation of the acceleration. If a combination of instructions *WorldAccLim, AccSet and PathAccLim is done,* the system reduces the acceleration/deceleration in following order

- according *WorldAccLim*
- according *AccSet*
- according *PathAccLim*

Limitations

Can only be used together with robot type IRB5400-04 with track motion.

The minimum acceleration allowed is 1 m/s^2.

Error handling

If the argument *On* is set to a value too low, the system variable ERRNO is set to ERR_ACC_TOO_LOW. This error can then be handled in the error handler.

Syntax

```
WorldAccLim
   ['\'On ':=' <expression (IN) of num >] | ['\'Off ]';'
```

Related information

Table 93

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Motion settings data	Data Types - *motsetdata*
Reduction of acceleration	Instructions - *AccSet*
Limitation of acceleration along the path	Instructions - *PathAccLim*

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 "com2:", 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 *com2:*. 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

Table 94

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

WriteAnyBin - Writes data to a binary serial channel or file

WriteAnyBin (Write Any Binary) is used to write any type of data to a binary serial channel or file.

Example

```
VAR iodev channel2;
VAR orient quat1 := [1, 0, 0, 0];
...
Open "com2:", channel2 \Bin;
WriteAnyBin channel2, quat1;
```

The orient data quat1 is written to the channel referred to by *channel2*.

Arguments

WriteAnyBin IODevice Data

IODevice **Data type:** *iodev*

The name (reference) of the binary serial channel or file for the writing operation.

Data **Data type:** *ANYTYPE*

The VAR or PERS containing the data to be written.

Program execution

As many bytes as required for the specified data are written to the specified binary serial channel or file.

Limitations

This instruction can only be used for serial channels or files that have been opened for binary writing.

The data to be written by this instruction must have a *value* data type of *atomic, string*, or *record* data type. *Semi-value* and *non-value* data types cannot be used.

Array data cannot be used.

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;
VAR robtarget cur_robt;

Open "com2:", 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 current robot position
    cur_robt := CRobT(\Tool:= tool1\WObj:= wobj1);
    WriteAnyBin channel, cur_robt;
ENDIF

Close channel;
```

The current position of the robot is written to a binary serial channel.

Syntax

WriteAnyBin
  [IODevice':='] <variable (**VAR**) of *iodev*>','
  [Data':='] <var or pers (**INOUT**) of *ANYTYPE*>';'

Related information

Table 95

	Described in:
Opening (etc.) of serial channels or files	RAPID Summary - *Communication*
Read data from a binary serial channel or file	Functions - *ReadAnyBin*

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 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 "com2:", 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

Table 96

	Described in:
Opening (etc.) of serial channels	RAPID Summary - *Communication*
Convert a string to a byte data	Functions - *StrToByte*
Byte data	Data Types - *byte*

WriteBlock - write block of data to device

WriteBlock (Write Block) is used to write a block of data to a device connected to the serial sensor interface. The data is fetched from a file on ramdisk or floppy disk.

The sensor interface communicates with a maximum of two sensors over serial channels using the RTP1 transport protocol.
The two channels must be named "laser1:" and "swg:".

This is an example of a sensor channel configuration.

```
COM_PHY_CHANN:
                    -name "sio1:" -type "sio"-Channel 1-Baudrate 19200
COM_TRP:
                    -Name "laser1:"-Type "RTP1" -PhyChnnel "sio1"
```

Example

```
CONST string SensorPar := "flp1:senpar.cfg";
CONST num ParBlock:= 1;

! Write sensor parameters from flp1:senpar.cfg to sensor datablock 1.

WriteBlock ParBlock, SensorPar;
```

Arguments

WriteBlock BlockNo FileName [\SensorNo]

BlockNo **Data type:** *num*

The argument *BlockNo* is used to select the data block in the sensor block to be written.

FileName **Data type:** *string*

The argument *FileName* is used to select a file from which data is written to the data block in the sensor selected by the *BlockNo* argument.

[\SensorNo] **Data type:** *num*

The optional SensorNo is used if more than one sensor is connected to the robot controller.

SensorNo 0 selects the sensor connected to the "laser1:" channel.
SensorNo 1 selects the sensor connected to the "swg:" channel.

If the argument is left out the default SensorNo 0 is used.

Fault management

Error constant (*ERRNO* value)	Description
SEN_NO_MEAS	Measurement failure
SEN_NOREADY	Sensor unable to handle command
SEN_GENERRO	General sensor error
SEN_BUSY	Sensor busy
SEN_UNKNOWN	Unknown sensor
SEN_EXALARM	External sensor error
SEN_CAALARM	Internal sensor error
SEN_TEMP	Sensor temperature error
SEN_VALUE	Illegal communication value
SEN_CAMCHECK	Sensor check failure
SEN_TIMEOUT	Communication error

Syntax

WriteBlock
 [BlockNo ':='] < expression (**IN**) of *num* >
 [FileName ':='] < expression (**IN**) of *string* >
 [('\' SensorNo ':=' < expression (**IN**) of *num* >)] ';'

Related information

Table 97

	Described in:
Write a sensor variable	Instructions - *WriteVar*
Write a sensor data block	Instructions - *WriteBlock*
Read a sensor data block	Instructions - *ReadBlock*
Configuration of sensor communication	System Parameters - *Communication*

WriteCfgData - Writes attribute of a system parameter

WriteCfgData is used to write one attribute of a named system parameter (configuration data).

Examples

WriteCfgData "/MOC/MOTOR_CALIB/ROB_1","cal_offset",offset1;

Writes the value of the num variable *offset1* as calibration offset for axis ROB_1.

WriteCfgData "/EIO/EIO_USER_SIGNAL/process_error","Unit",io_unit;

Writes the value of the string variable *io_unit* as the name of the I/O unit where the signal *process_error* is defined.

Arguments

WriteCfgData InstancePath Attribute CfgData

InstancePath **Data type:** ***string***

Specifies a path to the named parameter to be modified. The format of this string is /DOMAIN/TYPE/InstanceName

Attribute **Data type:** *string*

The name of the attribute of the parameter to be written.

CfgData **Data type:** *any type*

The variable from which the new data to store is readed.
Depending on the attribute type, valid types are bool, num, or string.

Program execution

The value of the attribute specified by the *Attribute* argument is set according to the value of the variable specified by the *CfgData* argument.

Limitations

The conversion from RAPID program units (mm, degree, second etc.) to system parameter units (m, radian, second etc.) for *CfgData* of data type *num* must be done by the user in the RAPID program.

You must manual restart the controller in order for the change to have effect.

Only named parameters can be accessed, i.e. parameters where the first attribute is ‘name’, ‘Name’, or ‘NAME’.

RAPID strings are limited to 80 characters. In some cases, this can be in theory too small for the definition of *InstancePath*, *Attribute*, or *CfgData*.

Error handling

If it is not possible to find the data specified with “*InstancePath* + *Attribute*” in the configuration database, the system variable ERRNO is set to ERR_CFG_NOTFND.

If the data type for parameter *CfgData* is not equal to the real data type for the found data specified with “*InstancePath* + *Attribute*” in the configuration database, the system variable ERRNO is set to ERR_CFG_ILLTYPE.

If the data for parameter *CfgData* is outside limits (max./min. value), the system variable ERRNO is set to ERR_CFG_LIMIT.

If trying to write internal write protected data, the system variable ERRNO is set to ERR_CFG_INTERNAL.

These errors can then be handled in the error handler.

Syntax

WriteCfgData
    [InstancePath ’:=’] < expression (**IN**) of *string* >’,’
    [Attribute ’:=’] < expression (**IN**) of *string* > ’,’
    [CfgData ’:=’] < variable (**VAR**) of *anytype* > ’;’

Related information

Table 98

	Described in:
Definition of string	Data types- *string*
Read attribute of a system parameter	Instructions - *ReadCfgData*
Get robot name in current task	Functions - *RobName*
Configuration	System Parameters

WriteRawBytes - Write rawbytes data

WriteRawBytes is used to write data of type rawbytes to a device opened with Open\Bin.

Example

```
VAR iodev io_device;
VAR rawbytes raw_data_out;
VAR rawbytes raw_data_in;
VAR num no_of_bytes;
VAR num float := 0.2;
VAR string answer;

ClearRawBytes raw_data_out;
PackDNHeader "10", "20 1D 24 01 30 64", raw_data;
PackRawBytes float, raw_data_out, (RawBytesLen(raw_data_out)+1) \Float4;

Open "dsqc328_1:", io_device \Bin;
WriteRawBytes io_device, raw_data_out;
no_of_bytes := 10;
ReadRawBytes io_device, raw_data_in \Time:=1;
Close io_device;

UnpackRawBytes raw_data_in, 1, answer \ASCII:=10;
```

In this example *raw_data_out* is cleared, and then packed with DeviceNet header and a float with value *0.2.*

A device, "*dsqc328_1:*", is opened and the current valid data in *raw_data_out* is written to the device. Then the program waits for at most *1* second to read from the device, which is stored in the *raw_data_in*.

After having closed the device "*dsqc328_1:*", the read data is unpacked as a string of *10* characters and stored in *answer.*

Arguments

WriteRawBytes IODevice RawData [\NoOfBytes]

IODevice **Data type:** *iodev*

IODevice is the identifier of the device to which *RawData* shall be written.

RawData **Data type:** *rawbytes*

RawData is the data container to be written to *IODevice*.

[\NoOfBytes] **Data type:** *num*

\NoOfBytes tells how many bytes of *RawData* should be written to *IODevice*, starting at index 1.

If *\NoOfBytes* is not present, the current length of valid bytes in the variable *RawData* is written to device *IODevice*.

Program execution

During program execution data is written to the device indicated by *IODevice*.

If using *WriteRawBytes* for field bus commands, such as DeviceNet, the field bus always sends an answer. The answer must be handle in RAPID with the *ReadRawBytes* instruction.

The current length of valid bytes in the *RawData* variable is not changed.

Error handling

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

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

Syntax

WriteRawBytes
  [IODevice ':='] < variable (**VAR**) of *iodev*> ','
  [RawData ':='] < variable (**VAR**) of *rawbytes*>
  ['\'NoOfBytes ':=' < expression (**IN**) of *num*>]';'

Related information

Table 99

	Described in:
rawbytes data	Data Types - *rawbytes*
Get the length of *rawbytes* data	Functions - *RawBytesLen*
Clear the contents of *rawbytes* data	Instructions - *ClearRawBytes*
Copy the contents of *rawbytes* data	Instructions - *CopyRawBytes*
Pack DeviceNet header into *rawbytes* data	Instructions - *PackDNHeader*
Pack data into *rawbytes* data	Instructions - *PackRawBytes*
Read *rawbytes* data	Instructions - *ReadRawBytes*
Unpack data from *rawbytes* data	Instructions - *UnpackRawBytes*

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 "com2:", 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

Table 100

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

WriteVar - write variable

WriteVar (Write Variable) is used to write a variable to a device connected to the serial sensor interface.

The sensor interface communicates with a maximum of two sensors over serial channels using the RTP1 transport protocol.
The two channels must be named “laser1:” and “swg:”.

This is an example of a sensor channel configuration.

```
COM_PHY_CHANN:
          -name “sio1:” -type “sio”-Channel 1      -Baudrate 19200
COM_TRP:
          -Name “laser1:”-Type “RTP1” -PhyChnnel “sio1”
```

Example

```
! Define variable numbers
CONST num SensorOn := 6;
CONST num XCoord := 8;
CONST num YCoord := 9;
CONST num ZCoord := 10;
VAR pos SensorPos;
! Request start of sensor meassurements
WriteVar SensorOn, 1;
! Read a cartesian position from the sensor.
SensorPos.x := WriteVar XCoord;
SensorPos.y := WriteVar YCoord;
SensorPos.z := WriteVar ZCoord;
! Stop sensor
WriteVar SensorOn, 0;
```

Arguments

WriteVar VarNo VarData [\SensorNo]

VarNo **Data type:** *num*

The argument *VarNo* is used to select variable .

VarData **Data type:** *num*

The argument *VarData* defines the data which is to be written to the variable selected by the *VarNo* argument.

[\SensorNo] **Data type:** *num*

The optional SensorNo is used if more than one sensor is connected to the robot controller.

SensorNo 0 selects the sensor connected to the “laser1:” channel.
SensorNo 1 selects the sensor connected to the “swg:” channel.

If the argument is left out the default SensorNo 0 is used.

Fault management

Error constant (*ERRNO* value)	Description
SEN_NO_MEAS	Measurement failure
SEN_NOREADY	Sensor unable to handle command
SEN_GENERRO	General sensor error
SEN_BUSY	Sensor busy
SEN_UNKNOWN	Unknown sensor
SEN_EXALARM	External sensor error
SEN_CAALARM	Internal sensor error
SEN_TEMP	Sensor temperature error
SEN_VALUE	Illegal communication value
SEN_CAMCHECK	Sensor check failure
SEN_TIMEOUT	Communication error

Syntax

WriteVar
  [VarNo ’:=’] < expression (**IN**) of *num* >
  [VarData ’:=’] < expression (**IN**) of *num* >
  [(’\’ SensorNo ’:=’ < expression (**IN**) of *num* >)] ’;’

Related information

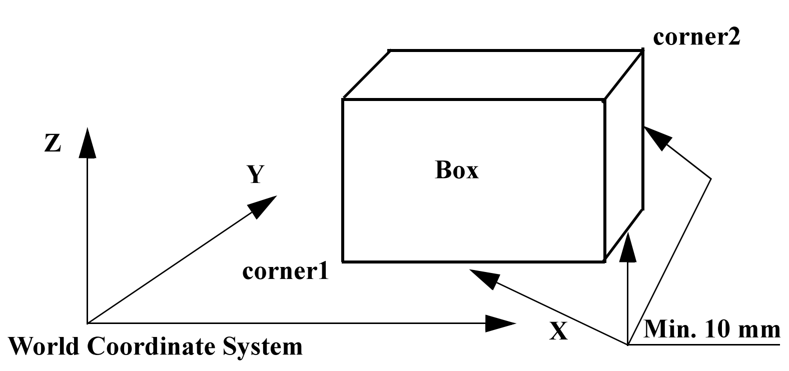
Table 101

	Described in:
Read a sensor variable	Instructions - *ReadVar*
Write a sensor data block	Instructions - *WriteBlock*
Read a sensor data block	Instructions - *ReadBlock*
Configuration of sensor communication	System Parameters - *Communication*

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 for 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

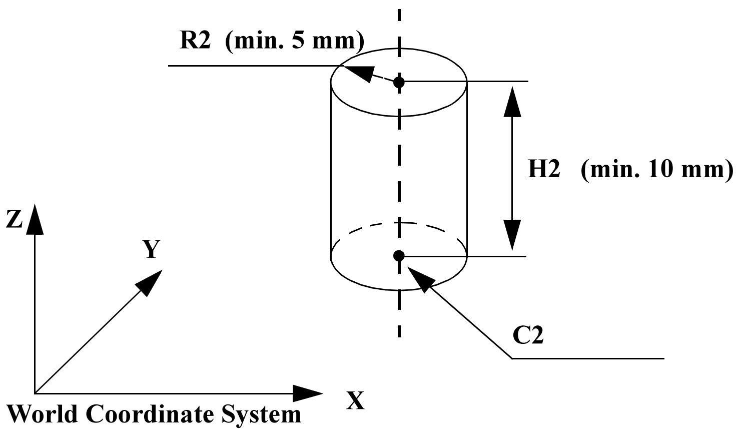
Table 102

	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*
Define a world zone for home joints	Instruction - *WZHomeJointDef*
Define a world zone for limit joints	Instruction - *WZLimJointDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

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

Table 103

	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 a world zone for home joints	Instruction - *WZHomeJointDef*
Define a world zone for limit joints	Instruction - *WZLimJointDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

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

Table 104

	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 or the robot/external axes (zone in joints) 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 *\Before* 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 (or robot/external joint position) is supervised and the output will be set, when the robot's TCP position (or robot/external joint position) is inside the volume (*\Inside*) or comes close to the border of the volume (*\Before*).

If use of *WZHomeJointDef* or *WZLimJointDef* together with *WZDOSet*, the digital output signal is set, only if all active axes with joint space supervision are before or inside the joint space.

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 *World-Zone*.

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

Table 105

	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	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

Table 106

	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

Table 107

	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*

WZHomeJointDef - Define a world zone for home joints

WZHomeJointDef (World Zone Home Joint Definition) is used to define a world zone in joints coordinates for both the robot and external axes to be used as a HOME or SERVICE position.

Example

```
VAR wzstationary home;
...
PROC power_on()
    VAR shapedata joint_space;
    CONST jointtarget home_pos := [ [ 0, 0, 0, 0, 0, -45],
        [ 0, 9E9, 9E9, 9E9, 9E9, 9E9] ];
    CONST jointtarget delta_pos  := [ [ 2, 2, 2, 2, 2, 2],
        [ 5, 9E9, 9E9, 9E9, 9E9, 9E9] ];
    ...
    WZHomeJointDef \Inside, joint_space, home_pos, delta_pos;
    WZDOSet \Stat, home \Inside, joint_space, do_home, 1;
ENDPROC
```

Definition and activation of stationary world zone *home*, that sets the signal *do_home to 1,* when all robot axes and the external axis *extax.eax_a* are at the joint position *home_pos* (within +/- *delta_pos* for each axes) during program execution and jogging. The variable *joint_space* of data type *shapedata* are used to transfer data from the instruction *WZHomeJointDef* to the instruction *WZDOSet*.

Arguments

WZHomeJointDef [\Inside] | [\Outside] Shape MiddleJointVal DeltaJointVal

[\Inside] **Data type:** *switch*

Define the joint space inside the *MiddleJointVal +/- DeltaJointVal.*

[\Outside] **Data type:** *switch*

Define the joint space outside the *MiddleJointVal +/- DeltaJointVal* (inverse joint space).

Shape **Data type:** *shapedata*

Variable for storage of the defined joint space (private data for the system).

MiddleJointVal **Data type:** *jointtarget*

The position in joint coordinates for the centre of the joint space to define. Specifies for each robot axes and external axes (degrees for rotational axes and mm for linear axes). Specifies in absolute joints (not in offset coordinate system *EOffsSet-EOffsOn* for external axes).
Value 9E9 for some axis means that the axis should not be supervised.
Not active external axis gives also 9E9 at programming time.

DeltaJointVal **Data type:** *jointtarget*

The +/- delta position in joint coordinates from the centre of the joint space.
The value must be greater than 0 for all axes to supervise.

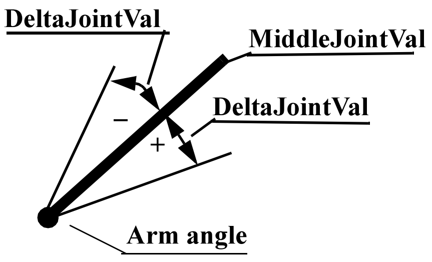


Figure 30 Definition of joint space for rotational axis

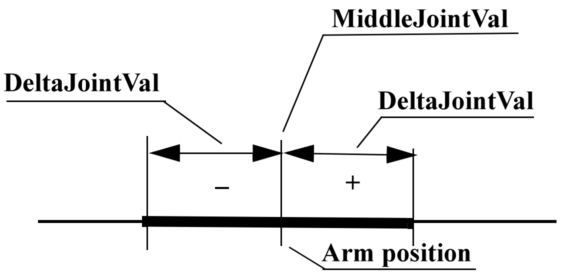


Figure 31 Definition of joint space for linear axis

Program execution

The definition of the joint space is stored in the variable of type *shapedata* (argument *Shape*), for future use in *WZLimSup* or *WZDOSet* instructions.

If use of *WZHomeJointDef* together with *WZDOSet*, the digital output signal is set, only if all active axes with joint space supervision are before or inside the joint space.

If use of *WZHomeJointDef* with outside joint space (argument *\Outside*) together with *WZLimSup*, the robot is stopped, as soon as one active axes with joint space supervision reach the joint space.

If use of *WZHomeJointDef* with inside joint space (argument *\Inside*) together with *WZLimSup*, the robot is stopped, as soon as the last active axes with joint space supervision reach the joint space. That means that one or several axes but not all active and supervised axes can be inside the joint space at the same time.

At execution of the instruction *ActUnit* or *DeactUnit* for activation or deactivation of mechanical units, will the supervision status for HOME position or work area limitation be updated.

Limitations

⚠ Only active mechanical units and it's active axes at activation time of the word zone (with instruction *WZDOSet* resp. *WZLimSup*), are included in the supervision of the HOME position resp. the limitatation of the working area. Besides that, the mecanical unit and it's axes must still be active at the movement from the program or jogging to be supervised.

For example, if one axis with supervision is outside it's HOME joint position but is deactivated, doesn't prevent the digital output signal for the HOME joint position to be set, if all other active axes with joint space supervision are inside the HOME joint position. At activation of that axis again, will it bee included in the supervision and the robot system will the be outside the HOME joint position and the digital output will be reset.

Syntax

WZHomeJointDef
  ['\'Inside] | ['\'Outside]','
  [Shape':=']<variable (**VAR**) of *shapedata*>','
  [MiddleJointVal ':=']<expression (**IN**) of *jointtarget*>','
  [DeltaJointVal ':=']<expression (**IN**) of *jointtarget*>';'

Related information

Table 108

	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*
Define sphere-shaped world zone	Instructions - *WZSphDef*
Define a world zone for limit joints	Instruction - *WZLimJointDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

WZLimJointDef - Define a world zone for limitation in joints

WZLimJointDef (World Zone Limit Joint Definition) is used to define a world zone in joints coordinates for both the robot and external axes to be used for limitation of the working area.

With *WZLimJointDef* it is possible to limitate the working area for each robot and external axes in the RAPID program, besides the limitation that can be done with *Configuration/Motion/Arm/robx_y/Upper Joint Bound ... Lower Joint Bound.*

Example

```
VAR wzstationary work_limit;
...
PROC power_on()
    VAR shapedata joint_space;
    CONST jointtarget low_pos  := [ [ -90, 9E9, 9E9, 9E9, 9E9, 9E9],
        [ -1000, 9E9, 9E9, 9E9, 9E9,
    CONST jointtarget high_pos := [ [ 90, 9E9, 9E9, 9E9,9E9, 9E9],
        [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
    ...
    WZLimJointDef \Outside, joint_space, low_pos, high_pos;
    WZLimSup \Stat, work_limit, joint_space;
ENDPROC
```

Definition and activation of stationary world zone *work_limit*, that limit the working area for robot axis 1 to -90 and +90 degreeds and the external axis *extax.eax_a* to -1000 mm during program execution and jogging. The variable *joint_space* of data type *shapedata* are used to transfer data from the instruction *WZLimJointDef* to the instruction *WZLimSup*.

Arguments

WZLimJointDef [\Inside] | [\Outside] Shape LowJointVal HighJointVal

[\Inside] **Data type:** *switch*

Define the joint space inside the *LowJointVal ... HighJointVal.*

[\Outside] **Data type:** *switch*

Define the joint space outside the *LowJointVal ... HighJointVal* (inverse joint space).

Shape **Data type:** *shapedata*

Variable for storage of the defined joint space (private data for the system).

LowJointVal **Data type:** *jointtarget*

The position in joint coordinates for the low limit of the joint space to define. Specifies for each robot axes and external axes (degrees for rotational axes and mm for linear axes). Specifies in absolute joints (not in offset coordinate system *EOffsSet-EOffsOn* for external axes).
Value 9E9 for some axis means that the axis should not be supervised for low limit. Not active external axis gives also 9E9 at programming time.

HighJointVal **Data type:** *jointtarget*

The position in joint coordinates for the high limit of the joint space to define. Specifies for each robot axes and external axes (degrees for rotational axes and mm for linear axes). Specifies in absolute joints (not in offset coordinate system *EOffsSet-EOffsOn* for external axes).
Value 9E9 for some axis means that the axis should not be supervised for high limit. Not active external axis gives also 9E9 at programming time.

(*HighJointVal-LowJointVal*) for each axis must be greater than 0 for all axes to supervise for both low and high limits.

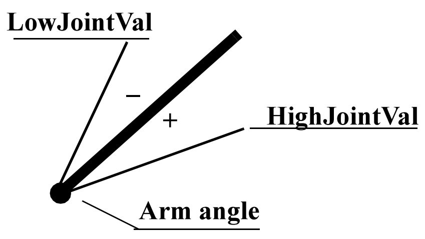


Figure 32 Definition of joint space for rotational axis

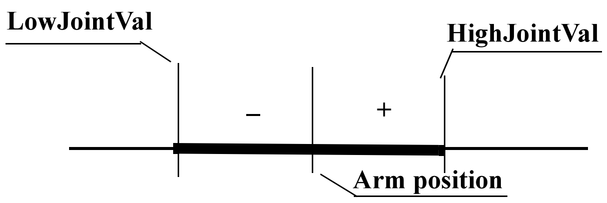


Figure 33 Definition of joint space for linear axis

Program execution

The definition of the joint space is stored in the variable of type *shapedata* (argument *Shape*), for future use in *WZLimSup* or *WZDOSet* instructions.

If use of *WZLimJointDef* together with *WZDOSet*, the digital output signal is set, only if all active axes with joint space supervision are before or inside the joint space.

If use of *WZLimJointDef* with outside joint space (argument *\Outside*) together with *WZLimSup*, the robot is stopped, as soon as one active axes with joint space supervision reach the joint space.

If use of *WZLimJointDef* with inside joint space (argument *\Inside*) together with *WZLimSup*, the robot is stopped, as soon as the last active axes with joint space supervision reach the joint space. That means that one or several axes but not all active and supervised axes can be inside the joint space at the same time.

At execution of the instruction *ActUnit* or *DeactUnit* will the supervision status be updated.

Limitations

Only active mechanical units and it's active axes at activation time of the word zone (with instruction *WZDOSet* resp. *WZLimSup*), are included in the supervision of the HOME position resp. the limitatation of the working area. Besides that, the mecanical unit and it's axes must still be active at the movement from the program or jogging to be supervised.

For example, if one axis with supervision is outside it's HOME joint position but is deactivated, doesn't prevent the digital output signal for the HOME joint position to be set, if all other active axes with joint space supervision are inside the HOME joint position. At activation of that axis again, will it bee included in the supervision and the robot system will the be outside the HOME joint position and the digital output will be reset.

Syntax

WZLimJointDef
    ['\'Inside] | ['\'Outside]','
    [Shape':=']<variable (**VAR**) of *shapedata*>','
    [LowJointVal ':=']<expression (**IN**) of *jointtarget*>','
    [HighJointVal ':=']<expression (**IN**) of *jointtarget*>';'

Related information

Table 109

	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*
Define sphere-shaped world zone	Instructions - *WZSphDef*
Define a world zone for home joints	Instruction - *WZHomeJointDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

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 or when the robot/external axes reaches the defined world zone in joints, 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 or the robot/external axes joint position is supervised. If it reaches the defined area the movement is stopped.

If use of *WZLimJointDef* or *WZHomeJointDef* with outside joint space (argument *\Outside*) together with *WZLimSup*, the robot is stopped, as soon as one active axes with joint space supervision reach the joint space.

If use of *WZLimJointDef* or *WZHomeJointDef* with inside joint space (argument *\Inside*) together with *WZLimSup*, the robot is stopped, as soon as the last active axes with joint space supervision reach the joint space. That means that one or several axes but not all active and supervised axes can be inside the joint space at the same time.

At execution of the instruction *ActUnit* or *DeactUnit* will the supervision status be updated.

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

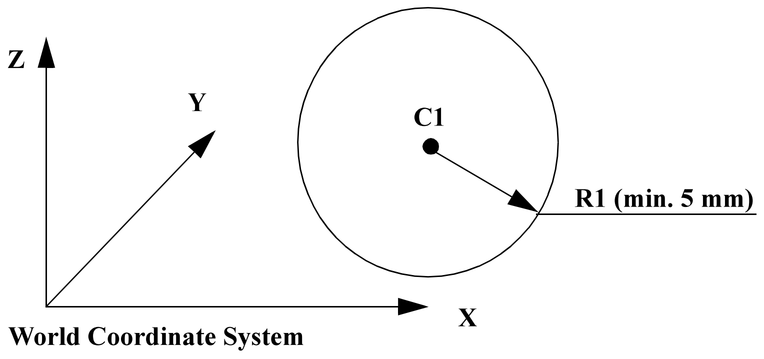
Table 110

	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*
Define a world zone for home joints	Instruction - *WZHomeJointDef*
Define a world zone for limit joints	Instruction - *WZLimJointDef*
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

Table 111

	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*
Define a world zone for home joints	Instruction - *WZHomeJointDef*
Define a world zone for limit joints	Instruction - *WZLimJointDef*
Activate world zone limit supervision	Instructions - *WZLimSup*
Activate world zone digital output set	Instructions - *WZDOSet*

ABB Automation Technologies AB
Robotics
SE-721 68 Västerås
SWEDEN
Telephone:	+46 (0) 21-34 40 00
Telefax:	+46 (0) 21-13 25 92

3HAC 16581-1 part 1b, Revision B , En