RAPID Reference Manual

BaseWare OS 2.1

Article number: 3HAC 7677-1
Issue: M2000/Rev.1

ABB Robotics Products AB
S-721 68 Västerås
Sweden

MAIN MENU

WELCOME to the Internal manual

Use the tool field in the Acrobat Reader to manoeuvre through the on-line manual.

The buttons **Go back** and **Go forward** take you step by step through the document, view to view in the order you have seen them.

The buttons **Previous/Next page** move forward or backwards in the document one page at the time. You can also use the buttons **PageUp**/**PageDown** on the key board.

The buttons **First/Last page** move forward or backwards to the last or first page of the document.

To mark text in the document. For use in another document the text can be copied with the command **Copy** in the menu **Edit**.

To mark graphics in the document (in the menu **Tools**). For use in another document the graphics can be copied in the same way as text, see above.

It is also possible to print the manual, use the function **Print....** in the menu **File**.

The cursor, in shape of a hand, changes to a pointing finger when it moves over a linked area. To jump to a linked view, click with the mouse button.

For more information, see the Acrobat Reader on-line manual under the menu **Help**.

Click on the Main menu button to continue to the User's Guide on-line Manual.

Main menu

CONTENTS

Page

Page

INDEX

S

T

CONTENTS

Introduction

This is a reference manual containing a detailed explanation of the programming language as well as all *data types*, *instructions* and *functions*. If you are programming off-line, this manual will be particularly useful in this respect.

When you start to program the robot it is normally better to start with the User's Guide until you are familiar with the system.

1 Other Manuals

Before using the robot for the first time, you should read *Basic Operation*. This will provide you with the basics of operating the robot.

The User's Guide provides step-by-step instructions on how to perform various tasks, such as how to move the robot manually, how to program, or how to start a program when running production.

The Product Manual describes how to install the robot, as well as maintenance procedures and troubleshooting. This manual also contains a *Product Specification* which provides an overview of the characteristics and performance of the robot.

2 How to Read this Manual

To answer the questions *Which instruction should I use?* or *What does this instruction mean?,* see *Chapter 3: RAPID Summary*. This chapter briefly describes all instructions, functions and data types grouped in accordance with the instruction pick-lists you use when programming. It also includes a summary of the syntax, which is particularly useful when programming off-line.

Chapter 5: Basic Characteristics explains the inner details of the language. You would not normally read this chapter unless you are an experienced programmer.

Chapter 6: Motion and I/O Principles describes the various coordinate systems of the robot, its velocity and other motion characteristics during different types of execution.

Chapters 7-9 describe all *data types*, *instructions* and *functions*. They are described in alphabetical order for your convenience.

This manual describes all the data and programs provided with the robot on delivery. In addition to these, there are a number of predefined data and programs supplied with the robot, either on diskette or, or sometimes already loaded. *Chapter 10: Predefined Data and Programs* describes what happens when these are loaded into the robot.

If you program off-line, you will find some tips in *Chapter 11: Programming off-line*.

Chapter 13-15 describes the functionality when the robot is equipped with Process-

Ware, i.e. ArcWare, SpotWare and GlueWare.

If you want to find out what a particular menu command does, you should refer to *Chapter 18: Quick Reference*. This chapter can also be used as a pocket guide when you are working with the robot.

To make things easier to locate and understand, *Chapter 20* contains an *index* and a *glossary*.

If the robot is delivered or upgraded with some extra functionality, this is described in *Chapter 19: Special Functionality in this Robot*.

2.1 Typographic conventions

The commands located under any of the five menu keys at the top of the teach pendant display are written in the form of **Menu: Command**. For example, to activate the Print command in the File menu, you choose **File: Print**.

The names on the function keys and in the entry fields are specified in bold italic typeface, e.g. ***Modpos***.

Words belonging to the actual programming language, such as instruction names, are written in italics, e.g. *MoveL*.

Examples of programs are always displayed in the same way as they are output to a diskette or printer. This differs from what is displayed on the teach pendant in the following ways:

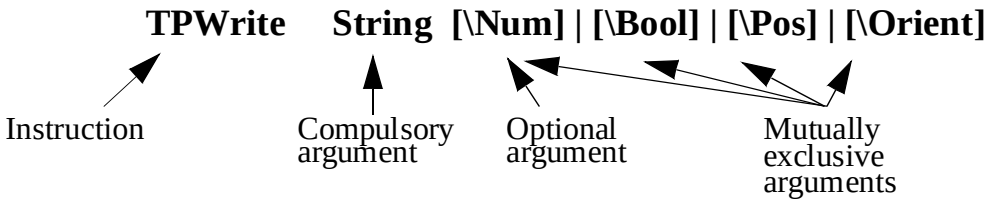
- Certain control words that are masked in the teach pendant display are printed, e.g. words indicating the start and end of a routine.
- Data and routine declarations are printed in the formal form, e.g. *VAR num reg1;*.

2.2 Syntax rules

Instructions and functions are described using both simplified syntax and formal syntax. If you use the teach pendant to program, you generally only need to know the simplified syntax, since the robot automatically makes sure that the correct syntax is used.

Simplified syntax

Example:



- Optional arguments are enclosed in square brackets [ ]. These arguments can be omitted.
- Arguments that are mutually exclusive, i.e. cannot exist in the instruction at the same time, are separated by a vertical bar |.
- Arguments that can be repeated an arbitrary number of times are enclosed in braces { }.

2.3 Formal syntax

Example:
```
TPWrite
    [String':='] <expression (IN) of string>
    ['\'Num':=' <expression (IN) of num> ] |
    ['\'Bool':=' <expression (IN) of bool> ] |
    ['\'Pos':=' <expression (IN) of pos> ] |
    ['\'Orient':=' <expression (IN) of orient> ]';'
```

- The text within the square brackets [ ] may be omitted.
- Arguments that are mutually exclusive, i.e. cannot exist in the instruction at the same time, are separated by a vertical bar |.
- Arguments that can be repeated an arbitrary number of times are enclosed in braces { }.
- Symbols that are written in order to obtain the correct syntax are enclosed in single quotation marks (apostrophes) ' '.
- The data type of the argument (italics) and other characteristics are enclosed in angle brackets < >. See the description of the parameters of a routine for more detailed information.

The basic elements of the language and certain instructions are written using a special syntax, EBNF. This is based on the same rules, but with some additions.

Example:
```
GOTO <identifier>';'
    <identifier> ::= <ident>
                   | <ID>
    <ident> ::= <letter> {<letter> | <digit> | '_'}
```

- The symbol ::= means *is defined as*.
- Text enclosed in angle brackets < > is defined in a separate line.

CONTENTS

Page

Page

1 The Structure of the Language

The program consists of a number of instructions which describe the work of the robot. Thus, there are specific instructions for the various commands, such as one to move the robot, one to set an output, etc.

The instructions generally have a number of associated arguments which define what is to take place in a specific instruction. For example, the instruction for resetting an output contains an argument which defines which output is to be reset; e.g. *Reset do5*. These arguments can be specified in one of the following ways:

- as a numeric value, e.g. *5* or *4.6*
- as a reference to data, e.g. *reg1*
- as an expression, e.g. *5+reg1*2*
- as a function call, e.g. *Abs(reg1)*
- as a string value, e.g. *"Producing part A"*

There are three types of routines – *procedures*, *functions* and *trap routines*.

- A procedure is used as a subprogram.
- A function returns a value of a specific type and is used as an argument of an instruction.
- Trap routines provide a means of responding to interrupts. A trap routine can be associated with a specific interrupt; e.g. when an input is set, it is automatically executed if that particular interrupt occurs.

Information can also be stored in data, e.g. tool data (which contains all information on a tool, such as its TCP and weight) and numerical data (which can be used, for example, to count the number of parts to be processed). Data is grouped into different data types which describe different types of information, such as tools, positions and loads. As this data can be created and assigned arbitrary names, there is no limit (except that imposed by memory) on the number of data. These data can exist either globally in the program or locally within a routine.

There are three kinds of data – *constants*, *variables* and *persistents*.

- A constant represents a static value and can only be assigned a new value manually.
- A variable can also be assigned a new value during program execution.
- A persistent can be described as a “persistent” variable. When a program is saved the initialization value reflects the current value of the persistent.

Other features in the language are:

- Routine parameters
- Arithmetic and logical expressions
- Automatic error handling
- Modular programs
- Multi tasking

2 Controlling the Program Flow

The program is executed sequentially as a rule, i.e. instruction by instruction. Sometimes, instructions which interrupt this sequential execution and call another instruction are required to handle different situations that may arise during execution.

2.1 Programming principles

The program flow can be controlled according to five different principles:

- By calling another routine (procedure) and, when that routine has been executed, continuing execution with the instruction following the routine call.
- By executing different instructions depending on whether or not a given condition is satisfied.
- By repeating a sequence of instructions a number of times or until a given condition is satisfied.
- By going to a label within the same routine.
- By stopping program execution.

2.2 Calling another routine

Instruction	Used to:
ProcCall	Call (jump to) another routine
CallByVar	Call procedures with specific names
RETURN	Return to the original routine

2.3 Program control within the routine

Instruction	Used to:
Compact IF	Execute one instruction only if a condition is satisfied
IF	Execute a sequence of different instructions depending on whether or not a condition is satisfied
FOR	Repeat a section of the program a number of times
WHILE	Repeat a sequence of different instructions as long as a given condition is satisfied
TEST	Execute different instructions depending on the value of an expression
GOTO	Jump to a label
label	Specify a label (line name)

2.4 Stopping program execution

Instruction	Used to:
Stop	Stop program execution
EXIT	Stop program execution when a program restart is not allowed
Break	Stop program execution temporarily for debugging purposes

3 Various Instructions

Various instructions are used to

- assign values to data
- wait a given amount of time or wait until a condition is satisfied
- insert a comment into the program
- load program modules.

3.1 Assigning a value to data

Data can be assigned an arbitrary value. It can, for example, be initialized with a constant value, e.g. 5, or updated with an arithmetic expression, e.g. *reg1+5*reg3*.

Instruction	Used to:
:=	Assign a value to data

3.2 Wait

The robot can be programmed to wait a given amount of time, or to wait until an arbitrary condition is satisfied; for example, to wait until an input is set.

Instruction	Used to:
WaitTime	Wait a given amount of time or to wait until the robot stops moving
WaitUntil	Wait until a condition is satisfied
WaitDI	Wait until a digital input is set
WaitDO	Wait until a digital output is set

3.3 Comments

Comments are only inserted into the program to increase its readability. Program execution is not affected by a comment.

Instruction	Used to:
comment	Comment on the program

3.4 Loading program modules

Program modules can be loaded from mass memory or erased from the program mem-

ory. In this way large programs can be handled with only a small memory.

3.5 Various functions

Function	Used to:
OpMode	Read the current operating mode of the robot
RunMode	Read the current program execution mode of the robot
Dim	Obtain the dimensions of an array
Present	Find out whether an optional parameter was present when a routine call was made
IsPers	Check whether a parameter is a persistent
IsVar	Check whether a parameter is a variable
Load	Load a program module into the program memory
UnLoad	Unload a program module from the program memory

3.6 Basic data

Data type	Used to define:
bool	Logical data (with the values true or false)
num	Numeric values (decimal or integer)
symnum	Numeric data with symbolic value
string	Character strings
switch	Routine parameters without value

4 Motion Settings

Some of the motion characteristics of the robot are determined using logical instructions that apply to all movements:

- Maximum velocity and velocity override
- Acceleration
- Management of different robot configurations
- Payload
- Behaviour close to singular points
- Program displacement
- Soft servo
- Tuning values

4.1 Programming principles

The basic characteristics of the robot motion are determined by data specified for each positioning instruction. Some data, however, is specified in separate instructions which apply to all movements until that data changes.

The general motion settings are specified using a number of instructions, but can also be read using the system variable *C_MOTSET* or *C_PROGDISP*.

Default values are automatically set (by executing the routine *SYS_RESET* in system module BASE)

- at a cold start-up,
- when a new program is loaded,
- when the program is started from the beginning.

4.2 Defining velocity

The absolute velocity is programmed as an argument in the positioning instruction. In addition to this, the maximum velocity and velocity override (a percentage of the programmed velocity) can be defined.

Instruction	Used to define:
VelSet	The maximum velocity and velocity override

4.3 Defining acceleration

When fragile parts, for example, are handled, the acceleration can be reduced for part of the program.

Instruction	Used to define:
AccSet	The maximum acceleration

4.4 Defining configuration management

The robot's configuration is normally checked during motion. If joint (axis-by-axis) motion is used, the correct configuration will be achieved. If linear or circular motion are used, the robot will always move towards the closest configuration, but a check is performed to see if it is the same as the programmed one. It is possible to change this, however.

Instruction	Used to define:
ConfJ	Configuration control on/off during joint motion
ConfL	Configuration check on/off during linear motion

4.5 Defining the payload

To achieve the best robot performance, the correct payload must be defined.

Instruction	Used to define:
GripLoad	The payload of the gripper

4.6 Defining the behaviour near singular points

The robot can be programmed to avoid singular points by changing the tool orientation automatically.

Instruction	Used to define:
SingArea	The interpolation method through singular points

4.7 Displacing a program

When part of the program must be displaced, e.g. following a search, a program displacement can be added.

Instruction	Used to:
PDispOn	Activate program displacement
PDispSet	Activate program displacement by specifying a value
PDispOff	Deactivate program displacement
EOffsOn	Activate an external axis offset
EOffsSet	Activate an external axis offset by specifying a value
EOffsOff	Deactivate an external axis offset

Function	Used to:
DefDFrame	Calculate a program displacement from three positions
DefFrame	Calculate a program displacement from six positions
ORobT	Remove program displacement from a position

4.8 Soft servo

One or more of the robot axes can be made “soft”. When using this function, the robot will be compliant and can replace, for example, a spring tool.

Instruction	Used to:
SoftAct	Activate the soft servo for one or more axes
SoftDeact	Deactivate the soft servo

4.9 Adjust the robot tuning values

In general, the performance of the robot is self-optimising; however, in certain extreme cases, overrunning, for example, can occur. You can adjust the robot tuning values to obtain the required performance.

Instruction	Used to:
TuneServo	Adjust the robot tuning values
TuneReset	Reset tuning to normal
tunetype	Represent the tuning type as a symbolic constant

4.10 Data for motion settings

Data type	Used to define:
motsetdata	Motion settings except program displacement
progdisp	Program displacement

5 Motion

The robot movements are programmed as pose-to-pose movements, i.e. “move from the current position to a new position”. The path between these two positions is then automatically calculated by the robot.

5.1 Programming principles

The basic motion characteristics, such as the type of path, are specified by choosing the appropriate positioning instruction.

The remaining motion characteristics are specified by defining data which are arguments of the instruction:

- Position data (end position for robot and external axes)
- Speed data (desired speed)
- Zone data (position accuracy)
- Tool data (e.g. the position of the TCP)
- Work-object data (e.g. the current coordinate system)

Some of the motion characteristics of the robot are determined using logical instructions which apply to all movements (See *Motion Settings* on page 10):

- Maximum velocity and velocity override
- Acceleration
- Management of different robot configurations
- Payload
- Behaviour close to singular points
- Program displacement
- Soft servo
- Tuning values

Both the robot and the external axes are positioned using the same instructions. The external axes are moved at a constant velocity, arriving at the end position at the same time as the robot.

5.2 Positioning instructions

Instruction	Type of movement:
MoveC	TCP moves along a circular path
MoveJ	Joint movement
MoveL	TCP moves along a linear path
MoveAbsJ	Absolute joint movement

5.3 Searching

During the movement, the robot can search for the position of a work object, for example. The searched position (indicated by a sensor signal) is stored and can be used later to position the robot or to calculate a program displacement.

Instruction	Type of movement:
SearchC	TCP along a circular path
SearchL	TCP along a linear path

5.4 Activating outputs or interrupts at specific positions

Normally, logical instructions are executed in the transition from one positioning instruction to another. If, however, special motion instructions are used, these can be executed instead when the robot is at a specific position.

Instruction	Used to:
TriggIO[1]	Define a trigg condition to set an output at a given position
*TriggInt*1	Define a trigg condition to execute a trap routine at a given position
TriggEquip[1]	Define a trigg condition to set an output at a given position with the possibility to include time compensation for the lag in the external equipment
*TriggC*1	Run the robot (TCP) circularly with an activated trigg condition
*TriggJ*1	Run the robot axis-by-axis with an activated trigg condition
*TriggL*1	Run the robot (TCP) linearly with an activated trigg condition

Data type	Used to define:
*triggdata*1	Trigg conditions

1. Only if the robot is equipped with the option “Advanced functions”

5.5 Motion control if an error/interrupt takes place

In order to rectify an error or an interrupt, motion can be stopped temporarily and then restarted again.

Instruction	Used to:
StopMove	Stop the robot movements
StartMove	Restart the robot movements
*StorePath*1	Store the last path generated
*RestoPath*1	Regenerate a path stored earlier

1. Only if the robot is equipped with the option “Advanced Functions”.

5.6 Controlling external axes

The robot and external axes are usually positioned using the same instructions. Some instructions, however, only affect the external axis movements.

Instruction	Used to:
DeactUnit	Deactivate an external mechanical unit
ActUnit	Activate an external mechanical unit

5.7 Independent axes

The robot’s axis 6, or an external axis, can be moved independently of other movements. The working area of an axis can also be reset, which will reduce the cycle times.

Function	Used to:
IndAMove[2]	Change an axis to independent mode and move the axis to an absolute position
IndCMove[2]	Change an axis to independent mode and start the axis moving continuously
IndDMove[2]	Change an axis to independent mode and move the axis a delta distance
IndRMove[2]	Change an axis to independent mode and move the axis to a relative position (within the axis revolution)
IndReset[2]	Change an axis to dependent mode or/and reset the working area
IndInpos[2]	Check whether an independent axis is in position
IndSpeed[2]	Check whether an independent axis has reached programmed speed

2. Only if the robot is equipped with the option “Advanced Motion”.

5.8 Position functions

Function	Used to:
Offs	Add an offset to a robot position, expressed in relation to the work object
RelTool	Add an offset, expressed in the tool coordinate system
CPos	Read the current position (only *x*, *y*, *z* of the robot)
CRobT	Read the current position (the complete *robtarget*)
CJointT	Read the current joint angles
ReadMotor	Read the current motor angles
CTool	Read the current tooldata value
CWObj	Read the current wobjdata value
ORobT	Remove a program displacement from a position
MirPos	Mirror a position

5.9 Motion data

Motion data is used as an argument in the positioning instructions.

Data type	Used to define:
robtarget	The end position
jointtarget	The end position for a *MoveAbsJ* instruction
speeddata	The speed
zonedata	The accuracy of the position (stop point or fly-by point)
tooldata	The tool coordinate system and the load of the tool
wobjdata	The work object coordinate system

5.10 Basic data for movements

Data type	Used to define:
pos	A position (x, y, z)
orient	An orientation
pose	A coordinate system (position + orientation)
confdata	The configuration of the robot axes
extjoint	The position of the external axes
robjoint	The position of the robot axes
o_robtarget	Original robot position when *Limit ModPos* is used
o_jointtarget	Original robot position when *Limit ModPos* is used for *MoveAbsJ*
loaddata	A load
mecunit	An external mechanical unit

6 Input and Output Signals

The robot can be equipped with a number of digital and analog user signals that can be read and changed from within the program.

6.1 Programming principles

The signal names are defined in the system parameters and, using these names, can be read from the program.

The value of an analog signal or a group of digital signals is specified as a numeric value.

6.2 Changing the value of a signal

Instruction	Used to:
InvertDO	Invert the value of a digital output signal
PulseDO	Generate a pulse on a digital output signal
Reset	Reset a digital output signal (to 0)
Set	Set a digital output signal (to 1)
SetAO	Change the value of an analog output signal
SetDO	Change the value of a digital output signal (symbolic value; e.g. *high/low*)
SetGO	Change the value of a group of digital output signals

6.3 Reading the value of an input signal

The value of an input signal can be read directly, e.g. IF di1=1 THEN ...

6.4 Reading the value of an output signal

Function	Used to read the value of:
DOutput	A digital output signal
GOutput	A group of digital output signals

6.5 Testing input on output signals

Instruction	Used to:
WaitDI	Wait until a digital input is set or reset
WaitDO	Wait until a digital output is set on reset

Function	Used to:
TestDI	Test whether a digital input is set

6.6 Data for input and output signals

Data type	Used to define:
dionum	The symbolic value of a digital signal
signalai	The name of an analog input signal *
signalao	The name of an analog output signal *
signaldi	The name of a digital input signal *
signaldo	The name of a digital output signal *
signalgi	The name of a group of digital input signals *
signalgo	The name of a group of digital output signals *

* Only to be defined using system parameters.

7 Communication

There are four possible ways to communicate via serial channels:

- Messages can be output to the teach pendant display and the user can answer questions, such as about the number of parts to be processed.
- Character-based information can be written to or read from text files on mass memory. In this way, for example, production statistics can be stored and processed later in a PC. Information can also be printed directly on a printer connected to the robot.
- Binary information can be transferred between the robot and a sensor, for example.
- Binary information can be transferred between the robot and another computer, for example, with a link protocol.

7.1 Programming principles

The decision whether to use character-based or binary information is dependent on how the equipment with which the robot communicates handles that information. A file, for example, can have data that is stored in character-based or binary form.

If communication is required in both directions simultaneously, binary transmission is necessary.

Each serial channel or file used must first be opened. On doing this, the channel/file receives a descriptor that is then used as a reference when reading/writing. The teach pendant can be used at all times and does not need to be opened.

Both text and the value of certain types of data can be printed.

7.2 Communicating using the teach pendant

Instruction	Used to:
TPErase	Clear the teach pendant operator display
TPWrite	Write text on the teach pendant operator display
ErrWrite	Write text on the teach pendant display and simultaneously store that message in the progam's error log.
TPReadFK	Label the function keys and to read which key is pressed
TPReadNum	Read a numeric value from the teach pendant

7.3 Reading from or writing to a character-based serial channel/file

Instruction	Used to:
Open[1]	Open a channel/file for reading or writing
*Write*1	Write text to the channel/file
*Close*1	Close the channel/file

Function	Used to:
*ReadNum*1	Read a numeric value
*ReadStr*1	Read a text string

7.4 Communicating using binary serial channels

Instruction	Used to:
*Open*1	Open a serial channel/file for binary transfer of data
*WriteBin*1	Write to a binary serial channel
*Close*1	Close the channel/file

Function	Used to:
*ReadBin*1	Read from a binary serial channel

7.5 Data for serial channels

Data type	Used to define:
iodev	A reference to a serial channel/file, which can then be used for reading and writing

1. Only if the robot is equipped with the option “Advanced functions”

8 Interrupts

Interrupts are used by the program to enable it to deal directly with an event, regardless of which instruction is being run at the time.

The program is interrupted, for example, when a specific input is set to one. When this occurs, the ordinary program is interrupted and a special trap routine is executed. When this has been fully executed, program execution resumes from where it was interrupted.

8.1 Programming principles

Each interrupt is assigned an interrupt identity. It obtains its identity by creating a variable (of data type *intnum*) and connecting this to a trap routine.

The interrupt identity (variable) is then used to order an interrupt, i.e. to specify the reason for the interrupt. This may be one of the following events:

- An input or output is set to one or to zero.
- A given amount of time elapses after an interrupt is ordered.
- A specific position is reached.

When an interrupt is ordered, it is also automatically enabled, but can be temporarily disabled. This can take place in two ways:

- All interrupts can be disabled. Any interrupts occurring during this time are placed in a queue and then automatically generated when interrupts are enabled again.
- Individual interrupts can be deactivated. Any interrupts occurring during this time are disregarded.

8.2 Connecting interrupts to trap routines

Instruction	Used to:
CONNECT	Connect a variable (interrupt identity) to a trap routine

8.3 Ordering interrupts

Instruction	Used to order:
ISignalDI	An interrupt from a digital input signal
SignalDO	An interrupt from a digital output signal
ITimer	A timed interrupt
TriggInt[1]	A position-fixed interrupt (from the Motion pick list)

8.4 Cancelling interrupts

Instruction	Used to:
IDelete	Cancel (delete) an interrupt

8.5 Enabling/disabling interrupts

Instruction	Used to:
ISleep	Deactivate an individual interrupt
IWatch	Activate an individual interrupt
IDisable	Disable all interrupts
IEnable	Enable all interrupts

8.6 Data type of interrupts

Data type	Used to define:
intnum	The identity of an interrupt

1. Only if the robot is equipped with the option “Advanced functions”

9 Error Recovery

Many of the errors that occur when a program is being executed can be handled in the program, which means that program execution does not have to be interrupted. These errors are either of a type detected by the robot, such as division by zero, or of a type that is detected by the program, such as errors that occur when an incorrect value is read by a bar code reader.

9.1 Programming principles

When an error occurs, the error handler of the routine is called (if there is one). It is also possible to create an error from within the program and then jump to the error handler.

If the routine does not have an error handler, a call will be made to the error handler in the routine that called the routine in question. If there is no error handler there either, a call will be made to the error handler in the routine that called that routine, and so on until the internal error handler of the robot takes over and outputs an error message and stops program execution.

In the error handler, errors can be handled using ordinary instructions. The system data *ERRNO* can be used to determine the type of error that has occurred. A return from the error handler can then take place in various ways.

In future releases, if the current routine does not have an error handler, the internal error handler of the robot takes over directly. The internal error handler outputs an error message and stops program execution with the program pointer at the faulty instruction.

So, a good rule already in this issue is as follows: if you want to call the error handler of the routine that called the current routine (propagate the error), then:

- Add an error handler in the current routine
- Add the instruction RAISE in this error handler.

9.2 Creating an error situation from within the program

Instruction	Used to:
RAISE	“Create” an error and call the error handler

9.3 Restarting/returning from the error handler

Instruction	Used to:
EXIT	Stop program execution in the event of a fatal error
RAISE	Call the error handler of the routine that called the current routine
RETRY	Re-execute the instruction that caused the error
TRYNEXT	Execute the instruction following the instruction that caused the error
RETURN	Return to the routine that called the current routine

9.4 Data for error handling

Data type	Used to define:
errnum	The reason for the error

10 System & Time

System and time instructions allow the user to measure, inspect and record time.

10.1 Programming principles

Clock instructions allow the user to use clocks that function as stopwatches. In this way the robot program can be used to time any desired event.

The current time or date can be retrieved in a string. This string can then be displayed to the operator on the teach pendant display or used to time and date-stamp log files.

It is also possible to retrieve components of the current system time as a numeric value. This allows the robot program to perform an action at a certain time or on a certain day of the week.

10.2 Using a clock to time an event

Instruction	Used to:
ClkReset	Reset a clock used for timing
ClkStart	Start a clock used for timing
ClkStop	Stop a clock used for timing

Function	Used to:
ClkRead	Read a clock used for timing

Data Type	Used for:
clock	Timing – stores a time measurement in seconds

10.3 Reading current time and date

Function	Used to:
CDate	Read the Current Date as a string
CTime	Read the Current Time as a string
GetTime	Read the Current Time as a numeric value

11 Mathematics

Mathematical instructions and functions are used to calculate and change the value of data.

11.1 Programming principles

Calculations are normally performed using the assignment instruction, e.g. *reg1:= reg2 + reg3 / 5*. There are also some instructions used for simple calculations, such as to clear a numeric variable.

11.2 Simple calculations on numeric data

Instruction	Used to:
Clear	Clear the value
Add	Add or subtract a value
Incr	Increment by 1
Decr	Decrement by 1

11.3 More advanced calculations

Instruction	Used to:
:=	Perform calculations on any type of data

11.4 Arithmetic functions

Function	Used to:
Abs	Calculate the absolute value
Round	Round a numeric value
Trunc	Truncate a numeric value
Sqrt	Calculate the square root
Exp	Calculate the exponential value with the base “e”
Pow	Calculate the exponential value with an arbitrary base
ACos	Calculate the arc cosine value
ASin	Calculate the arc sine value
ATan	Calculate the arc tangent value in the range [-90,90]
ATan2	Calculate the arc tangent value in the range [-180,180]
Cos	Calculate the cosine value
Sin	Calculate the sine value
Tan	Calculate the tangent value
EulerZYX	Calculate Euler angles from an orientation
OrientZYX	Calculate the orientation from Euler angles
PoseInv	Invert a pose
PoseMult	Multiply a pose
PoseVect	Multiply a pose and a vector

12 Spot Welding

The SpotWare package provides support for spot welding applications that are equipped with a weld timer and on/off weld gun.

The SpotWare application provides fast and accurate positioning combined with gun manipulation, process start and supervision of an external weld timer.

Communication with the welding equipment is carried out by means of digital inputs and outputs. Some serial weld timer interfaces are also supported: Bosch PSS5000, NADEX, ABB Timer. See separate documentation.

It should be noted that SpotWare and SpotWare Plus, even more, are packages that can be extensively customised. The user routines are meant to be adapted to the environmental situation.

12.1 Spot welding features

The SpotWare package contains the following features:

- Fast and accurate positioning
- Handling of an on/off gun with two strokes
- Dual/single gun
- Gun pre-closing
- User defined supervision of the surrounding equipment before weld start
- User defined supervision of the surrounding equipment after the weld
- Monitoring of the external weld timer
- Weld error recovery with automatic rewelding
- Return to the spot weld position
- Spot counters
- Time- or signal-dependent motion release after a weld
- Quick start after a weld
- User-defined service routines
- Presetting and checking of gun pressure
- Simulated welding
- Reverse execution with gun control
- Parallel and serial weld timer interfaces
- Supports both program and start triggered weld timers

SpotWare Plus also contains the following features:

- User defined open/close gun and supervision
- User defined pressure setting
- User defined preclose time calculation
- SpotL current data information
- User defined autonomous supervision, such as state controlled weld current signal and water cooling start.
- Manual weld, gun open and gun close initiated by digital input
- Weld process start disregarding the in position event, is possible

12.2 Principles of SpotWare

When a SpotL instruction is executed, one main sequence consisting of motion and logical instructions is executed.

During the motion it is possible to set predefined signals but not execution of RAPID-code.

For well defined entries, calls to user routines offer adaptations to the plant environment. A number of predefined parameters are also available to shape the behaviour of the SpotL instruction.

The RAPID part of SpotL is executing only when the user's program is running. If the user program stops, then the internal RAPID execution of SpotL will stop immediately.

Supported equipment:

- One weld timer monitoring with standard parallel (some serial) interface
- Single/dual on-off gun with one signal/tip for closing the gun and one for setting opening gap
- Pressure level preset 1..4.

12.3 Principles of SpotWare Plus

SpotWare Plus is based on a separate handling of motion, spot-welding and supervision. On its way towards the programmed position, the motion task will trigger actions in the spot-welding tasks.

The triggers are activated by virtual digital signals.

The tasks work with their own internal encapsulated variables and with persistents which are fully transparent for all tasks.

A program stop will only stop the motion task execution. The process and supervision

carry on their tasks until they come to a well defined process stop. For example, this will make the gun open after a finished weld, although the program has stopped.

The opening and closing of the gun are always executed by RAPID routines, even if activated manually from the I/O window on the teach-pendant. These gun routines may be changed from the simple on/off default functionality to a more complex like analog gun control.

Since the process and supervision tasks are acting on I/O triggers, they will be executed either by the trig that was sent by the motion (SpotL) or by manual activation (teach pendant or external). This offers the possibility of performing a stand-alone weld anywhere without programming a new position.

It is also possible to define new supervision events and to connect them to digital signal triggers. By default, a state dependent weld power and water cooling signal control are implemented.

Supported equipment:

- One weld timer monitoring with standard parallel (some serial) interface
- Any type of single/dual gun close and gun gap control.
- Any type of pressure preset.
- Event controlled SpotL-independent spot weld equipment such as contactors etc.

12.4 Programming principles

Both the robot's linear movement and the spot weld process control are embedded in one instruction, *SpotL*.

The spot welding process is specified by:

- Spotdata: spot weld process data
- Gundata: spot weld gun data
- The system module SWUSER, SWUSRF and SWUSRC: RAPID routines and global data for customising purposes. See *Predefined Data and Programs ProcessWare*.
- System parameters: the I/O configuration. See User's Guide - System Parameters

12.5 Spot welding instructions

Instruction	Used to:
SpotL	Control the motion, gun closure/opening and the welding process Move the TCP along a linear path and perform a spot weld at the end position

12.6 Spot welding data

Data type	Used to define:
spotdata	The spot weld process control
gundata	The spot weld gun

13 Arc Welding

The ArcWare package supports most welding functions. Crater-filling and scraping starts can, for example, be programmed. Using ArcWare, the whole welding process can be controlled and monitored by the robot via a number of different digital and analog inputs and outputs.

13.1 Programming principles

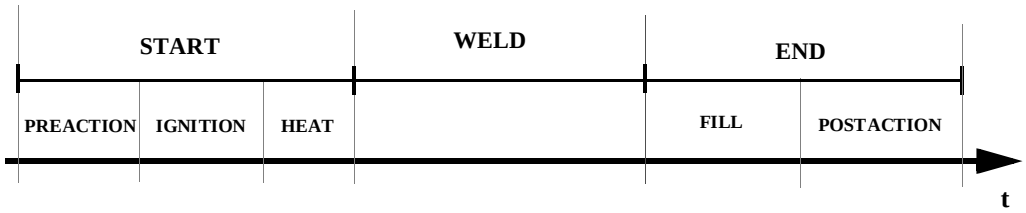
The same instructions are used both to control the robot's movements and the actual welding process. The arc welding instructions indicate which weld data and seam data are used in the weld in question.

The weld settings for the actual weld phase are defined in weld data. The start and end phase are defined in seam data.

Any weaving is defined in weave data, which is also identified by the arc welding instruction.

Certain functions, such as a scraping start, are defined in the system parameters.

The welding process is divided into the following phases:



13.2 Arc welding instructions

Instruction	Type of movement:
ArcC	TCP along a circular path
ArcL	TCP along a linear path

13.3 Arc welding data

Data type	Used to define:
welddata	The weld phase
seamdata	The start and end phase of a weld
weavedata	The weaving characteristics
trackdata	Seamtracking via the serial sensor interface

14 GlueWare

The GlueWare package provides support for gluing applications that are equipped with one or two gluing guns.

The GlueWare application provides fast and accurate positioning combined with gun manipulation, process start and stop.

Communication with the glueing equipment is carried out by means of digital and analog outputs.

14.1 Glueing features

The GlueWare package contains the following features:

- Fast and accurate positioning
- Handling of on/off guns as well as proportional guns
- Two different guns can be handled in the same program, each gun controlled by one digital signal (on/off) and two analog signals (flows)
- Gun pre-opening and pre-closing respectively
- Simulated glueing

14.2 Programming principles

Both the robot's movement and the glue process control are embedded in one instruction, *GlueL* and *GlueC* respectively.

The glueing process is specified by:

- Gundata: glue gun data. See *Data types - ggundata*.
- The system module GLUSER: RAPID routines and global data for customizing purposes. See *Predefined Data and Programs - System Module GLUSER.*
- System parameters: the I/O configuration. See *System Parameters - Glueing*

14.3 Glue instructions

Instruction	Used to:
GlueL	Move the TCP along a linear path and perform glueing with the given data
GlueC	Move the TCP along a circular path and perform glueing with the given data

14.4 Glue data

Data type	Used to define:
ggundata	The used glue gun

15 External Computer Communication

The robot can be controlled from a superordinate computer. In this case, a special communications protocol is used to transfer information.

15.1 Programming principles

As a common communications protocol is used to transfer information from the robot to the computer and vice versa, the robot and computer can understand each other and no programming is required. The computer can, for example, change values in the program's data without any programming having to be carried out (except for defining this data). Programming is only necessary when program-controlled information has to be sent from the robot to the superordinate computer.

15.2 Sending a program-controlled message from the robot to a computer

Instruction	Used to:
SCWrite[1]	Send a message to the superordinate computer

1. Only if the robot is equipped with the option "RAP Serial Link".

16 Service Instructions

A number of instructions are available to test the robot system. See the chapter on Troubleshooting Tools in the Product Manual for more information.

16.1 Directing a value to the robot's test signal

A reference signal, such as the speed of a motor, can be directed to an analog output signal located on the backplane of the robot.

Instruction	Used to:
TestSign	Define and activate a test signal

Data type	Used to define:
testsignal	The type of test signal

17 String Functions

String functions are used for operations with strings such as copying, concatenation, comparison, searching, conversion, etc.

17.1 Basic Operations

Data type	Used to define:
string	String. Predefined constants STR_DIGIT, STR_UPPER, STR_LOWER and STR_WHITE
Instruction/Operator	**Used to:**
:=	Assign a value (copy of string)
+	String concatenation
Function	**Used to:**
StrLen	Find string length
StrPart	Obtain part of a string

17.2 Comparison and Searching

Operator	Used to:
=	Test if equal to
<>	Test if not equal to
Function	**Used to:**
StrMemb	Check if character belongs to a set
StrFind	Search for character in a string
StrMatch	Search for pattern in a string
StrOrder	Check if strings are in order

17.3 Conversion

Function	Used to:
NumToStr	Convert a numeric value to a string
ValToStr	Convert a value to a string
StrToVal	Convert a string to a value
StrMap	Map a string

18 Syntax Summary

18.1 Instructions

Data **:=** Value

AccSet Acc Ramp

ActUnit MecUnit

Add Name AddValue

Break

CallBy Var Name Number

Clear Name

ClkReset Clock

ClkStart Clock

ClkStop Clock

Close IODevice

! Comment

ConfJ [\On] | [\Off]

ConfL [\On] | [\Off]

CONNECT Interrupt **WITH** Trap routine

DeactUnit MecUnit

Decr Name

EOffsSet EAxOffs

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

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

GOTO Label

GripLoad Load

IDelete Interrupt

IF Condition **...**

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

ENDIF

Incr Name

IndAMove MecUnit Axis [\ToAbsPos] | [\ToAbsNum] Speed [\Ramp]

IndCMove MecUnit Axis Speed [\Ramp]

IndDMove MecUnit Axis Delta Speed [\Ramp]

IndReset MecUnit Axis [\RefPos] | [\RefNum] | [\Short] | [\Fwd] | [\Bwd] | [\Old]

IndRMove MecUnit Axis [\ToRelPos] | [\ToRelNum] | [\Short] | [\Fwd] | [\Bwd] Speed [\Ramp]

InvertDO Signal

ISignalDI [\Single] Signal TriggValue Interrupt

ISignalDO [\Single] Signal TriggValue Interrupt

ISleep Interrupt

ITimer [\Single] Time Interrupt

IWatch Interrupt

Label**:**

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

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

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

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

Open Object [\File] IODevice [\Read] | [\Write] | [\Append] | [\Bin]

PDispOn [\Rot] [\ExeP] ProgPoint Tool [\WObj]

PDispSet DispFrame

Procedure { Argument }

PulseDO [\PLength] Signal

RAISE [Error no]

Reset Signal

RETURN [Return value]

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

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

Set Signal

SetAO Signal Value

SetDO [\SDelay] Signal Value

SetGO Signal Value

SingArea [\Wrist] | [\Arm] | [\Off]

SoftAct Axis Softness [\Ramp]

Stop [\NoRegain]

TEST Test data **{CASE** Test value {**, Test value**} **:** ...}
[**DEFAULT:** ...] **ENDTEST**

TPReadFK Answer String FK1 FK2 FK3 FK4 FK5 [\MaxTime] [\DIBreak] [\BreakFlag]

TPReadNum Answer String [\MaxTime] [\DIBreak] [\BreakFlag]

TPWrite String [\Num] | [\Bool] | [\Pos] | [\Orient]

TriggC CirPoint ToPoint Speed [\T] Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj]

TriggInt TriggData Distance [\Start] | [\Time] Interrupt

TriggIO TriggData Distance [\Start] | [\Time] [\DOp] | [\GOp] | [\AOp] SetValue [\DODelay] | [\AORamp]

TriggJ ToPoint Speed [\T] Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj]

TriggL ToPoint Speed [\T] Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj]

TuneServo MecUnit Axis TuneValue

TuneServo MecUnit Axis TuneValue [\Type]

UnLoad FilePath [\File]

VelSet Override Max

WaitDI Signal Value [\MaxTime] [\TimeFlag]

WaitDO Signal Value [\MaxTime] [\TimeFlag]

WaitTime [\InPos] Time

WaitUntil [\InPos] Cond [\MaxTime] [\TimeFlag]

WHILE Condition DO ... ENDWHILE

Write IODevice String [\Num] | [\Bool] | [\Pos] | [\Orient] [\NoNewLine]

WriteBin IODevice Buffer NChar

18.2 Functions

Abs (Input)

ACos (Value)

ASin (Value)

ATan (Value)

ATan2 (Y X)

ClkRead (Clock)

Cos (Angle)

CPos ([Tool] [\WObj])

CRobT ([Tool] [\WObj])

DefDFrame (OldP1 OldP2 OldP3 NewP1 NewP2 NewP3)

DefFrame (NewP1 NewP2 NewP3 [\Origin])

Dim (ArrPar DimNo)

DOutput (Signal)

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

Exp (Exponent)

GOutput (Signal)

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

IndInpos MecUnit Axis

IndSpeed MecUnit Axis [\InSpeed] | [\ZeroSpeed]

IsPers (DatObj)

IsVar (DatObj)

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

NumToStr (Val Dec [\Exp])

Offs (Point XOffset YOffset ZOffset)

OrientZYX (ZAngle YAngle XAngle)

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

PoseInv (Pose)

PoseMult (Pose1 Pose2)

PoseVect (Pose Pos)

Pow (Base Exponent)

Present (OptPar)

ReadBin (IODevice [\Time])

ReadMotor [\MecUnit] Axis

ReadNum (IODevice [\Time])

ReadStr (IODevice [\Time])

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

Round (Val [\Dec])

Sin (Angle)

Sqrt (Value)

StrFind (Str ChPos Set [\NotInSet])

StrLen (Str)

StrMap (Str FromMap ToMap)

StrMatch (Str ChPos Pattern)

StrMemb (Str ChPos Set)

StrOrder (Str1 Str2 Order)

StrPart (Str ChPos Len)

StrToVal (Str Val)

Tan (Angle)

TestDI (Signal)

Trunc (Val [\Dec])

ValToStr (Val)

CONTENTS

Page

Page

1 Basic Elements

1.1 Identifiers

Identifiers are used to name modules, routines, data and labels;

e.g.
```
MODULE module_name
PROC routine_name()
VAR pos data_name;
label_name:
```

The first character in an identifier must be a letter. The other characters can be letters, digits or underscores "_".

The maximum length of any identifier is 16 characters, each of these characters being significant. Identifiers that are the same except that they are typed in the upper case, and vice versa, are considered the same.

Reserved words

The words listed below are reserved. They have a special meaning in the RAPID language and thus must not be used as identifiers.

There are also a number of predefined names for data types, system data, instructions, and functions, that must not be used as identifiers. See Chapters 7, 8, 9, 10 ,13, 14 and 15 in this manual.

AND	BACKWARD	CASE	CONNECT
CONST	DEFAULT	DIV	DO
ELSE	ELSEIF	ENDFOR	ENDFUNC
ENDIF	ENDMODULE	ENDPROC	ENDTEST
ENDTRAP	ENDWHILE	ERROR	EXIT
FALSE	FOR	FROM	FUNC
GOTO	IF	INOUT	LOCAL
MOD	MODULE	NOSTEPIN	NOT
NOVIEW	OR	PERS	PROC
RAISE	READONLY	RETRY	RETURN
STEP	SYSMODULE	TEST	THEN
TO	TRAP	TRUE	TRYNEXT
VAR	VIEWONLY	WHILE	WITH
XOR			

1.2 Spaces and new-line characters

The RAPID programming language is a free format language, meaning that spaces can be used anywhere except for in:

- identifiers
- reserved words
- numerical values
- placeholders.

New-line, tab and form-feed characters can be used wherever a space can be used, except for within comments.

Identifiers, reserved words and numeric values must be separated from one another by a space, a new-line, tab or form-feed character.

Unnecessary spaces and new-line characters will automatically be deleted from a program loaded into the program memory. Consequently, programs loaded from diskette and then stored again might not be identical.

1.3 Numeric values

A numeric value can be expressed as

- an integer, e.g. 3, -100, 3E2
- a decimal number, e.g. 3.5, -0.345, -245E-2

The value must be in the range specified by the ANSI IEEE 754-1985 standard (single precision) float format.

1.4 Logical values

A logical value can be expressed as TRUE or FALSE.

1.5 String values

A string value is a sequence of characters (ISO 8859-1) and control characters (non-ISO 8859-1 characters in the numeric code range 0-255). Character codes can be included, making it possible to include non-printable characters (binary data) in the string as well.

Example:
```
"This is a string"
"This string ends with the BEL control character \07"
```

If a backslash (which indicates character code) or double quote character is included, it must be written twice.

Example:
```
"This string contains a "" character"
"This string contains a \\ character"
```

1.6 Comments

Comments are used to make the program easier to understand. They do not affect the meaning of the program in any way.

A comment starts with an exclamation mark "!" and ends with a new-line character. It occupies an entire line and cannot occur between two modules;

e.g.
```
! comment
IF reg1 > 5 THEN
    ! comment
    reg2 := 0;
ENDIF
```

1.7 Placeholders

Placeholders can be used to temporarily represent parts of a program that are "not yet defined". A program that contains placeholders is syntactically correct and may be loaded into the program memory.

Placeholder	Represents:
<DDN>	data declaration
<RDN>	routine declaration
<PAR>	formal optional alternative parameter
<ALT>	optional formal parameter
<DIM>	formal (conformant) array dimension
<SMT>	instruction
<VAR>	data object (variable, persistent or parameter) reference
<EIT>	else if clause of if instruction
<CSE>	case clause of test instruction
<EXP>	expression
<ARG>	procedure call argument
<ID>	identifier

1.8 File header

A program file starts with the following file header:

```
%%%
 VERSION:1                    (Program version M94 or M94A)
 LANGUAGE:ENGLISH             (or some other language:
%%%                            GERMAN or FRENCH)
```

1.9 Syntax

Identifiers

<identifier> ::=
<ident>
| **<ID>**
<ident> ::= <letter> {<letter> | <digit> | '_'}

Numeric values

<num literal> ::=
<integer> [<exponent>]
| <integer> '.' [<integer>] [<exponent>]
| [<integer>] '.' <integer> [<exponent>]
<integer> ::= <digit> {<digit>}
<exponent> ::= ('E' | 'e') ['+' | '-'] <integer>

Logical values

<bool literal> ::= **TRUE** | **FALSE**

String values

<string literal> ::= '"' {<character> | <character code> } '"'
<character code> ::= '\' <hex digit> <hex digit>
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f

Comments

<comment> ::=
'!' {<character> | <tab>} <newline>

Characters

<character> ::= -- *ISO 8859-1* --
<newline> ::= -- *newline control character* --
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::=
<upper case letter>
| <lower case letter>

<upper case letter> ::=

A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S | T
| U | V | W | X | Y | Z | À | Á | Â | Ã
| Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í
| Î | Ï | [1] | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
| Ù | Ú | Û | Ü | [2] | [3] | ß

<lower case letter> ::=

a | b | c | d | e | f | g | h | i | j
| k | l | m | n | o | p | q | r | s | t
| u | v | w | x | y | z | ß | à | á | â
| ã| ä | å | æ | ç | è | é | ê | ë | ì
| í | î | ï | [1] | ñ | ò | ó | ô | õ | ö
| ø | ù | ú | û | ü | [2] | [3] | ÿ

1) Icelandic letter eth.
2) Letter Y with acute accent.
3) Icelandic letter thorn.

2 Modules

The program is divided into *program* and *system modules*. The program can also be divided into *modules* (see Figure 1).

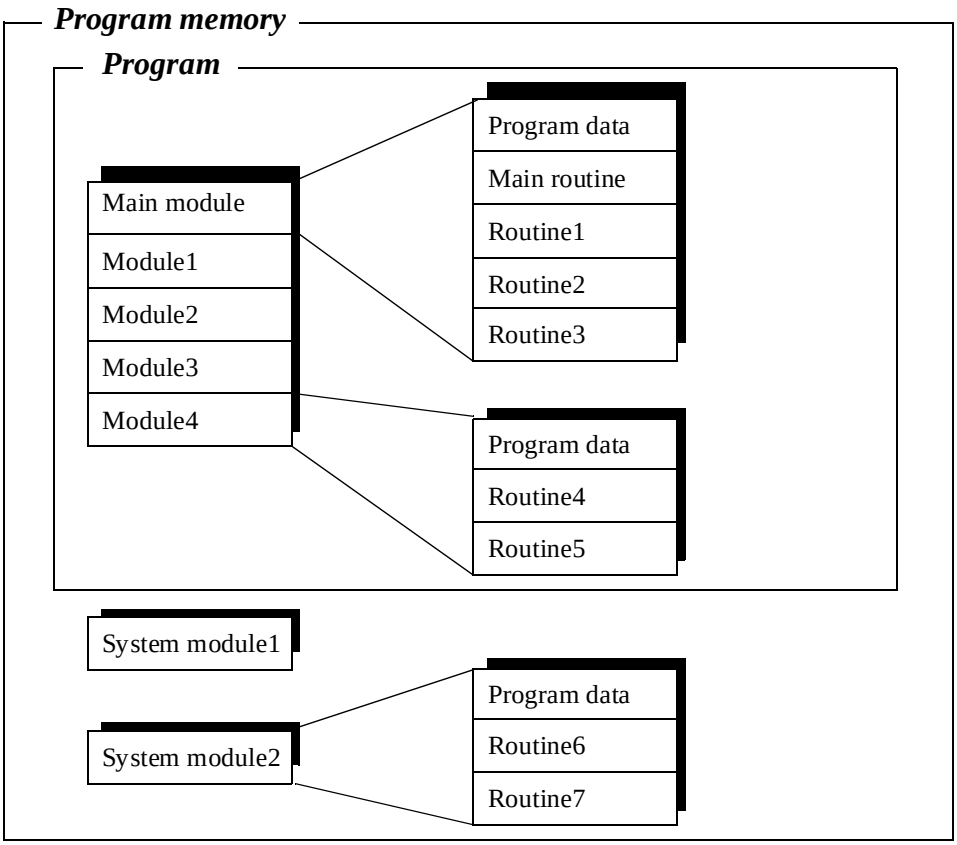


Figure 1 The program can be divided into modules.

2.1 Program modules

A program module can consist of different data and routines. Each module, or the whole program, can be copied to diskette, RAM disk, etc., and vice versa.

One of the modules contains the entry procedure, a global procedure called *main*. Executing the program means, in actual fact, executing the main procedure. The program can include many modules, but only one of these will have a main procedure.

A module may, for example, define the interface with external equipment or contain geometrical data that is either generated from CAD systems or created on-line by digitizing (teach programming).

Whereas small installations are often contained in one module, larger installations may have a main module that references routines and/or data contained in one or several other modules.

2.2 System modules

System modules are used to define common, system-specific data and routines, such as tools. They are not included when a program is saved, meaning that any update made to a system module will affect all existing programs currently in, or loaded at a later stage into the program memory.

2.3 Module declarations

A module declaration specifies the name and attributes of that module. These attributes can only be added off-line, not using the teach pendant. The following are examples of the attributes of a module:

Attribute	If specified, the module:
SYSMODULE	is a system module, otherwise a program module
NOSTEPIN	cannot be entered during stepwise execution
VIEWONLY	cannot be modified
READONLY	cannot be modified, but the attribute can be removed
NOVIEW	cannot be viewed, only executed. Global routines can be reached from other modules and are always run as NOSTEPIN. The current values for global data can be reached from other modules or from the data window on the teach pendant. A module or a program containing a NOVIEW program module cannot be saved. Therefore, NOVIEW should primarily be used for system modules. NOVIEW can only be defined off-line from a PC.

e.g.
```
MODULE module_name (SYSMODULE, VIEWONLY)
    !data declarations
    !routine declarations
ENDMODULE
```

A module may not have the same name as another module or a global routine or data.

2.4 Syntax

Module declaration

<module declaration> ::=
MODULE <module name> [<module attribute list>]
<data declaration list>
<routine declaration list>
ENDMODULE
<module name> ::= <identifier>
<module attribute list> ::= ‘(‘ <module attribute> { ‘,’ <module attribute> } ‘)’

<module attribute> ::=
SYSMODULE
| **NOVIEW**
| **NOSTEPIN**
| **VIEWONLY**
| **READONLY**

(***Note.*** If two or more attributes are used they must be in the above order, the NOVIEW attribute can only be specified alone or together with the attribute SYSMODULE.)

<data declaration list> ::= { <data declaration> }
<routine declaration list> ::= { <routine declaration> }

3 Routines

There are three types of routines (subprograms): *procedures*, *functions* and *traps*.

- Procedures do not return a value and are used in the context of instructions.
- Functions return a value of a specific type and are used in the context of expressions.
- Trap routines provide a means of dealing with interrupts. A trap routine can be associated with a specific interrupt and then, if that particular interrupt occurs at a later stage, will automatically be executed. A trap routine can never be explicitly called from the program.

3.1 Routine scope

The scope of a routine denotes the area in which the routine is visible. The optional local directive of a routine declaration classifies a routine as local (within the module), otherwise it is global.

Example:
```
LOCAL PROC local_routine (...
PROC global_routine (...
```

The following scope rules apply to routines (see the example in Figure 2):

- The scope of a global routine may include any module.
- The scope of a local routine comprises the module in which it is contained.
- Within its scope, a local routine hides any global routine or data with the same name.
- Within its scope, a routine hides instructions and predefined routines and data with the same name.

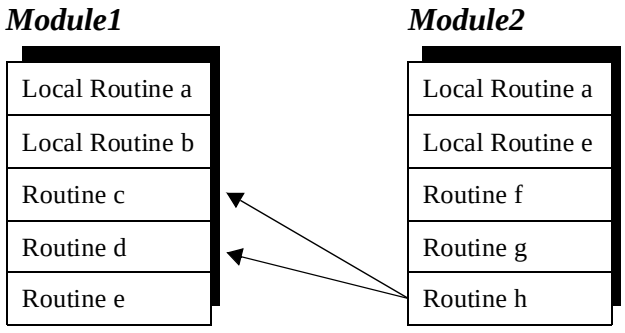


Figure 2 Example: The following routines can be called from Routine h:
Module1 - Routine c, d.
Module2 - All routines.

A routine may not have the same name as another routine or data in the same module. A global routine may not have the same name as a module or a global routine or global data in another module.

3.2 Parameters

The parameter list of a routine declaration specifies the arguments (actual parameters) that must/can be supplied when the routine is called.

There are four different types of parameters (in the access mode):

- Normally, a parameter is used only as an input and is treated as a routine variable. Changing this variable will not change the corresponding argument.
- An INOUT parameter specifies that a corresponding argument must be a variable (entire, element or component) or an entire persistent which can be changed by the routine.
- A VAR parameter specifies that a corresponding argument must be a variable (entire, element or component) which can be changed by the routine.
- A PERS parameter specifies that a corresponding argument must be an entire persistent which can be changed by the routine.

If an INOUT, VAR or PERS parameter is updated, this means, in actual fact, that the argument itself is updated, i.e. it makes it possible to use arguments to return values to the calling routine.

Example:
```
PROC routine1 (num in_par, INOUT num inout_par,
               VAR num var_par, PERS num pers_par)
```

A parameter can be optional and may be omitted from the argument list of a routine call. An optional parameter is denoted by a backslash "\" before the parameter.

Example:
```
PROC routine2 (num required_par \num optional_par)
```

The value of an optional parameter that is omitted in a routine call may not be referenced. This means that routine calls must be checked for optional parameters before an optional parameter is used.

Two or more optional parameters may be mutually exclusive (i.e. declared to exclude each other), which means that only one of them may be present in a routine call. This is indicated by a stroke "|" between the parameters in question.

Example:
```
PROC routine3 (\num exclude1 | num exclude2)
```

The special type, *switch*, may (only) be assigned to optional parameters and provides a means to use switch arguments, i.e. arguments that are only specified by names (not values). A value cannot be transferred to a switch parameter. The only way to use a switch parameter is to check for its presence using the predefined function, *Present*.

Example:
```
PROC routine4 (\switch on | switch off)
    ...
    IF Present (off ) THEN
    ...
ENDPROC
```

Arrays may be passed as arguments. The degree of an array argument must comply

with the degree of the corresponding formal parameter. The dimension of an array parameter is “conformant” (marked with “*”). The actual dimension thus depends on the dimension of the corresponding argument in a routine call. A routine can determine the actual dimension of a parameter using the predefined function, *Dim*.

Example: PROC routine5 (VAR num pallet{*,*})

3.3 Routine termination

The execution of a procedure is either explicitly terminated by a RETURN instruction or implicitly terminated when the end (ENDPROC, BACKWARD or ERROR) of the procedure is reached.

The evaluation of a function must be terminated by a RETURN instruction.

The execution of a trap routine is explicitly terminated using the RETURN instruction or implicitly terminated when the end (ENDTRAP or ERROR) of that trap routine is reached. Execution continues from the point where the interrupt occurred.

3.4 Routine declarations

A routine can contain routine declarations (including parameters), data, a body, a backward handler (only procedures) and an error handler (see Figure 3). Routine declarations cannot be nested, i.e. it is not possible to declare a routine within a routine.

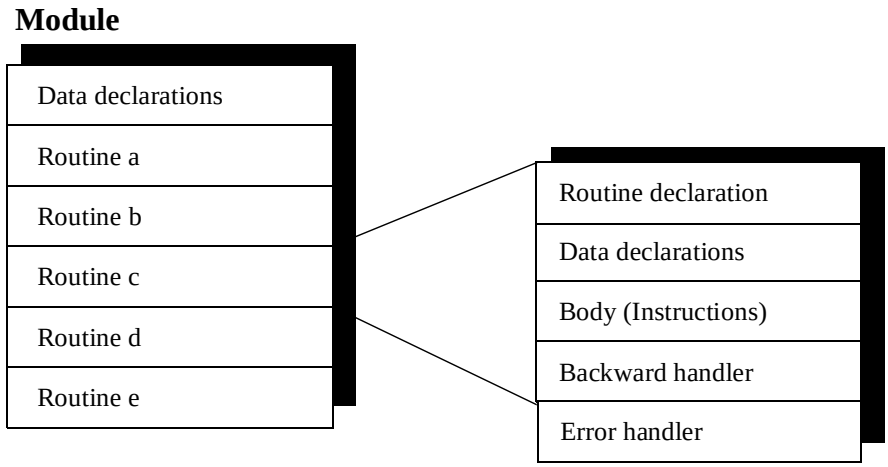


Figure 3 A routine can contain declarations, data, a body, a backward handler and an error handler.

Procedure declaration

Example: Multiply all elements in a num array by a factor;

```
PROC arrmul( VAR num array{*}, num factor)
   FOR index FROM 1 TO dim( array, 1 ) DO
      array{index} := array{index} * factor;
   ENDFOR
ENDPROC
```

Function declaration

A function can return any data type value, but not an array value.

Example: Return the length of a vector;

```
FUNC num veclen (pos vector)
 RETURN
Sqrt(Pow(vector.x,2)+Pow(vector.y,2)+Pow(vector.z,2));
ENDFUNC
```

Trap declaration

Example: Respond to feeder empty interrupt;

```
TRAP feeder_empty
  wait_feeder;
  RETURN;
ENDTRAP
```

3.5 Procedure call

When a procedure is called, the arguments that correspond to the parameters of the procedure shall be used:

- Mandatory parameters must be specified. They must also be specified in the correct order.
- Optional arguments can be omitted.
- Conditional arguments can be used to transfer parameters from one routine call to another.

See the Chapter *Using function calls in expressions* on page 29 for more details.

The procedure name may either be statically specified by using an identifier (*early binding*) or evaluated during runtime from a string type expression (*late binding*). Even though early binding should be considered to be the “normal” procedure call form, late binding sometimes provides very efficient and compact code. Late binding is defined by putting percent signs before and after the string that denotes the name of the procedure.

Example:
```
! early binding
test products_id
case 1:
    proc1 x, y, z;
case 2:
    proc2 x, y, z;
case 3:
    ...
! same example using binding
% “proc” + NumToStr(product_id, 0) % x, y, z;
    ...
! same example again using another variant of late binding
VAR string procname {3} :=[“proc1”, “proc2”, “proc3”];
    ...
% procname{product_id} % x, y, z;
    ...
```

Note that the late binding is available for procedure calls only, and not for function calls.

3.6 Syntax

Routine declaration

```
<routine declaration> ::=
        [LOCAL] ( <procedure declaration>
                | <function declaration>
                | <trap declaration> )
        | <comment>
        | <RDN>
```

Parameters

```
<parameter list> ::=
        <first parameter declaration> { <next parameter declaration> }
<first parameter declaration> ::=
        <parameter declaration>
        | <optional parameter declaration>
        | <PAR>
<next parameter declaration> ::=
        ’,’ <parameter declaration>
         | <optional parameter declaration>
         | ’,’ <PAR>
<optional parameter declaration> ::=
        ’\’ ( <parameter declaration> | <ALT> )
            { ’|’ ( <parameter declaration> | <ALT> ) }
```

<parameter declaration> ::=
[**VAR** | **PERS** | **INOUT**] <data type>
<identifier> [’{’ (’*’ { ’,’ ’*’ }) | **<DIM>**] ’}’
| ’switch’ <identifier>

Procedure declaration

<procedure declaration> ::=
PROC <procedure name>
’(’ [<parameter list>] ’)’
<data declaration list>
<instruction list>
[**BACKWARD** <instruction list>]
[**ERROR** <instruction list>]
ENDPROC
<procedure name> ::= <identifier>
<data declaration list> ::= {<data declaration>}

Function declaration

<function declaration> ::=
FUNC <value data type>
<function name>
’(’ [<parameter list>] ’)’
<data declaration list>
<instruction list>
[**ERROR** <instruction list>]
ENDFUNC
<function name> ::= <identifier>

Trap routine declaration

<trap declaration> ::=
TRAP <trap name>
<data declaration list>
<instruction list>
[**ERROR** <instruction list>]
ENDTRAP
<trap name> ::= <identifier>

Procedure call

<procedure call> ::= <procedure> [<procedure argument list>] ’;’
<procedure> ::=
<identifier>
| ’%’ <expression> ’%’
<procedure argument list> ::= <first procedure argument> { <procedure argument> }

<first procedure argument> ::=
<required procedure argument>
| <optional procedure argument>
| <conditional procedure argument>
| **<ARG>**
<procedure argument> ::=
’,’ <required procedure argument>
| <optional procedure argument>
| <conditional procedure argument>
| ’,’ **<ARG>**
<required procedure argument> ::= [<identifier> ’:=’] <expression>
<optional procedure argument> ::= ’\’ <identifier> [’:=’ <expression>]
<conditional procedure argument> ::= ’\’ <identifier> ’?’ (<parameter> | **<VAR>**)

4 Data Types

There are two different kinds of data types:

- An *atomic* type is atomic in the sense that it is not defined based on any other type and cannot be divided into parts or components, e.g. *num*.
- A *record* data type is a composite type with named, ordered components, e.g. *pos*. A component may be of an atomic or record type.

A record value can be expressed using an *aggregate* representation;

e.g. [300, 500, depth] pos record aggregate value.

A specific component of a record data can be accessed by using the name of that component;

e.g. pos1.x := 300; assignment of the x-component of pos1.

4.1 Non-value data types

Each available data type is either a *value* data type or a *non-value* data type. Simply speaking, a value data type represents some form of “value”. Non-value data cannot be used in value-oriented operations:

- Initialisation
- Assignment (:=)
- Equal to (=) and not equal to (<>) checks
- TEST instructions
- IN (access mode) parameters in routine calls
- Function (return) data types

The input data types (*signalai, signaldi, signalgi*) are of the data type *semi value*. These data can be used in value-oriented operations, except initialisation and assignment.

In the description of a data type it is only specified when it is a semi value or a non-value data type.

4.2 Equal (alias) data types

An *alias* data type is defined as being equal to another type. Data with the same data types can be substituted for one another.

Example:
```
VAR dionum high:=1;
VAR num level;          This is OK since dionum is an alias
level:= high;           data type for num
```

4.3 Syntax

<data type> ::= <identifier>

5 Data

There are three kinds of data: *variables, persistents* and *constants*.

- A variable can be assigned a new value during program execution.
- A persistent can be described as a "persistent" variable. This is accomplished by letting an update of the value of a persistent automatically cause the initialisation value of the persistent declaration to be updated. (When a program is saved the initialisation value of any persistent declaration reflects the current value of the persistent.)
- A constant represents a static value and cannot be assigned a new value.

A data declaration introduces data by associating a name (identifier) with a data type. Except for predefined data and loop variables, all data used must be declared.

5.1 Data scope

The scope of data denotes the area in which the data is visible. The optional local directive of a data declaration classifies data as local (within the module), otherwise it is global. Note that the local directive may only be used at module level, not inside a routine.

Example:
```
LOCAL VAR num local_variable;
VAR num global_variable;
```

Data declared outside a routine is called *program data*. The following scope rules apply to program data:

- The scope of predefined or global program data may include any module.
- The scope of local program data comprises the module in which it is contained.
- Within its scope, local program data hides any global data or routine with the same name (including instructions and predefined routines and data).

Program data may not have the same name as other data or a routine in the same module. Global program data may not have the same name as other global data or a routine in another module. A persistent may not have the same name as another persistent in the same program.

Data declared inside a routine is called *routine data*. Note that the parameters of a routine are also handled as routine data. The following scope rules apply to routine data:

- The scope of routine data comprises the routine in which it is contained.
- Within its scope, routine data hides any other routine or data with the same name.

See the example in Figure 4.

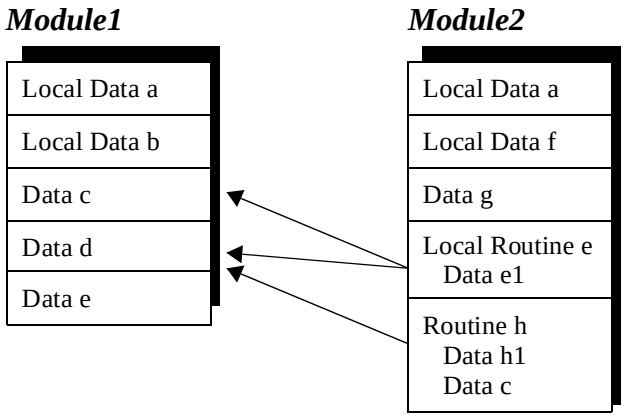


Figure 4 Example: The following data can be called from routine e:
Module1: Data c, d.
Module2: Data a, f, g, e1.
The following data can be called from routine h:
Module1: Data d.
Module2: Data a, f, g, h1, c.

Routine data may not have the same name as other data or a label in the same routine.

5.2 Variable declaration

A variable is introduced by a variable declaration.

Example: VAR num x;

Variables of any type can be given an array (of degree 1, 2 or 3) format by adding dimensional information to the declaration. A dimension is an integer value greater than 0.

Example: VAR pos pallet{14, 18};

Variables with value types may be initialised (given an initial value). The expression used to initialise a program variable must be constant. Note that the value of an uninitialized variable may be used, but it is undefined, i.e. set to zero.

Example:
```
VAR string author_name := "John Smith";
VAR pos start := [100, 100, 50];
VAR num maxno{10} := [1, 2, 3, 9, 8, 7, 6, 5, 4, 3];
```

The initialisation value is set when:

- the program is opened,
- the program is executed from the beginning of the program.

5.3 Persistent declaration

Persistents can only be declared at module level, not inside a routine, and must be given an initial value. The initialisation value must be a single value (without data or operands), or a single aggregate with members which, in turn, are single values or single aggregates.

Example: PERS pos refpnt := [100.23, 778.55, 1183.98];

Persistents of any type can be given an array (of degree 1, 2 or 3) format by adding dimensional information to the declaration. A dimension is an integer value greater than 0.

Example: PERS pos pallet{14, 18};

Note that if the value of a persistent is updated, this automatically causes the initialisation value of the persistent declaration to be updated.

Example:
```
PERS num reg1 := 0;
  ...
reg1 := 5;
```
After execution, the program looks like this:
```
PERS num reg1 := 5;
  ...
reg1 := 5;
```

5.4 Constant declaration

A constant is introduced by a constant declaration. The value of a constant cannot be modified.

Example: CONST num pi := 3.141592654;

A constant of any type can be given an array (of degree 1, 2 or 3) format by adding dimensional information to the declaration. A dimension is an integer value greater than 0.

Example:
```
CONST pos seq{3} := [[614, 778, 1020],
                    [914, 998, 1021],
                    [814, 998, 1022]];
```

5.5 Initiating data

In the table below, you can what is happening in various activities, e.g. warm and cold start.

Table1

System event / Affects	Power on (Warm start)	Open program	Close or New program	Start program (Move PP to main)	Start program (Move PP to Routine)	Start program (After cycle)	Start program (After stop)	Start program (Move PP to cursor)
Constant	Unchanged	Init	Unchanged	Init	Init	Unchanged	Unchanged	Unchanged
Variable	Unchanged	Init	Unchanged	Init	Init	Unchanged	Unchanged	Unchanged
Persistent	Unchanged	Init	Unchanged	Init	Init	Unchanged	Unchanged	Unchanged
Com-manded interrupts	Re-ordered	Disappears	Disappears	Unchanged	Unchanged	Unchanged	Unchanged	Unchanged
Start up routine Sys_reset (with motn ??? set-tings)	Not run	Run*	Run	Run	Not run	Not run	Not run	Not run
Files	Unchanged	Closes	Closes	Closes	Closes	Unchanged	Unchanged	Unchanged
Path	Recreated after power off	Disappears	Disappears	Disappears	Disappears	Unchanged	Unchanged	Disappears

* Not run when memory is empty, only when a program must first be deleted.

5.6 Syntax

Data declaration

<data declaration> ::=
[**LOCAL**] (<variable declaration>
| <persistent declaration>
| <constant declaration>)
| **<comment>**
| **<DDN>**

Variable declaration

<variable declaration> ::=
VAR <data type> <variable definition> ';'
<variable definition> ::=
<identifier> ['{' <dim> { ',' <dim> } '}']
[':=' <constant expression>]
<dim> ::= <constant expression>

Persistent declaration

<persistent declaration> ::=
PERS <data type> <persistent definition> ';'

<persistent definition> ::=
<identifier> [’{’ <dim> { ’,’ <dim> } ’}’]
’:=’ <literal expression>

Constant declaration

<constant declaration> ::=
CONST <data type> <constant definition> ’;’
<constant definition> ::=
<identifier> [’{’ <dim> { ’,’ <dim> } ’}’]
’:=’ <constant expression>
<dim> ::= <constant expression>

6 Instructions

Instructions are executed in succession unless a program flow instruction or an interrupt or error causes the execution to continue at some other place.

Most instructions are terminated by a semicolon ";". A label is terminated by a colon ":". Some instructions may contain other instructions and are terminated by specific keywords:

Instruction	Termination word
IF	ENDIF
FOR	ENDFOR
WHILE	ENDWHILE
TEST	ENDTEST

Example:
```
WHILE index < 100 DO
    .
    index := index + 1;
ENDWHILE
```

6.1 Syntax

<instruction list> ::= { <instruction> }
<instruction> ::=
[<instruction according to separate chapter in this manual>
| **<SMT>**

7 Expressions

An expression specifies the evaluation of a value. It can be used, for example:

- in an assignment instruction e.g. a:=*3*b/c*;
- as a condition in an IF instruction e.g. IF *a>=3* THEN ...
- as an argument in an instruction e.g. WaitTime *time*;
- as an argument in a function call e.g. a:=Abs*(3*b)*;

7.1 Arithmetic expressions

An arithmetic expression is used to evaluate a numeric value.

Example: 2*pi*radius

Table 2 shows the different types of operations possible.

Table 2

Operator	Operation	Operand types	Result type
+	addition	num + num	num[3)]
+	unary plus; keep sign	+num or +pos	same[1)3)]
+	vector addition	pos + pos	pos
-	subtraction	num - num	num[3)]
-	unary minus; change sign	-num or -pos	same[1)3)]
-	vector subtraction	pos - pos	pos
*	multiplication	num * num	num[3)]
*	scalar vector multiplication	num * pos or pos * num	pos
*	vector product	pos * pos	pos
*	linking of rotations	orient * orient	orient
/	division	num / num	num
DIV [2)]	integer division	num DIV num	num
MOD [2)]	integer modulo; remainder	num MOD num	num

1. The result receives the same type as the operand. If the operand has an alias data type, the result receives the alias "base" type (num or pos).

2. Integer operations, e.g. 14 DIV 4=3, 14 MOD 4=2.
(Non-integer operands are illegal.)

3. Preserves integer (exact) representation as long as operands and result are kept within the integer subdomain of the num type.

7.2 Logical expressions

A logical expression is used to evaluate a logical value (TRUE/FALSE).

Example: a>5 AND b=3

Table 3 shows the different types of operations possible.

Table 3

Operator	Operation	Operand types	Result type
<	less than	num < num	bool
<=	less than or equal to	num <= num	bool
=	equal to	any 1)= any 1)	bool
>=	greater than or equal to	num >= num	bool
>	greater than	num > num	bool
<>	not equal to	any 1) <> any 1)	bool
AND	and	bool AND bool	bool
XOR	exclusive or	bool XOR bool	bool
OR	or	bool OR bool	bool
NOT	unary not; negation	NOT bool	bool

1) Only value data types. Operands must have equal types.

a AND b

b \ a	True	False
True	True	False
False	False	False

a XOR b

b \ a	True	False
True	False	True
False	True	False

a OR b

b \ a	True	False
True	True	True
False	True	False

NOT b

b	
True	False
False	True

7.3 String expressions

A string expression is used to carry out operations on strings.

Example: “IN” + “PUT” gives the result “INPUT”

Table 4 shows the one operation possible.

Table 4

Operator	Operation	Operand types	Result type
+	string concatenation	string + string	string

7.4 Using data in expressions

An entire variable, persistent or constant can be a part of an expression.

Example: 2*pi*radius

Arrays

A variable, persistent or constant declared as an array can be referenced to the whole array or a single element.

An array element is referenced using the index number of the element. The index is an integer value greater than 0 and may not violate the declared dimension. Index value 1 selects the first element. The number of elements in the index list must fit the declared degree (1, 2 or 3) of the array.

```
Example:    VAR num row{3};
            VAR num column{3};
            VAR num value;
               .
            value := column{3};      only one element in the array
            row := column;           all elements in the array
```

Records

A variable, persistent or constant declared as a record can be referenced to the whole record or a single component.

A record component is referenced using the component name.

```
Example:    VAR pos home;
            VAR pos pos1;
            VAR num yvalue;
            ..
            yvalue := home.y;        the Y component only
            pos1 := home;            the whole position
```

7.5 Using aggregates in expressions

An aggregate is used for record or array values.

Example:	pos := [x, y, 2*x];	pos record aggregate
	posarr := [[0, 0, 100], [0,0,z]];	pos array aggregate

It must be possible to determine the data type of an aggregate the context. The data type of each aggregate member must be equal to the type of the corresponding member of the determined type.

Example	**VAR** pos pl;	
	p1 :=[1, -100, 12];	aggregate type pos - determined by p1
	IF [1, -100, 12] = [a,b,b,] THEN	illegal since the data type of neither of the aggregates can be determined by the context.

7.6 Using function calls in expressions

A function call initiates the evaluation of a specific function and receives the value returned by the function.

Example: Sin(angle)

The arguments of a function call are used to transfer data to (and possibly from) the called function. The data type of an argument must be equal to the type of the corresponding parameter of the function. Optional arguments may be omitted but the order of the (present) arguments must be the same as the order of the formal parameters. In addition, two or more optional arguments may be declared to exclude each other, in which case, only one of them may be present in the argument list.

A required (compulsory) argument is separated from the preceding argument by a comma “,”. The formal parameter name may be included or omitted.

Example:	Polar(3.937, 0.785398)	two required arguments
	Polar(Dist:=3.937, Angle:=0.785398)...	using names

An optional argument must be preceded by a backslash “\” and the formal parameter name. A switch type argument is somewhat special; it may not include any argument expression. Instead, such an argument can only be either "present" or "not present".

Example:	Cosine(45)	one required argument
	Cosine(0.785398\Rad)	... and one switch
	Dist(p2)	one required argument
	Dist(\distance:=pos1, p2)	... and one optional

Conditional arguments are used to support smooth propagation of optional arguments through chains of routine calls. A conditional argument is considered to be “present” if the specified optional parameter (of the calling function) is present, otherwise it is

simply considered to be omitted. Note that the specified parameter must be optional.

Example:
```
PROC Read_from_file (iodev File \num Maxtime)
    ..
    character:=ReadBin (File \Time?Maxtime);
! Max. time is only used if specified when calling the routine
! Read_from_file
    ..
ENDPROC
```

The parameter list of a function assigns an *access mode* to each parameter. The access mode can be either *in*, *inout*, *var* or *pers:*

- An IN parameter (default) allows the argument to be any expression. The called function views the parameter as a constant.
- An INOUT parameter requires the corresponding argument to be a variable (entire, array element or record component) or an entire persistent. The called function gains full (read/write) access to the argument.
- A VAR parameter requires the corresponding argument to be a variable (entire, array element or record component). The called function gains full (read/write) access to the argument.
- A PERS parameter requires the corresponding argument to be an entire persistent. The called function gains full (read/update) access to the argument.

7.7 Priority between operators

The relative priority of the operators determines the order in which they are evaluated. Parentheses provide a means to override operator priority. The rules below imply the following operator priority:

* / DIV MOD	- highest
+ -	
< > <> <= >= =	
AND	
XOR OR NOT	- lowest

An operator with high priority is evaluated prior to an operator with low priority. Operators of the same priority are evaluated from left to right.

Example

Expression	Evaluation order	Comment
a + b + c	(a + b) + c	left to right rule
a + b * c	a + (b * c)	* higher than +
a OR b OR c	(a OR b) OR c	Left to right rule

a AND b OR c AND d	(a AND b) OR (c AND d)	AND higher than OR
a < b AND c < d	(a < b) AND (c < d)	< higher than AND

7.8 Syntax

Expressions

```
<expression> ::=
                <expr>
                | <EXP>
<expr> ::= [ NOT ] <logical term> { ( OR | XOR ) <logical term> }
<logical term> ::= <relation> { AND <relation> }
<relation> ::= <simple expr> [ <relop> <simple expr> ]
<simple expr> ::= [ <addop> ] <term> { <addop> <term> }
<term> ::= <primary> { <mulop> <primary> }
<primary> ::=
                <literal>
                | <variable>
                | <persistent>
                | <constant>
                | <parameter>
                | <function call>
                | <aggregate>
                | '(' <expr> ')'
```

Operators

```
<relop> ::= '<' | '<=' | '=' | '>' | '>=' | '<>'
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/' | DIV | MOD
```

Constant values

```
<literal> ::= <num literal>
                | <string literal>
                | <bool literal>
```

Data

```
<variable> ::=
                <entire variable>
                | <variable element>
                | <variable component>
<entire variable> ::= <ident>
<variable element> ::= <entire variable> '{' <index list> '}'
```

<index list> ::= <expr> { ’,’ <expr> }
<variable component> ::= <variable> ’.’ <component name>
<component name> ::= <ident>
<persistent> ::=
<entire persistent>
| <persistent element>
| <persistent component>
<constant> ::=
<entire constant>
| <constant element>
| <constant component>

Aggregates

<aggregate> ::= ’[’ <expr> { ’,’ <expr> } ’]’

Function calls

<function call> ::= <function> ’(’ [<function argument list>] ’)’
<function> ::= <ident>
<function argument list> ::= <first function argument> { <function argument> }
<first function argument> ::=
<required function argument>
| <optional function argument>
| <conditional function argument>
<function argument> ::=
’,’ <required function argument>
| <optional function argument>
| <conditional function argument>
<required function argument> ::= [<ident> ’:=’] <expr>
<optional function argument> ::= ’\’ <ident> [’:=’ <expr>]
<conditional function argument> ::= ’\’ <ident> ’?’ <parameter>

Special expressions

<constant expression> ::= <expression>
<literal expression> ::= <expression>
<conditional expression> ::= <expression>

Parameters

<parameter> ::=
<entire parameter>
| <parameter element>
| <parameter component>

8 Error Recovery

An execution error is an abnormal situation, related to the execution of a specific piece of a program. An error makes further execution impossible (or at least hazardous). “Overflow” and “division by zero” are examples of errors. Errors are identified by their unique *error number* and are always recognized by the robot. The occurrence of an error causes suspension of the normal program execution and the control is passed to an *error handler*. The concept of error handlers makes it possible to respond to and, possibly, recover from errors that arise during program execution. If further execution is not possible, the error handler can at least assure that the program is given a graceful abortion.

8.1 Error handlers

Any routine may include an error handler. The error handler is really a part of the routine, and the scope of any routine data also comprises the error handler of the routine. If an error occurs during the execution of the routine, control is transferred to its error handler.

Example:

```
FUNC num safediv( num x, num y)
    RETURN x / y;
ERROR
    IF ERRNO = ERR_DIVZERO THEN
        TPWrite "The number cannot be equal 0";
        RETURN x;
    ENDIF
    !All other errors are propagated RAISE;
ENDFUNC
```

The system variable *ERRNO* contains the error number of the (most recent) error and can be used by the error handler to identify that error. After any necessary actions have been taken, the error handler can:

- Resume execution, starting with the instruction in which the error occurred. This is done using the RETRY instruction. If this instruction causes the same error again, up to five error recoveries will take place; after that execution will stop.
- Resume execution, starting with the instruction following the instruction in which the error occurred. This is done using the TRYNEXT instruction.
- Return control to the caller of the routine using the RETURN instruction. If the routine is a function, the RETURN instruction must specify an appropriate return value.
- Propagate the error to the caller of the routine using the RAISE instruction.

An error handler should normally end with RAISE or EXIT, in order to take care of errors that cannot be handled.

In a chain of routine calls, each routine may have its own error handler. If an error

occurs in a routine that does not contain an error handler (or if the error is explicitly propagated using the raise instruction) the same error is raised again at the point where the routine was called – the error is *propagated*. If the top of the call chain is reached without any error handler being found, the *system error handler* is called. The system error handler just reports the error and stops the execution. If an error is propagated from a trap routine this will result in immediate termination of the execution by the system error handler.

Error recovery is not available for instructions in the backward handler. Such errors are always propagated to the system error handler.

In addition to errors detected and raised by the robot, a program can explicitly raise errors using the RAISE instruction. This facility can be used to recover from complex situations. It can, for example, be used to escape from deeply-nested code positions. Error numbers 1-90 may be used in the raise instruction. Explicitly-raised errors are treated exactly like errors raised by the system.

Note that it is not possible to recover from or respond to errors that occur within an error clause. Such errors are always propagated to the system error handler.

9 Interrupts

Interrupts are program-defined events, identified by *interrupt numbers*. An interrupt occurs when an *interrupt condition* is true. Unlike errors, the occurrence of an interrupt is not directly related to (synchronous with) a specific code position. The occurrence of an interrupt causes suspension of the normal program execution and control is passed to a *trap routine*.

Even though the robot immediately recognizes the occurrence of an interrupt (only delayed by the speed of the hardware), its response – calling the corresponding trap routine – can only take place at specific program positions, namely:

- when the next instruction is entered,
- any time during the execution of a waiting instruction, e.g. *WaitUntil*,
- any time during the execution of a movement instruction, e.g. *MoveL*.

This normally results in a delay of 5-120 ms between interrupt recognition and response, depending on what type of movement is being performed at the time of the interrupt.

The raising of interrupts may be *disabled* and *enabled*. If interrupts are disabled, any interrupt that occurs is queued and not raised until interrupts are enabled again. Note that the interrupt queue may contain more than one waiting interrupt. Queued interrupts are raised in *FIFO* order. Interrupts are always disabled during the execution of a trap routine.

When running stepwise and when the program has been stopped, no interrupts will be handled. Interrupts that are generated under these circumstances are not dealt with.

The maximal number of interrupts at the same time is 40.

9.1 Interrupt manipulation

Defining an interrupt makes it known to the robot. The definition specifies the interrupt condition and enables the interrupt.

```
Example:    VAR intnum sig1int;
              .
            ISignalDI di1, high, sig1int;
```

An enabled interrupt may in turn be disabled (and vice versa).

```
Example:    ISleep sig1int;                 disabled
              .
            IWatch sig1int;                 enabled
```

Deleting an interrupt removes its definition. It is not necessary to explicitly remove an interrupt definition, but a new interrupt cannot be defined to an interrupt variable until the previous definition has been deleted.

Example: IDelete sig1int;

9.2 Trap routines

Trap routines provide a means of dealing with interrupts. A trap routine can be connected to a particular interrupt using the CONNECT instruction. When an interrupt occurs, control is immediately transferred to the associated trap routine (if any). If an interrupt occurs, that does not have any connected trap routine, this is treated as a fatal error, i.e. causes immediate termination of program execution.

Example:

```
VAR intnum empty;
VAR intnum full;

   .PROC main()

   CONNECT empty WITH etrap;        connect trap routines
   CONNECT full WITH ftrap;
   ISignalDI di1, high, empty;      define feeder interrupts
   ISignalDI di3, high, full;
   .
   IDelete empty;
   IDelete full;
ENDPROC

TRAP etrap                          responds to "feeder
   open_valve;                      empty" interrupt
   RETURN;
ENDTRAP

TRAP ftrap                          responds to "feeder full"
   close_valve;                     interrupt
   RETURN;
ENDTRAP
```

Several interrupts may be connected to the same trap routine. The system variable *INTNO* contains the interrupt number and can be used by a trap routine to identify an interrupt. After the necessary action has been taken, a trap routine can be terminated using the RETURN instruction or when the end (ENDTRAP or ERROR) of the trap routine is reached. Execution continues from the place where the interrupt occurred.

10 Backward execution

A program can be executed backwards one instruction at a time. The following general restrictions are valid for backward execution:

- The instructions IF, FOR, WHILE and TEST cannot be executed backwards.
- It is not possible to step backwards out of a routine when reaching the beginning of the routine.

10.1 Backward handlers

Procedures may contain a backward handler that defines the backward execution of a procedure call.

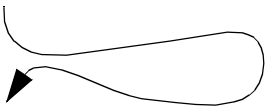
The backward handler is really a part of the procedure and the scope of any routine data also comprises the backward handler of the procedure.

Example:

```
PROC MoveTo ()
   MoveL p1,v500,z10,tool1;
   MoveL p2,v500,z10,tool1;
   MoveL p3,v500,z10,tool1;
BACKWARD
   MoveL p3,v100,z10,tool1;
   MoveL p2,v100,z10,tool1;
   MoveL p1,v100,z10,tool1;
ENDPROC
```

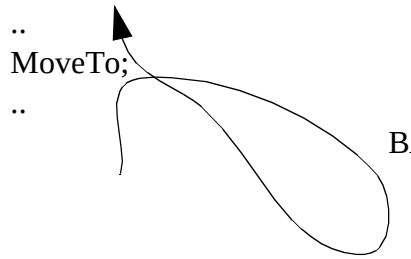
When the procedure is called during forward execution, the following occurs:

```
..
MoveTo;
..
```



```
PROC MoveTo ()
   MoveL p1,v500,z10,tool1;
   MoveL p2,v500,z10,tool1;
   MoveL p3,v500,z10,tool1;
BACKWARD
   MoveL p3,v100,z10,tool1;
   MoveL p2,v100,z10,tool1;
   MoveL p1,v100,z10,tool1;
ENDPROC
```

When the procedure is called during backwards execution, the following occurs:




```
PROC MoveTo ()
    MoveL p1,v500,z10,tool1;
    MoveL p2,v500,z10,tool1;
    MoveL p3,v500,z10,tool1;
BACKWARD
    MoveL p3,v100,z10,tool1;
    MoveL p2,v100,z10,tool1;
    MoveL p1,v100,z10,tool1;
ENDPROC
```

Instructions in the backward or error handler of a routine may not be executed backwards. Backward execution cannot be nested, i.e. two instructions in a call chain may not simultaneously be executed backwards.

A procedure with no backward handler cannot be executed backwards. A procedure with an empty backward handler is executed as “no operation”.

11 Multitasking

The events in a robot cell are often in parallel, so why are the programs not in parallel?

Multitasking RAPID is a way to execute programs in (pseudo) parallel with the normal execution. The execution is started at power on and will continue for ever, unless an error occurs in that program. One parallel program can be placed in the background or foreground of another program. It can also be on the same level as another program.

To use this function the robot must be configured with one extra TASK for each background program**.**

Up to 10 different tasks can be run in pseudo parallel. Each task consists of a set of modules, in the same way as the normal program. All the modules are local in each task.

Variables and constants are local in each task, but persistents are not. A persistent with the same name and type is reachable in all tasks. If two persistents have the same name, but their type or size differ, a runtime error will occur.

A task has its own trap handling and the event routines are triggered only on its own task system states (e.g. Start/Stop/Restart...).

There are a few restrictions on the use of Multitasking RAPID.

- Do not mix up parallel programs with a PLC. The response time is the same as the interrupt response time for one task. This is true, of course, when the task is not in the background of another busy program
- There is only one physical Teach Pendent, so be careful that a TPWrite request is not mixed in the Operator Window for all tasks.
- When running a Wait instruction in manual mode, a simulation box will come up after 3 seconds. This will only occur in the main task.
- Move instructions can only be executed in the main task (the task bind to program instance 0, see User's guide - System parameters).
- The execution of a task will halt during the time that some other tasks are accessing the file system, that is if the operator chooses to save or open a program, or if the program in a task uses the load/erase/read/write instructions.
- The Teach Pendent cannot access other tasks than the main task. So, the development of RAPID programs for other tasks can only be done if the code is loaded into the main task, or off-line.

For all settings, see User's Guide - System parameters.

11.1 Synchronising the tasks

In many applications a parallel task only supervises some cell unit, quite independently of the other tasks being executed. In such cases, no synchronisation mechanism is necessary. But there are other applications which need to know what the main task is

doing, for example.

Synchronising using polling

This is the easiest way to do it, but the performance will be the slowest.

Persistents are then used together with the instructions *WaitUntil*, *IF*, *WHILE* or *GOTO*.

If the instruction *WaitUntil* is used, it will poll internally every 100 ms. Do not poll more frequently in other implementations.

Example

TASK 0

```
MODULE module1
PERS BOOL startsync:=FALSE;
PROC main()

   startsync:= TRUE;
   .

ENDPROC
ENDMODULE
```

TASK 1

```
MODULE module2
PERS BOOL startsync:=FALSE;
PROC main()

   WaitUntil startsync;
   .

ENDPROC
ENDMODULE
```

Synchronising using an interrupt

To use this method, a cross connection must be prepared between two I/O-signals; one output cross connected to one input. The instruction *SetDO*, *ISignalDI* or *WaitDI* is then used.

Example

In this example do1 is connected to di1.

TASK 0

```
MODULE module1
PERS BOOL startsync:=FALSE;
PROC main()

SetDO do1,1;
   .

ENDPROC
ENDMODULE
```

TASK 1

```
MODULE module2
LOCAL VAR intnum isiint1;
PROC main()

CONNECT isiint1 WITH isi_trap;
ISignalDI di1, 1, isiint1;

WHILE TRUE DO
    WaitTime 200;
ENDWHILE

ENDPROC

TRAP isi_trap

     .

ENDTRAP
ENDMODULE
```

11.2 Intertask communication

All types of data can be sent between two (or more) tasks with persistent variables.

A persistent variable is global in all tasks. The persistent variable must be of the same type and size (array dimension) in all tasks that declared it. Otherwise a runtime error will occur.

All declarations must specify an init value to the persistent variable, but only the first module loaded with the declaration will use it.

Example

TASK 0

```
MODULE module1
PERS BOOL startsync:=FALSE;
PERS STRING stringtosend:="";
PROC main()
```

```
   stringtosend:=”this is a test”;

   startsync:= TRUE

ENDPROC
ENDMODULE
```

TASK 1

```
MODULE module2
PERS BOOL startsync:=FALSE;
PERS STRING stringtosend:=””;
PROC main()

   WaitUntil startsync;

   !read string
   IF stringtosend = “this is a test” THEN

ENDPROC
ENDMODULE
```

11.3 Type of task

Each extra task (not 0) is started in the system start sequence. If the task is of type STATIC, it will be restarted at the current position (where PP was when the system was powered off), but if the type is set to SEMISTATIC, it will be restarted from the beginning each time the power is turned on.

It is also possible to set the task to type NORMAL, then it will behave in the same was as task 0 (the main task, controlling the robot movement). The teach pendent can only be used to start task 0, so the only way to start other NORMAL tasks is to use CommunicationWare.

11.4 Priorities

The way to run the tasks as default is to run all tasks at the same level in a round robbin way (one basic step on each instance). But it is possible to change the priority of one task by putting the task in the background of another. Then the background will only execute when the foreground is waiting for some events, or has stopped the execution (idle). A robot program with move instructions will be in an idle state most of the time.

The example below describes some situations where the system has 10 tasks (see Figure 5)

Round robbin chain 1: tasks 0, 1, and 8 are busy

Round robbin chain 2:	tasks 0, 3, 4, 5 and 8 are busy tasks 1 and 2 are idle
Round robbin chain 3:	tasks 2, 4 and 5 are busy tasks 1, 2, 9 and 10 are idle.
Round robbin chain 4:	tasks 6 and 7 are busy tasks 0, 1, 2, 3 4, 5, 8 and 9 are idle

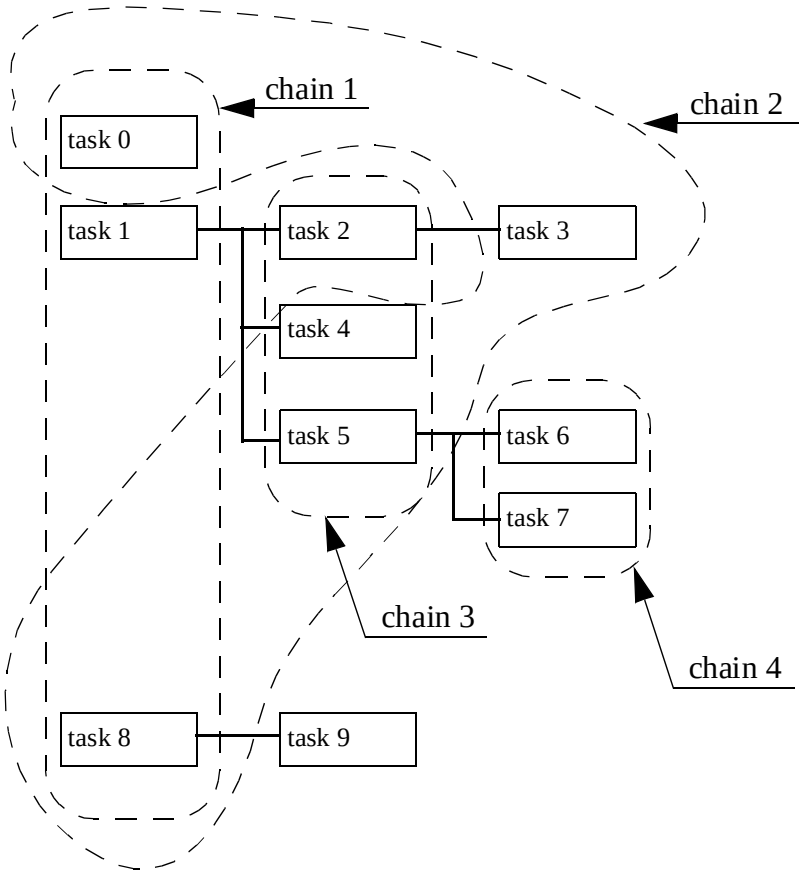


Figure 5 The tasks can have different priorities.

11.5 Task sizes

The system will supply a memory area with an installation depending on size. That area is shared by all tasks.

If the memory for a task is defined in the system parameters, it will be given the specified amount of memory.

All tasks without size specifications will share the free part of the supplied area in the following way. If the size of the main task is unspecified, it will be given the percent-

age of the free area that is specified in the *MemoryTask0* parameter (see System parameters). The remainder of the free area will be split up among the other tasks in equal parts.

The value of a persistent variable will be stored in a separate part of the system, and not affect the memory area above. See System parameters - *AveragePers*.

11.6 Something to think about

When you specify task priorities, you must think about the following:

- Always use the interrupt mechanism or loops with delays in supervision tasks. Otherwise the teach pendant will never have any time to interact with the user. And if the supervision task is in foreground, it will never allow another task to execute in the background.

CONTENTS

Page

1 Coordinate Systems

1.1 The robot's tool centre point (TCP)

The position of the robot and its movements are always related to the tool centre point. This point is normally defined as being somewhere on the tool, e.g. in the muzzle of a glue gun, at the centre of a gripper or at the end of a grading tool.

Several TCPs (tools) may be defined, but only one may be active at any one time. When a position is recorded, it is the position of the TCP that is recorded. This is also the point that moves along a given path, at a given velocity.

If the robot is holding a work object and working on a stationary tool, a stationary TCP is used. If that tool is active, the programmed path and speed are related to the work object. See *Stationary TCPs* on page 10.

1.2 Coordinate systems used to determine the position of the TCP

The tool (TCP's) position can be specified in different coordinate systems to facilitate programming and readjustment of programs.

The coordinate system defined depends on what the robot has to do. When no coordinate system is defined, the robot's positions are defined in the base coordinate system.

1.2.1 Base coordinate system

In a simple application, programming can be done in the base coordinate system; here the z-axis is coincident with axis 1 of the robot (see Figure 1).

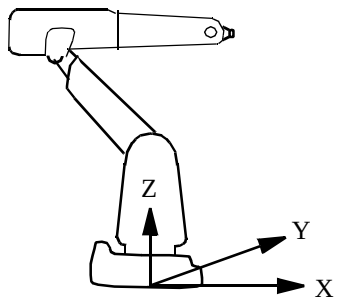


Figure 1 The base coordinate system.

The base coordinate system is located on the base of the robot:

- *The origin* is situated at the intersection of axis 1 and the base mounting surface.
- *The xy plane* is the same as the base mounting surface.
- *The x-axis* points forwards.
- *The y-axis* points to the left (from the perspective of the robot).
- *The z-axis* points upwards.

1.2.2 World coordinate system

If the robot is floor-mounted, programming in the base coordinate system is easy. If, however, the robot is mounted upside down (suspended), programming in the base coordinate system is more difficult because the directions of the axes are not the same as the principal directions in the working space. In such cases, it is useful to define a world coordinate system. The world coordinate system will be coincident with the base coordinate system, if it is not specifically defined.

Sometimes, several robots work within the same working space at a plant. A common world coordinate system is used in this case to enable the robot programs to communicate with one another. It can also be advantageous to use this type of system when the positions are to be related to a fixed point in the workshop. See the example in Figure 2.

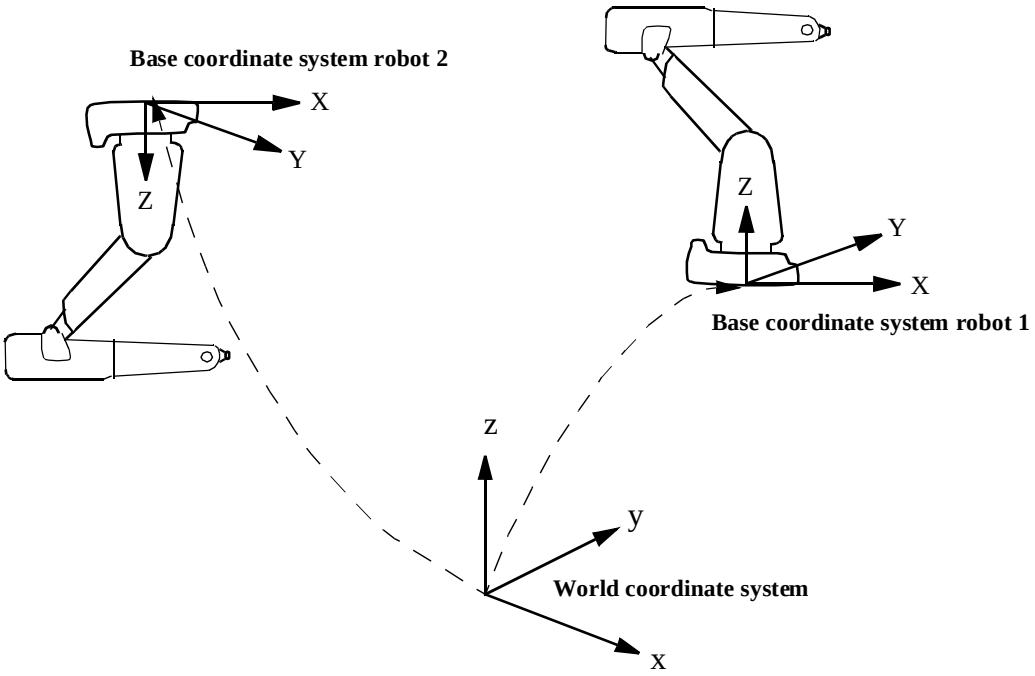


Figure 2 Two robots (one of which is suspended) with a common world coordinate system.

1.2.3 User coordinate system

A robot can work with different fixtures or working surfaces having different positions and orientations. A user coordinate system can be defined for each fixture. If all positions are stored in object coordinates, you will not need to reprogram if a fixture must be moved or turned. By moving/turning the user coordinate system as much as the fixture has been moved/turned, all programmed positions will follow the fixture and no reprogramming will be required.

The user coordinate system is defined based on the world coordinate system (see Figure 3).

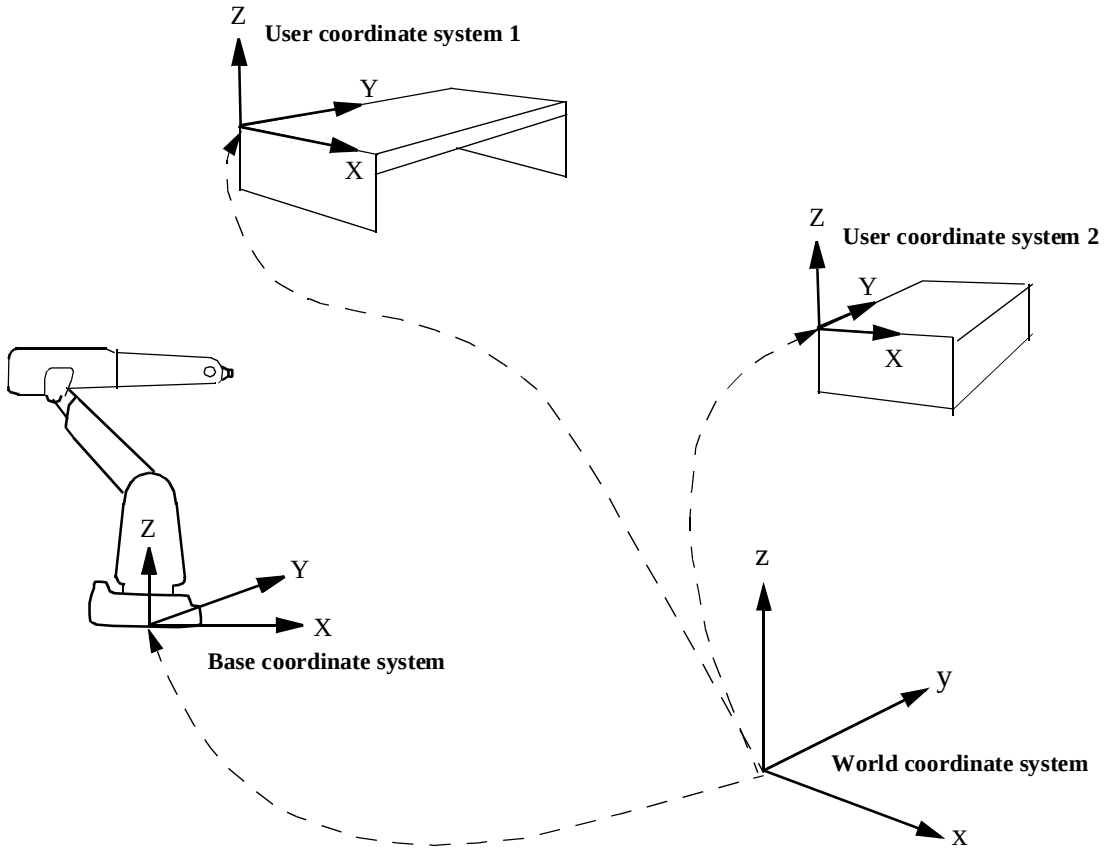


Figure 3 Two user coordinate systems describe the position of two different fixtures.

1.2.4 Object coordinate system

The user coordinate system is used to get different coordinate systems for different fixtures or working surfaces. A fixture, however, may include several work objects that are to be processed or handled by the robot. Thus, it often helps to define a coordinate system for each object in order to make it easier to adjust the program if the object is moved or if a new object, the same as the previous one, is to be programmed at a different location. A coordinate system referenced to an object is called an object coordinate system. This coordinate system is also very suited to off-line programming since the positions specified can usually be taken directly from a drawing of the work object. The object coordinate system can also be used when jogging the robot.

The object coordinate system is defined based on the user coordinate system (see Figure 4).

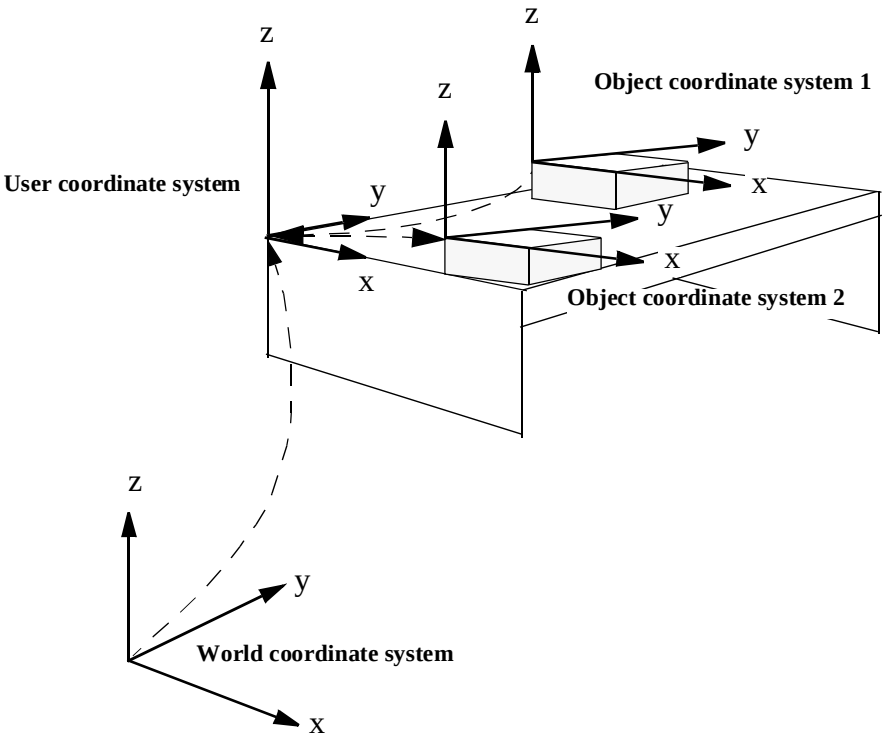


Figure 4 Two object coordinate systems describe the position of two different work objects located in the same fixture.

The programmed positions are always defined relative to an object coordinate system. If a fixture is moved/turned, this can be compensated for by moving/turning the user coordinate system. Neither the programmed positions nor the defined object coordinate systems need to be changed. If the work object is moved/turned, this can be compensated for by moving/turning the object coordinate system.

If the user coordinate system is movable, that is, coordinated external axes are used, then the object coordinate system moves with the user coordinate system. This makes it possible to move the robot in relation to the object even when the workbench is being manipulated.

1.2.5 Displacement coordinate system

Sometimes, the same path is to be performed at several places on the same object. To avoid having to re-program all positions each time, a coordinate system, known as the displacement coordinate system, is defined. This coordinate system can also be used in conjunction with searches, to compensate for differences in the positions of the individual parts.

The displacement coordinate system is defined based on the object coordinate system (see Figure 5).

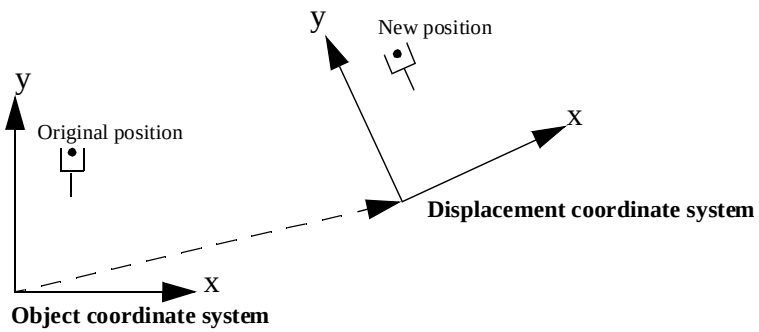


Figure 5 If program displacement is active, all positions are displaced.

1.2.6 Coordinated external axes

Coordination of user coordinate system

If a work object is placed on an external mechanical unit, that is moved whilst the robot is executing a path defined in the object coordinate system, a movable user coordinate system can be defined. The position and orientation of the user coordinate system will, in this case, be dependent on the axes rotations of the external unit. The programmed path and speed will thus be related to the work object (see Figure 6) and there is no need to consider the fact that the object is moved by the external unit.

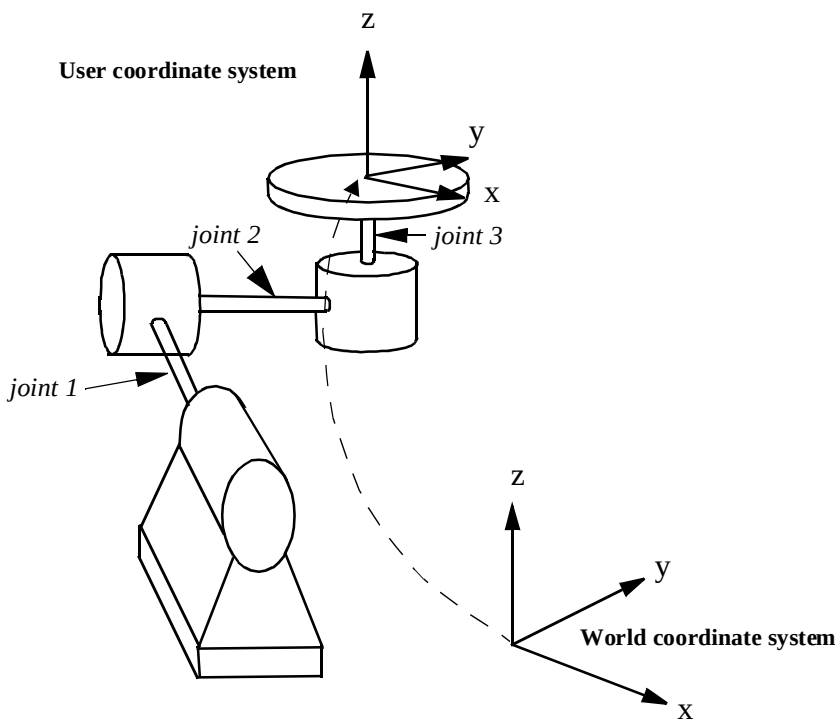


Figure 6 A user coordinate system, defined to follow the movements of a 3-axis external mechanical unit.

Coordination of base coordinate system

A movable coordinate system can also be defined for the base of the robot. This is of interest for the installation when the robot is mounted on a track or a gantry, for example. The position and orientation of the base coordinate system will, as for the moveable user coordinate system, be dependent on the movements of the external unit. The programmed path and speed will be related to the object coordinate system (Figure 7) and there is no need to think about the fact that the robot base is moved by an external unit. A coordinated user coordinate system and a coordinated base coordinate system can both be defined at the same time.

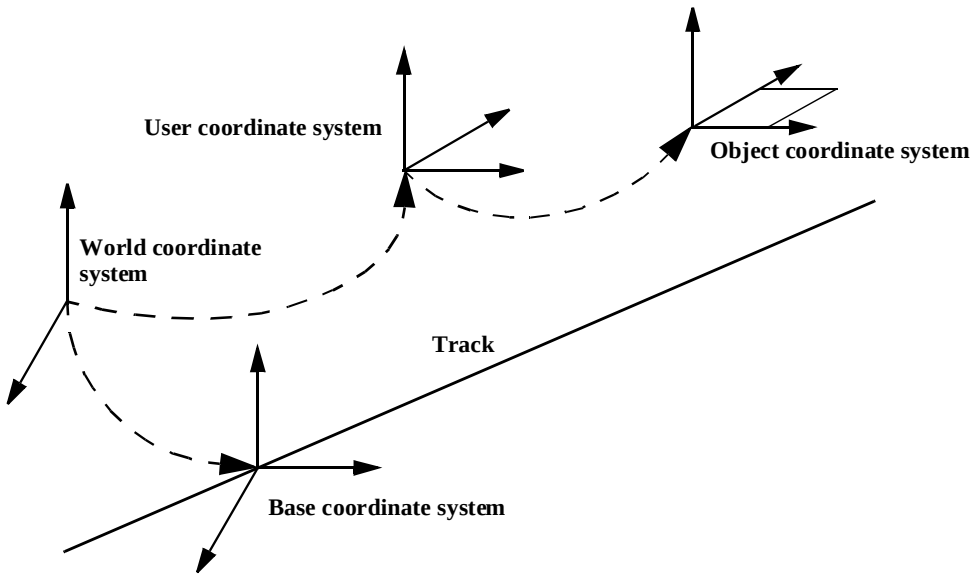


Figure 7 Coordinated interpolation with a track moving the base coordinate system of the robot.

To be able to calculate the user and the base coordinate systems when involved units are moved, the robot must be aware of:

- The calibration positions of the user and the base coordinate systems
- The relations between the angles of the external axes and the translation/rotation of the user and the base coordinate systems.

These relations are defined in the system parameters.

1.3 Coordinate systems used to determine the direction of the tool

The orientation of a tool at a programmed position is given by the orientation of the tool coordinate system. The tool coordinate system is referenced to the wrist coordinated system, defined at the mounting flange on the wrist of the robot.

1.3.1 Wrist coordinate system

In a simple application, the wrist coordinate system can be used to define the orientation of the tool; here the z-axis is coincident with axis 6 of the robot (see Figure 8).

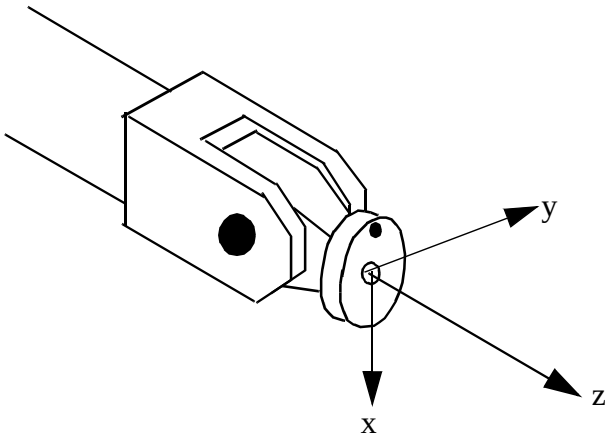


Figure 8 The wrist coordinate system.

The wrist coordinate system cannot be changed and is always the same as the mounting flange of the robot in the following respects:

- *The origin* is situated at the centre of the mounting flange (on the mounting surface).
- *The x-axis* points in the opposite direction, towards the control hole of the mounting flange.
- *The z-axis* points outwards, at right angles to the mounting flange.

1.3.2 Tool coordinate system

The tool mounted on the mounting flange of the robot often requires its own coordinate system to enable definition of its TCP, which is the origin of the tool coordinate system. The tool coordinate system can also be used to get appropriate motion directions when jogging the robot.

If a tool is damaged or replaced, all you have to do is redefine the tool coordinate system. The program does not normally have to be changed.

The TCP (origin) is selected as the point on the tool that must be correctly positioned, e.g. the muzzle on a glue gun. The tool coordinate axes are defined as those natural for the tool in question.

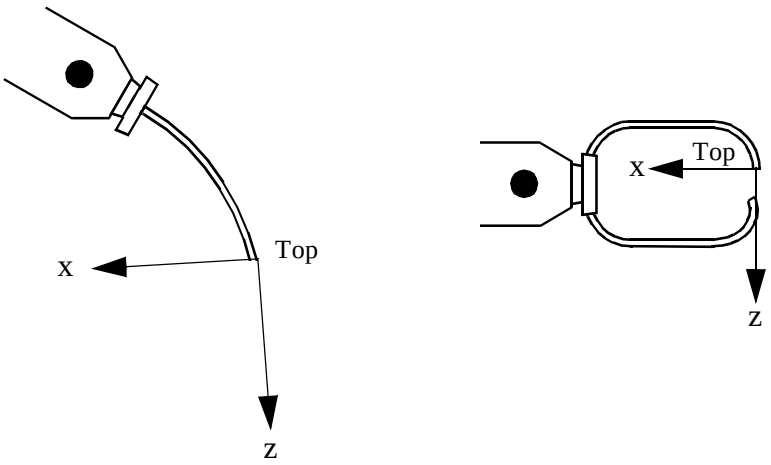


Figure 9 Tool coordinate system, as usually defined for an arc-welding gun (left) and a spot welding gun (right).

The tool coordinate system is defined based on the wrist coordinate system (see Figure 10).

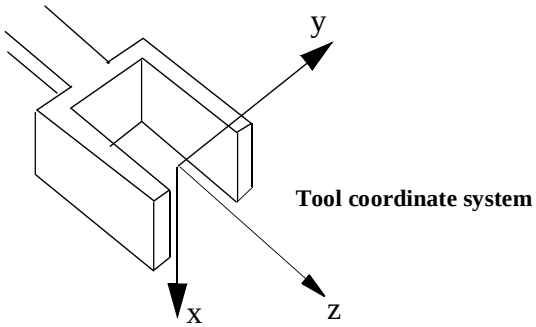


Figure 10 The tool coordinate system is defined relative to the wrist coordinate system, here for a gripper.

1.3.3 Stationary TCPs

If the robot is holding a work object and working on a stationary tool, a stationary TCP is used. If that tool is active, the programmed path and speed are related to the work object held by the robot.

This means that the coordinate systems will be reversed, as in Figure 11.

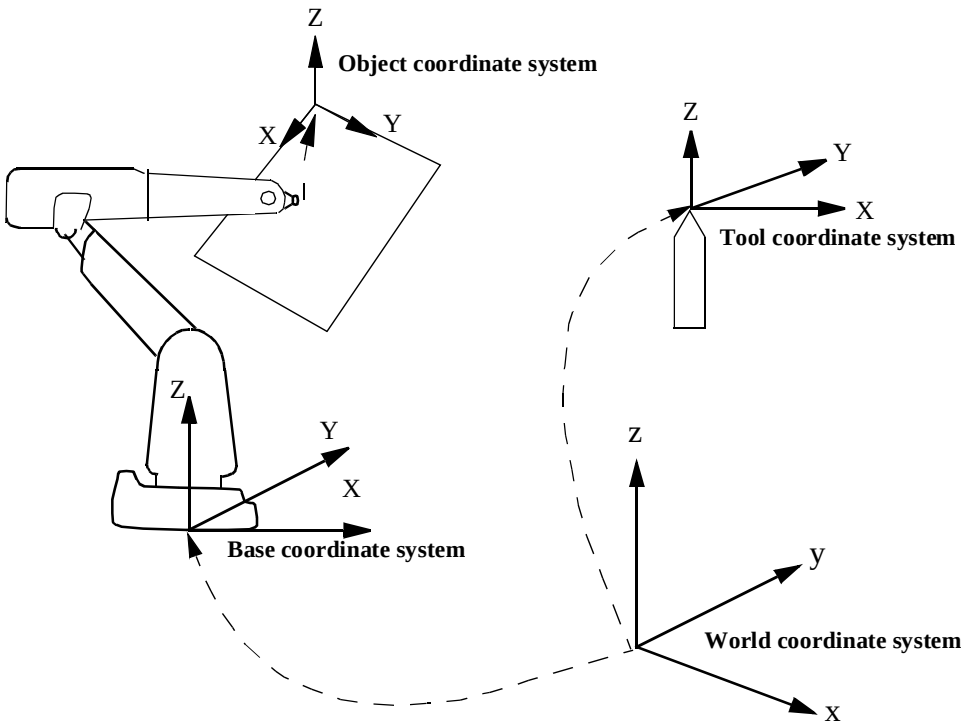


Figure 11 If a stationary TCP is used, the object coordinate system is usually based on the wrist coordinate system.

In the example in Figure 11, neither the user coordinate system nor program displacement is used. It is, however, possible to use them and, if they are used, they will be related to each other as shown in Figure 12.

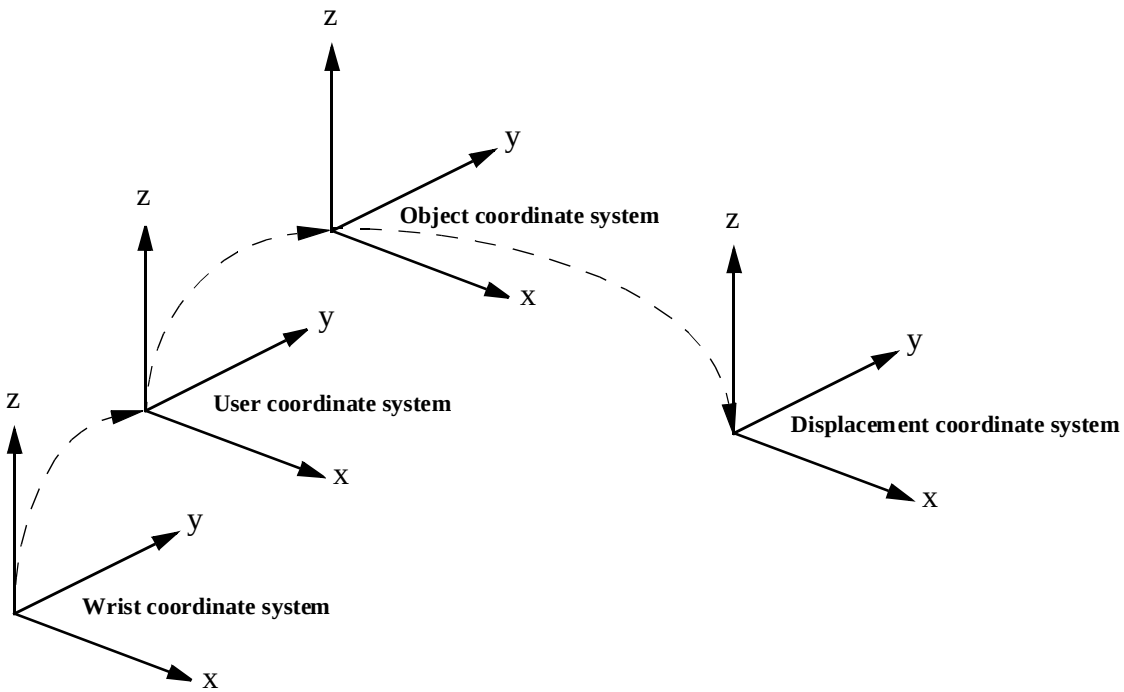


Figure 12 Program displacement can also be used together with stationary TCPs.

1.4 Related information

	Described in:
Definition of the world coordinate system	User's Guide - System Parameters
Definition of the user coordinate system	User's Guide - Calibration Data Types - *wobjdata*
Definition of the object coordinate system	User's Guide - Calibration Data Types - *wobjdata*
Definition of the tool coordinate system	User's Guide - Calibration Data Types - *tooldata*
Definition of a tool centre point	User's Guide - Calibration Data Types - *tooldata*
Definition of displacement frame	User's Guide - Calibration RAPID Summary - *Motion Settings*
Jogging in different coordinate systems	User's Guide - Jogging

2 Positioning during Program Execution

2.1 General

During program execution, positioning instructions in the robot program control all movements. The main task of the positioning instructions is to provide the following information on how to perform movements:

- The destination point of the movement (defined as the position of the tool centre point, the orientation of the tool, the configuration of the robot and the position of the external axes).
- The interpolation method used to reach the destination point, e.g. joint interpolation, linear interpolation or circle interpolation.
- The velocity of the robot and external axes.
- The zone data (defines how the robot and the external axes are to pass the destination point).
- The coordinate systems (tool, user and object) used for the movement.

As an alternative to defining the velocity of the robot and the external axes, the time for the movement can be programmed. This should, however, be avoided if the weaving function is used. Instead the velocities of the orientation and external axes should be used to limit the speed, when small or no TCP-movements are made.

In material handling and palletising applications with intensive and frequent movements, the drive system supervision may trip out and stop the robot in order to prevent overheating of drives or motors. If this occurs, the cycle time needs to be slightly increased by reducing programmed speed or acceleration.

2.2 Interpolation of the position and orientation of the tool

2.2.1 Joint interpolation

When path accuracy is not too important, this type of motion is used to move the tool quickly from one position to another. Joint interpolation also allows an axis to move from any location to another within its working space, in a single movement.

All axes move from the start point to the destination point at constant axis velocity (see Figure 13).

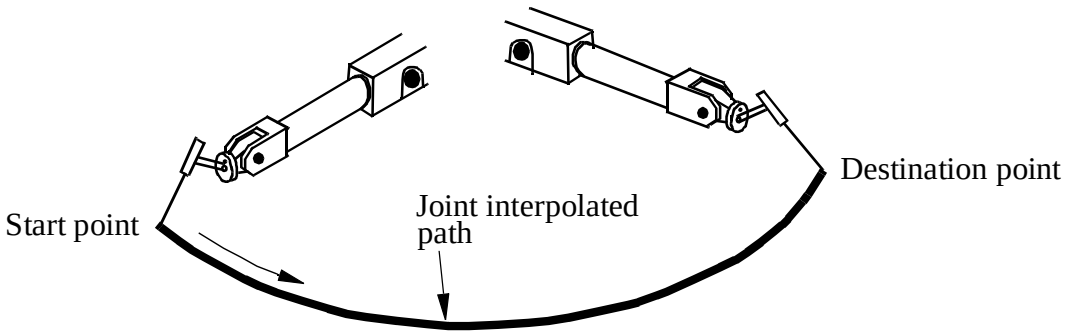


Figure 13 Joint interpolation is often the fastest way to move between two points as the robot axes follow the closest path between the start point and the destination point (from the perspective of the axis angles).

The velocity of the tool centre point is expressed in mm/s (in the object coordinate system). As interpolation takes place axis-by-axis, the velocity will not be exactly the programmed value.

During interpolation, the velocity of the limiting axis, i.e. the axis that travels fastest relative to its maximum velocity in order to carry out the movement, is determined. Then, the velocities of the remaining axes are calculated so that all axes reach the destination point at the same time.

All axes are coordinated in order to obtain a path that is independent of the velocity. Acceleration is automatically optimised to the max performance of the robot.

2.2.2 Linear interpolation

During linear interpolation, the TCP travels along a straight line between the start and destination points (see Figure 14).

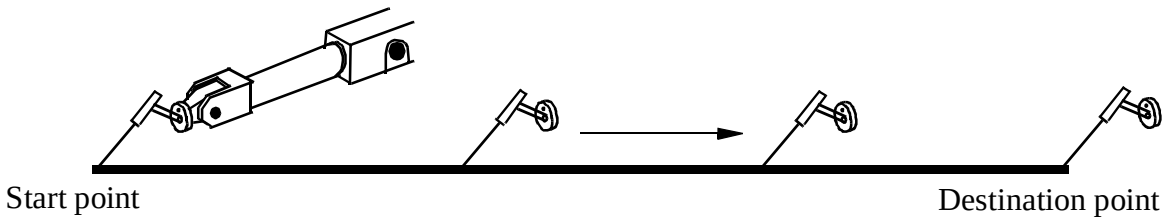


Figure 14 Linear interpolation without reorientation of the tool.

To obtain a linear path in the object coordinate system, the robot axes must follow a non-linear path in the axis space. The more non-linear the configuration of the robot is, the more accelerations and decelerations are required to make the tool move in a straight line and to obtain the desired tool orientation. If the configuration is extremely non-linear (e.g. in the proximity of wrist and arm singularities), one or more of the axes will require more torque than the motors can give. In this case, the velocity of all axes will automatically be reduced.

The orientation of the tool remains constant during the entire movement unless a reorientation has been programmed. If the tool is reoriented, it is rotated at constant velocity.

A maximum rotational velocity (in degrees per second) can be specified when rotating the tool. If this is set to a low value, reorientation will be smooth, irrespective of the velocity defined for the tool centre point. If it is a high value, the reorientation velocity is only limited by the maximum motor speeds. As long as no motor exceeds the limit for the torque, the defined velocity will be maintained. If, on the other hand, one of the motors exceeds the current limit, the velocity of the entire movement (with respect to both the position and the orientation) will be reduced.

All axes are coordinated in order to obtain a path that is independent of the velocity. Acceleration is optimised automatically.

2.2.3 Circular interpolation

A circular path is defined using three programmed positions that define a circle segment. The first point to be programmed is the start of the circle segment. The next point is a support point (circle point) used to define the curvature of the circle, and the third point denotes the end of the circle (see Figure 15).

The three programmed points should be dispersed at regular intervals along the arc of the circle to make this as accurate as possible.

The orientation defined for the support point is used to select between the short and the long twist for the orientation from start to destination point.

If the programmed orientation is the same relative to the circle at the start and the destination points, and the orientation at the support is close to the same orientation relative to the circle, the orientation of the tool will remain constant relative to the path.

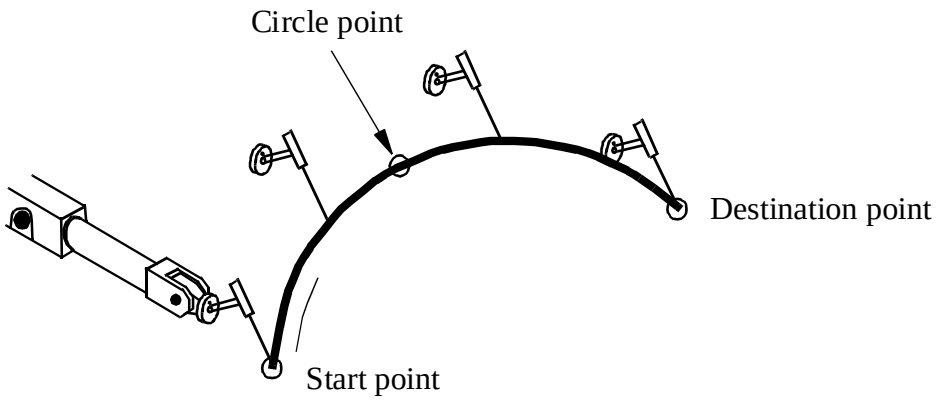


Figure 15 Circular interpolation with a short twist for part of a circle (circle segment) with a start point, circle point and destination point.

However, if the orientation at the support point is programmed closer to the orientation rotated 180°, the alternative twist is selected (see Figure 16).

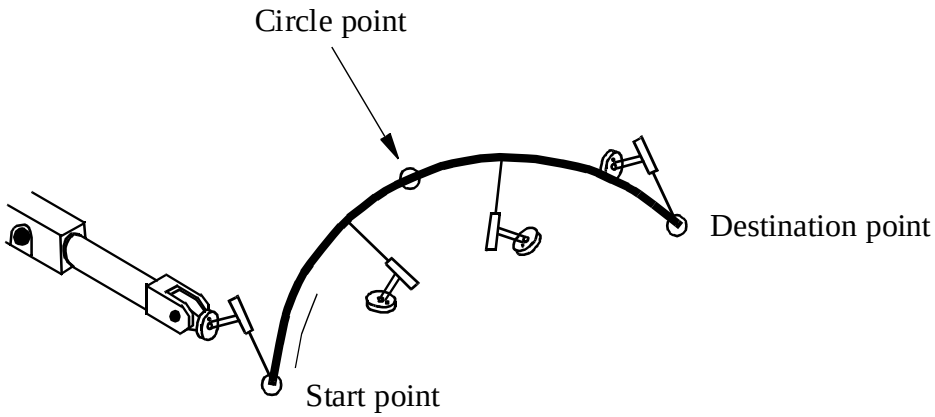


Figure 16 Circular interpolation with a long twist for orientation is achieved by defining the orientation in the circle point in the opposite direction compared to the start point.

As long as all motor torques do not exceed the maximum permitted values, the tool will move at the programmed velocity along the arc of the circle. If the torque of any of the motors is insufficient, the velocity will automatically be reduced at those parts of the circular path where the motor performance is insufficient.

All axes are coordinated in order to obtain a path that is independent of the velocity. Acceleration is optimised automatically.

2.2.4 SingArea\Wrist

During execution in the proximity of a singular point, linear or circular interpolation may be problematic. In this case, it is best to use modified interpolation, which means that the wrist axes are interpolated axis-by-axis, with the TCP following a linear or circular path. The orientation of the tool, however, will differ somewhat from the programmed orientation.

In the *SingArea\Wrist* case the orientation in the circle support point will be the same as programmed. However, the tool will not have a constant direction relative to the circle plane as for normal circular interpolation. If the circle path passes a singularity, the orientation in the programmed positions sometimes must be modified to avoid big wrist movements, which can occur if a complete wrist reconfiguration is generated when the circle is executed (joints 4 and 6 moved 180 degrees each).

2.3 Interpolation of corner paths

The destination point is defined as a stop point in order to get point-to-point movement. This means that the robot and any external axes will stop and that it will not be possible to continue positioning until the velocities of all axes are zero and the axes are close to their destinations.

Fly-by points are used to get continuous movements past programmed positions. In this way, positions can be passed at high speed without having to reduce the speed unnecessarily. A fly-by point generates a corner path (parabola path) past the programmed

position, which generally means that the programmed position is never reached. The beginning and end of this corner path are defined by a zone around the programmed position (see Figure 17).

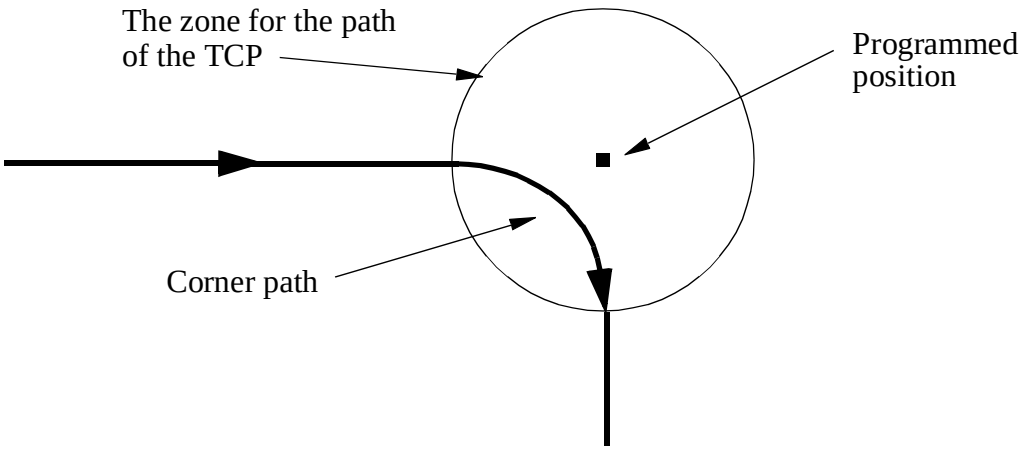


Figure 17 A fly-by point generates a corner path to pass the programmed position.

All axes are coordinated in order to obtain a path that is independent of the velocity. Acceleration is optimised automatically.

2.3.1 Joint interpolation in corner paths

The size of the corner paths (zones) for the TCP movement is expressed in mm (see Figure 18). Since the interpolation is performed axis-by-axis, the size of the zones (in mm) must be recalculated in axis angles (radians). This calculation has an error factor (normally max. 10%), which means that the true zone will deviate somewhat from the one programmed.

If different speeds have been programmed before or after the position, the transition from one speed to the other will be smooth and take place within the corner path without affecting the actual path.

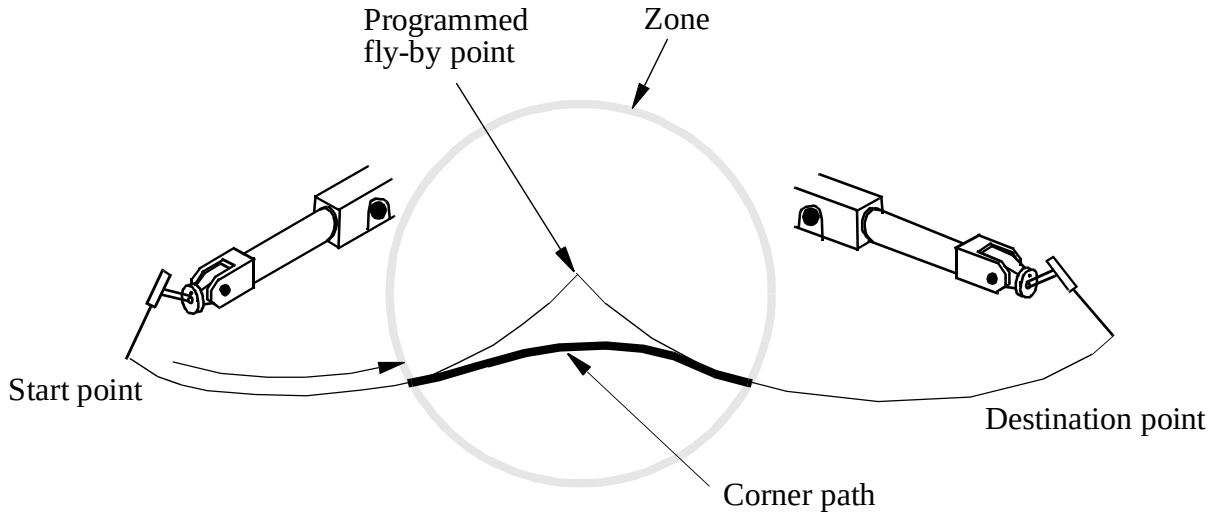


Figure 18 During joint interpolation, a corner path is generated in order to pass a fly-by point.

2.3.2 Linear interpolation of a position in corner paths

The size of the corner paths (zones) for the TCP movement is expressed in mm (see Figure 19).

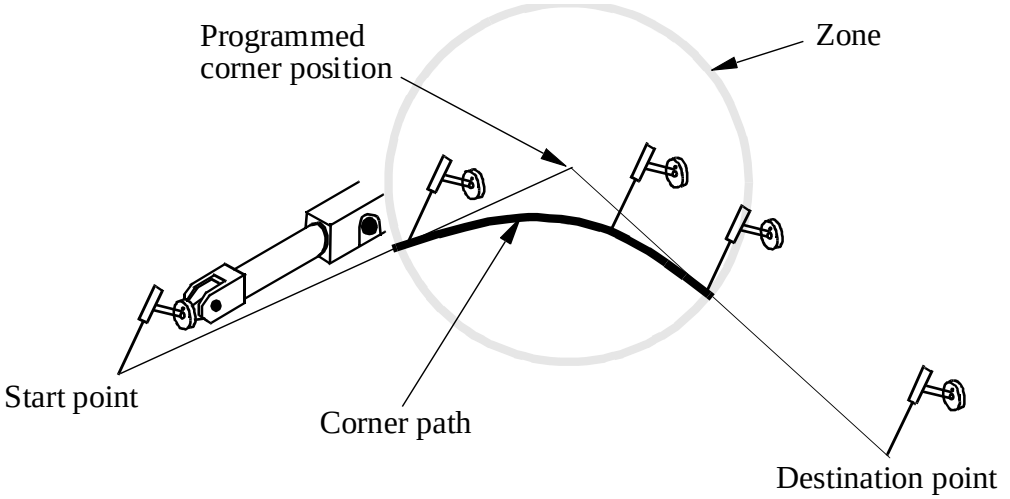


Figure 19 During linear interpolation, a corner path is generated in order to pass a fly-by point.

If different speeds have been programmed before or after the corner position, the transition will be smooth and take place within the corner path without affecting the actual path.

If the tool is to carry out a process (such as arc-welding, gluing or water cutting) along the corner path, the size of the zone can be adjusted to get the desired path. If the shape of the parabolic corner path does not match the object geometry, the programmed positions can be placed closer together, making it possible to approximate the desired path using two or more smaller parabolic paths.

2.3.3 Linear interpolation of the orientation in corner paths

Zones can be defined for tool orientations, just as zones can be defined for tool positions. The orientation zone is usually set larger than the position zone. In this case, the reorientation will start interpolating towards the orientation of the next position before the corner path starts. The reorientation will then be smoother and it will probably not be necessary to reduce the velocity to perform the reorientation.

The tool will be reoriented so that the orientation at the end of the zone will be the same as if a stop point had been programmed (see Figure 20a-c).

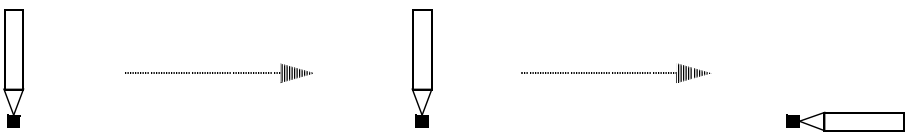


Figure 20a Three positions with different tool orientations are programmed as above.

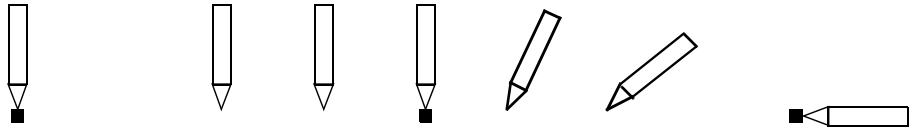


Figure 20b If all positions were stop points, program execution would look like this.

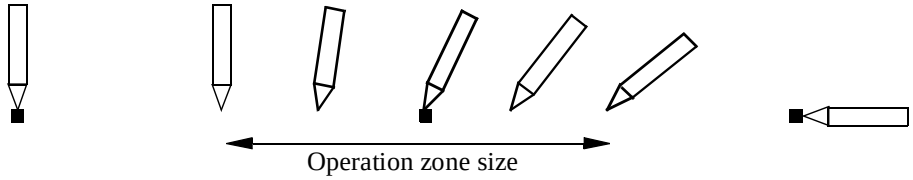


Figure 20c If the middle position was a fly-by point, program execution would look like this.

The orientation zone for the tool movement is normally expressed in mm. In this way, you can determine directly where on the path the orientation zone begins and ends. If the tool is not moved, the size of the zone is expressed in angle of rotation degrees instead of TCP-mm.

If different reorientation velocities are programmed before and after the fly-by point, and if the reorientation velocities limit the movement, the transition from one velocity to the other will take place smoothly within the corner path.

2.3.4 Interpolation of external axes in corner paths

Zones can also be defined for external axes, in the same manner as for orientation. If the external axis zone is set to be larger than the TCP zone, the interpolation of the external axes towards the destination of the next programmed position, will be started before the TCP corner path starts. This can be used for smoothing external axes movements in the same way as the orientation zone is used för the smoothing of the wrist movements.

2.3.5 Corner paths when changing the interpolation method

Corner paths are also generated when one interpolation method is exchanged for another. The interpolation method used in the actual corner paths is chosen in such a way as to make the transition from one method to another as smooth as possible. If the corner path zones for orientation and position are not the same size, more than one interpolation method may be used in the corner path (see Figure 21).

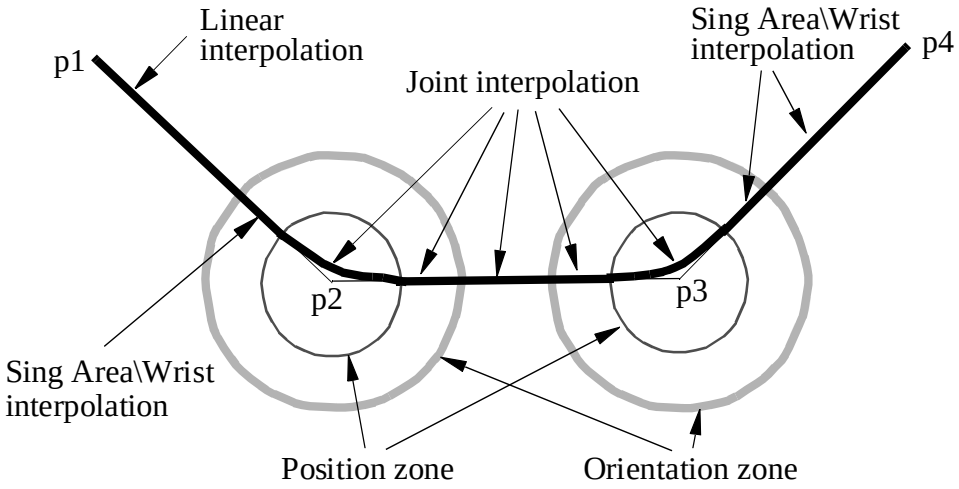


Figure 21 Interpolation when changing from one interpolation method to another. Linear interpolation has been programmed between p1 and p2; joint interpolation between p2 and p3; and Sing Area\Wrist interpolation between p3 and p4.

If the interpolation is changed from a normal TCP-movement to a reorientation without a TCP-movement or vice versa, no corner zone will be generated. The same will be the case if the interpolation is changed to or from an external joint movement without TCP-movement.

2.3.6 Interpolation when changing coordinate system

When there is a change of coordinate system in a corner path, e.g. a new TCP or a new work object, joint interpolation of the corner path is used. This is also applicable when changing from coordinated operation to non-coordinated operation, or vice versa.

2.3.7 Corner paths with overlapping zones

If programmed positions are located close to each other, it is not unusual for the programmed zones to overlap. To get a well-defined path and to achieve optimum velocity at all times, the robot reduces the size of the zone to half the distance from one overlapping programmed position to the other (see Figure 22). The same zone radius is always used for inputs to or outputs from a programmed position, in order to obtain symmetrical corner paths.

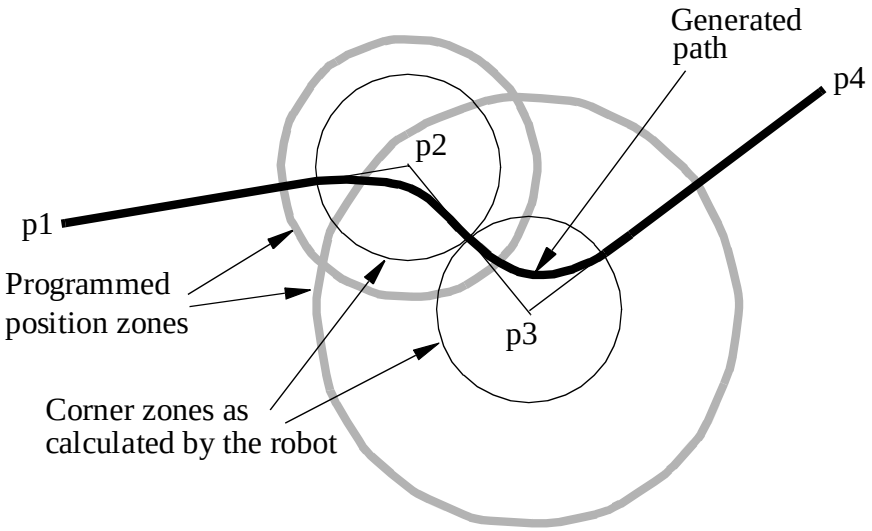


Figure 22 Interpolation with overlapping position zones. The zones around p2 and p3 are larger than half the distance from p2 to p3. Thus, the robot reduces the size of the zones to make them equal to half the distance from p2 to p3, thereby generating symmetrical corner paths within the zones.

Both position and orientation corner path zones can overlap. As soon as one of these corner path zones overlap, that zone is reduced (see Figure 23).

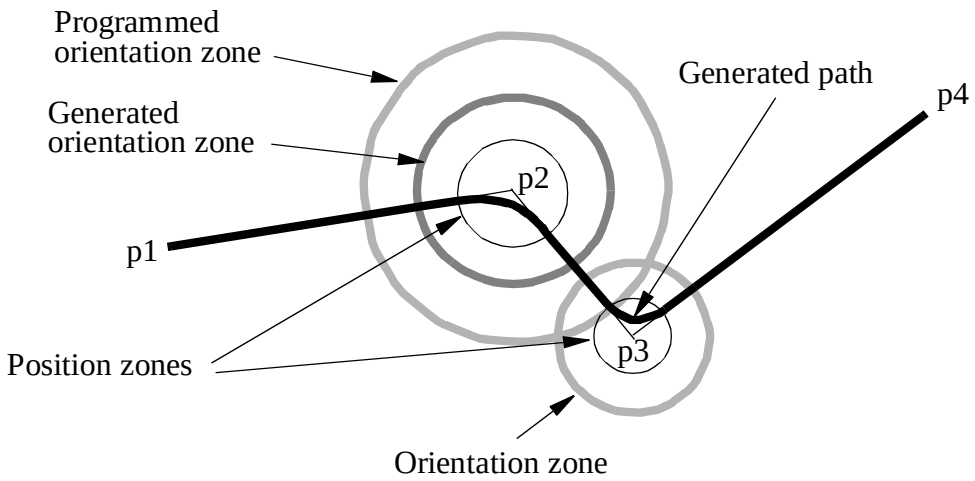


Figure 23 Interpolation with overlapping orientation zones. The orientation zone at p2 is larger than half the distance from p2 to p3 and is thus reduced to half the distance from p2 to p3. The position zones do not overlap and are consequently not reduced; the orientation zone at p3 is not reduced either.

2.3.8 Planning time for fly-by points

Occasionally, if the next movement is not planned in time, programmed fly-by points can give rise to a stop point. This may happen when:

- A number of logical instructions with long program execution times are

programmed between short movements.

- The points are very close together at high speeds.

If stop points are a problem then use concurrent program execution.

2.4 Independent axes

An independent axis is an axis moving independently of other axes in the robot system. It is possible to change an axis to independent mode and later back to normal mode again.

A special set of instructions handles the independent axes. Four different move instructions specify the movement of the axis. For instance, the *IndCMove* instruction starts the axis for continuous movement. The axis then keeps moving at a constant speed (regardless of what the robot does) until a new independent-instruction is executed.

To change back to normal mode a reset instruction, *IndReset*, is used. The reset instruction can also set a new reference for the measurement system - a type of new synchronization of the axis. Once the axis is changed back to normal mode it is possible to run it as a normal axis.

2.4.1 Program execution

An axis is immediately change to independent mode when an *Ind_Move* instruction is executed. This takes place even if the axis is being moved at the time, such as when a previous point has been programmed as a fly-by point, or when simultaneous program execution is used.

If a new *Ind_Move* instruction is executed before the last one is finished, the new instruction immediately overrides the old one.

If a program execution is stopped when an independent axis is moving, that axis will stop. When the program is restarted the independent axis starts automatically. No active coordination between independent and other axes in normal mode takes place.

If a loss of voltage occurs when an axis is in independent mode, the program cannot be restarted. An error message is then displayed, and the program must be started from the beginning.

Note that a mechanical unit may not be deactivated when one of its axes is in independent mode.

2.4.2 Stepwise execution

During stepwise execution, an independent axis is executed only when another instruction is being executed. The movement of the axis will also be stepwise in line with the execution of other instruments, see Figure 24.

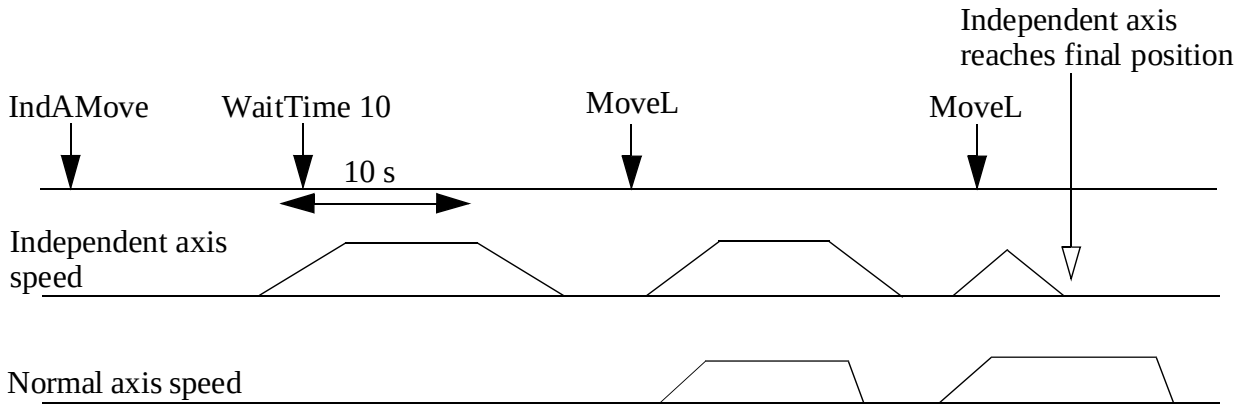


Figure 24 Stepwise execution of independent axes.

2.4.3 Jogging

Axes in independent mode cannot be jogged. If an attempt is made to execute the axis manually, the axis does not move and an error message is displayed. Execute an *IndReset* instruction or move the program pointer to main, in order to leave the independent mode.

2.4.4 Working range

The physical working range is the total movement of the axis.

The logical working range is the range used by RAPID instructions and read in the jogging window.

After synchronization (updated revolution counter), the physical and logical working range coincide. By using the *IndReset* instruction the logical working area can be moved, see Figure 25.

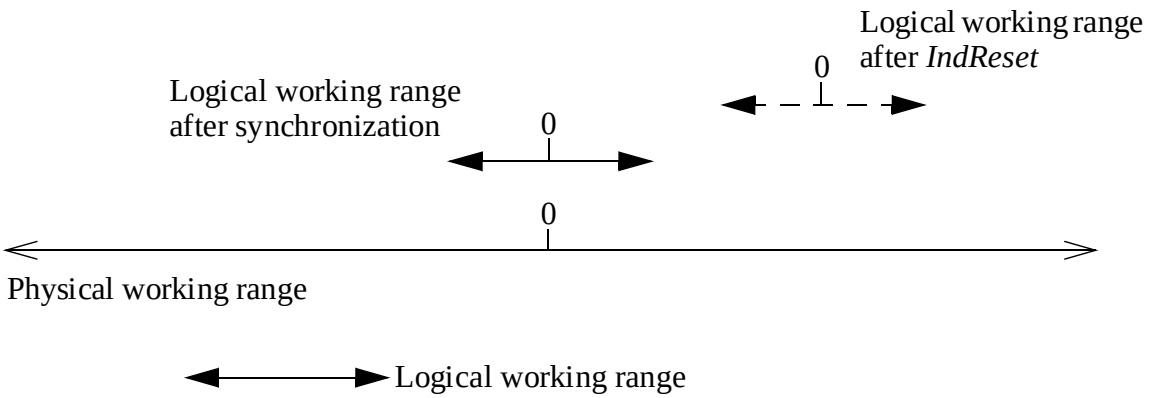


Figure 25 The logical working range can be moved, using the instruction IndReset.

The resolution of positions is decreased when moving away from logical position 0.

Low resolution together with stiff tuned controller can result in unacceptable torque, noise and controller instability. Check the controller tuning and axis performance close to the working range limit at installation. Also check if the position resolution and path performance are acceptable.

2.4.5 Speed and acceleration

In manual mode with reduced speed, the speed is reduced to the same level as if the axis was running as non-independent. Note that the *IndSpeed\InSpeed* function will not be TRUE if the axis speed is reduced.

The *VelSet* instruction and speed correction in percentage via the production window, are active for independent movement. Note that correction via the production window inhibits TRUE value from the *IndSpeed\InSpeed* function.

In independent mode, the lowest value of acceleration and deceleration, specified in the configuration file, is used both for acceleration and deceleration. This value can be reduced by the ramp value in the instruction (1 - 100%). The *AccSet* instruction does not affect axes in independent mode.

2.4.6 Robot axes

Only robot axis 6 can be used as independent axis. Normally only the *IndReset* instruction is used for this axis.

If axis 6 is used as an independent axis, singularity problems may occur because the normal 6-axes coordinate transform function is still used. If a problem occurs, execute the same program with axis 6 in normal mode. Modify the points or use *SingArea\Wrist* or *MoveJ* instructions.

The axis 6 is also internally active in the path performance calculation. A result of this is that an internal movement of axis 6 can reduce the speed of the other axes in the system.

The independent working range for axis 6 is defined with axis 4 and 5 in home position. If axis 4 or 5 is out of home position the working range for axis 6 is moved due to the gear coupling. However, the position read from teach pendant for axis 6 is compensated with the positions of axis 4 and 5 via the gear coupling.

2.5 Soft Servo

In some applications there is a need for a servo, which acts like a mechanical spring. This means that the force from the robot on the work object will increase as a function of the distance between the programmed position (behind the work object) and the contact position (robot tool - work object).

The relationship between the position deviation and the force, is defined by a parameter called ***softness.*** The higher the softness parameter, the larger the position deviation

required to obtain the same force.

The softness parameter is set in the program and it is possible to change the softness values anywhere in the program. Different softness values can be set for different joints and it is also possible to mix joints having normal servo with joints having soft servo.

Activation and deactivation of soft servo as well as changing of softness values can be made when the robot is moving. When this is done, a tuning will be made between the different servo modes and between different softness values to achieve smooth transitions. The tuning time can be set from the program with the parameter ramp. With *ramp = 1*, the transitions will take 0.5 seconds, and in the general case the transition time will be *ramp x 0.5* in seconds.

Note that deactivation of soft servo should not be done when there is a force between the robot and the work object.

With high softness values there is a risk that the servo position deviations may be so big that the axes will move outside the working range of the robot.

2.6 Path Resolution

The robot has a parameter named “path resolution”, which can be used in robot installations having external axes with long deceleration times. In such applications the warning “50082 Deceleration too long” will be reported, simultaneously generating a quick-stop. The path resolution parameter will then need to be increased until the problem disappears.

The need for tuning the path resolution parameter will increase when:

- The acceleration value of an external axis (and the robot) is decreased (*Acc Set*, first parameter)
- The acceleration derivative is decreased (*Acc Set*, second parameter)
- The speed is increased
- The distance between close programmed positions are decreased
- The number of simultaneously controlled axes is increased
- Coordinated interpolation is used.

It is important to use a path resolution value which is as small as possible to achieve a high path resolution also at high speed.

2.7 Stop and restart

A movement can be stopped in three different ways:

1. *For a normal stop* the robot will stop on the path, which makes a restart easy.

2. *For a stiff stop* the robot will stop in a shorter time than for the normal stop, but the

deceleration path will not follow the programmed path. This stop method is, for example, used for search stop when it is important to stop the motion as soon as possible.

3. *For a quick-stop* the mechanical brakes are used to achieve a deceleration distance, which is as short as specified for safety reasons. The path deviation will usually be bigger for a quick-stop than for a stiff stop.

After a stop (any of the types above) a restart can always be made on the interrupted path. If the robot has stopped outside the programmed path, the restart will begin with a return to the position on the path, where the robot should have stopped.

A restart following a power failure is equivalent to a restart after a quick-stop. It should be noted that the robot will always return to the path before the interrupted program operation is restarted, even in cases when the power failure occurred while a logical instruction was running. When restarting, all times are counted from the beginning; for example, positioning on time or an interruption in the instruction *WaitTime*.

2.8 Related information

	Described in:
Definition of speed	Data Types - *speeddata*
Definition of zones (corner paths)	Data Types - *zonedata*
Instruction for joint interpolation	Instructions - *MoveJ*
Instruction for linear interpolation	Instructions - *MoveL*
Instruction for circular interpolation	Instructions - *MoveC*
Instruction for modified interpolation	Instructions - *SingArea*
Singularity	Motion and I/O Principles- *Singularity*
Concurrent program execution	Motion and I/O Principles- *Syncronization with logical instructions*

3 Synchronization with logical instructions

Instructions are normally executed sequentially in the program. Nevertheless, logical instructions can also be executed at specific positions or during an ongoing movement.

A logical instruction is any instruction that does not generate a robot movement or an external axis movement, e.g. an I/O instruction.

3.1 Sequential program execution at stop points

If a positioning instruction has been programmed as a stop point, the subsequent instruction is not executed until the robot and the external axes have come to a standstill, i.e. when the programmed position has been attained (see Figure 26).

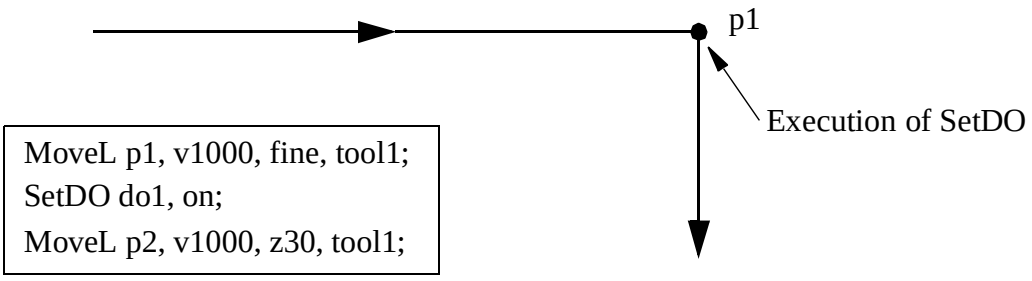


Figure 26 A logical instruction after a stop point is not executed until the destination position has been reached.

3.2 Sequential program execution at fly-by points

If a positioning instruction has been programmed as a fly-by point, the subsequent logical instructions are executed some time before reaching the largest zone (for position, orientation or external axes). See Figure 27. These instructions are then executed in order.

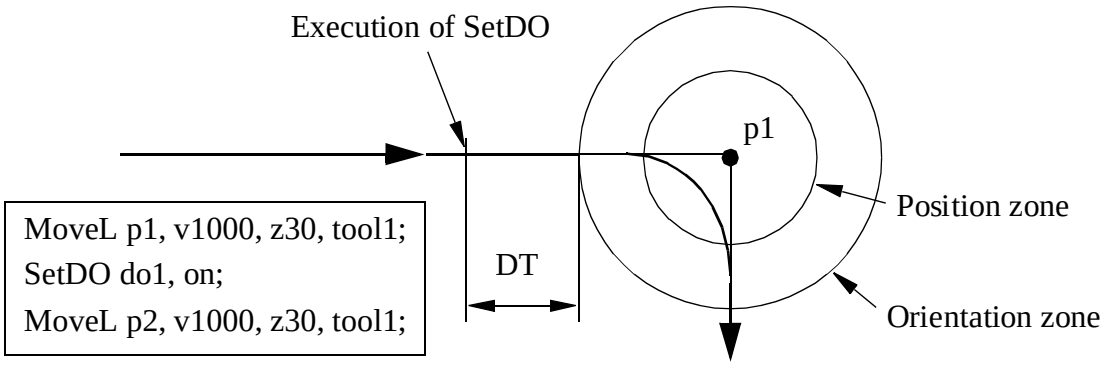


Figure 27 A logical instruction following a fly-by point is executed before reaching the largest zone.

The time at which they are executed *(DT*) comprises the following time components:

- The time it takes for the robot to plan the next move: approx. 0.1 seconds.
- The robot delay (servo lag) in seconds: 0 - 1.0 seconds depending on the velocity and the actual deceleration performance of the robot.

3.3 Concurrent program execution

Concurrent program execution can be programmed using the argument *\Conc* in the positioning instruction. This argument is used to:

- Execute one or more logical instructions at the same time as the robot moves in order to reduce the cycle time (e.g. used when communicating via serial channels).

When a positioning instruction with the argument *\Conc* is executed, the following logical instructions are also executed (in sequence):

- If the robot is not moving, or if the previous positioning instruction ended with a stop point, the logical instructions are executed as soon as the current positioning instruction starts (at the same time as the movement). See Figure 28.
- If the previous positioning instruction ends at a fly-by point, the logical instructions are executed at a given time (*DT*) before reaching the largest zone (for position, orientation or external axes). See Figure 29.

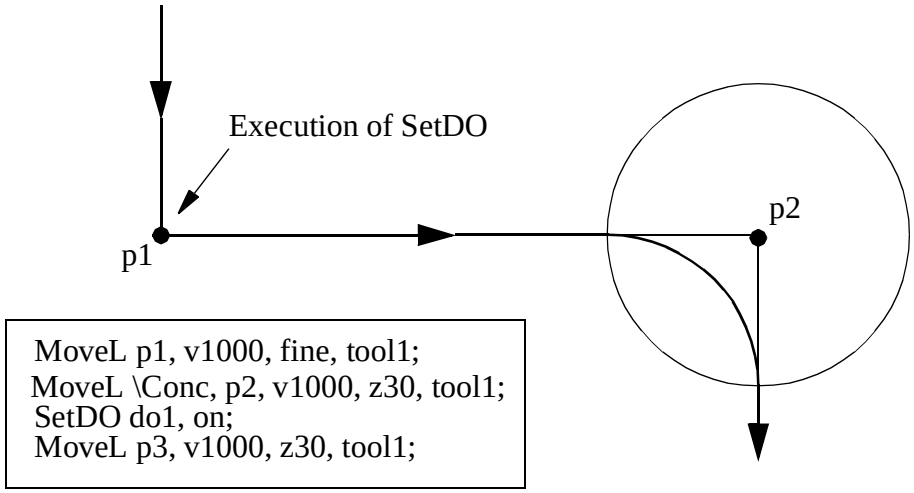


Figure 28 In the case of concurrent program execution after a stop point, a positioning instruction and subsequent logical instructions are started at the same time.

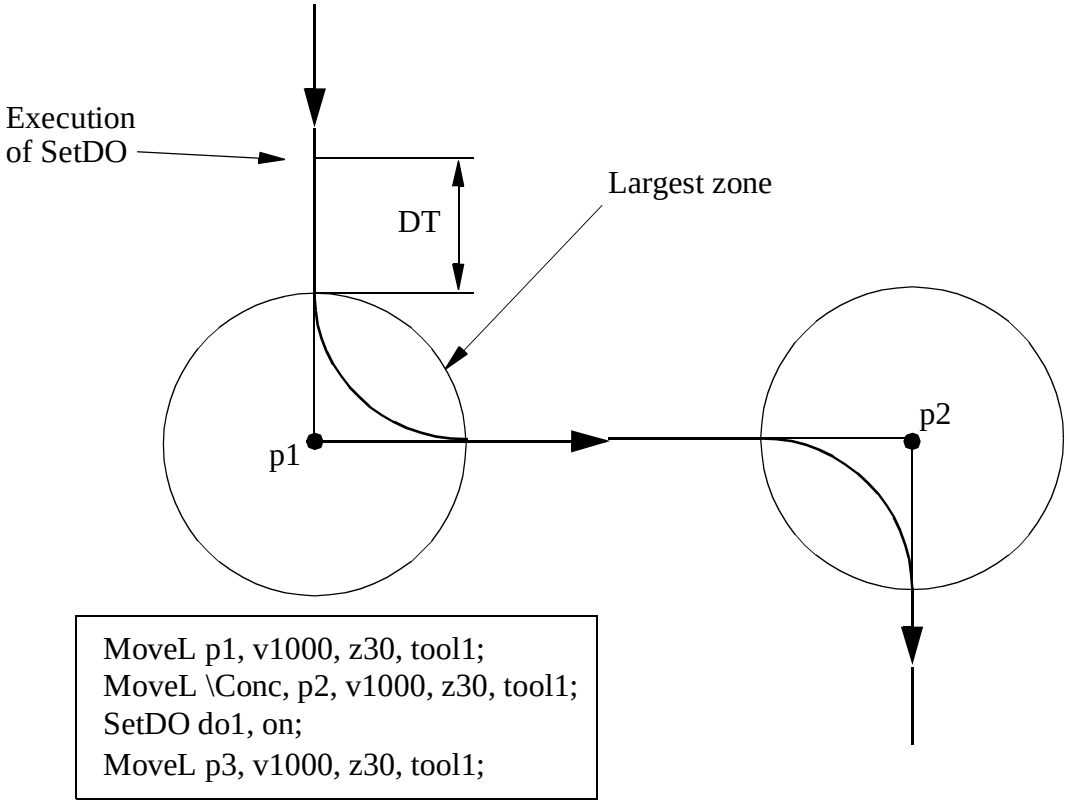


Figure 29 In the case of concurrent program execution after a fly-by point, the logical instructions start executing before the positioning instructions with the argument \Conc are started.

Instructions which indirectly affect movements, such as *ConfL* and *SingArea*, are executed in the same way as other logical instructions. They do not, however, affect the movements ordered by previous positioning instructions.

If several positioning instructions with the argument *\Conc* and several logical instructions in a long sequence are mixed, the following applies:

- Positioning instructions are executed after the previous positioning instruction has been executed.
- Logical instructions are executed directly, in the order they were programmed. This takes place at the same time as the movement (see Figure 30) meaning that logical instructions are executed at an earlier stage on the path than they were programmed.
- Movements cannot be restarted after a program has been stopped, if two or more positioning instructions in sequence are waiting to be executed.

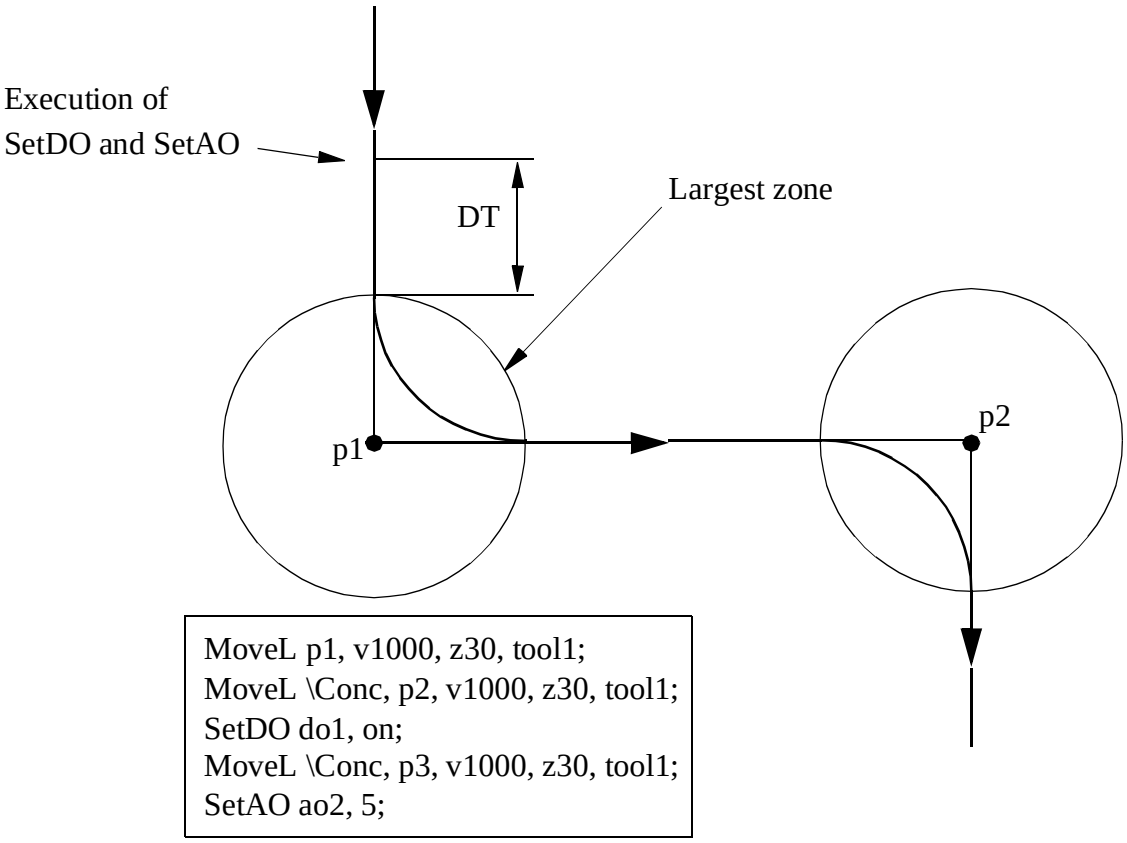


Figure 30 If several positioning instructions with the argument \Conc are programmed in sequence, all connected logical instructions are executed at the same time as the first position is executed.

During concurrent program execution, the following instructions are programmed to end the sequence and subsequently re-synchronize positioning instructions and logical instructions:

- a positioning instruction without the argument *\Conc*,
- the instruction *WaitTime* or *WaitUntil* with the argument *\Inpos*.

3.4 Path synchronization

In order to synchronize process equipment (for applications such as gluing, painting and arc welding) with the robot movements, different types of path synchronization signals can be generated.

With a so-called positions event, a trig signal will be generated when the robot passes a predefined position on the path. With a time event, a signal will be generated in a predefined time before the robot stops in a stop position. Moreover, the control system also handles weave events, which generate pulses at predefined phase angles of a weave motion.

All the position synchronized signals can be achieved both before (look ahead time) and after (delay time) the time that the robot passes the predefined position. The position is given by a programmed position and can be tuned as a path distance before the programmed position.

The accuracy is ±2 ms. On the part of the path just before the final position (max. 12 mm at 500 mm/s), the signal can however be delayed by 24 ms.

If a power failure occurs when running a Trigg instruction, the trigg condition is activated from the new starting point. This in turn means that the signal will be set incorrectly if the argument *\Start* is used. If the trigg condition had already occurred before the power failure, then the trigg event will be activated once again.

3.5 Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Definition of zone size	Data Types - *zonedata*

4 Robot Configuration

It is usually possible to attain the same robot tool position and orientation in several different ways, using different sets of axis angles. We call these different robot configurations. If, for example, a position is located approximately in the middle of a work cell, some robots can get to that position from above and from below (see Figure 31). This can also be achieved by turning the front part of the robot upper arm (axis 4) upside down while rotating axes 5 and 6 to the desired position and orientation (see Figure 32).

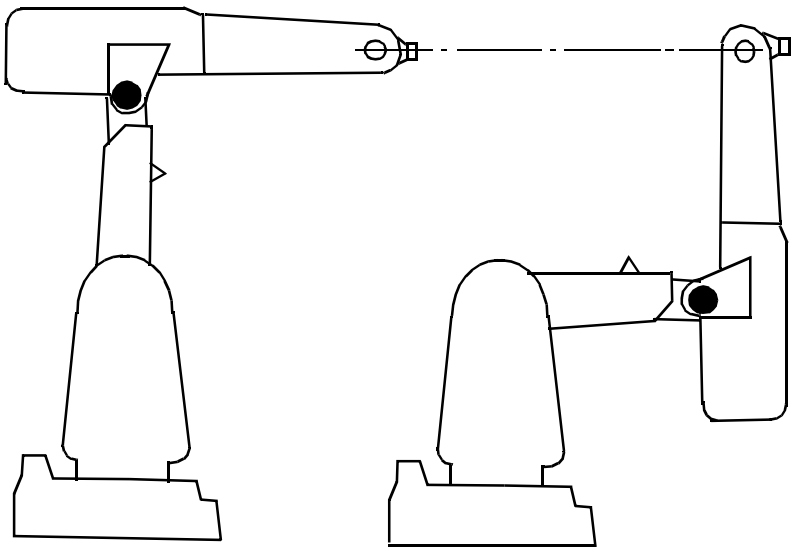


Figure 31 Two different arm configurations used to attain the same position and orientation. In one of the configurations, the arms point upwards and to attain the other configuration, axis 1 must be rotated 180 degrees.

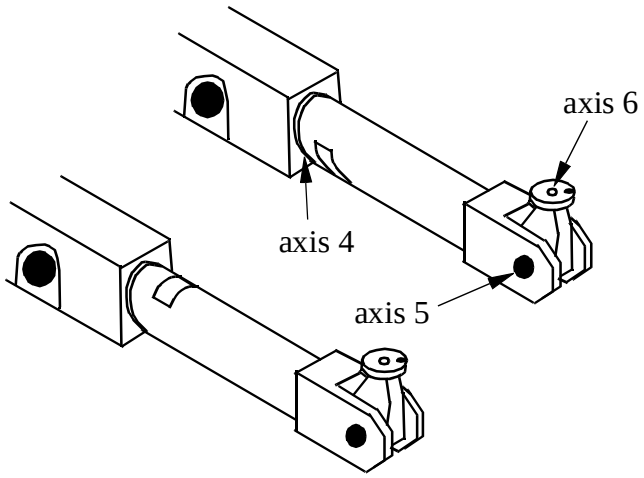


Figure 32 Two different wrist configurations used to attain the same position and orientation. In the configuration in which the front part of the upper arm points upwards (lower), axis 4 has been rotated 180 degrees, axis 5 through 180 degrees and axis 6 through 180 degrees in order to attain the configuration in which the front part of the upper arm points downwards (upper).

Usually you want the robot to attain the same configuration during program execution as the one you programmed. To do this, you can make the robot check the configuration and, if the correct configuration is not attained, program execution will stop. If the configuration is not checked, the robot may unexpectedly start to move its arms and wrists which, in turn, may cause it to collide with peripheral equipment.

The robot configuration is specified by defining the appropriate quarter revolutions of axes 1, 4 and 6. If both the robot and the programmed position have the same quarter revolution for these axes, the robot configuration is correct.

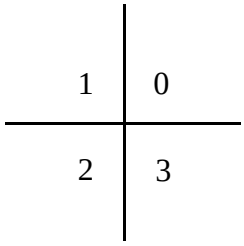


Figure 33 Quarter revolution for a positive joint angle: $int\left(joint - \frac{angle}{\pi/2}\right)$.

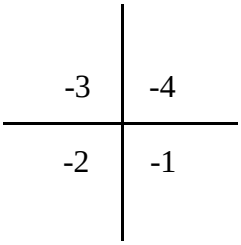


Figure 34 Quarter revolution for a negative joint angle: $int\left(joint - \frac{angle}{\pi/2} - 1\right)$.

The configuration check involves comparing the configuration of the programmed position with that of the robot.

During linear movement, the robot always moves to the closest possible configuration. If, however, the configuration check is active, program execution stops as soon as any one of the axes deviates more than the specified number of degrees.

During axis-by-axis or modified linear movement using a configuration check, the robot always moves to the programmed axis configuration. If the programmed position and orientation are not achieved, program execution stops before starting the movement. If the configuration check is not active, the robot moves to the specified position and orientation with the closest configuration.

When the execution of a programmed position is stopped because of a configuration

error, it may often be caused by one or more of the following reasons:

- The position is programmed off-line with a faulty configuration.
- The robot tool has been changed causing the robot to take another configuration than was programmed.
- The position is subject to an active frame operation (displacement, user, object, base).

The correct configuration in the destination position can be found by positioning the robot near it and reading the configuration on the teach pendant.

If the configuration parameters change because of active frame operation, the configuration check can be deactivated.

4.1 Robot configuration data for 6400C

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

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

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

Figure 35 shows an example of a forward configuration and a backward configuration giving the same position and orientation.

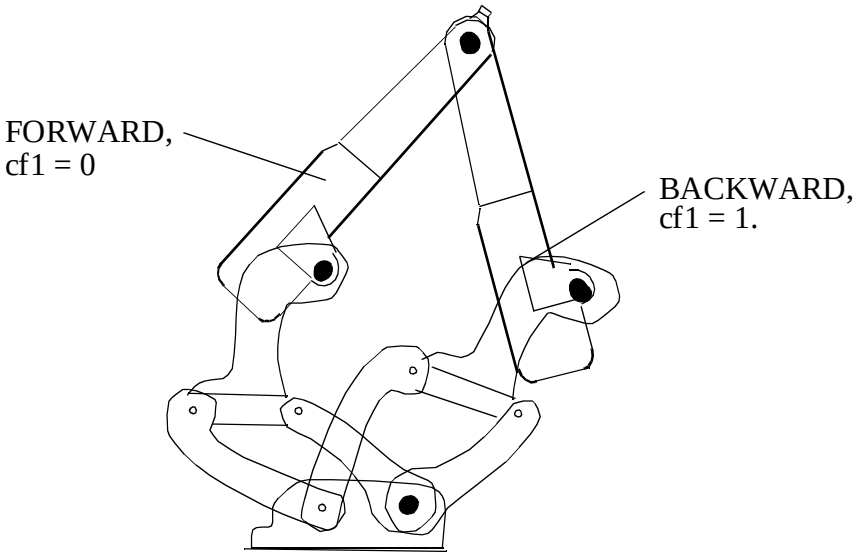


Figure 35 Same position and orientation with two different main axes configuration.

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

4.2 Related information

	Described in:
Definition of robot configuration	Data Types - *confdata*
Activating/deactivating the configuration check	RAPID Summary - *Motion Settings*

5 Singularities

Some positions in the robot working space can be attained using an infinite number of robot configurations to position and orient the tool. These positions, known as singular points (singularities), constitute a problem when calculating the robot arm angles based on the position and orientation of the tool.

Generally speaking, a robot has two types of singularities: arm singularities and wrist singularities. Arm singularities are all configurations where the wrist centre (the intersection of axes 4, 5 and 6) ends up directly above axis 1 (see Figure 36). Wrist singularities are configurations where axis 4 and axis 6 are on the same line, i.e. axis 5 has an angle equal to 0 (see Figure 37).

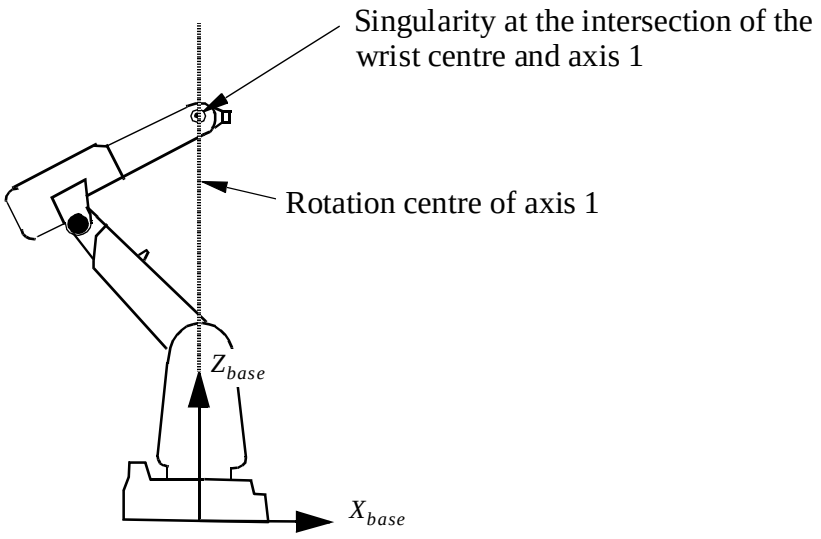


Figure 36 Arm singularity occurs where the wrist centre and axis 1 intersect.

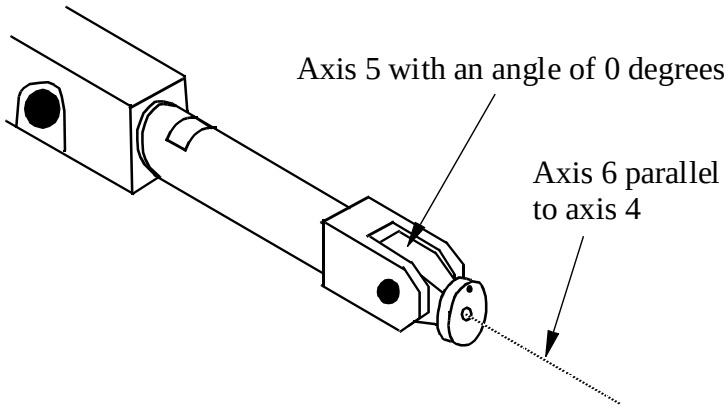


Figure 37 Wrist singularity occurs when axis 5 is 0 degrees.

Singularity points/IRB 6400C

Between the robot's working space, with the arm directed forward and the arm directed backward, there is a singularity point above the robot. There is also a singularity point on the sides of the robot. These points contain a singularity and have kinematic limitations. A position in these points cannot be specified as forward/ backward and can only be reached with *MoveAbsJ*. When the robot is in a singular point:

- It is only possible to use *MoveAbsJ* or to jog the robot axis by axis.

5.1 Program execution through singularities

During joint interpolation, the robot never has any problem passing singular points.

When executing a linear or circular path close to a singularity, the velocities in some joints (1 and 6/4 and 6) may be very high. In order not to exceed the maximum joint velocities, the linear path velocity is reduced.

The high joint velocities may be reduced by using the mode (*Sing Area\Wrist*) when the wrist axes are interpolated in joint angles while still maintaining the linear path of the robot tool. An orientation error compared to the full linear interpolation is however introduced.

Note that the robot configuration changes dramatically when the robot passes close to a singularity with linear or circular interpolation. In order to avoid the reconfiguration, the first position on the other side of the singularity should be programmed with an orientation that makes the reconfiguration unnecessary.

Also note that the robot should not be in its singularity when external joints only are moved, as this may cause robot joints to make unnecessary movements.

5.2 Jogging through singularities

During joint interpolation, the robot never has any problem passing singular points.

During linear interpolation the robot can pass singular points but at a decreased speed.

5.3 Related information

	Described in:
Controlling how the robot is to act on execution near singular points	Instructions - *SingArea*

6 I/O Principles

The robot generally has one or more I/O boards. Each of the boards has several digital and/or analog channels which must be connected to logical signals before they can be used. This is carried out in the system parameters and has usually already been done using standard names before the robot is delivered. Logical names must always be used during programming.

A physical channel can be connected to several logical names, but can also have no logical connections (see Figure 38).

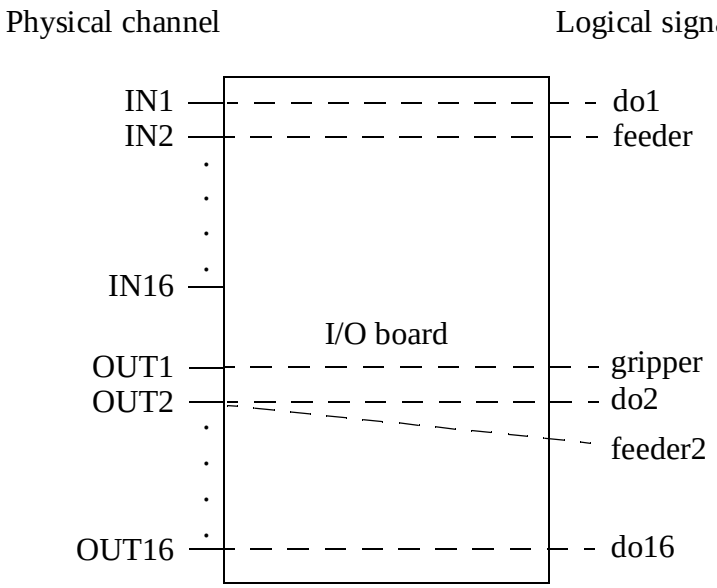


Figure 38 To be able to use an I/O board, its channels must be given logical names. In the above example, the physical output 2 is connected to two different logical names. IN16, on the other hand, has no logical name and thus cannot be used.

6.1 Signal characteristics

The characteristics of a signal are depend on the physical channel used as well as how the channel is defined in the system parameters. The physical channel determines time delays and voltage levels (see the Product Specification). The characteristics, filter times and scaling between programmed and physical values, are defined in the system parameters.

When the power supply to the robot is switched on, all signals are set to zero. They are not, however, affected by emergency stops or similar events.

An output can be set to one or zero from within the program. This can also be done using a delay or in the form of a pulse. If a pulse or a delayed change is ordered for an output, program execution continues. The change is then carried out without affecting the rest of the program execution. If, on the other hand, a new change is ordered for the same output before the given time elapses, the first change is not carried out (see Figure 39).

```
SetDO \SDelay:=1, do1;
WaitTime 0.5;
PulseDO do1;
```

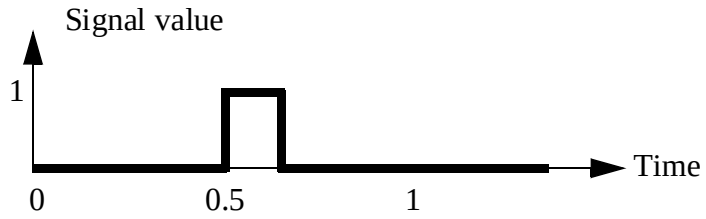


Figure 39 The instruction SetDO is not carried out at all because a new command is given before the time delay has elapsed.

6.2 System signals

Logical signals can be interconnected by means of special system functions. If, for example, an input is connected to the system function *Start*, a program start is automatically generated as soon as this input is enabled. These system functions are generally only enabled in automatic mode. For more information, see Chapter 9, System Parameters, or the chapter on *Installation and Commissioning - PLC Communication* in the Product Manual.

6.3 Cross connections

Digital signals can be interconnected in such a way that they automatically affect one another:

- An output signal can be connected to one or more input or output signals.
- An input signal can be connected to one or more input or output signals.
- If the same signal is used in several cross connections, the value of that signal is the same as the value that was last enabled (changed).
- Cross connections can be interlinked, in other words, one cross connection can affect another. They must not, however, be connected in such a way so as to form a ”vicious circle”, e.g. cross-connecting *di1* to *di2* whilst *di2* is cross-connected to *di1*.
- If there is a cross connection on an input signal, the corresponding physical connection is automatically disabled. Any changes to that physical channel will thus not be detected.

- Pulses or delays are not transmitted over cross connections.
- Logical conditions can be defined using NOT, AND and OR (Option: Advanced functions).

Examples:

- *di2=di1*
- *di3=di2*
- *do4=di2*

If *di1* changes, *di2*, *di3* and *do4* will be changed to the corresponding value.

- *do8=do7*
- *do8=di5*

If *do7* is set to 1, *do8* will also be set to 1. If *di5* is then set to 0, *do8* will also be changed (in spite of the fact that *do7* is still 1).

- *do5=di6* and *do1*

Do5 is set to 1 if both *di6* and *do1* is set to 1.

6.4 Limitations

A maximum of 10 signals can be pulsed at the same time and a maximum of 20 signals can be delayed at the same time.

6.5 Related information

	Described in:
Definition of I/O boards and signals	User's Guide - System Parameters
Instructions for handling I/O	RAPID Summary - *Input and Output Signals*
Manual manipulation of I/O	User's Guide - Inputs and Outputs

bool Logical values

Bool is used for logical values (true/false).

Description

The value of data of the type *bool* can be either *TRUE* or *FALSE*.

Examples

```
flag1 := TRUE;
```

flag is assigned the value TRUE.

```
VAR bool highvalue;
VAR num reg1;
   .
highvalue := reg1 > 100;
```

highvalue is assigned the value *TRUE* if *reg1 is greater than 100*; otherwise, *FALSE* is assigned.

```
IF highvalue Set do1;
```

The *do1* signal is set if *highvalue* is *TRUE*.

```
highvalue := reg1 > 100;
mediumvalue := reg1 > 20 AND NOT highvalue;
```

mediumvalue is assigned the value *TRUE* if *reg1* is between *20* and *100*.

Related information

	Described in:
Logical expressions	Basic Characteristics - *Expressions*
Operations using logical values	Basic Characteristics - *Expressions*

clock Time measurement

Clock is used for time measurement. A *clock* functions like a stopwatch used for timing.

Description

Data of the type *clock* stores a time measurement in seconds and has a resolution of 0.01 seconds.

Example

```
VAR clock clock1;

    ClkReset clock1;
```

The clock, *clock1*, is declared and reset. Before using *ClkReset*, *ClkStart*, *ClkStop* and *ClkRead*, you must declare a variable of data type *clock* in your program.

Limitations

The maximum time that can be stored in a clock variable is approximately 49 days (4,294,967 seconds). The instructions *ClkStart*, *ClkStop* and *ClkRead* report clock overflows in the very unlikely event that one occurs.

A clock must be declared as a *VAR* variable type, not as a *persistent* variable type.

Characteristics

Clock is a non-value data type and cannot be used in value-oriented operations.

Related Information

	Described in:
Summary of Time and Date Instructions	RAPID Summary - *System & Time*
Non-value data type characteristics	Basic Characteristics - *Data Types*

confdata Robot configuration data

Confdata is used to define the axis configurations of the robot.

Description

All positions of the robot are defined and stored using rectangular coordinates. When calculating the corresponding axis positions, there will often be two or more possible solutions. This means that the robot is able to achieve the same position, i.e. the tool is in the same position and with the same orientation, with several different positions or configurations of the robots axes.

To unambiguously denote one of these possible configurations, the robot configuration is specified using three axis values. These values define the current quadrant of axis 1, axis 4 and axis 6. The quadrants are numbered 0, 1, 2, etc. (they can also be negative). The quadrant number is connected to the current joint angle of the axis. For each axis, quadrant 0 is the first quarter revolution, 0-90°, in a positive direction from the zero position; quadrant 1 is the next revolution, 90-180°, etc. Quadrant -1 is the revolution 0° - (-90°), etc. (see Figure 1).

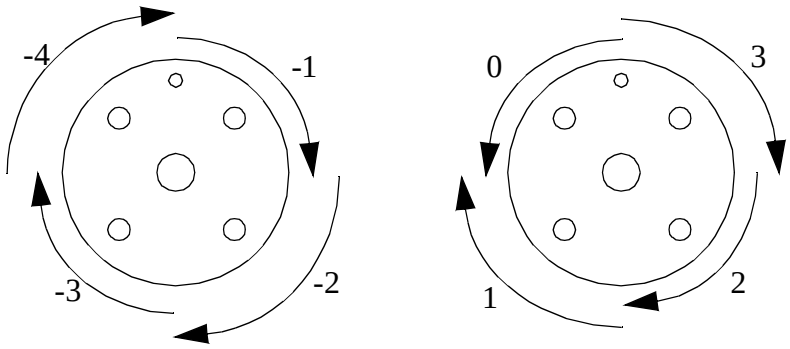


Figure 1 The configuration quadrants for axis 6.

Robot configuration data for 6400C

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

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

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

Figure 2 shows an example of a forward configuration and a backward configuration giving the same position and orientation.

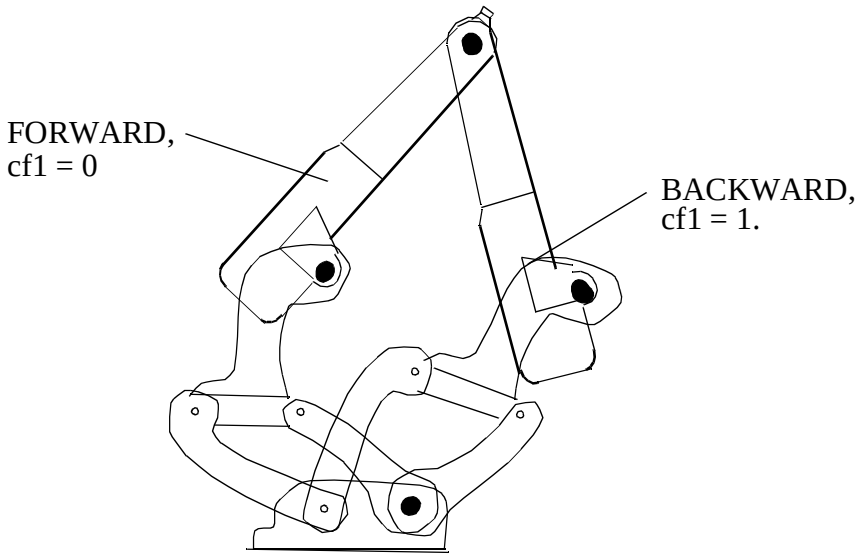


Figure 2 Same position and orientation with two different main axes configurations.

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

Components

cf1 Data type: *num*

The current quadrant of axis 1, expressed as a positive or negative integer.

cf4 Data type: *num*

The current quadrant of axis 4, expressed as a positive or negative integer.

cf6 Data type: *num*

The current quadrant of axis 6, expressed as a positive or negative integer.

cfx Data type: *num*

This component is not used at present.

Example

VAR confdata conf15 := [1, -1, 0, 0]

A robot configuration *conf15* is defined as follows:

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

Structure

< dataobject of *confdata* >
  < *cf1* of *num* >
  < *cf4* of *num* >
  < *cf6* of *num* >
  < *cf*x of *num* >

Related information

	Described in:
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Handling configuration data	Motion and I/O Principles - *Robot Configuration*

dionum Digital values 0 - 1

Dionum (digital input output numeric) is used for digital values (0 or 1).

This data type is used in conjunction with instructions and functions that handle digital input or output signals.

Description

Data of the type *dionum* represents a digital value 0 or 1.

Examples

CONST dionum close := 1;

Definition of a constant *close* with a value equal to *1*.

SetDO grip1, close;

The signal *grip1* is set to *close*, i.e. 1.

Error handling

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

Characteristics

Dionum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Summary input/output instructions	RAPID Summary - *Input and Output Signals*
Configuration of I/O	User's Guide - *System Parameters*
Alias data types	Basic Characteristics- *Data Types*

errnum Error number

Errnum is used to describe all recoverable (non fatal) errors that occur during program execution, such as division by zero.

Description

If the robot detects an error during program execution, this can be dealt with in the error handler of the routine. Examples of such errors are values that are too high and division by zero. The system variable *ERRNO*, of type *errnum*, is thus assigned different values depending on the nature of an error. The error handler may be able to correct an error by reading this variable and then program execution can continue in the correct way.

An error can also be created from within the program using the RAISE instruction. This particular type of error can be detected in the error handler by specifying an error number as an argument to RAISE.

Examples

```
reg1 := reg2 / reg3;
.
ERROR
   IF ERRNO = ERR_DIVZERO THEN
      reg3 := 1;
      RETRY;
   ENDIF
```

If *reg3* = 0, the robot detects an error when division is taking place. This error, however, can be detected and corrected by assigning *reg3* the value *1*. Following this, the division can be performed again and program execution can continue.

```
CONST errnum machine_error := 1;
.
IF di1=0 RAISE machine_error;
.
ERROR
   IF ERRNO=machine_error RAISE;
```

An error occurs in a machine (detected by means of the input signal *di1*). A jump is made to the error handler in the routine which, in turn, calls the error handler of the calling routine where the error may possibly be corrected. The constant, *machine_error*, is used to let the error handler know exactly what type of error has occurred.

Predefined data

The system variable ERRNO can be used to read the latest error that occurred. A number of predefined constants can be used to determine the type of error that has occurred.

Name	Cause of error
ERR_ALRDYCNT	The interrupt variable is already connected to a TRAP routine
ERR_ARGDUPCND	More than one present conditional argument for the same parameter
ERR_ARGNOTPER	Argument is not a persistent reference
ERR_ARGNOTVAR	Argument is not a variable reference
ERR_AXIS_ACT	Axis is not active
ERR_AXIS_IND	Axis is not independent
ERR_AXIS_MOVING	Axis is moving
ERR_AXIS_PAR	Parameter axis in instruction TestSign and SetCurrRef is wrong.
ERR_CALLPROC	Procedure call error at runtime (late binding)
ERR_CNTNOTVAR	CONNECT target is not a variable reference
ERR_DEV_MAXTIME	Timeout when executing a ReadBin, ReadNum or a ReadStr instruction
ERR_DIVZERO	Division by zero
ERR_EXCRTYMAX	Max. number of retries exceeded
ERR_EXECPHR	An attempt was made to execute an instruction using a place holder
ERR_FILEACC	A file is accessed incorrectly
ERR_FILNOTFND	File not found
ERR_FILEOPEN	A file cannot be opened
ERR_FNCNORET	No return value
ERR_ILLDIM	Incorrect array dimension
ERR_ILLQUAT	Attempt to use illegal orientation (quaternion) valve
ERR_ILLRAISE	Error number in RAISE out of range
ERR_INOMAX	No more interrupt numbers available
ERR_MAXINTVAL	The integer value is too large
ERR_NEGARG	Negative argument is not allowed
ERR_NOTARR	Data is not an array
ERR_NOTEQDIM	The array dimension used when calling the routine does not coincide with its parameters
ERR_NOTINTVAL	Not an integer value
ERR_NOTPRES	A parameter is used, despite the fact that the corresponding argument was not used at the routine call
ERR_OUTOFBND	The array index is outside the permitted limits
ERR_REFUNKDAT	Reference to unknown entire data object

ERR_REFUNKFUN	Reference to unknown function
ERR_REFUNKPRC	Reference to unknown procedure
ERR_REFUNKTRP	Reference to unknown trap
ERR_PATHDIST	Too long regain distance for StartMove instruction
ERR_RCVDATA	An attempt was made to read non numeric data with ReadNum
ERR_SC_WRITE	Error when sending to external computer
ERR_STEP_PAR	Parameter Step in SetCurrRef is wrong
ERR_STRTOOLNG	The string is too long
ERR_TP_DIBREAK	A TPRead instruction was interrupted by a digital input
ERR_TP_MAXTIME	Timeout when executing a TPRead instruction
ERR_UNIT_PAR	Parameter Mech_unit in TestSign and SetCurrRef is wrong
ERR_UNKINO	Unknown interrupt number
ERR_UNLOAD	Unload error
ERR_WAIT_MAXTIME	Timeout when executing a WaitDI or WaitUntil instruction
ERR_WHLSEARCH	No search stop

Characteristics

Errnum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Error recovery	RAPID Summary - *Error Recovery* Basic Characteristics - *Error Recovery*
Data types in general, alias data types	Basic Characteristics - *Data Types*

extjoint Position of external joints

Extjoint is used to define the axis positions of external axes, positioners or workpiece manipulators.

Description

The robot can control up to six external axes in addition to its six internal axes, i.e. a total of twelve axes. The six external axes are logically denoted: a, b, c, d, e, f. Each such logical axis can be connected to a physical axis and, in this case, the connection is defined in the system parameters.

Data of the type *extjoint* is used to hold position values for each of the logical axes a - f.

For each logical axis connected to a physical axis, the position is defined as follows:

- For rotating axes – the position is defined as the rotation in degrees from the calibration position.
- For linear axes – the position is defined as the distance in mm from the calibration position.

If a logical axis is not connected to a physical one, the value 9E9 is used as a position value, indicating that the axis is not connected. At the time of execution, the position data of each axis is checked and it is checked whether or not the corresponding axis is connected. If the stored position value does not comply with the actual axis connection, the following applies:

- If the position is not defined in the position data (value is 9E9), the value will be ignored if the axis is connected and not activated. But if the axis is activated, it will result in an error.
- If the position is defined in the position data, although the axis is not connected, the value will be ignored.

If an external axis offset is used (instruction *EOffsOn* or *EOffsSet*), the positions are specified in the ExtOffs coordinate system.

Components

eax_a *(external axis a)* Data type: *num*

The position of the external logical axis “a”, expressed in degrees or mm (depending on the type of axis).

eax_b *(external axis b)* Data type: *num*

The position of the external logical axis “b”, expressed in degrees or mm (depending on the type of axis).

...

eax_f *(external axis f)* Data type: *num*

The position of the external logical axis “f”, expressed in degrees or mm (depending on the type of axis).

Example

VAR extjoint axpos10 := [11, 12.3, 9E9, 9E9, 9E9, 9E9] ;

The position of an external positioner, *axpos10,* is defined as follows:

- The position of the external logical axis “a” is set to 11, expressed in degrees or mm (depending on the type of axis).
- The position of the external logical axis “b” is set to 12.3, expressed in degrees or mm (depending on the type of axis).
- Axes c to f are undefined.

Structure

< dataobject of *extjoint* >
< *eax_a* of *num* >
< *eax_b* of *num* >
< *eax_c* of *num* >
< *eax_d* of *num* >
< *eax_e* of *num* >
< *eax_f* of *num* >

Related information

	Described in:
Position data	Data Types - *robtarget*
ExtOffs coordinate system	Instructions - *EOffsOn*

intnum Interrupt identity

Intnum (interrupt numeric) is used to identify an interrupt.

Description

When a variable of type *intnum* is connected to a trap routine, it is given a specific value identifying the interrupt. This variable is then used in all dealings with the interrupt, such as when ordering or disabling an interrupt.

More than one interrupt identity can be connected to the same trap routine. The system variable *INTNO* can thus be used in a trap routine to determine the type of interrupt that occurs.

Examples

```
VAR intnum feeder_error;
.
CONNECT feeder_error WITH correct_feeder;
ISignalDI di1, 1, feeder_error;
```

An interrupt is generated when the input *di1* is set to *1*. When this happens, a call is made to the *correct_feeder* trap routine.

```
VAR intnum feeder1_error;
VAR intnum feeder2_error;
.
PROC init_interrupt();
.
   CONNECT feeder1_error WITH correct_feeder;
   ISignalDI di1, 1, feeder1_error;
   CONNECT feeder2_error WITH correct_feeder;
   ISignalDI di2, 1, feeder2_error;
.
ENDPROC
.
TRAP correct_feeder
   IF INTNO=feeder1_error THEN
   .
   ELSE
   .
   ENDIF
.
ENDTRAP
```

An interrupt is generated when either of the inputs *di1* or *di2* is set to *1*. A call is then made to the *correct_feeder* trap routine. The system variable INTNO is used in the trap routine to find out which type of interrupt has occurred.

Characteristics

Intnum is an alias data type for *num* and thus inherits its properties.

Related information

	Described in:
Summary of interrupts	RAPID Summary - *Interrupts*
Alias data types	Basic Characteristics- *Data Types*

iodev Serial channels and files

Iodev (I/O device) is used for serial channels, such as printers and files.

Description

Data of the type *iodev* contains a reference to a file or serial channel. It can be linked to the physical unit by means of the instruction *Open* and then used for reading and writing.

Example

```
VAR iodev file;
.
Open "flp1:LOGDIR/INFILE.DOC", file\Read;
input := ReadNum(file);
```

The file *INFILE.DOC* is opened for reading. When reading from the file, *file* is used as a reference instead of the file name.

Characteristics

Iodev is a non-value data type.

Related information

	Described in:
Communication via serial channels	RAPID Summary - *Communication*
Configuration of serial channels	User's Guide - *System Parameters*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

jointtarget Joint position data

Jointtarget is used to define the position that the robot and the external axes will move to with the instruction *MoveAbsJ*.

Description

Jointtarget defines each individual axis position, for both the robot and the external axes.

Components

robax *(robot axes)* Data type: *robjoint*

Axis positions of the robot axes in degrees.

Axis position is defined as the rotation in degrees for the respective axis (arm) in a positive or negative direction from the axis calibration position.

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

The position is defined as follows for each individual axis (*eax_a, eax_b ... eax_f*):

- For rotating axes, the position is defined as the rotation in degrees from the calibration position.
- For linear axes, the position is defined as the distance in mm from the calibration position.

External axes *eax_a* ... are logical axes. How the logical axis number and the physical axis number are related to each other is defined in the system parameters.

The value 9E9 is defined for axes which are not connected. If the axes defined in the position data differ from the axes that are actually connected on program execution, the following applies:

- If the position is not defined in the position data (value 9E9) the value will be ignored, if the axis is connected and not activated. But if the axis is activated it will result in error.
- If the position is defined in the position data yet the axis is not connected, the value is ignored.

Examples

```
CONST jointtarget calib_pos := [ [ 0, 0, 0, 0, 0, 0], [ 0, 9E9, 9E9, 9E9, 9E9, 9E9] ];
```

The normal calibration position for IRB2400 is defined in *calib_pos* by the data type *jointtarget*. The normal calibration position 0 (degrees or mm) is also defined for the external logical axis a. The external axes b to f are undefined.

Structure

```
< dataobject of jointtarget >
   < robax of robjoint >
      < rax_1 of num >
      < rax_2 of num >
      < rax_3 of num >
      < rax_4 of num >
      < rax_5 of num >
      < rax_6 of num >
   < extax of extjoint >
      < eax_a of num >
      < eax_b of num >
      < eax_c of num >
      < eax_d of num >
      < eax_e of num >
      < eax_f of num >
```

Related information

	Described in:
Move to joint position	Instructions - *MoveAbsJ*
Positioning instructions	RAPID Summary - *Motion*
Configuration of external axes	User's Guide - *System Parameters*

loaddata Load data

Loaddata is used to describe loads attached to the mechanical interface of the robot (the robot's mounting flange).

Load data usually defines the payload of the robot, i.e. the load held in the robot gripper. The tool load is specified in the tool data (*tooldata*) which includes load data.

Description

Specified loads are used to set up a model of the dynamics of the robot so that the robot movements can be controlled in the best possible way.

If incorrect load data is specified, it often has the following consequences:

- If the specified load data is greater than the value of the true load;
  -> The robot will not be used to its maximum capacity
  -> Impaired path accuracy including running the risk of overshooting.
- If the specified load data is less than the value of the true load;
  -> Impaired path accuracy including running the risk of overshooting.

The payload is connected/disconnected using the instruction *GripLoad*.

Components

mass Data type: *num*

The weight of the load in kg.

cog *(centre of gravity)* Data type: *pos*

The centre of gravity of the load (x, y and z) in mm.

The centre of gravity of a payload is defined using the tool coordinate system (the object coordinate system when a stationary tool is used).

The centre of gravity of a tool is defined using the wrist coordinate system.

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

The orientation of the axes of moment of the load at the centre of gravity, expressed as a quaternion (q1, q2, q3 and q4).

The axes of moment of a payload are defined using the tool coordinate system, with the origin (zero) located at the centre of gravity of the load (see Figure 1) – the object coordinate system when a stationary tool is used. If the orientation is defined as q1=1 and q2 ... q4=0, this means that the axes of moment are parallel to the tool coordinate axes.

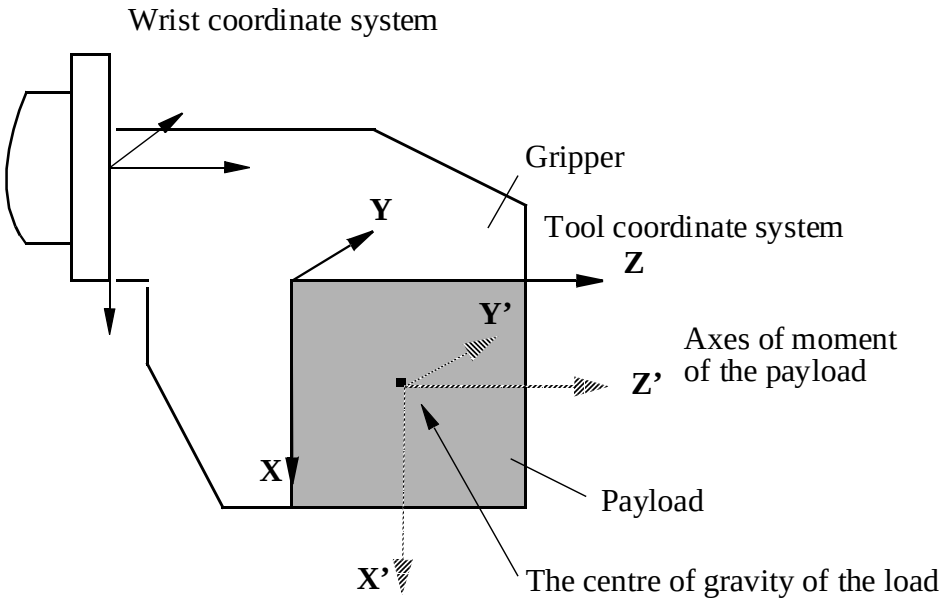


Figure 1 The centre of gravity of the payload and axes of moment (the axes of moment are parallel to the tool coordinate axes in this example).

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

The moment of inertia of the load around the x'-axis, expressed in kgm^2.

The orientation of the axes of moment and the moment of inertia do not generally need to be defined. If, however, the moment of inertia is defined correctly, this will increase the level of accuracy of the path when handling large sheets of metal, etc. If the moment of inertia of all of the components *ix*, *iy* and *iz* is equal to 0 kgm², this implies a spot load.

The moment of inertia is calculated using various formulae depending on the shape of the load and, due to lack of space, is not described here (see other sources of reference). Normally, the moment of inertia must only be defined when the distance from the mounting flange to the centre of gravity is less than the dimension of the load (see Figure 2).

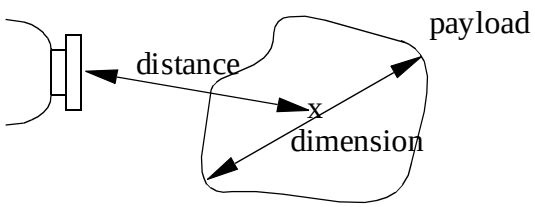


Figure 2 The moment of inertia must normally be defined when the distance is less than the dimension.

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

The moment of inertia of the load around the y'-axis, expressed in kgm^2.

For more information, see *ix*.

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

The moment of inertia of the load around the z'-axis, expressed in kgm^2.

For more information, see *ix*.

Examples

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

The payload in Figure 1 is described using the following values:

- Weight *5* kg.
- The centre of gravity is x = *50*, y = *0* and z = *50* mm in the tool coordinate system.
- The payload is a spot load.

```
Set gripper;
WaitTime 0.3;
GripLoad piece1;
```

Connection of the payload, *piece1*, specified at the same time as the robot grips the load *piece1*.

```
Reset gripper;
WaitTime 0.3;
GripLoad load0;
```

Disconnection of a payload, specified at the same time as the robot releases a payload.

Limitations

The payload should only be defined as a persistent variable (PERS) and not within a routine. Current values are then saved when storing the program on diskette and are retrieved on loading.

Arguments of the type load data in the *GripLoad* instruction should only be an entire persistent (not array element or record component).

Predefined data

The load *load0* defines a payload, the weight of which is equal to 0 kg, i.e. no load at all. This load is used as the argument in the instruction *GripLoad* to disconnect a payload.

The load *load0* can always be accessed from the program, but cannot be changed (it is stored in the system module *BASE*).

PERS loaddata load0 := [0, [0, 0, 0], [1, 0, 0, 0],0, 0 ,0];

Structure

< dataobject of *loaddata* >
< *mass* of *num* >
< *cog* of *pos* >
< *x* of *num* >
< *y* of *num* >
< *z* of *num* >
< *aom* of *orient* >
< *q1* of *num* >
< *q2* of *num* >
< *q3* of *num* >
< *q4* of *num* >
< *ix* of *num* >
< *iy* of *num* >
< *iz* of *num* >

Related information

	Described in:
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Definition of tool loads	Data Types - *tooldata*
Activation of payloads	Instructions - *GripLoad*

mecunit Mechanical unit

Mecunit is used to define the different mechanical units which can be controlled and accessed from the robot and the program.

The names of the mechanical units are defined in the system parameters and, consequently, must not be defined in the program.

Description

Data of the type *mecunit* only contains a reference to the mechanical unit.

Limitations

Data of the type *mecunit* must not be defined in the program. The data type can, on the other hand, be used as a parameter when declaring a routine.

Predefined data

The mechanical units defined in the system parameters can always be accessed from the program (installed data).

Characteristics

Mecunit is a *non-value* data type. This means that data of this type does not permit value-oriented operations.

Related information

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

motsetdata Motion settings data

Motsetdata is used to define a number of motion settings that affect all positioning instructions in the program:

- Max. velocity and velocity override
- Acceleration data
- Behaviour around singular points
- Management of different robot configurations
- Payload

This data type does not normally have to be used since these settings can only be set using the instructions *VelSet*, *AccSet*, *SingArea*, *ConfJ*, *ConfL*, and *GripLoad*.

The current values of these motion settings can be accessed using the system variable *C_MOTSET*.

Description

The current motion settings (stored in the system variable *C_MOTSET*) affect all movements.

Components

vel.oride Data type: *veldata/num*

Velocity as a percentage of programmed velocity.

vel.max Data type: *veldata/num*

Maximum velocity in mm/s.

acc.acc Data type: *accdata/num*

Acceleration and deceleration as a percentage of the normal values.

acc.ramp Data type: *accdata/num*

The rate by which acceleration and deceleration increases as a percentage of the normal values.

sing.wrist Data type: *singdata/bool*

The orientation of the tool is allowed to deviate somewhat in order to prevent wrist singularity.

sing.arm Data type: *singdata/bool*

The orientation of the tool is allowed to deviate somewhat in order to prevent arm singularity.

sing.base Data type: *singdata/bool*

The orientation of the tool is not allowed to deviate.

conf.jsup Data type: *confsupdata/bool*

Supervision of joint configuration is active during joint movement.

conf.lsup Data type: *confsupdata/bool*

Supervision of joint configuration is active during linear and circular movement.

conf.ax1 Data type: *confsupdata/num*

Maximum permitted deviation in degrees for axis 1 (not used in this version).

conf.ax4 Data type: *confsupdata/num*

Maximum permitted deviation in degrees for axis 4 (not used in this version).

conf.ax6 Data type: *confsupdata/num*

Maximum permitted deviation in degrees for axis 6 (not used in this version).

grip.load Data type:*gripdata/loaddata*

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

grip.acc Data type:*gripdata/num*

Acceleration data for current load (not used in this version).

Limitations

One and only one of the components *sing.wrist*, *sing.arm* or *sing.base* may have a value equal to TRUE.

Example

```
IF C_MOTSET.vel.oride > 50 THEN
    ...
ELSE
    ...
ENDIF
```

Different parts of the program are executed depending on the current velocity override.

Predefined data

C_MOTSET describes the current motion settings of the robot and can always be accessed from the program (installed data). *C_MOTSET*, on the other hand, can only be changed using a number of instructions, not by assignment.

The following default values for motion parameters are set

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

```
PERS motsetdata C_MOTSET := [
   [ 100, 500 ],                          -> veldata
   [ 100, 100 ],                          -> accdata
   [ FALSE, FALSE, TRUE ],                -> singdata
   [ TRUE, TRUE, 30, 45, 90],             -> confsupdata
   [ [ 0, [ 0, 0, 0 ], [ 1, 0, 0, 0 ], 0, 0, 0 ], -1 ]   -> gripdata
   ]
```

Structure

<dataobject of *motsetdata*>
- <*vel* of *veldata* > -> Affect by instruction VelSet
  - < *oride* of *num* >
  - < *max* of *num* >
- <*acc* of *accdata* > -> Affect by instruction AccSet
  - < *acc* of *num* >
  - < *ramp* of *num* >
- <*sing* of *singdata* > -> Affect by instruction SingArea
  - < *wrist* of *bool* >
  - < *arm* of *bool* >
  - < *base* of *bool* >
- <*conf* of *confsupdata* > -> Affect by instructions ConfJ and ConfL
  - < *jsup* of *bool* >
  - <*lsup* of *bool* >
  - < *ax1* of *num* >
  - < *ax4* of *num* >
  - < *ax6* of *num* >
- <*grip* of *gripdata* > -> Affect by instruction GripLoad
  - < *load* of *loaddata* >
    - < *mass* of *num*>
    - < *cog* of *pos* >
      - < *x* of *num* >
      - < *y* of *num* >
      - < *z* of *num* >
    - <*aom* of *orient* >
      - < *q1* of *num* >
      - < *q2* of *num* >
      - < *q3* of *num* >
      - < *q4* of *num* >
    - < *ix* of *num* >
    - < *iy* of *num*>
    - < *iz* of *num* >
  - < *acc* of *num* >

Related information

	Described in:
Instructions for setting motion parameters	RAPID Summary - *Motion Settings*

num Numeric values (registers)

Num is used for numeric values; e.g. counters.

Description

The value of the *num* data type may be

- an integer; e.g. -5,
- a decimal number; e.g. 3.45.

It may also be written exponentially; e.g.2E3 (= $2*10^3$ = 2000), 2.5E-2 (= 0.025).

Integers between -8388607 and +8388608 are always stored as exact integers.

Decimal numbers are only approximate numbers and should not, therefore, be used in *is equal to* or *is not equal to* comparisons. In the case of divisions, and operations using decimal numbers, the result will also be a decimal number; i.e. not an exact integer.

E.g.

```
a := 10;
b := 5;
IF a/b=2 THEN
...
```

As the result of a/b is not an integer, this condition is not necessarily satisfied.

Example

```
VAR num reg1;
  .
reg1 := 3;
```

reg1 is assigned the value *3*.

```
a := 10 DIV 3;
b := 10 MOD 3;
```

Integer division where *a* is assigned an integer (=3) and *b* is assigned the remainder (=1).

Predefined data

The constant pi (π) is already defined in the system module *BASE*.

```
CONST num pi := 3.1415926;
```

Related information

	Described in:
Numeric expressions	Basic Characteristics - *Expressions*
Operations using numeric values	Basic Characteristics - *Expressions*

o_jointtarget Original joint position data

o_jointtarget (original joint target) is used in combination with the function *Absolute Limit Modpos*. When this function is used to modify a position, the original position is stored as a data of the type *o_jointtarget*.

Description

If the function *Absolute Limit Modpos* is activated and a named position in a movement instruction is modified with the function *Modpos*, then the original programmed position is saved.

Example of a program before *Modpos*:

```
CONST jointtarget jpos40     := [[0, 0, 0, 0, 0, 0],
                                 [0, 9E9, 9E9, 9E9, 9E9, 9E9]];
...
MoveAbsJ jpos40, v1000, z50, tool1;
```

The same program after *ModPos* in which the point *jpos40 is* corrected to 2 degrees for robot axis 1:

```
CONST jointtarget jpos40        := [[2, 0, 0, 0, 0, 0],
                                    [0, 9E9, 9E9, 9E9, 9E9, 9E9]];
CONST o_jointtarget o_jpos40 := [[0, 0, 0, 0, 0, 0],
                                    [0, 9E9, 9E9, 9E9, 9E9, 9E9]];
...
MoveAbsJ jpos40, v1000, z50, tool1;
```

The original programmed point has now been saved in *o_jpos40* (by the data type *o_jointtarget*) and the modified point saved in *jpos40* (by the data type *jointtarget*).

By saving the original programmed point, the robot can monitor that further *Modpos* of the point in question are within the acceptable limits from the original programmed point.

The fixed name convention means that an original programmed point with the name *xxxxx* is saved with the name *o_xxxxx* by using *Absolute Limit Modpos*.

Components

robax *(robot axes)* Data type: *robjoint*

Axis positions of the robot axes in degrees.

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

Structure

```
< dataobject of o_jointtarget >
   < robax of robjoint>
      < rax_1 of num >
      < rax_2 of num >
      < rax_3 of num >
      < rax_4 of num >
      < rax_5 of num >
      < rax_6 of num >
   < extax of extjoint >
      < eax_a of num >
      < eax_b of num >
      < eax_c of num >
      < eax_d of num >
      < eax_e of num >
      < eax_f of num >
```

Related information

	Described in:
Position data	Data Types - *Jointtarget*
Configuration of Limit Modpos	User's Guide - *System Parameters*

orient Orientation

Orient is used for orientations (such as the orientation of a tool) and rotations (such as the rotation of a coordinate system).

Description

The orientation is described in the form of a quaternion which consists of four elements: *q1*, *q2*, *q3* and *q4*. For more information on how to calculate these, see below.

Components

q1 Data type: *num*

Quaternion 1.

q2 Data type: *num*

Quaternion 2.

q3 Data type: *num*

Quaternion 3.

q4 Data type: *num*

Quaternion 4.

Example

```
VAR orient orient1;
.
orient1 := [1, 0, 0, 0];
```

The *orient1* orientation is assigned the value q1=1, q2-q4=0; this corresponds to no rotation.

Limitations

The orientation must be normalised; i.e. the sum of the squares must equal 1:

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

What is a Quaternion?

The orientation of a coordinate system (such as that of a tool) can be described by a rotational matrix that describes the direction of the axes of the coordinate system in relation to a reference system (see Figure 1).

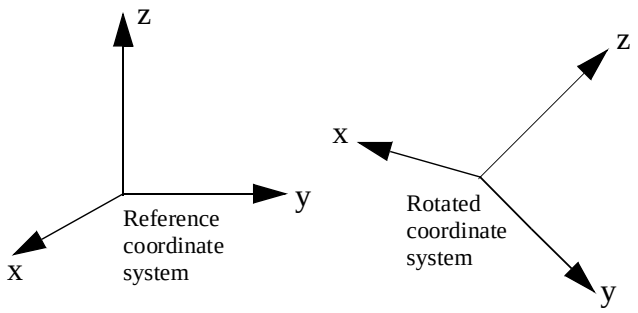


Figure 1 The rotation of a coordinate system is described by a quaternion.

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

$\mathbf{x} = (x_1, x_2, x_3)$

$\mathbf{y} = (y_1, y_2, y_3)$

$\mathbf{z} = (z_1, z_2, z_3)$

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

These three vectors can be put together in a matrix, a rotational matrix, where each of the vectors form one of the columns:

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

A quaternion is just a more concise way to describe this rotational matrix; the quaternions are calculated based on the elements of the rotational matrix:

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

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

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

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

Example 1

A tool is orientated so that its Z'-axis points straight ahead (in the same direction as the X-axis of the base coordinate system). The Y'-axis of the tool corresponds to the Y-axis of the base coordinate system (see Figure 2). How is the orientation of the tool defined in the position data (robtarget)?

The orientation of the tool in a programmed position is normally related to the coordinate system of the work object used. In this example, no work object is used and the base coordinate system is equal to the world coordinate system. Thus, the orientation is related to the base coordinate system.

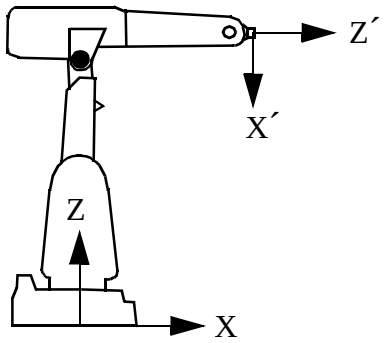


Figure 2 The direction of a tool in accordance with example 1.

The axes will then be related as follows:

$\mathbf{x}' = -\mathbf{z} = (0, 0, -1)$

$\mathbf{y}' = \mathbf{y} = (0, 1, 0)$

$\mathbf{z}' = \mathbf{x} = (1, 0, 0)$

Which corresponds to the following rotational matrix: $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$

The rotational matrix provides a corresponding quaternion:

$$q1 = \frac{\sqrt{0+1+0+1}}{2} = \frac{\sqrt{2}}{2} = 0{,}707$$

$$q2 = \frac{\sqrt{0-1-0+1}}{2} = 0$$

$$q3 = \frac{\sqrt{1-0-0+1}}{2} = \frac{\sqrt{2}}{2} = 0{,}707 \qquad \text{sign } q3 = \text{sign } (1+1) = +$$

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

Example 2

The direction of the tool is rotated 30^{o} about the X'- and Z'-axes in relation to the wrist coordinate system (see Figure 2). How is the orientation of the tool defined in the tool data?

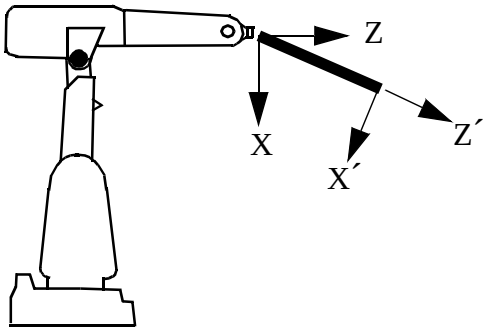


Figure 3 The direction of the tool in accordance with example 2.

The axes will then be related as follows:

x' = (cos30^o, 0, -sin30^o)

x' = (0, 1, 0)

x' = (sin30^o, 0, cos30^o)

Which corresponds to the following rotational matrix: $\begin{bmatrix} \cos 30° & 0 & \sin 30° \\ 0 & 1 & 0 \\ -\sin 30° & 0 & \cos 30° \end{bmatrix}$

The rotational matrix provides a corresponding quaternion:

$$q1 = \frac{\sqrt{\cos 30° + 1 + \cos 30° + 1}}{2} = 0{,}965926$$

$$q2 = \frac{\sqrt{\cos 30° - 1 - \cos 30° + 1}}{2} = 0$$

$$q3 = \frac{\sqrt{1 - \cos 30° - \cos 30° + 1}}{2} = 0{,}258819 \qquad \text{sign } q3 = \text{sign } (\sin 30^o + \sin 30^o) = +$$

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

Structure

<dataobject of *orient*>
  <*q1* of *num*>
  <*q2* of *num*>
  <*q3* of *num*>
  <*q4* of *num*>

Related information

	Described in:
Operations on orientations	Basic Characteristics - *Expressions*

o_robtarget Original position data

o_robtarget (*original robot target*) is used in combination with the function *Absolute Limit Modpos*. When this function is used to modify a position, the original position is stored as a data of the type *o_robtarget*.

Description

If the function *Absolute Limit Modpos* is activated and a named position in a movement instruction is modified with the function *Modpos*, then the original programmed position is saved.

Example of a program before *Modpos*:

```
CONST robtarget p50       := [[500, 500, 500], [1, 0, 0, 0], [1, 1, 0, 0],
                              [500, 9E9, 9E9, 9E9, 9E9, 9E9] ];
...
MoveL p50, v1000, z50, tool1;
```

The same program after *ModPos* in which the point *p50 is* corrected to 502 in the x-direction:

```
CONST robtarget p50       := [[502, 500, 500], [1, 0, 0, 0], [1, 1, 0, 0],
                              [500, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST o_robtarget o_p50   := [[500, 500, 500], [1, 0, 0, 0], [1, 1, 0, 0],
                              [ 500, 9E9, 9E9, 9E9, 9E9, 9E9] ];
...
MoveL p50, v1000, z50, tool1;
```

The original programmed point has now been saved in *o_p50* (by the data type *o_robtarget*) and the modified point saved in *p50* (by the data type *robtarget*).

By saving the original programmed point, the robot can monitor that further *Modpos* of the point in question are within the acceptable limits from the original programmed point.

The fixed name convention means that an original programmed point with the name *xxxxx* is saved with the name *o_xxxxx* by using *Absolute Limit Modpos*.

Components

trans *(translation)* Data type: *pos*

The position (x, y and z) of the tool centre point expressed in mm.

rot *(rotation)* Data type: *orient*

The orientation of the tool, expressed in the form of a quaternion (q1, q2, q3 and q4).

robconf *(robot configuration)* Data type: *confdata*

The axis configuration of the robot (cf1, cf4, cf6 and cfx).

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

Structure

< dataobject of *o_robtarget* >
< *trans* of *pos* >
< *x* of *num* >
< *y* of *num* >
< *z* of *num* >
< *rot* of *orient* >
< *q1* of *num* >
< *q2* of *num* >
< *q3* of *num* >
< *q4* of *num* >
< *robconf* of *confdata* >
< *cf1* of *num* >
< *cf4* of *num* >
< *cf6* of *num* >
< *cfx* of *num* >
< *extax* of *extjoint* >
< *eax*_a of *num* >
< *eax*_b of *num* >
< *eax*_c of *num* >
< *eax*_d of *num* >
< *eax*_e of *num* >
< *eax*_f of *num* >

Related information

	Described in:
Position data	Data Types - *Robtarget*
Configuration of Limit Modpos	User's Guide - *System Parameters*

pos Positions (only X, Y and Z)

Pos is used for positions (only X, Y and Z).

The *robtarget* data type is used for the robot's position including the orientation of the tool and the configuration of the axes.

Description

Data of the type *pos* describes the coordinates of a position: X, Y and Z.

Components

x Data type: *num*

The X-value of the position.

y Data type: *num*

The Y-value of the position.

z Data type: *num*

The Z-value of the position.

Examples

```
VAR pos pos1;
.
pos1 := [500, 0, 940];
```

The *pos1* position is assigned the value: X=500 mm, Y=0 mm, Z=940 mm.

```
pos1.x := pos1.x + 50;
```

The *pos1* position is shifted *50* mm in the X-direction.

Structure

```
<dataobject of pos>
    <x of num>
    <y of num>
    <z of num>
```

Related information

	Described in:
Operations on positions	Basic Characteristics - *Expressions*
Robot position including orientation	Data Types- *robtarget*

pose Coordinate transformations

Pose is used to change from one coordinate system to another.

Description

Data of the type *pose* describes how a coordinate system is displaced and rotated around another coordinate system. The data can, for example, describe how the tool coordinate system is located and oriented in relation to the wrist coordinate system.

Components

trans *(translation)* Data type: *pos*

The displacement in position (x, y and z) of the coordinate system.

rot *(rotation)* Data type: *orient*

The rotation of the coordinate system.

Example

```
VAR pose frame1;
.
frame1.trans := [50, 0, 40];
frame1.rot := [1, 0, 0, 0];
```

The *frame1* coordinate transformation is assigned a value that corresponds to a displacement in position, where X=*50* mm, Y=*0* mm, Z=*40* mm; there is, however, no rotation.

Structure

```
<dataobject of pose>
    <trans of pos>
    <rot of orient>
```

Related information

	Described in:
What is a Quaternion?	Data Types - *orient*

progdisp Program displacement

Progdisp is used to store the current program displacement of the robot and the external axes.

This data type does not normally have to be used since the data is set using the instructions *PDispSet, PDispOn, PDispOff, EOffsSet, EOffsOn* and *EOffsOff*. It is only used to temporarily store the current value for later use.

Description

The current values for program displacement can be accessed using the system variable *C_PROGDISP*.

For more information, see the instructions *PDispSet, PDispOn, EOffsSet* and *EOffsOn*.

Components

pdisp *(program displacement)* Data type: *pose*

The program displacement for the robot, expressed using a translation and an orientation. The translation is expressed in mm.

eoffs *(external offset)* Data type: *extjoint*

The offset for each of the external axes. If the axis is linear, the value is expressed in mm; if it is rotating, the value is expressed in degrees.

Example

```
VAR progdisp progdisp1;
.
SearchL sen1, psearch, p10, v100, tool1;
PDispOn \ExeP:=psearch, *, tool1;
EOffsOn \ExeP:=psearch, *;
.
progdisp1:=C_PROGDISP;
PDispOff;
EOffsOff;
.
PDispSet progdisp1.pdisp;
EOffsSet progdisp1.eoffs;
```

First, a program displacement is activated from a searched position. Then, it is temporarily deactivated by storing the value in the variable *progdisp1* and, later on, re-activated using the instructions *PDispSet* and *EOffsSet*.

Predefined data

The system variable *C_PROGDISP* describes the current program displacement of the robot and external axes, and can always be accessed from the program (installed data). *C_PROGDISP*, on the other hand, can only be changed using a number of instructions, not by assignment.

Structure

```
< dataobject of progdisp >
    <pdisp of pose>
        < trans of pos >
            < x of num >
            < y of num >
            < z of num >
        < rot of orient >
            < q1 of num >
            < q2 of num >
            < q3 of num >
            < q4 of num >
    < eoffs of extjoint >
        < eax_a of num >
        < eax_b of num >
        < eax_c of num >
        < eax_d of num >
        < eax_e of num >
        < eax_f of num >
```

Related information

	Described in:
Instructions for defining program displacement	RAPID Summary - *Motion Settings*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

robjoint Joint position of robot axes

Robjoint is used to define the axis position in degrees of the robot axes.

Description

Data of the type *robjoint* is used to store axis positions in degrees of the robot axes 1 to 6. Axis position is defined as the rotation in degrees for the respective axis (arm) in a positive or negative direction from the axis calibration position.

Components

rax_1 *(robot axis 1)* Data type: *num*

The position of robot axis 1 in degrees from the calibration position.

...

rax_6 *(robot axis 6)* Data type: *num*

The position of robot axis 6 in degrees from the calibration position.

Structure

< dataobject of *robjoint* >
< *rax_1* of *num* >
< *rax_2* of *num* >
< *rax_3* of *num* >
< *rax_4* of *num* >
< *rax_5* of *num* >
< *rax_6* of *num* >

Related information

	Described in:
Joint position data	Data Types - *jointtarget*
Move to joint position	Instructions - *MoveAbsJ*

robtarget Position data

Robtarget (*robot target*) is used to define the position of the robot and external axes.

Description

Position data is used to define the position in the positioning instructions to which the robot and external axes are to move.

As the robot is able to achieve the same position in several different ways, the axis configuration is also specified. This defines the axis values if these are in any way ambiguous, for example:

- if the robot is in a forward or backward position,
- if axis 4 points downwards or upwards,
- if axis 6 has a negative or positive revolution.

The position is defined based on the coordinate system of the work object, including any program displacement. If the position is programmed with some other work object than the one used in the instruction, the robot will not move in the expected way. Make sure that you use the same work object as the one used when programming positioning instructions. Incorrect use can injure someone or damage the robot or other equipment.

Components

trans *(translation)* Data type: *pos*

The position (x, y and z) of the tool centre point expressed in mm.

The position is specified in relation to the current object coordinate system, including program displacement. If no work object is specified, this is the world coordinate system.

rot *(rotation)* Data type: *orient*

The orientation of the tool, expressed in the form of a quaternion (q1, q2, q3 and q4).

The orientation is specified in relation to the current object coordinate system, including program displacement. If no work object is specified, this is the world coordinate system.

robconf *(robot configuration)* Data type: *confdata*

The axis configuration of the robot (cf1, cf4, cf6 and cfx). This is defined in the form of the current quarter revolution of axis 1, axis 4 and axis 6. The first pos-

itive quarter revolution 0-90 ^{o} is defined as 0. The component cfx is not used at present.

For more information, see data type *confdata*.

extax *(external axes)* Data type: *extjoint*

The position of the external axes.

The position is defined as follows for each individual axis (*eax_a*, *eax_b* ... *eax_f*):

- For rotating axes, the position is defined as the rotation in degrees from the calibration position.
- For linear axes, the position is defined as the distance in mm from the calibration position.

External axes *eax_a* ... are logical axes. How the logical axis number and the physical axis number are related to each other is defined in the system parameters.

The value 9E9 is defined for axes which are not connected. If the axes defined in the position data differ from the axes that are actually connected on program execution, the following applies:

- If the position is not defined in the position data (value 9E9), the value will be ignored, if the axis is connected and not activated. But if the axis is activated, it will result in an error.
- If the position is defined in the position data although the axis is not connected, the value is ignored.

Examples

```
CONST robtarget p15 := [ [600, 500, 225.3], [1, 0, 0, 0], [1, 1, 0, 0],
[ 11, 12.3, 9E9, 9E9, 9E9, 9E9] ];
```

A position *p15* is defined as follows:

- The position of the robot: *x = 600*, *y = 500* and *z = 225.3* mm in the object coordinate system.
- The orientation of the tool in the same direction as the object coordinate system.
- The axis configuration of the robot: axes 1 and 4 in position 90-180^{o}, axis 6 in position 0-90^{o}.
- The position of the external logical axes, a and b, expressed in degrees or mm (depending on the type of axis). Axes c to f are undefined.

```
VAR robtarget p20;
. . .
p20 := CRobT();
p20 := Offs(p20,10,0,0);
```

The position *p20* is set to the same position as the current position of the robot by calling the function *CRobT*. The position is then moved *10* mm in the x-direction.

Limitations

When using the configurable edit function *Absolute Limit Modpos*, the number of characters in the name of the data of the type *robtarget*, is limited to 14 (in other cases 16).

Structure

< dataobject of *robtarget* >
- < *trans* of *pos* >
  - < *x* of *num* >
  - < *y* of *num* >
  - < *z* of *num* >
- < *rot* of *orient* >
  - < *q1* of *num* >
  - < *q2* of *num* >
  - < *q3* of *num* >
  - < *q4* of *num* >
- < *robconf* of *confdata* >
  - < *cf1* of *num* >
  - < *cf4* of *num* >
  - < *cf6* of *num* >
  - < *cf*x of *num* >
- < *extax* of *extjoint* >
  - < *eax*_a of *num* >
  - < *eax*_b of *num* >
  - < *eax*_c of *num* >
  - < *eax*_d of *num* >
  - < *eax*_e of *num* >
  - < *eax*_f of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Handling configuration data	Motion and I/O Principles - *Robot Configuration*
Configuration of external axes	User’s Guide - *System Parameters*
What is a quaternion?	Data Types - *Orient*

signal*xx* Digital and analog signals

Data types within *signalxx* are used for digital and analog input and output signals.

The names of the signals are defined in the system parameters and are consequently not to be defined in the program.

Description

Data type	Used for
signalai	analog input signals
signalao	analog output signals
signaldi	digital input signals
signaldo	digital output signals
signalgi	groups of digital input signals
signalgo	groups of digital output signals

Variables of the type *signalxo* only contain a reference to the signal. The value is set using an instruction, e.g. *DOutput*.

Variables of the type *signalxi* contain a reference to a signal as well as a method to retrieve the value. The value of the input signal is returned when a function is called, e.g. *DInput*, or when the variable is used in a value context, e.g. *IF signal_y=1 THEN*.

Limitations

Data of the data type *signalxx* may not be defined in the program. However, if this is in fact done, an error message will be displayed as soon as an instruction or function that refers to this signal is executed. The data type can, on the other hand, be used as a parameter when declaring a routine.

Predefined data

The signals defined in the system parameters can always be accessed from the program by using the predefined signal variables (installed data). It should however be noted that if other data with the same name is defined, these signals cannot be used.

Characteristics

Signalxo is a *non-value* data type. Thus, data of this type does not permit value-oriented operations.

Signalxi is a semi-value data type.

Related information

	Described in:
Summary input/output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	User’s Guide - *System Parameters*
Characteristics of non-value data types	Basic Characteristics - *Data Types*

speeddata Speed data

Speeddata is used to specify the velocity at which both the robot and the external axes move.

Description

Speed data defines the velocity:

- at which the tool centre point moves,
- of the reorientation of the tool,
- at which linear or rotating external axes move.

When several different types of movement are combined, one of the velocities often limits all movements. The velocity of the other movements will be reduced in such a way that all movements will finish executing at the same time.

The velocity is also restricted by the performance of the robot. This differs, depending on the type of robot and the path of movement.

Components

v_tcp *(velocity tcp)* Data type: *num*

The velocity of the tool centre point (TCP) in mm/s.

If a stationary tool or coordinated external axes are used, the velocity is specified relative to the work object.

v_ori *(velocity orientation)* Data type: *num*

The velocity of reorientation about the TCP expressed in degrees/s.

If a stationary tool or coordinated external axes are used, the velocity is specified relative to the work object.

v_leax *(velocity linear external axes)* Data type: *num*

The velocity of linear external axes in mm/s.

v_reax *(velocity rotational external axes)* Data type: *num*

The velocity of rotating external axes in degrees/s.

Example

VAR speeddata vmedium := [1000, 30, 200, 15];

The speed data *vmedium* is defined with the following velocities:

- *1000* mm/s for the TCP.
- *30* degrees/s for reorientation of the tool.
- *200* mm/s for linear external axes.
- *15* degrees/s for rotating external axes.

vmedium.v_tcp := 900;

The velocity of the TCP is changed to *900* mm/s.

Predefined data

A number of speed data are already defined in the system module *BASE*.

Name	TCP speed	Orientation	Linear ext. axis	Rotating ext. axis
v5	5 mm/s	500°/s	5000 mm/s	1000°/s
v10	10 mm/s	500°/s	5000 mm/s	1000°/s
v20	20 mm/s	500°/s	5000 mm/s	1000°/s
v30	30 mm/s	500°/s	5000 mm/s	1000°/s
v40	40 mm/s	500°/s	5000 mm/s	1000°/s
v50	50 mm/s	500°/s	5000 mm/s	1000°/s
v60	60 mm/s	500°/s	5000 mm/s	1000°/s
v80	80 mm/s	500°/s	5000 mm/s	1000°/s
v100	100 mm/s	500°/s	5000 mm/s	1000°/s
v150	150 mm/s	500°/s	5000 mm/s	1000°/s
v200	200 mm/s	500°/s	5000 mm/s	1000°/s
v300	300 mm/s	500°/s	5000 mm/s	1000°/s
v400	400 mm/s	500°/s	5000 mm/s	1000°/s
v500	500 mm/s	500°/s	5000 mm/s	1000°/s
v600	600 mm/s	500°/s	5000 mm/s	1000°/s
v800	800 mm/s	500°/s	5000 mm/s	1000°/s
v1000	1000 mm/s	500°/s	5000 mm/s	1000°/s
v1500	1500 mm/s	500°/s	5000 mm/s	1000°/s
v2000	2000 mm/s	500°/s	5000 mm/s	1000°/s
v2500	2500 mm/s	500°/s	5000 mm/s	1000°/s
v3000	3000 mm/s	500°/s	5000 mm/s	1000°/s
v4000	4000 mm/s	500°/s	5000 mm/s	1000°/s
v5000	5000 mm/s	500°/s	5000 mm/s	1000°/s
vmax	5000 mm/s	500°/s	5000 mm/s	1000°/s
v6000	6000 mm/s	500°/s	5000 mm/s	1000°/s
v7000	7000 mm/s	500°/s	5000 mm/s	1000°/s

Structure

< dataobject of *speeddata* >
  < *v_tcp* of *num* >
  < *v_ori* of *num* >
  < *v_leax* of *num* >
  < *v_reax* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Motion/Speed in general	Motion and I/O Principles - *Positioning during Program Execution*
Defining maximum velocity	Instructions - *VelSet*
Configuration of external axes	User’s Guide - *System Parameters*
Motion performance	Product Specification

string Strings

String is used for character strings.

Description

A character string consists of a number of characters (a maximum of 80) enclosed by quotation marks (''),

e.g. "This is a character string".

If the quotation marks are to be included in the string, they must be written twice,

e.g. "This string contains a ""character".

Example

```
VAR string text;
.
text := "start welding pipe 1";
TPWrite text;
```

The text *start welding pipe 1* is written on the teach pendant.

Limitations

A string may have from 0 to 80 characters; inclusive of extra quotation marks.

A string may contain any of the characters specified by ISO 8859-1 as well as control characters (non-ISO 8859-1 characters with a numeric code between 0-255).

Predefined data

A number of predefined string constants (installed data) can be used together with string functions.

Name	Character set
STR_DIGIT	<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
STR_UPPER	<upper case letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í | Î | Ï | 1) | Ñ | Ò | Ó | Ô | Õ | Ö | Ø | Ù | Ú | Û | Ü | 2) | 3)
STR_LOWER	<lower case letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | à | á | â | ã | ä | å | æ | ç | è | é | ê | ë | ì | í | î | ï | 1) | ñ | ò | ó | ô | õ | ö | ø | ù | ú | û | ü | 2) | 3) | ß | ÿ
STR_WHITE	<blank character> ::=

1) Icelandic letter eth.
2) Letter Y with acute accent.
3) Icelandic letter thorn.

Related information

	Described in:
Operations using strings	Basic Characteristics - *Expressions*
String values	Basic Characteristics - *Basic Elements*

symnum Symbolic number

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

Description

A *symnum* constant is intended to be used when checking the return value from the functions *OpMode* and *RunMode*. See example below.

Example

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

Predefined data

The following symbolic constants of the data type *symnum* are predefined and can be used when checking return values from the functions *OpMode* and *RunMode*.

Value	Symbolic constant	Comment
0	RUN_UNDEF	Undefined running mode
1	RUN_CONT_CYCLE	Continuous or cycle running mode
2	RUN_INSTR_FWD	Instruction forward running mode
3	RUN_INSTR_BWD	Instruction backward running mode
4	RUN_SIM	Simulated running mode

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

Characteristics

Symnum is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Data types in general, alias data types	Basic Characteristics - *Data Types*

tooldata Tool data

Tooldata is used to describe the characteristics of a tool, e.g. a welding gun or a gripper.

If the tool is fixed in space (a stationary tool), common tool data is defined for this tool and the gripper holding the work object.

Description

Tool data affects robot movements in the following ways:

- The tool centre point (TCP) refers to a point that will satisfy the specified path and velocity performance. If the tool is reoriented or if coordinated external axes are used, only this point will follow the desired path at the programmed velocity.
- If a stationary tool is used, the programmed speed and path will relate to the work object.
- The load of the tool is used to control the robot's movements in the best possible way. An incorrect load can cause overshooting, for example.
- Programmed positions refer to the position of the current TCP and the orientation in relation to the tool coordinate system. This means that if, for example, a tool is replaced because it is damaged, the old program can still be used if the tool coordinate system is redefined.

Tool data is also used when jogging the robot to:

- Define the TCP that is not to move when the tool is reoriented.
- Define the tool coordinate system in order to facilitate moving in or rotating about the tool directions.

Components

robhold *(robot hold)* Data type: *bool*

Defines whether or not the robot is holding the tool:

- *TRUE* -> The robot is holding the tool.
- *FALSE* -> The robot is not holding the tool, i.e. a stationary tool.

tframe *(tool frame)* Data type: *pose*

The tool coordinate system, i.e.:

- The position of the TCP (x, y and z) in mm.
- The tool directions, expressed in the form of a quaternion (q1, q2, q3 and q4).

Both the position and rotation are defined using the wrist coordinate system (see Figure 1.) If a stationary tool is used, the definition is defined in relation to the world coordinate system.

If the direction of the tool is not specified, the tool coordinate system and the wrist coordinate system will coincide.

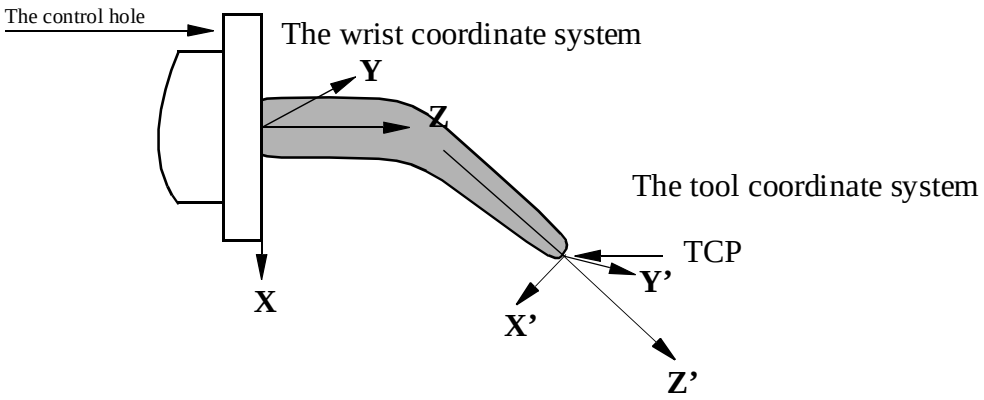


Figure 1 Definition of the tool coordinate system.

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

The load of the tool, i.e.:

- The weight of the tool in kg.
- The centre of gravity of the tool (x, y and z) in mm.
- The axes of moment of the tool, expressed as a quaternion (q1, q2, q3, q4).
- The moment of inertia of the tool about the axes of moment x, y and z, expressed in kgm^2.
  If all components are defined as being 0 kgm^2, the tool is handled as if it were a spot load.

For more information, see the data type *loaddata*.

If a stationary tool is used, you define the load of the gripper holding the work object.

Note that only the load of the tool is to be specified. The payload handled by a gripper is connected and disconnected by means of the instruction *GripLoad*.

Examples

```
PERS tooldata gripper := [ TRUE, [[97.4, 0, 223.1], [0.924, 0, 0.383 ,0]],
   [5, [23, 0, 75], [1, 0, 0, 0], 0, 0, 0]];
```

The tool in Figure 1 is described using the following values:

- The robot is holding the tool.
- The TCP is located at a point *223.1* mm straight out from axis 6 and *97.4* mm along the X-axis of the wrist coordinate system.
- The X and Z directions of the tool are rotated 45^{o} in relation to the wrist coordinate system.
- The tool weighs 5 kg.
- The centre of gravity is located at a point *75* mm straight out from axis 6 and *23* mm along the X-axis of the wrist coordinate system.
- The load can be considered a spot load, i.e. without any moment of inertia.

```
gripper.tframe.trans.z := 225.2;
```

The TCP of the tool, *gripper*, is adjusted to *225.2* in the z-direction.

Limitations

The tool data should be defined as a persistent variable *(PERS)* and should not be defined within a routine. The current values are then saved when the program is stored on diskette and are retrieved on loading.

Arguments of the type tool data in any motion instruction should only be an entire persistent (not array element or record component).

Predefined data

The tool *tool0* defines the wrist coordinate system, with the origin being the centre of the mounting flange. *Tool0* can always be accessed from the program, but can never be changed (it is stored in system module *BASE*).

```
PERS tooldata tool0 := [ TRUE, [ [0, 0, 0], [1, 0, 0 ,0] ],
                              [0, [0, 0, 0], [1, 0, 0, 0], 0, 0, 0] ];
```

Structure

```
< dataobject of tooldata >
   < robhold of bool >
   < tframe of pose >
      < trans of pos >
         < x of num >
         < y of num >
         < z of num >
      < rot of orient >
         < q1 of num >
         < q2 of num >
         < q3 of num >
         < q4 of num >
   < tload of loaddata >
      < mass of num >
      < cog of pos >
         < x of num >
         < y of num >
         < z of num >
      < aom of orient >
         < q1 of num >
         < q2 of num >
         < q3 of num >
         < q4 of num >
      < ix of num >
      < iy of num >
      < iz of num >
```

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Definition of payload	Instructions - *Gripload*

triggdata Positioning events - trigg

Triggdata is used to store data about a positioning event during a robot movement.

A positioning event can take the form of setting an output signal or running an interrupt routine at a specific position along the movement path of the robot.

Description

To define the conditions for the respective measures at a positioning event, variables of the type *triggdata* are used. The data contents of the variable are formed in the program using one of the instructions *TriggIO* or *TriggInt*, and are used by one of the instructions *TriggL*, *TriggC* or *TriggJ*.

Example

```
VAR triggdata gunoff;

TriggIO gunoff, 5 \DOp:=gun, off;

TriggL p1, v500, gunoff, fine, gun1;
```

The digital output signal *gun* is set to the value *off* when the TCP is at a position *5* mm before the point *p1*.

Characteristics

Triggdata is a non-value data type.

Related information

	Described in:
Definition of triggs	Instructions - *TriggIO, TriggInt*
Use of triggs	Instructions - *TriggL, TriggC, TriggJ*
Characteristics of non-value data types	Basic Characteristics- *Data Types*

tunetype Servo tune type

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

Description

A *tunetype* constant is indented to be used as an argument to the instruction *TuneServo.* See example below.

Example

```
TuneServo MHA160R1, 1, 110 \Type:= TUNE_KP;
```

Predefined data

The following symbolic constants of the data type *tunetype* are predefined and can be used as argument for the instruction *TuneServo.*

Value	Symbolic constant	Comment
0	TUNE_DF	Reduces overshoots
1	TUNE_KP	Affects position control gain
2	TUNE_KV	Affects speed control gain
3	TUNE_TI	Affects speed control integration time

Characteristics

Tunetype is an alias data type for *num* and consequently inherits its characteristics.

Related information

	Described in:
Data types in general, alias data types	Basic Characteristics - *Data Types*

wobjdata Work object data

Wobjdata is used to describe the work object that the robot welds, processes, moves within, etc.

Description

If work objects are defined in a positioning instruction, the position will be based on the coordinates of the work object. The advantages of this are as follows:

- If position data is entered manually, such as in off-line programming, the values can often be taken from a drawing.
- Programs can be reused quickly following changes in the robot installation. If, for example, the fixture is moved, only the user coordinate system has to be redefined.
- Variations in how the work object is attached can be compensated for. For this, however, some sort of sensor will be required to position the work object.

If a stationary tool or coordinated external axes are used the work object must be defined, since the path and velocity would then be related to the work object instead of the TCP.

Work object data can also be used for jogging:

- The robot can be jogged in the directions of the work object.
- The current position displayed is based on the coordinate system of the work object.

Components

robhold *(robot hold)* Data type: *bool*

Defines whether or not the robot is holding the work object:

- *TRUE* -> The robot is holding the work object, i.e. using a stationary tool.
- *FALSE* -> The robot is not holding the work object, i.e. the robot is holding the tool.

ufprog *(user frame programmed)* Data type: *bool*

Defines whether or not a fixed user coordinate system is used:

- *TRUE* -> Fixed user coordinate system.
- *FALSE* -> Movable user coordinate system, i.e. coordinated external axes are used.

ufmec *(user frame mechanical unit)* Data type: *string*

The mechanical unit with which the robot movements are coordinated. Only specified in the case of movable user coordinate systems (*ufprog* is *FALSE*).

Specified with the name that is defined in the system parameters, e.g. "orbit_a".

uframe *(user frame)* Data type: *pose*

The user coordinate system, i.e. the position of the current work surface or fixture (see Figure 1):

- The position of the origin of the coordinate system (x, y and z) in mm.
- The rotation of the coordinate system, expressed as a quaternion (q1, q2, q3, q4).

If the robot is holding the tool, the user coordinate system is defined in the world coordinate system (in the wrist coordinate system if a stationary tool is used).

When coordinated external axes are used (*ufprog* is *FALSE*), the user coordinate system is defined in the system parameters.

oframe *(object frame)* Data type: *pose*

The object coordinate system, i.e. the position of the current work object (see Figure 1):

- The position of the origin of the coordinate system (x, y and z) in mm.
- The rotation of the coordinate system, expressed as a quaternion (q1, q2, q3, q4).

The object coordinate system is defined in the user coordinate system.

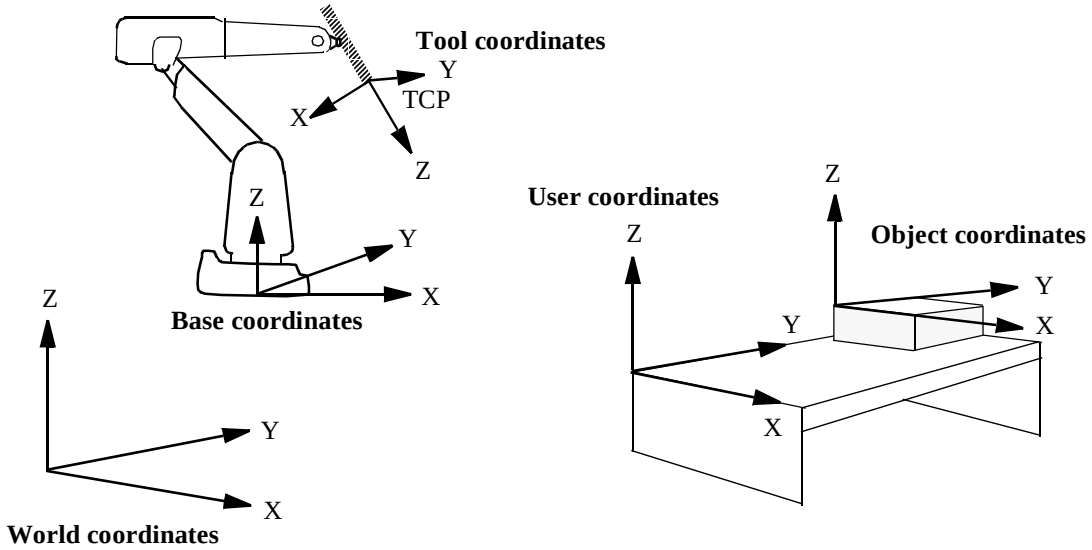


Figure 1 The various coordinate systems of the robot (when the robot is holding the tool).

Example

```
PERS wobjdata wobj2 :=[ FALSE, TRUE, "", [ [300, 600, 200], [1, 0, 0 ,0] ],
   [ [0, 200, 30], [1, 0, 0 ,0] ] ];
```

The work object in Figure 1 is described using the following values:

- The robot is not holding the work object.
- The fixed user coordinate system is used.
- The user coordinate system is not rotated and the coordinates of its origin are x= *300*, y = *600* and z = *200* mm in the world coordinate system.
- The object coordinate system is not rotated and the coordinates of its origin are x= *0*, y= *200* and z= *30* mm in the user coordinate system.

```
wobj2.oframe.trans.z := 38.3;
```

- The position of the work object *wobj2* is adjusted to *38.3* mm in the z-direction.

Limitations

The work object data should be defined as a persistent variable (*PERS)* and should not be defined within a routine. The current values are then saved when the program is stored on diskette and are retrieved on loading.

Arguments of the type work object data in any motion instruction should only be an entire persistent (not array element or record component).

Predefined data

The work object data *wobj0* is defined in such a way that the object coordinate system coincides with the world coordinate system. The robot does not hold the work object.

Wobj0 can always be accessed from the program, but can never be changed (it is stored in system module *BASE*).

```
PERS wobjdata wobj0 :=    [ FALSE, TRUE, "", [ [0, 0, 0], [1, 0, 0 ,0] ],
                          [ [0, 0, 0], [1, 0, 0 ,0] ] ];
```

Structure

< dataobject of *wobjdata* >
  < *robhold* of *bool* >
  < *ufprog* of *bool*>
  < *ufmec* of *string* >
  < *uframe* of *pose* >
    < *trans* of *pos* >
      < *x* of *num* >
      < *y* of *num* >
      < *z* of *num* >
    < *rot* of *orient* >
      < *q1* of *num* >
      < *q2* of *num* >
      < *q3* of *num* >
      < *q4* of *num* >
  < *oframe* of *pose* >
    < *trans* of *pos* >
      < *x* of *num* >
      < *y* of *num* >
      < *z* of *num* >
    < *rot* of *orient* >
      < *q1* of *num* >
      < *q2* of *num* >
      < *q3* of *num* >
      < *q4* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*
Coordinated external axes	Motion and I/O Principles - *Coordinate Systems*
Calibration of coordinated external axes	User’s Guide - *System Parameters*

zonedata Zone data

Zonedata is used to specify how a position is to be terminated, i.e. how close to the programmed position the axes must be before moving towards the next position.

Description

A position can be terminated either in the form of a stop point or a fly-by point.

A stop point means that the robot and external axes must reach the specified position (stand still) before program execution continues with the next instruction.

A fly-by point means that the programmed position is never attained.
Instead, the direction of motion is changed before the position is reached.
Two different zones (ranges) can be defined for each position:

- The zone for the TCP path.
- The extended zone for reorientation of the tool and for external axes.

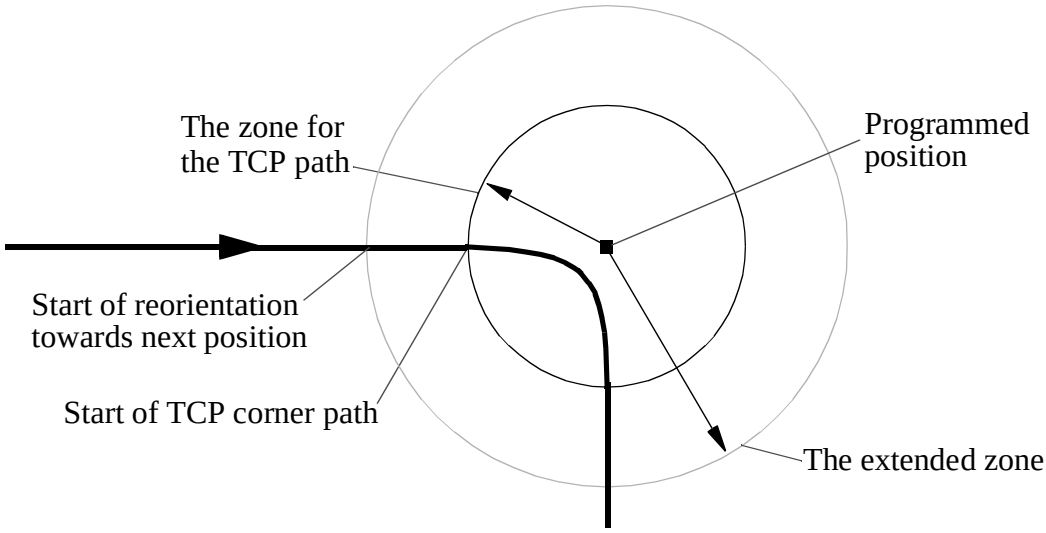


Figure 1 The zones for a fly-by point.

Zones function in the same way during joint movement, but the zone size may differ somewhat from the one programmed.

The zone size cannot be larger than half the distance to the closest position (forwards or backwards). If a larger zone is specified, the robot automatically reduces it.

The zone for the TCP path

A corner path (parabola) is generated as soon as the edge of the zone is reached (see Figure 1).

The zone for reorientation of the tool

Reorientation starts as soon as the TCP reaches the extended zone. The tool is reoriented in such a way that the orientation is the same leaving the zone as it would have been in the same position if stop points had been programmed. Reorientation will be smoother if the zone size is increased, and there is less of a risk of having to reduce the velocity to carry out the reorientation.

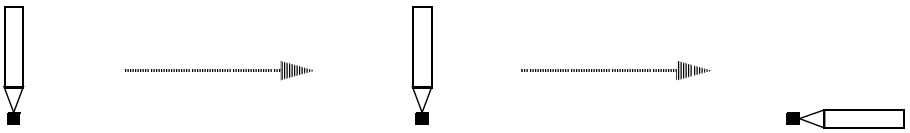


Figure 2a Three positions are programmed, the last with different tool orientation.

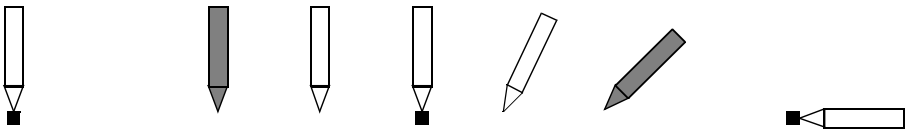


Figure 2b If all positions were stop points, program execution would look like this.

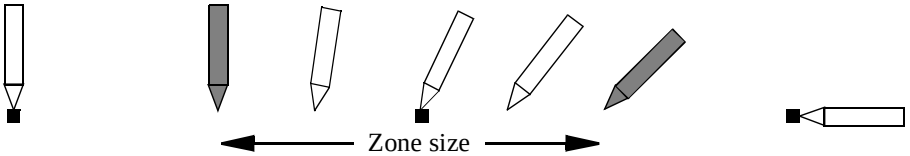


Figure 2c If the middle position was a fly-by point, program execution would look like this

The zone for external axes

External axes start to move towards the next position as soon as the TCP reaches the extended zone. In this way, a slow axis can start accelerating at an earlier stage and thus execute more evenly.

Reduced zone

With large reorientations of the tool or with large movements of the external axes, the extended zone and even the TCP zone can be reduced by the robot. The zone will be defined as the smallest relative size of the zone based upon the zone components (see next page) and the programmed motion.

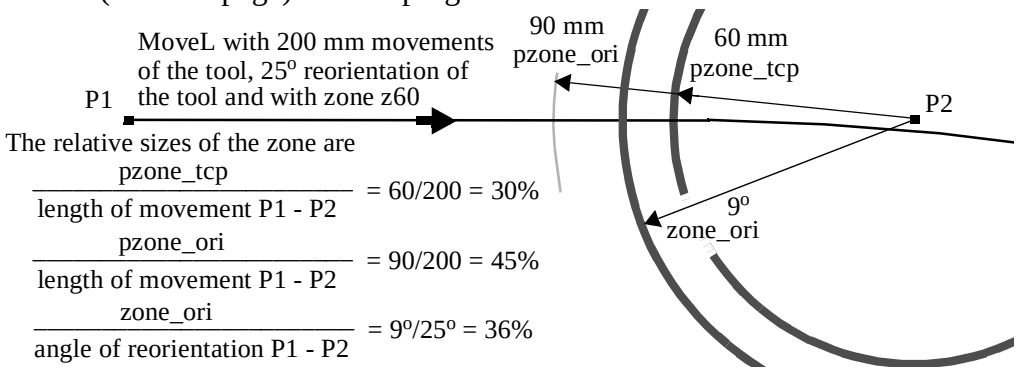


Figure 3 Example of reduced zone to 36% of the motion

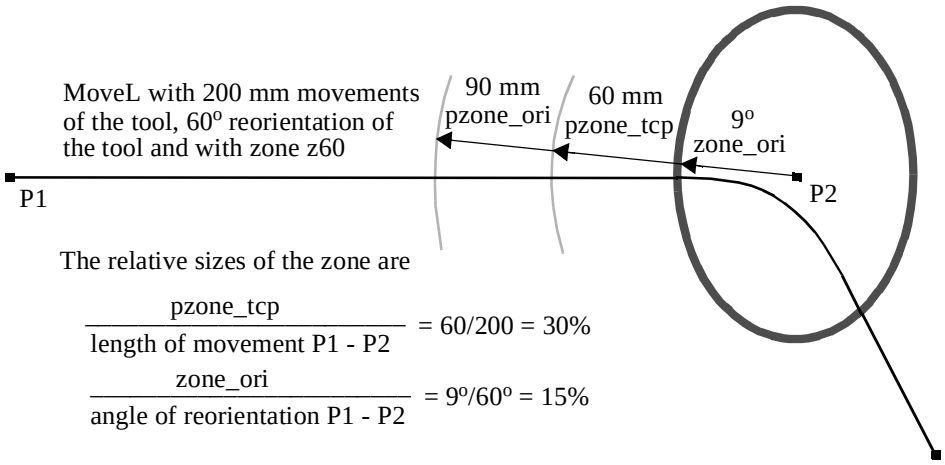


Figure 4 Example of reduced zone to 15% of the motion

Components

finep *(fine point)* Data type: *bool*

Defines whether the movement is to terminate as a stop point (fine point) or as a fly-by point.

- *TRUE* -> The movement terminates as a stop point.
  The remaining components in the zone data are not used.
- *FALSE* -> The movement terminates as a fly-by point.

pzone_tcp *(path zone TCP)* Data type: *num*

The size (the radius) of the TCP zone in mm.

The extended zone will be defined as the smallest relative size of the zone based upon the following components and the programmed motion.

pzone_ori *(path zone orientation)* Data type: *num*

The zone size (the radius) for the tool reorientation. The size is defined as the distance of the TCP from the programmed point in mm.

The size must be larger than the corresponding value for *pzone_tcp*.
If a lower value is specified, the size is automatically increased to make it the same as *pzone_tcp*.

pzone_eax *(path zone external axes)* Data type: *num*

The zone size (the radius) for external axes. The size is defined as the distance of the TCP from the programmed point in mm.

The size must be larger than the corresponding value for *pzone_tcp*.
If a lower value is specified, the size is automatically increased to make it the same as *pzone_tcp*.

zone_ori *(zone orientation)* Data type: *num*

The zone size for the tool reorientation in degrees. If the robot is holding the work object, this means an angle of rotation for the work object.

zone_leax *(zone linear external axes)* Data type: *num*

The zone size for linear external axes in mm.

zone_reax *(zone rotational external axes)* Data type: *num*

The zone size for rotating external axes in degrees.

Examples

VAR zonedata path := [FALSE, 25, 40, 40, 10, 35, 5];

The zone data *path* is defined by means of the following characteristics:

- The zone size for the TCP path is *25* mm.
- The zone size for the tool reorientation is *40* mm (TCP movement).
- The zone size for external axes is *40* mm (TCP movement).

If the TCP is standing still, or there is a large reorientation, or there is a large external axis movement, with respect to the zone, the following apply instead:

- The zone size for the tool reorientation is *10* degrees.
- The zone size for linear external axes is *35* mm.
- The zone size for rotating external axes is *5* degrees.

path.pzone_tcp := 40;

The zone size for the TCP path is adjusted to *40* mm.

Predefined data

A number of zone data are already defined in the system module *BASE*.

Stop points

Name	
fine	0 mm

Fly-by points

	TCP movement			Tool reorientation		
Name	TCP path	Orientation	Ext. axis	Orientation	Linear axis	Rotating axis
z1	1 mm	1 mm	1 mm	0.1 ^{o}	1 mm	0.1 ^{o}
z5	5 mm	8 mm	8 mm	0.8 ^{o}	8 mm	0.8 ^{o}
z10	10 mm	15 mm	15 mm	1.5 ^{o}	15 mm	1.5 ^{o}
z15	15 mm	23 mm	23 mm	2.3 ^{o}	23 mm	2.3 ^{o}
z20	20 mm	30 mm	30 mm	3.0 ^{o}	30 mm	3.0^{o}
z30	30 mm	45 mm	45 mm	4.5 ^{o}	45 mm	4.5 ^{o}
z40	40 mm	60 mm	60 mm	6.0 ^{o}	60 mm	6.0 ^{o}
z50	50 mm	75 mm	75 mm	7.5 ^{o}	75 mm	7.5 ^{o}
z60	60 mm	90 mm	90 mm	9.0 ^{o}	90 mm	9.0 ^{o}
z80	80 mm	120 mm	120 mm	12 ^{o}	120 mm	12 ^{o}
z100	100 mm	150 mm	150 mm	15 ^{o}	150 mm	15 ^{o}
z150	150 mm	225 mm	225 mm	23 ^{o}	225 mm	23 ^{o}
z200	200 mm	300 mm	300 mm	30 ^{o}	300 mm	30 ^{o}

Structure

< data object of *zonedata* >
< *finep* of *bool* >
< *pzone_tcp* of *num* >
< *pzone_ori* of *num* >
< *pzone_eax* of *num* >
< *zone_ori* of *num* >
< *zone_leax* of *num* >
< *zone_reax* of *num* >

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*
Movements/Paths in general	Motion and I/O Principles - *Positioning during Program Execution*
Configuration of external axes	User’s Guide - *System Parameters*

System Data

System data is the internal data of the robot that can be accessed and read by the program. It can be used to read the current status, e.g. the current maximum velocity.

The following table contains a list of all system data.

Name	Description	Data Type	Changed by	See also
C_MOTSET	Current motion settings, i.e.: - max. velocity and velocity override - max. acceleration - movement about singular points - monitoring the axis configuration - payload in gripper	motsetdata	Instructions - VelSet - AccSet - SingArea - ConfL,ConfJ - GripLoad	Data Types - *motsetdata* Instructions - *VelSet* Instructions - *AccSet* Instructions - *SingArea* Instructions - *ConfL, ConfJ* Instructions - *GripLoad*
C_PROGDISP	Current program displacement for robot and external axes.	progdisp	Instructions - PDispSet - PDispOn - PDispOff - EOffsSet - EOffsOn - EOffsOff	Data Types - *progdisp* Instructions - *PDispSet* Instructions - *PDispOn* Instructions - *PDispOff* Instructions - *EOffsSet* Instructions - *EOffsOn* Instructions - *EOffsOff*
ERRNO	The latest error that occurred	errnum	The robot	Data Types - *errnum* RAPID Summary - *Error Recovery*
INTNO	The latest interrupt that occurred	intnum	The robot	Data Types - *intnum* RAPID Summary -*Interrupts*

“:=” Assigns a value

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

Examples

reg1 := 5;

reg1 is assigned the value *5*.

reg1 := reg2 - reg3;

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

counter := counter + 1;

counter is incremented by one.

Arguments

Data := Value

Data Data type: All

The data that is to be assigned a new value.

Value Data type: Same as Data

The desired value.

Examples

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

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

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

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

Limitations

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

- a constant
- a non-value data type.

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

Syntax

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

Related information

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

AccSet Reduces the acceleration

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

Examples

AccSet 50, 100;

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

AccSet 100, 50;

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

Arguments

AccSet Acc Ramp

Acc Data type: *num*

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

Ramp Data type: *num*

The rate at which acceleration and deceleration increases as a percentage of the normal values (see Figure 1). Jerking can be restricted by reducing this value.
100% corresponds to maximum rate. Maximum value: 100%.
Input value < 10% gives 10% of maximum rate.

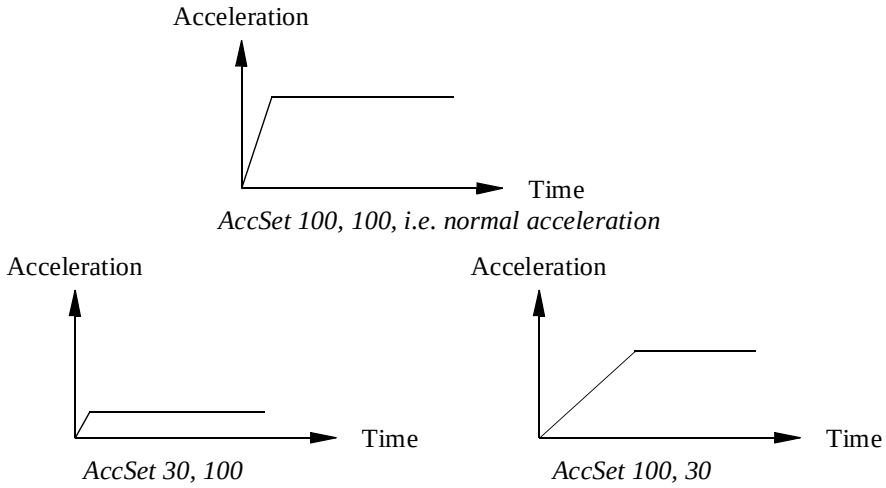


Figure 1 Reducing the acceleration results in smoother movements.

Program execution

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

The default values (100%) are automatically set

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

Syntax

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

Related information

	Described in:
Positioning instructions	RAPID Summary - *Motion*

ActUnit Activates a mechanical unit

ActUnit is used to activate a mechanical unit.

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

Example

ActUnit orbit_a;

Activation of the *orbit_a* mechanical unit.

Arguments

ActUnit MecUnit

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

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

Program execution

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

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

Limitations

Instruction ActUnit cannot be used in

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

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

Syntax

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

Related information

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

Add Adds a numeric value

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

Examples

Add reg1, 3;

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

Add reg1, -reg2;

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

Arguments

Add Name AddValue

Name Data type: *num*

The name of the variable or persistent to be changed.

AddValue Data type: *num*

The value to be added.

Syntax

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

Related information

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

Break Break program execution

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

Example

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

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

Program execution

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

Syntax

Break';'

Related information

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

CallByVar Call a procedure by a variable

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

Example

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

The procedure *proc2* is called.

Arguments

CallByVar Name Number

Name Data type: *string*

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

Number Data type: *num*

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

Example

Static selection of procedure call

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

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

Dynamic selection of procedure call with RAPID syntax

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

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

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

Dynamic selection of procedure call with CallByVar

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

The procedure *proc2* is called.

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

Limitations

Can only be used to call procedures without parameters.

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

Error handling

If the specified procedure is not found, the system variable ERRNO is set to ERR_CALLPROC.

Syntax

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

Related information

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

Clear Clears the value

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

Example

Clear reg1;

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

Arguments

Clear Name

Name Data type: *num*

The name of the variable or persistent to be cleared.

Syntax

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

Related information

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

ClkReset Resets a clock used for timing

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

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

Example

ClkReset clock1;

The clock *clock1* is reset.

Arguments

ClkReset Clock

Clock Data type: *clock*

The name of the clock to reset.

Program execution

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

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

Syntax

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

Related Information

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

ClkStart Starts a clock used for timing

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

Example

ClkStart clock1;

The clock *clock1* is started.

Arguments

ClkStart Clock

Clock Data type: *clock*

The name of the clock to start.

Program execution

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

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

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

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

Example

```
VAR clock clock2;

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

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

Syntax

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

Related Information

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

ClkStop Stops a clock used for timing

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

Example

ClkStop clock1;

The clock *clock1* is stopped.

Arguments

ClkStop Clock

Clock Data type: *clock*

The name of the clock to stop.

Program execution

When a clock is stopped, it will stop running.

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

Syntax

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

Related Information

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

Close Closes a file or serial channel

Close is used to close a file or serial channel.

Example

Close channel2;

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

Arguments

Close IODevice

IODevice Data type: *iodev*

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

Program execution

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

Syntax

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

Related information

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

comment Comment

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

Example

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

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

Arguments

! Comment

Comment Text string

Any text.

Program execution

Nothing happens when you execute this instruction.

Syntax

(EBNF)
'!' {<character>} <newline>

Related information

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

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

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

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

Examples

IF reg1 > 5 GOTO next;

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

IF counter > 10 Set do1;

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

Arguments

IF Condition ...

Condition Data type: *bool*

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

Syntax

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

Related information

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

ConfJ Controls the configuration during joint movement

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

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

Examples

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

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

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

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

Arguments

ConfJ [\On] | [\Off]

\On Data type: *switch*

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

\Off Data type: *switch*

The robot always moves to the closest axis configuration.

Program execution

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

If the argument *\Off* is chosen, the robot always moves to the closest axis configuration. This may be different to the programmed one if the configuration has been incorrectly

specified manually, or if a program displacement has been carried out.

The control is active by default. This is automatically set

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

Syntax

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

Related information

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

ConfL Monitors the configuration during linear movement

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

Examples

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

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

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

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

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

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

Arguments

ConfL [\On] | [\Off]

\On Data type: *switch*

The robot configuration is monitored.

\Off Data type: *switch*

The robot configuration is not monitored.

Program execution

During linear or circular movement, the robot always moves to the programmed position and orientation that has the closest possible axis configuration. If the argument *\On* (or no argument) is chosen, then the robot checks that the programmed wrist axes con-

figuration is reachable from the current position. If it cannot be reached, the program execution stops. However, it is possible to restart the program again, although the wrist axes will continue to the wrong configuration. At a stop point, the robot will check that the configurations of all axes are achieved, not only the wrist axes.

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

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

Monitoring is active by default. This is automatically set

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

Syntax

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

Related information

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

CONNECT Connects an interrupt to a trap routine

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

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

Example

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

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

Arguments

CONNECT Interrupt WITH Trap routine

Interrupt Data type: *intnum*

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

Trap routine Identifier

The name of the trap routine.

Program execution

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

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

Limitations

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

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

Error handling

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

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

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

These errors can be handled in the ERROR handler.

Syntax

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

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

Related information

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

DeactUnit Deactivates a mechanical unit

DeactUnit is used to deactivate a mechanical unit.

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

Examples

```
DeactUnit orbit_a;
```

Deactivation of the *orbit_a* mechanical unit.

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

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

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

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

Arguments

DeactUnit MecUnit

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

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

Program execution

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

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

Limitations

Instruction DeactUnit cannot be used

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

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

Syntax

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

Related information

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

Decr Decrements by 1

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

Example

Decr reg1;

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

Arguments

Decr Name

Name Data type: *num*

The name of the variable or persistent to be decremented.

Example

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

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

Syntax

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

Related information

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

EOffsOff Deactivates an offset for external axes

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

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

Examples

```
EOffsOff;
```

Deactivation of the offset for external axes.

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

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

Program execution

Active offsets for external axes are reset.

Syntax

EOffsOff ';'

Related information

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

EOffsOn Activates an offset for external axes

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

Examples

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

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

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

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

Arguments

EOffsOn [\ExeP] ProgPoint

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

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

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

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

Program execution

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

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

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

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

The external axes' offset is automatically reset

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

Example

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

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

Syntax

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

Related information

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

EOffsSet Activates an offset for external axes using a value

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

Example

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

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

- The ExtOffs coordinate system is displaced 100 mm for the logical axis “a” (see Figure 1).
- As long as this offset is active, all positions will be displaced 100 mm in the direction of the x-axis.

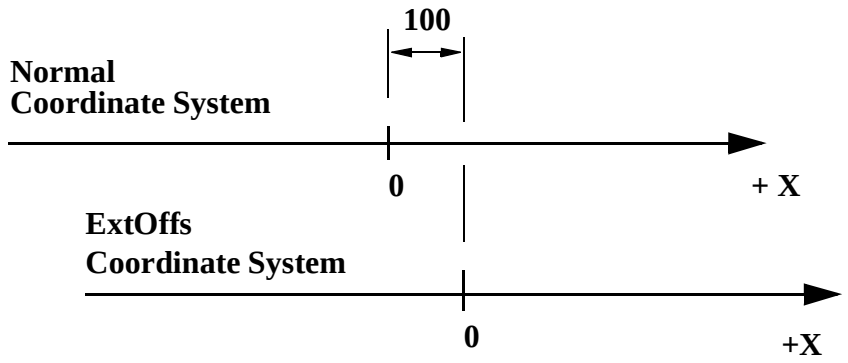


Figure 1 Displacement of an external axis.

Arguments

EOffsSet EAxOffs

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

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

- mm for linear axes
- degrees for rotating axes

Program execution

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

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

The external axes' offset is automatically reset

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

Syntax

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

Related information

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

ErrWrite Write an Error Message

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

Example

```
ErrWrite “PLC error” , “Fatal error in PLC”;
Stop;
```

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

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

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

Arguments

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

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

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

Header Data type: *string*

Error message heading (max. 24 characters).

Reason Data type: string

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

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

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

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

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

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

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

Program execution

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

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

Syntax

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

Related information

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

EXIT Terminates program execution

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

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

Example

```
EXIT;
```

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

Syntax

```
EXIT ';'
```

Related information

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

FOR Repeats a given number of times

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

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

Example

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

Repeats the *routine1* procedure *10* times.

Arguments

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

Loop counter Identifier

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

Start value Data type: *Num*

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

End value Data type: *Num*

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

Step value Data type: *Num*

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

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

Example

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

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

Program execution

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

Limitations

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

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

Syntax

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

<loop variable> ::= <identifier>

Related information

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

GOTO Goes to a new instruction

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

Examples

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

Program execution continues with the instruction following *next*.

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

The *next* program loop is executed five times.

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

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

Arguments

GOTO Label

Label Identifier

The label from where program execution is to continue.

Limitations

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

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

instruction.

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

Syntax

(EBNF)
GOTO <identifier>’;’

Related information

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

GripLoad Defines the payload of the robot

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

Examples

GripLoad piece1;

The robot gripper holds a load called *piece1*.

GripLoad load0;

The robot gripper releases all loads.

Arguments

GripLoad Load

Load Data type: *loaddata*

The load data that describes the current payload.

Program execution

The specified load affects the performance of the robot.

The default load, 0 kg, is automatically set

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

Syntax

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

Related information

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

IDelete Cancels an interrupt

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

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

Example

IDelete feeder_low;

The interrupt *feeder_low* is cancelled.

Arguments

IDelete Interrupt

Interrupt Data type: *intnum*

The interrupt identity.

Program execution

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

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

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

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

Syntax

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

Related information

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

IDisable Disables interrupts

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

Example

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

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

Program execution

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

Syntax

IDisable';'

Related information

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

IEnable Enables interrupts

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

Example

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

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

Program execution

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

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

Syntax

IEnable‘;’

Related information

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

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

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

Examples

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

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

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

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

Arguments

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

Condition Data type: *bool*

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

Example

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

ENDIF

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

Program execution

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

Syntax

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

Related information

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

Incr Increments by 1

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

Example

Incr reg1;

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

Arguments

Incr Name

Name Data type: *num*

The name of the variable or persistent to be changed.

Example

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

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

Syntax

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

Related information

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

IndAMove Independent Absolute position Movement

IndAMove is used to change an axis to independent mode and move the axis to a specific position.

An independent axis is an axis moving independently of other axes in the robot system. As program execution continues immediately, it is possible to execute other instructions (including positioning instructions) during the time the independent axis is moving.

If the axis is to be moved within a revolution, the instruction *IndRMove* should be used instead. If the move is to occur a short distance from the current position, the instruction *IndDMove* must be used.

Example

IndAMove Station_A,2\ToAbsPos:=p4,20;

Axis *2* of *Station_A* is moved to the position p4 at the speed *20* degrees/s.

Arguments

IndAMove MecUnit Axis [\ToAbsPos] | [\ToAbsNum] Speed [\Ramp]

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).

[\ToAbsPos] *(To Absolute Position)* Data type: *robtarget*

Axis position specified as a *robtarget*. Only the component for this specific axis is used. The value is used as an absolute position value in degrees (mm for linear axes).

The axis position will be affected if the axis is displaced using the instruction *EOffsSet* or *EOffsOn*.

For robot axes, the argument *\ToAbsNum* is to be used instead.

[\ToAbsNum] *(To Absolute Numeric value)* Data type: *num*

Axis position defined in degrees (mm for linear axis).

Using this argument, the position will NOT be affected by any displacement, e.g. *EOffsSet* or *PDispOn*.

Same function as *\ToAbsPos* but the position is defined as a numeric value to make it easy to manually change the position.

Speed Data type: *num*

Axis speed in degrees/s (mm/s for linear axis).

[\Ramp] Data type: *num*

Decrease acceleration and deceleration from maximum performance (1 - 100%, 100% = maximum performance).

Program execution

When *IndAMove* is executed, the specified axis starts to move at the programmed speed to the specified axis position. If *\Ramp* is programmed, there will be a reduction of acceleration/deceleration.

To change the axis back to normal mode, the *IndReset* instruction is used. In connection with this, the logical position of the axis can be changed, so that a number of revolutions are erased from the position, for example, to avoid rotating back for the next movement.

The speed can be altered by executing another *IndAMove* instruction (or another *Ind_Move* instruction). If a speed in the opposite direction is selected, the axis stops and then accelerates to the new speed and direction.

For stepwise execution of the instruction, the axis is set in independent mode only. The axis begins its movement when the next instruction is executed, and continues as long as program execution takes place. For more information see Chapter 6, Motion and I/O principles.

When the program pointer is moved to the start of the program, or to a new routine, all axes are automatically set to normal, without changing the measurement system (equivalent to executing the instruction *IndReset\Old*).

Note that an *IndAMove* instruction after an *IndCMove* operation can result in the axis spinning back the movement performed in the *IndCMove* instruction. To prevent this, use an *IndReset* instruction before the *IndAMove*, or use an *IndRMove* instruction.

Limitations

Axes in independent mode cannot be jogged. If an attempt is made to execute the axis manually, the axis will not move, and an error message will be displayed. Execute an *IndReset* instruction or move the program pointer to main, in order to leave independent mode.

If a loss of voltage occurs when an axis is in independent mode, the program cannot be restarted. An error message is displayed and the program must be started from the beginning.

Example

```
ActUnit Station_A;
weld_stationA;
IndAMove Station_A,1\ToAbsNum:=90,20\Ramp:=50;
ActUnit Station_B;
weld_stationB_1;
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;
DeactUnit Station_A;
weld_stationB_2;
```

Station_A is activated and the welding is started in station A.

Station_A (axis 1) is then moved to the 90 degrees position while the robot is welding in station B. The speed of the axis is 20 degrees/s . The speed is changed with acceleration/deceleration reduced to 50% of max performance.

When station A reaches this position, it is deactivated and reloading can take place in the station at the same time as the robot continues to weld in station B.

Error handling

If the axis is not activated, the system variable ERRNO is set to ERR_AXIS_ACT. This error can then be handled in the error handler.

Syntax

IndAMove
[MecUnit':='] < variable (**VAR**) of *mecunit*> ','
[Axis':='] < expression (**IN**) of *num*>
['\'ToAbsPos':='] < expression (**IN**) of *robtarget*>
| ['\' ToAbsNum':='] < expression (**IN**) of *num*> ','
[Speed ':='] < expression (**IN**) of *num*>
['\' Ramp':=' < expression (**IN**) of *num* >] ';'

Related information

	Described in:
Independent axes in general	Motion and I/O Principles - *Program execution*
Change back to normal mode	Instructions - *IndReset*
Reset the measurement system	Instructions - *IndReset*
Move an independent axis to a specific position within current revolution	Instructions - *IndRMove*
Move an independent axis a specific distance	Instructions - *IndDMove*
Check the speed status for independent axes	Functions - *IndSpeed*
Check the position status for independent axes	Functions - *IndInpos*

IndCMove Independent Continuous Movement

IndCMove is used to change an axis to independent mode and start the axis moving continuously at a specific speed.

An independent axis is an axis moving independently of other axes in the robot system. As program execution continues immediately, it is possible to execute other instructions (including positioning instructions) during the time the independent axis is moving.

Example

IndCMove Station_A,2,-30.5;

Axis *2* of *Station_A* starts to move in a negative direction at a speed of *30.5* degrees/s.

Arguments

IndCMove MecUnit Axis Speed [\Ramp]

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).

Speed Data type: *num*

Axis speed in degrees/s (mm/s for linear axis).
The direction of movement is specified as the sign of the speed argument.

[\Ramp] Data type: *num*

Decrease acceleration and deceleration from maximum performance (1 - 100%, 100% = maximum performance).

Program execution

When *IndCMove* is executed, the specified axis starts to move at the programmed speed. The direction of movement is specified as the sign of the speed argument. If *\Ramp* is programmed there will be a reduction of acceleration/deceleration.

To change the axis back to normal mode, the *IndReset* instruction is used. The logical position of the axis can be changed in connection with this - an number of full revolutions can be erased, for example, to avoid rotating back for the next movement.

The speed can be changed by executing a further *IndCMove* instruction. If a speed in the opposite direction is ordered, the axis stops and then accelerates to the new speed and direction. To stop the axis, speed argument 0 can be used. It will then still be in independent mode.

During stepwise execution of the instruction, the axis is set in independent mode only. The axis starts its movement when the next instruction is executed, and continues as long as program execution continues. For more information see Chapter 6, Motion and I/O principles.

When the program pointer is moved to the beginning of the program, or to a new routine, all axes are set automatically to normal mode, without changing the measurement system (equivalent to executing the instruction *IndReset\Old*).

Limitations

The resolution of the axis position worsens, the further it is moved from its logical zero position (usually the middle of the working area). To achieve high resolution again, the logical working area can be set to zero with the instruction *IndReset*. For more information see Chapter 6, Motion and I/O Principles.

Axes in independent mode cannot be jogged. If an attempt is made to execute the axis manually, the axis will not move, and an error message will be displayed. Execute an *IndReset* instruction or move the program pointer to main, in order to leave independent mode.

If a loss of voltage occurs when the axis is in independent mode, the program cannot be restarted. An error message is displayed, and the program must be started from the beginning.

Example

```
IndCMove Station_A,2,20;
WaitUntil IndSpeed(Station_A,2 \InSpeed) = TRUE;
WaitTime 0.2;
MoveL p10, v1000, fine, tool1;
IndCMove Station_A,2,-10\Ramp:=50;
MoveL p20, v1000, z50, tool1;
IndRMove Station_A,2 \ToRelPos:=p1 \Short,10;
MoveL p30, v1000, fine, tool1;
WaitUntil IndInpos(Station_A,2 ) = TRUE;
WaitTime 0.2;
IndReset Station_A,2 \RefPos:=p40\Short;
MoveL p40, v1000, fine, tool1;
```

Axis *2* of *Station_A* starts to move in a positive direction at a speed of 20 degrees/s. When this axis has reached the selected speed the robot axes start to move.

When the robot reaches position *p10*, the external axis changes direction and rotates at a speed of 10 degrees/s . The change of speed is performed with acceleration/deceleration reduced to 50% of maximum performance. At the same time, the robot executes towards *p20*.

Axis *2* of *Station_A* is then stopped as quickly as possible in position *p1* within the current revolution.

When axis 2 has reached this position, and the robot has stopped in position p30, axis 2 returns to normal mode again. The measurement system offset for this axis is changed a whole number of axis revolutions so that the actual position is as close as possible to *p40*.

When the robot is then moved to position *p40*, axis *2* of *Station_A* will be moved via the shortest route to position *p40* (max ±180 degrees).

Error handling

If the axis is not activated, the system variable ERRNO is set to ERR_AXIS_ACT. This error can then be handled in the error handler.

Syntax

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

Related information

	Described in:
Independent axes in general	Motion and I/O Principles - *Program execution*
Change back to normal mode	Instructions - *IndReset*
Reset the measurement system	Instructions - *IndReset*
Move an independent axis to a specific position	Instructions - *IndAMove*, *IndRMove*
Move an independent axis a specific distance	Instructions - *IndDMove*
Check the speed status for independent axes	Functions - *IndSpeed*
Check the position status for independent axes	Functions - *IndInpos*

IndDMove Independent Delta position Movement

IndDMove is used to change an axis to independent mode and move the axis a specific distance.

An independent axis is an axis moving independently of other axes in the robot system. As program execution continues immediately, it is possible to execute other instructions (including positioning instructions) during the time the independent axis is moving.

If the axis is to be moved to a specific position, the instruction *IndAMove* or *IndRMove* must be used instead.

Example

IndDMove Station_A,2,-30,20;

Axis *2* of *Station_A* is moved 30 degrees in a negative direction at a speed of *20* degrees/s.

Arguments

IndDMove MecUnit Axis Delta Speed [\Ramp]

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).

Delta Data type: *num*

The distance which the current axis is to be moved, expressed in degrees (mm for linear axes). The sign specifies the direction of movement.

Speed Data type: *num*

Axis speed in degrees/s (mm/s for linear axis).

[\Ramp] Data type: *num*

Decrease acceleration and deceleration from maximum performance (1 - 100%, 100% = maximum performance).

Program execution

When *IndAMove* is executed, the specified axis starts to move at the programmed speed for the specified distance. The direction of movement is specified as the sign of the *Delta* argument. If *\Ramp* is programmed there will be a reduction of acceleration/ deceleration.

If the axis is moving, the new position is calculated from the momentary position of the axis, when the instruction *IndDMove* is executed. If an *IndDMove* instruction with distance 0 is executed, the axis will stop and then move back to the position which the axis had when the instruction was executed.

To change the axis back to normal mode, the *IndReset* instruction is used. The logical position of the axis can be changed in connection with this - a number of full revolutions can be erased from the position, for example, to avoid rotating back for the next movement.

The speed can be changed by running a further *IndDMove* instruction (or another *Ind_Move* instruction). If a speed in the opposite direction is selected, the axis stops and then accelerates to the new speed and direction.

During stepwise execution of the instruction, the axis is set in independent mode only. The axis starts its movement when the next instruction is executed, and continues as long as program execution continues. For more information see Chapter 6, Motion and I/O principles.

When the program pointer is moved to the beginning of the program, or to a new routine, all axes are automatically set to normal mode, without changing the measurement system (equivalent to running the instruction *IndReset \Old*).

Limitations

Axes in independent mode cannot be jogged. If an attempt is made to execute the axis manually, the axis will not move, and an error message will be displayed. Execute an *IndReset* instruction or move the program pointer to main, in order to leave independent mode.

If a loss of voltage occurs when the axis is in independent mode, the program cannot be restarted. An error message is displayed, and the program must be started from the beginning.

Example

```
IndAMove Robot,6\ToAbsNum:=90,20;
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;
IndDMove Station_A,2,-30,20;
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;
IndDMove Station_A,2,400,20;
```

Axis 6 of the robot is moved to the following positions:
90 degrees
60 degrees
460 degrees (1 revolution + 100 degrees).

Error handling

If the axis is not activated, the system variable ERRNO is set to ERR_AXIS_ACT. This error can then be handled in the error handler.

Syntax

IndDMove
[MecUnit':='] < variable (**VAR**) of *mecunit*> ','
[Axis':='] < expression (**IN**) of *num*> ','
[Delta':='] < expression (**IN**) of *num*>','
[Speed ':='] < expression (**IN**) of *num*>
['\' Ramp':=' < expression (**IN**) of *num* >] ';'

Related information

	Described in:
Independent axes in general	Motion and I/O Principles - *Program execution*
Change back to normal mode	Instructions - *IndReset*
Reset the measurement system	Instructions - *IndReset*
Move an independent axis to a specific position	Instructions - *IndAMove*, *IndRMove*
Check the speed status for independent axes	Functions - *IndSpeed*
Check the position status for independent axes	Functions - *IndInpos*

IndReset Independent Reset

IndReset is used to change an independent axis back to normal mode. At the same time, the measurement system for rotational axes can be moved a number of axis revolutions.

Example

```
IndCMove Station_A,2,5;
MoveL *,v1000,fine,tool1;
IndCMove Station_A,2,0;
WaitUntil IndSpeed(Station_A,2\ZeroSpeed);
WaitTime 0.2
IndReset Station_A,2;
```

Axis *2* of *Station _A* is first moved in independent mode and then changed back to normal mode. The axis will keep its position.

Note that the current independent axis, and the normal axes, should not move when the instruction *IndReset* is executed. This is because the previous position is a stop point, and an *IndCMove* instruction is executed at zero speed. Furthermore, a pause of 0.2 seconds is used to ensure that the correct status has been achieved.

Arguments

IndReset MecUnit Axis [\RefPos] | [\RefNum] [\Short] | [\Fwd] | [\Bwd] | [\Old]

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).

[\RefPos] *(Reference Position)* Data type: *robtarget*

Axis position specified as a *robtarget*. Only the component for this specific axis is used. The position must be inside the normal working range.

For robot axes, the argument *\RefNum* is to be used instead.

The argument is only to be defined together with the argument *\Short*, *\Fwd* or *\Bwd*. It is not allowed together with the argument *\Old*.

[\RefNum] *(Reference Numeric value)* Data type: *num*

Axis position defined in degrees (mm for linear axis). The position must be inside the normal working range.

The argument is only to be defined together with the argument *\Short*, *\Fwd* or *\Bwd*. It is not allowed together with the argument *\Old*.

Same function as *\RefPos* but the position is defined as a numeric value to make it easy to change the position manually.

[\Short] Data type: *switch*

The measurement system will change a whole number of revolutions on the axis side so that the axis will be as close as possible to the specified *\RefPos* or *\RefNum* position. If a positioning instruction with the same position is executed after *IndReset*, the axis will travel the shortest route, less than ±180 degrees, in order to reach the position.

[\Fwd] *(Forward)* Data type: *switch*

The measurement system will change a whole number of revolutions on the axis side so that the reference position will be on the positive side of the specified *\RefPos* or *\RefNum* position. If a positioning instruction with the same position is executed after *IndReset*, the axis will turn in a positive direction less than 360 degrees in order to reach the position.

[\Bwd] *(Backward)* Data type: *switch*

The measurement system will change a whole number of revolutions on the axis side so that the reference position will be on the negative side of the specified *\RefPos* or *\RefNum* position. If a positioning instruction with the same position is executed after *IndReset*, the axis will turn in a negative direction less than 360 degrees in order to reach the position.

[\Old] Data type: *switch*

Keeps the old position. Note that resolution is decreased in positions far away from zero.

If no argument *\Short*, *\Fwd*, *\Bwd* or *\Old* is specified - *\Old* is used as default value.

Program execution

When *IndReset* is executed, it changes the independent axis back to normal mode. At the same time, the measurement system for the axis can be moved by a whole number of axis revolutions.

The instruction may also be used in normal mode in order to change the measurement system.

Note that the position is used only to adjust the measurement system - the axis will not move to the position.

Limitations

The instruction may only be executed when all active axes running in normal mode are standing still. The independent mode axis which is going to be changed to normal mode must also be stationary. For axes in normal mode this is achieved by executing a move instruction with the argument *fine*. The independent axis is stopped by an *IndCMove* with *Speed*:=0 (followed by a wait period of 0.2 seconds), *IndRMove*, *IndAMove* or *IndDMove* instruction.

The resolution of positions is decreased when moving away from logical position 0. An axis which progressively rotates further and further from the position 0 should thus be set to zero using the instruction *IndReset* with an argument other than *\Old*.

The measurement system cannot be changed for linear axes.

To ensure a proper start after *IndReset* of an axis with a relative measured measurement system (synchronization switches), an extra time delay of 0.12 seconds must be added after the *IndReset* instruction.

Example

```
IndAMove Station_A,1\ToAbsNum:=750,50;
WaitUntil IndInpos(Station_A,1);
WaitTime 0.2;
IndReset Station_A,1 \RefNum:=0 \Short;
.
IndAMove Station_A,1\ToAbsNum:=750,50;
WaitUntil IndInpos(Station_A,1);
WaitTime 0.2;
IndReset Station_A,1 \RefNum:=300 \Short;
```

Axis 1 in *Station_A* is first moved independently to the 750 degrees position (2 revolutions and 30 degrees). At the same time as it changes to normal mode, the logical position is set to 30 degrees.

Axis 1 in *Station_A* is subsequently moved to the 750 degrees position (2 revolutions and 30 degrees). At the same time as it changes to normal mode, the logical position is set to 390 degrees (1 revolution and 30 degrees).

Error handling

If the axis is moving, the system variable ERRNO is set to ERR_AXIS_MOVING.

If the axis is not activated, the system variable ERRNO is set to ERR_AXIS_ACT. This error can then be handled in the error handler.

Syntax

IndReset
[MecUnit':='] < variable (**VAR**) of *mecunit*> ','
[Axis':='] < expression (**IN**) of *num*>
['\' RefPos':=' < expression (**IN**) of robtarget>]
| ['\' RefNum':=' < expression (**IN**) of num>]
['\' Short] | ['\' Fwd] | ['\' Bwd] | ['\' Old]';'

Related information

	Described in:
Independent axes in general	Motion and I/O Principles - *Program execution*
Change an axis to independent mode	Instructions - *IndAMove*, *IndCMove*, *IndDMove*, *IndRMove*
Check the speed status for independent axes	Functions - *IndSpeed*
Check the position status for independent axes	Functions - *IndInpos*

IndRMove Independent Relative position Movement

IndRMove is used to change a rotational axis to independent mode and move the axis to a specific position within one revolution.

An independent axis is an axis moving independently of other axes in the robot system. As program execution continues immediately, it is possible to execute other instructions (including positioning instructions) during the time the independent axis is moving.

If the axis is to be moved to an absolute position (several revolutions) or if the axis is linear, the instruction *IndAMove* is used instead. If the movement is to take place a certain distance from the current position, the instruction *IndDMove* must be used.

Example

```
IndRMove Station_A,2\ToRelPos:=p5 \Short,20;
```

Axis *2* of *Station_A* is moved the shortest route to position *p5* within one revolution (maximum rotation ± 180 degrees) at a speed of *20* degrees/s.

Arguments

IndRMove MecUnit Axis [\ToRelPos] | [\ToRelNum] [\Short] | [\Fwd] | [\Bwd] Speed [\Ramp]

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).

[\ToRelPos] *(To Relative Position)* Data type: *robtarget*

Axis position specified as a *robtarget*. Only the component for this specific axis is used. The value is used as a position value in degrees within one axis revolution. This means that the axis moves less than one revolution.

The axis position will be affected if the axis is displaced using the instruction *EOffsSet* or *EOffsOn*.

For robot axes, the argument *\ToRelNum* is to be used instead.

[\ToRelNum] *(To Relative Numeric value)* Data type: *num*

Axis position defined in degrees.

Using this argument, the position will NOT be affected by any displacement, e.g. *EOffsSet* or *PDispOn*.

Same function as *\ToRelPos* but the position is defined as a numeric value to make it easy to change the position manually.

[\Short] Data type: *switch*

The axis is moved the shortest route to the new position. This means that the maximum rotation will be 180 degrees in any direction. The direction of movement therefore depends on the current location of the axis.

[\Fwd] *(Forward)* Data type: *switch*

The axis is moved in a positive direction to the new position. This means that the maximum rotation will be 360 degrees and always in a positive direction (increased position value).

[\Bwd] *(Backward)* Data type: *switch*

The axis is moved in a negative direction to the new position. This means that the maximum rotation will be 360 degrees and always in a negative direction (decreased position value).

If *\Short*, *\Fwd* or *\Bwd* argument is omitted, *\Short* is used as default value.

Speed Data type: *num*

Axis speed in degrees/s.

[\Ramp] Data type: *num*

Decrease acceleration and deceleration from maximum performance (1 - 100%, 100% = maximum performance).

Program execution

When *IndRMove* is executed, the specified axis starts to move at the programmed speed to the specified axis position, but only a maximum of one revolution. If *\Ramp* is programmed, there will be a reduction of acceleration/deceleration.

To change the axis back to normal mode, the *IndReset* instruction is used. The logical position of the axis can be changed in connection with this - a number of full revolutions can be erased from the position, for example, to avoid rotating back for the next movement.

The speed can be changed by running a further *IndRMove* instruction (or another *Ind_Move* instruction). If a speed in the opposite direction is selected, the axis stops and then accelerates to the new speed and direction.

During stepwise execution of the instruction, the axis is set in independent mode only. The axis starts its movement when the next instruction is executed, and continues as long as program execution continues. For more information see Chapter 6, Motion and I/O principles.

When the program pointer is moved to the beginning of the program, or to a new routine, all axes are automatically set to normal mode, without changing the measurement system (equivalent to running the instruction *IndReset \Old*).

Limitations

Axes in independent mode cannot be jogged. If an attempt is made to execute the axis manually, the axis will not move, and an error message will be displayed. Execute an *IndReset* instruction or move the program pointer to main, in order to leave independent mode.

If a loss of voltage occurs when the axis is in independent mode, the program cannot be restarted. An error message is displayed, and the program must be started from the beginning.

Examples

```
IndRMove Station_A,1\ToRelPos:=p5 \Fwd,20\Ramp:=50;
```

Axis *1* of *Station_A* starts to move in a positive direction to the position *p5* within one revolution (maximum rotation 360 degrees) at a speed of *20* degrees/s. The speed is changed with acceleration/deceleration reduced to 50% of maximum performance.

```
IndAMove Station_A,1\ToAbsNum:=90,20;
WaitUntil IndInpos(Station_A,1 ) = TRUE;
IndRMove Station_A,1\ToRelNum:=80 \Fwd,20;
WaitTime 0.2;
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;
IndRMove Station_A,1\ToRelNum:=50 \Bwd,20;
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;
IndRMove Station_A,1\ToRelNum:=150 \Short,20;
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;
IndAMove Station_A,1\ToAbsNum:=10,20;
```

Axis 1 of *Station_A* is moved to the following positions:
90 degrees
440 degrees (1 revolution + 80 degrees)
410 degrees (1 revolution + 50 degrees)
510 degrees (1 revolution + 150 degrees)
10 degrees

Error handling

If the axis is not activated, the system variable ERRNO is set to ERR_AXIS_ACT. This error can then be handled in the error handler.

Syntax

IndRMove
[MecUnit':='] < variable (**VAR**) of *mecunit*> ','
[Axis':='] < expression (**IN**) of *num*>
['\'ToRelPos':=' < expression (**IN**) of *robtargets*>]
| ['\'ToRelNum':=' < expression (**IN**) of *num*>]
['\'Short] | ['\' Fwd] | ['\' Bwd] ','
[Speed ':='] < expression (**IN**) of *num*>
['\'Ramp':=' < expression (**IN**) of *num* >] ';'

Related information

	Described in:
Independent axes in general	Motion and I/O Principles - *Program execution*
Change back to normal mode	Instructions - *IndReset*
Reset the measurement system	Instructions - *IndReset*
Move an independent axis to an absolute position	Instructions - *IndAMove*
Move an independent axis a specific distance	Instructions - *IndDMove*
More examples	Instructions - *IndCMove*
Check the speed status for independent axes	Functions - *IndSpeed*
Check the position status for independent axes	Functions - *IndInpos*

InvertDO Inverts the value of a digital output signal

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

Example

InvertDO do15;

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

Arguments

InvertDO Signal

Signal Data type: *signaldo*

The name of the signal to be inverted.

Program execution

The current value of the signal is inverted (see Figure 1).

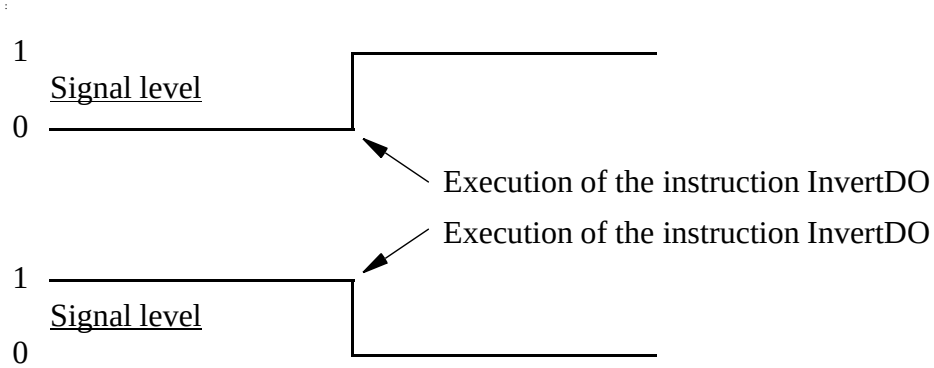


Figure 1 Inversion of a digital output signal.

Syntax

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

Related information

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

ISignalDI Orders interrupts from a digital input signal

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

System signals can also generate interrupts.

Examples

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

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

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

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

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

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

Arguments

ISignalDI [\Single] Signal TriggValue Interrupt

[\Single] Data type: *switch*

Specifies whether the interrupt is to occur once or cyclically.

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

Signal Data type: *signaldi*

The name of the signal that is to generate interrupts.

TriggValue Data type: *dionum*

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

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

Interrupt Data type: *intnum*

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

Program execution

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

If the signal changes to the specified value before the interrupt is ordered, no interrupt occurs (see Figure 1).

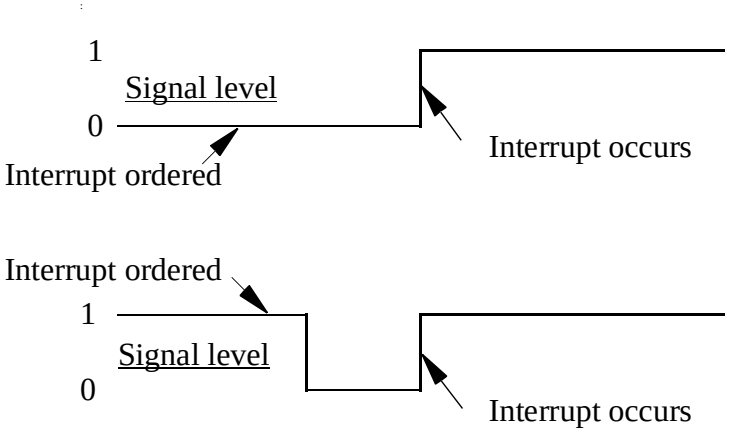


Figure 1 Interrupts from a digital input signal at signal level 1.

Limitations

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

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

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

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

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

Syntax

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

Related information

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

ISignalDO Interrupts from a digital output signal

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

System signals can also generate interrupts.

Examples

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

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

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

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

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

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

Arguments

ISignalDO [\Single] Signal TriggValue Interrupt

[\Single] Data type: *switch*

Specifies whether the interrupt is to occur once or cyclically.

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

Signal Data type: *signaldo*

The name of the signal that is to generate interrupts.

TriggValue Data type: *dionum*

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

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

Interrupt Data type: *intnum*

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

Program execution

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

If the signal changes to the specified value before the interrupt is ordered, no interrupt occurs (see Figure 1).

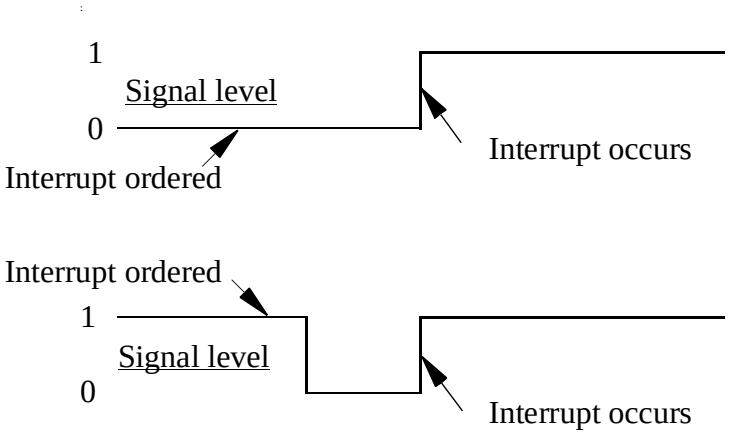


Figure 1 Interrupts from a digital output signal at signal level 1.

Limitations

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

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

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

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

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

Syntax

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

Related information

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

ISleep Deactivates an interrupt

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

Example

ISleep sig1int;

The interrupt *sig1int* is deactivated.

Arguments

ISleep Interrupt

Interrupt Data type: *intnum*

The variable (interrupt identity) of the interrupt.

Program execution

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

Example

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

Communication across the ch1 serial channel is monitored by means of interrupts which are generated every *60* seconds. The trap routine checks whether the

communication is working. When, however, communication is in progress, these interrupts are not permitted.

Error handling

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

Syntax

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

Related information

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

ITimer Orders a timed interrupt

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

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

Examples

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

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

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

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

Arguments

ITimer [\Single] Time Interrupt

[\Single] Data type: *switch*

Specifies whether the interrupt is to occur once or cyclically.

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

Time Data type: *num*

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

The value is specified in seconds. If *Single* is set, this time may not be less than 0.1 seconds. The corresponding time for cyclical interrupts is 0.5 seconds.

Interrupt Data type: *intnum*

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

Program execution

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

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

Example

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

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

Limitations

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

Syntax

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

Related information

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

IWatch Activates an interrupt

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

Example

IWatch sig1int;

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

Arguments

IWatch Interrupt

Interrupt Data type: *intnum*

Variable (interrupt identity) of the interrupt.

Program execution

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

Example

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

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

Error handling

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

Syntax

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

Related information

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

label Line name

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

Example

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

Program execution continues with the instruction following *next*.

Arguments

Label:

Label Identifier

The name you wish to give the line.

Program execution

Nothing happens when you execute this instruction.

Limitations

The label must not be the same as

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

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

Syntax

(EBNF)
<identifier>':'

Related information

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

Load Load a program module during execution

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

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

Example

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

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

Arguments

Load FilePath [\File]

FilePath

Data type: *string*

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

[\File]

Data type: *string*

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

Program execution

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

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

After the program module is loaded it will be linked and initialised. The initialisation of the loaded module sets all variables at module level to their init values. Unresolved references will be accepted if the system parameter for *Load* is set (*BindRef* = NO). However, when the program is started, the teach pendant function Program/File/Check program will not check for unresolved references if the parameter *BindRef* = NO. There will be a run time error on execution of an unresolved reference.

Examples

Load "ram1disk:DOORDIR/DOOR1.MOD";

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

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

Same as above but another syntax.

Limitations

It is not allowed to load a program module that contains a main routine.

TRAP routines, system I/O events and other program tasks cannot execute during the loading.

Avoid ongoing robot movements during the loading.

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

A program stop during execution of the *Load* instruction results in a guard stop with motors off and the error message "20025 Stop order timeout" will be displayed on the Teach Pendant.

Error handling

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

Syntax

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

Related information

	Described in:
Unload a program module	Instructions - *UnLoad*
Accept unresolved references	System Parameters - *Controller* System Parameters - *Tasks* System Parameters - *BindRef*

MoveAbsJMoves the robot to an absolute joint position

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

This instruction need only be used when:

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

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

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

Examples

MoveAbsJ p50, v1000, z50, tool2;

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

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

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

Arguments

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

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

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

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

If this argument is omitted, the subsequent instruction is only executed after the robot has reached the specified zone.

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

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

Speed Data type: *speeddata*

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

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

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

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

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

Zone Data type: *zonedata*

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

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

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

Tool Data type: *tooldata*

The tool in use during the movement.

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

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

The work object used during the movement.

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

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

Program execution

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

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

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

Examples

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

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

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

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

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

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

Error handling

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

Limitations

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

In order to be able to run backwards with the instruction *MoveAbsJ* involved, and

avoiding problems with singular points or ambiguous areas, it is essential that the subsequent instructions fulfil certain requirements, as follows (see Figure 1).

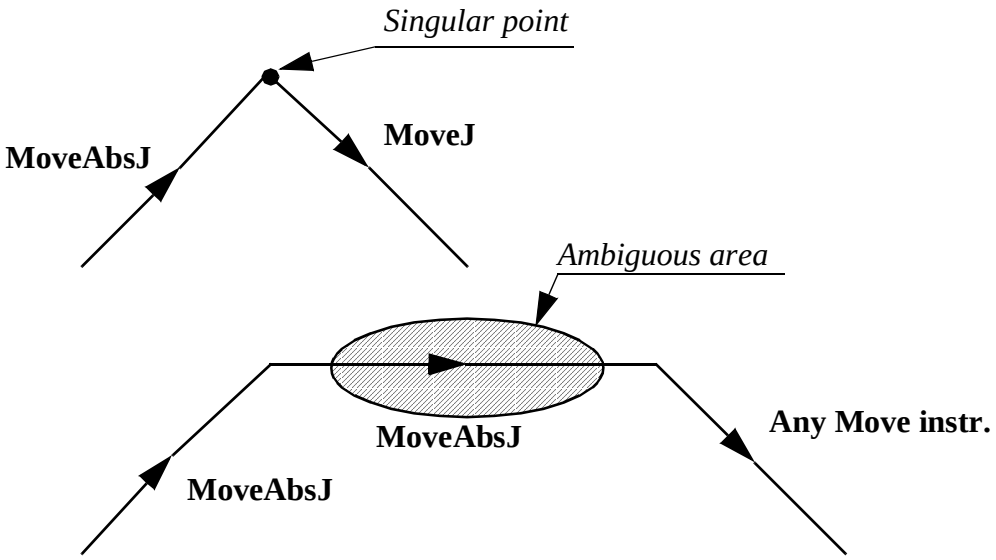


Figure 1 Limitation for backward execution with MoveAbsJ.

Syntax

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

Related information

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

MoveC Moves the robot circularly

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

Examples

MoveC p1, p2, v500, z30, tool2;

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

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

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

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

A complete circle is performed if the positions are the same as those shown in Figure 1.

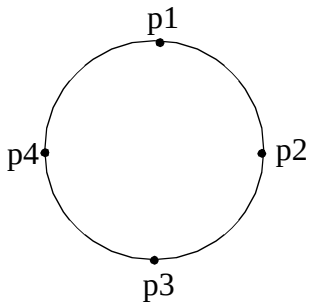


Figure 1 A complete circle is performed by two MoveC instructions.

Arguments

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

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

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

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

If this argument is omitted, the subsequent instruction is only executed after the robot has reached the specified zone.

CirPoint Data type: *robtarget*

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

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

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

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

Zone Data type: *zonedata*

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

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

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

Tool Data type: *tooldata*

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

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

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

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

Program execution

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

- The TCP of the tool is moved circularly at constant programmed velocity.
- The tool is reoriented at a constant velocity from the orientation at the start position to the orientation at the destination point.
- The reorientation is performed relative to the circular path. Thus, if the orientation relative to the path is the same at the start and the destination points, the relative orientation remains unchanged during the movement (see Figure 2).

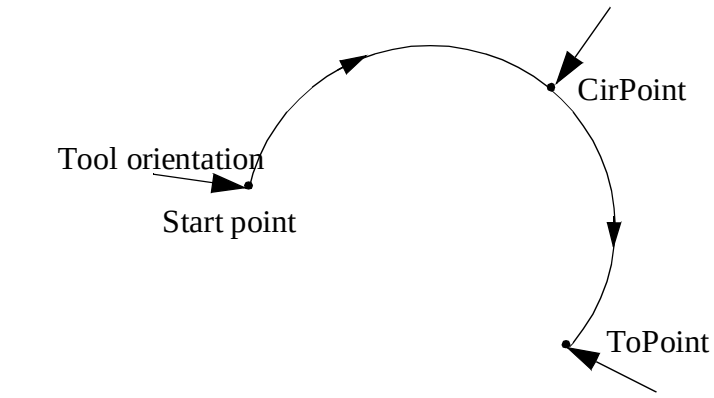


Figure 2 Tool orientation during circular movement.

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

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

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

Examples

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

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

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

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

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

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

Limitations

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

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

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

Syntax

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

Related information

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

MoveJ Moves the robot by joint movement

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

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

Examples

MoveJ p1, vmax, z30, tool2;

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

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

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

Arguments

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

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

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

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

If this argument is omitted, the subsequent instruction is only executed after the robot has reached the specified zone.

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

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

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

Zone Data type: *zonedata*

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

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

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

Tool Data type: *tooldata*

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

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

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

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

Program execution

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

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

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

Examples

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

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

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

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

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

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

Syntax

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

Related information

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

MoveL Moves the robot linearly

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

Example

MoveL p1, v1000, z30, tool2;

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

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

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

Arguments

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

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

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

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

If this argument is omitted, the subsequent instruction is only executed after the robot has reached the specified zone.

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

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

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

Zone Data type: *zonedata*

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

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

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

Tool Data type: *tooldata*

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

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

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

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

Program execution

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

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

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

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

Examples

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

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

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

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

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

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

Syntax

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

Related information

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

Open Opens a file or serial channel

Open is used to open a file or serial channel for reading or writing.

Example

```
VAR iodev logfile;
.
Open "flp1:LOGDIR" \File:= "LOGFILE1.DOC",logfile;
```

The file *LOGFILE1.DOC* in unit *flp1:* (diskette), directory *LOGDIR*, is opened for writing. The reference name *logfile* is used later in the program when writing to the file.

Arguments

Open Object [\File] IODevice [\Read] | [\Write] | [\Append] | [\Bin]

Object Data type: *string*

The I/O object that is to be opened, e.g. "flp1:", "ram1disk:".

[\File] Data type: *string*

The name of the file. This name can also be specified in the argument *Object*, e.g. "flp1:LOGDIR/LOGFILE.DOC".

IODevice Data type: *iodev*

A reference to the file or serial channel to open. This reference is then used for reading from and writing to the file/channel.

The arguments *\Read*, *\Write*, *\Append* and *\Bin* are mutually exclusive. If none of these are specified, the instruction acts in the same way as the *\Write* argument.

[\Read] Data type: *switch*

Opens a character-based file or serial channel for reading. When reading from a file, the reading is started from the beginning of the file.

[\Write] Data type: *switch*

Opens a character-based file or serial channel for writing. If the selected file already exists, its contents are deleted. Anything subsequently written is written at the start of the file.

[\Append] Data type: *switch*

Opens a character-based file or serial channel for writing. If the selected file already exists, anything subsequently written is written at the end of the file.

[\Bin] Data type: *switch*

Opens a binary serial channel for reading and writing.
Works as append, i.e. file pointer at end of file.

Example

```
VAR iodev printer;
.
Open "sio1:", printer \Bin;
Write printer, "This is a message to the printer";
Close printer;
```

The serial channel *sio1:* is opened for binary reading and writing. The reference name *printer* is used later when writing to and closing the serial channel.

Program execution

The specified serial channel/file is activated so that it can be read from or written to. Several files can be open on the same unit at the same time.

Error handling

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

Syntax

```
Open
   [Object ':='] <expression (IN) of string>
   ['\'File':=' <expression (IN) of string>']' ','
   [IODevice ':='] <variable (VAR) of iodev>
   ['\'Read] | ['\'Write] | ['\'Append] | ['\'Bin] ';'
```

Related information

	Described in:
Writing to and reading from serial channels and files	RAPID Summary - *Communication*

PDispOff Deactivates program displacement

PDispOff (Program Displacement Off) is used to deactivate a program displacement.

Program displacement is activated by the instruction *PDispSet* or *PDispOn* and applies to all movements until some other program displacement is activated or until program displacement is deactivated.

Examples

```
PDispOff;
```

Deactivation of a program displacement.

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

A program displacement is defined as the difference between the positions *p10* and *p11*. This displacement affects the movement to *p20* and *p30*, but not to *p40*.

Program execution

Active program displacement is reset. This means that the program displacement coordinate system is the same as the object coordinate system, and thus all programmed positions will be related to the latter.

Syntax

PDispOff ';'

Related information

	Described in:
Definition of program displacement using two positions	Instructions - *PDispOn*
Definition of program displacement using values	Instructions - *PDispSet*

PDispOn Activates program displacement

PDispOn (Program Displacement On) is used to define and activate a program displacement using two robot positions.

Program displacement is used, for example, after a search has been carried out, or when similar motion patterns are repeated at several different places in the program.

Examples

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

Activation of a program displacement (parallel movement). This is calculated based on the difference between positions *p10* and *p20*.

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

Activation of a program displacement (parallel movement). Since a stop point has been used in the previous instruction, the argument \ExeP does not have to be used. The displacement is calculated on the basis of the difference between the robot's actual position and the programmed point (*) stored in the instruction.

```
PDispOn \Rot \ExeP:=p10, p20, tool1;
```

Activation of a program displacement including a rotation. This is calculated based on the difference between positions *p10* and *p20*.

Arguments

PDispOn [\Rot] [\ExeP] ProgPoint Tool [\WObj]

[\Rot] *(Rotation)* Data type: *switch*

The difference in the tool orientation is taken into consideration and this involves a rotation of the program.

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

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

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

The robot's original position at the time of programming.

Tool Data type: *tooldata*

The tool used during programming, i.e. the TCP to which the *ProgPoint* position is related.

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

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

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

The arguments *Tool* and *\WObj* are used both to calculate the *ProgPoint* during programming and to calculate the current position during program execution if no ExeP argument is programmed.

Program execution

Program displacement means that the ProgDisp coordinate system is translated in relation to the object coordinate system. Since all positions are related to the ProgDisp coordinate system, all programmed positions will also be displaced. See Figure 1.

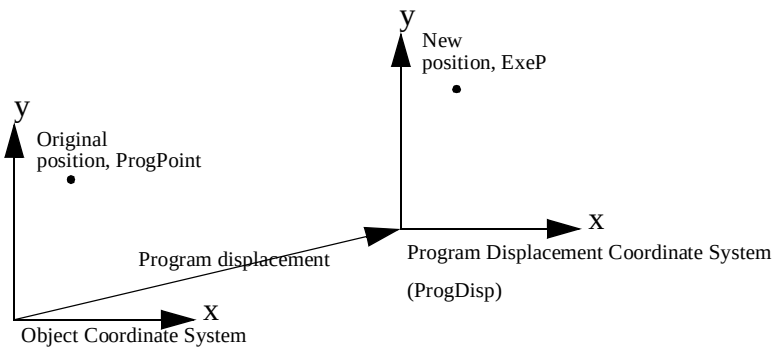


Figure 1 Displacement of a programmed position using program displacement.

Program displacement is activated when the instruction *PDispOn* is executed and remains active until some other program displacement is activated (the instruction *PDispSet* or *PDispOn*) or until program displacement is deactivated (the instruction *PDispOff*).

Only one program displacement can be active at any one time. Several *PDispOn* instructions, on the other hand, can be programmed one after the other and, in this case, the different program displacements will be added.

Program displacement is calculated as the difference between *ExeP* and *ProgPoint*. If *ExeP* has not been specified, the current position of the robot at the time of the program execution is used instead. Since it is the actual position of the robot that is used, the robot should not move when *PDispOn* is executed.

If the argument *\Rot* is used, the rotation is also calculated based on the tool orientation

at the two positions. The displacement will be calculated in such a way that the new position (*ExeP)* will have the same position and orientation in relation to the displaced coordinate system, ProgDisp, as the old position (*ProgPoint*) had in relation to the original coordinate system (see Figure 2).

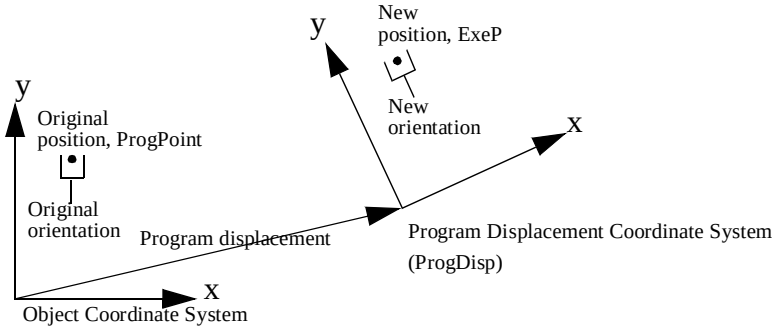


Figure 2 Translation and rotation of a programmed position.

The program displacement is automatically reset

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

Example

```
PROC draw_square()
   PDispOn *, tool1;
   MoveL *, v500, z10, tool1;
   MoveL *, v500, z10, tool1;
   MoveL *, v500, z10, tool1;
   MoveL *, v500, z10, tool1;
   PDispOff;
ENDPROC
.
MoveL p10, v500, fine, tool1;
draw_square;
MoveL p20, v500, fine, tool1;
draw_square;
MoveL p30, v500, fine, tool1;
draw_square;
```

The routine *draw_square* is used to execute the same motion pattern at three different positions, based on the positions *p10, p20* and *p30.* See Figure 3.

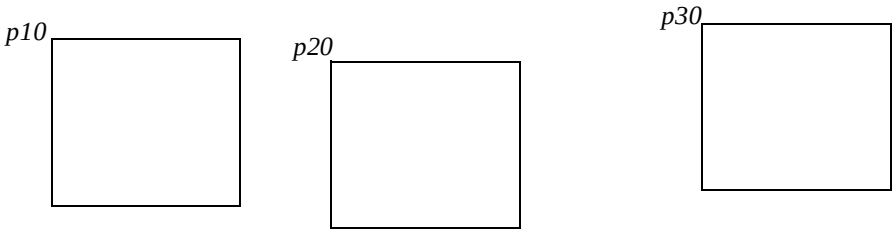


Figure 3 Using program displacement, motion patterns can be reused.

```
SearchL sen1, psearch, p10, v100, tool1\WObj:=fixture1;
PDispOn \ExeP:=psearch, *, tool1 \WObj:=fixture1;
```

A search is carried out in which the robot's searched position is stored in the position *psearch*. Any movement carried out after this starts from this position using a program displacement (parallel movement). The latter is calculated based on the difference between the searched position and the programmed point (*) stored in the instruction. All positions are based on the *fixture1* object coordinate system.

Syntax

```
PDispOn
   [ '\' Rot ',' ]
   [ '\' ExeP ':=' ] < expression (IN) of robtarget > ',']
   [ ProgPoint ':=' ] < expression (IN) of robtarget > ','
   [ Tool ':=' ] < persistent (PERS) of tooldata>
   [ '\'WObj ':=' < persistent (PERS) of wobjdata> ] ';'
```

Related information

	Described in:
Deactivation of program displacement	Instructions - *PDispOff*
Definition of program displacement using values	Instructions - *PDispSet*
Coordinate systems *tems*	Motion Principles - *Coordinate Sys-*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
More examples	Instructions - *PDispOff*

PDispSet Activates program displacement using a value

PDispSet (Program Displacement Set) is used to define and activate a program displacement using values.

Program displacement is used, for example, when similar motion patterns are repeated at several different places in the program.

Example

```
VAR pose xp100 :=  [ [100, 0, 0], [1, 0, 0, 0] ];
.
PDispSet xp100;
```

Activation of the *xp100* program displacement, meaning that:

- The ProgDisp coordinate system is displaced 100 mm from the object coordinate system, in the direction of the positive x-axis (see Figure 1).
- As long as this program displacement is active, all positions will be displaced 100 mm in the direction of the x-axis.

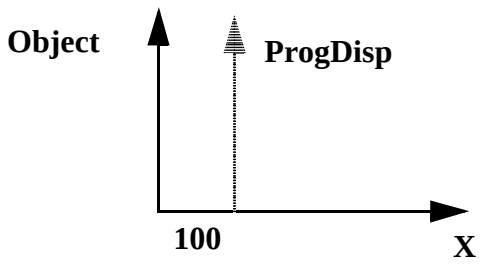


Figure 1 A 100 mm-program displacement along the x-axis.

Arguments

PDispSet DispFrame

DispFrame *(Displacement Frame)* Datatyp: *pose*

The program displacement is defined as data of the type *pose*.

Program execution

Program displacement involves translating and/or rotating the ProgDisp coordinate system relative to the object coordinate system. Since all positions are related to the ProgDisp coordinate system, all programmed positions will also be displaced. See Figure 2.

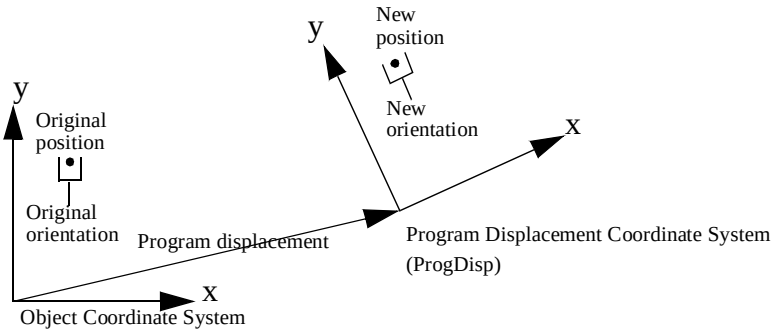


Figure 2 Translation and rotation of a programmed position.

Program displacement is activated when the instruction *PDispSet* is executed and remains active until some other program displacement is activated (the instruction *PDispSet* or *PDispOn*) or until program displacement is deactivated (the instruction *PDispOff*).

Only one program displacement can be active at any one time. Program displacements cannot be added to one another using *PDispSet*.

The program displacement is automatically reset

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

Syntax

PDispSet
[DispFrame ':='] < expression (**IN**) of *pose*> ';'

Related information

	Described in:
Deactivation of program displacement	Instructions - *PDispOff*
Definition of program displacement using two positions	Instructions - *PDispOn*
Definition of data of the type *pose*	Data Types - *pose*
Coordinate systems	Motion Principles- *Coordinate Systems*
Examples of how program displacement can be used	Instructions - *PDispOn*

ProcCall Calls a new procedure

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

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

Examples

```
weldpipe1;
```

Calls the *weldpipe1* procedure.

```
errormessage;
Set do1;
   .

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

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

Arguments

Procedure { Argument }

Procedure Identifier

The name of the procedure to be called.

Argument Data type: In accordance with the procedure declaration

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

Example

```
weldpipe2 10, lowspeed;
```

Calls the *weldpipe2* procedure, including two arguments.

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

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

Limitations

The procedure's arguments must agree with its parameters:

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

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

Syntax

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

<procedure> ::= <identifier>

Related information

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

PulseDO Generates a pulse on a digital output signal

PulseDO is used to generate a pulse on a digital output signal.

Examples

PulseDO do15;

A pulse with a pulse length of 0.2 s is generated on the output signal *do15*.

PulseDO \PLength:=1.0, ignition;

A pulse of length *1.0 s* is generated on the signal *ignition*.

Arguments

PulseDO [\PLength] Signal

[\PLength] *(Pulse Length)* Data type: *num*

The length of the pulse in seconds (0.1 - 32s).
If the argument is omitted, a 0.2 second pulse is generated.

Signal Data type: *signaldo*

The name of the signal on which a pulse is to be generated.

Program execution

A pulse is generated with a specified pulse length (see Figure 1).

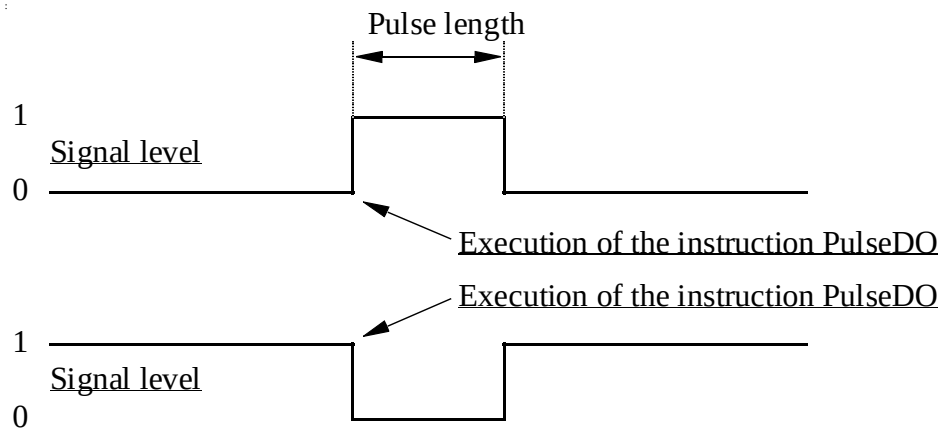


Figure 1 Generation of a pulse on a digital output signal.

The next instruction is executed directly after the pulse starts. The pulse can then be set/ reset without affecting the rest of the program execution.

Limitations

The length of the pulse has a resolution of 0.01 seconds. Programmed values that differ from this are rounded off.

Syntax

PulseDO
[’\’ PLength ’:=’ < expression (**IN**) of *num* > ’,’]
[Signal ’:=’] < variable (**VAR**) of *signaldo* > ’;’

Related information

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

RAISE Calls an error handler

RAISE is used to create an error in the program and then to call the error handler of the routine. *RAISE* can also be used in the error handler to propagate the current error to the error handler of the calling routine.

This instruction can, for example, be used to jump back to a higher level in the structure of the program, e.g. to the error handler in the main routine, if an error occurs at a lower level.

Example

```
IF ...
   IF ...
      IF ...
         RAISE escape1;
   .
ERROR
   IF ERRNO=escape1 RAISE;
```

The routine is interrupted to enable it to remove itself from a low level in the program. A jump occurs to the error handler of the called routine.

Arguments

RAISE [Error no.]

Error no. Data type: *errnum*

Error number: Any number between 1 and 90 which the error handler can use to locate the error that has occurred (the *ERRNO* system variable).

Error number must be specified outside the error handler in a RAISE instruction in order to be able to transfer execution to the error handler of that routine.

If the instruction is present in a routine's error handler, the error number may not be specified. In this case, the error is propagated to the error handler of the calling routine.

Program execution

Program execution continues in the routine's error handler. After the error handler has been executed, program execution can continue with:

- the routine that called the routine in question (RETURN),
- the error handler of the routine that called the routine in question (RAISE).

If the RAISE instruction is present in a routine's error handler, program execution continues in the error handler of the routine that called the routine in question. The same error number remains active.

If the RAISE instruction is present in a trap routine, the error is dealt with by the system's error handler.

Error handling

If the error number is out of range, the system variable ERRNO is set to ERR_ILLRAISE (see "Data types - errnum"). This error can be handled in the error handler.

Syntax

(EBNF)
RAISE [<error number>] ';'

<error number> ::= <expression>

Related information

	Described in:
Error handling	Basic Characteristics - *Error Recovery*

Reset Resets a digital output signal

Reset is used to reset the value of a digital output signal to zero.

Examples

Reset do15;

The signal *do15* is set to 0.

Reset weld;

The signal *weld* is set to 0.

Arguments

Reset Signal

Signal Data type: *signaldo*

The name of the signal to be reset to zero.

Program execution

The true value depends on the configuration of the signal. If the signal is inverted in the system parameters, this instruction causes the physical channel to be set to 1.

Syntax

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

Related information

	Described in:
Setting a digital output signal	Instructions - *Set*
Input/Output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	System Parameters

RestoPath Restores the path after an interrupt

RestoPath is used to restore a path that was stored at a previous stage using the instruction *StorePath*.

Example

RestoPath;

Restores the path that was stored earlier using *StorePath*.

Program execution

The current movement path of the robot and the external axes is deleted and the path stored earlier using *StorePath* is restored. Nothing moves, however, until the instruction *StartMove* is executed or a return is made using *RETRY* from an error handler.

Example

```
ArcL p100, v100, seam1, weld5, weave1, z10, gun1;
...
ERROR
   IF ERRNO=AW_WELD_ERR THEN
      gun_cleaning;
      RETRY;
   ENDIF
...
PROC gun_cleaning()
   VAR robtarget p1;
   StorePath;
   p1 := CRobT();
   MoveL pclean, v100, fine, gun1;
   ...
   MoveL p1, v100, fine, gun1;
   RestoPath;
ENDPROC
```

In the event of a welding error, program execution continues in the error handler of the routine, which, in turn, calls *gun_cleaning*. The movement path being executed at the time is then stored and the robot moves to the position *pclean* where the error is rectified. When this has been done, the robot returns to the position where the error occurred, *p1*, and stores the original movement once again. The weld then automatically restarts, meaning that the robot is first reversed along the path before welding starts and ordinary program execution can continue.

Limitations

The movement being executed at the time must first be stopped, using a stop point for example, before the instruction *RestoPath* is executed.

The movement instruction which precedes this instruction should be terminated with a stop point, in order to be able to restart in this instruction following a power failure.

Syntax

RestoPath';'

Related information

	Described in:
Storing paths	Instructions - *StorePath*
More examples	Instructions - *StorePath*

RETRY Restarts following an error

RETRY is used to restart program execution after an error has occurred.

Example

```
reg2 := reg3/reg4;
   .
ERROR
   IF ERRNO = ERR_DIVZERO THEN
      reg4 := 1;
      RETRY;
   ENDIF
```

An attempt is made to divide *reg3* by *reg4*. If reg4 is equal to 0 (division by zero), a jump is made to the error handler, which initialises reg4. The *RETRY* instruction is then used to jump from the error handler and another attempt is made to complete the division.

Program execution

Program execution continues with (re-executes) the instruction that caused the error.

Error handling

If the maximum number of retries (5) is exceeded, the system variable ERRNO is set to ERR_EXCRTYMAX (see "Data types - errnum"). This error can be handled in the error handler.

Limitations

The instruction can only exist in a routine's error handler. If the error was created using a *RAISE* instruction, program execution cannot be restarted with a *RETRY* instruction, then the instruction *TRYNEXT* should be used.

Syntax

RETRY ';'

Related information

	Described in:
Error handlers	Basic Characteristics- *Error Recovery*
Continue with the next instruction	Instructions - *TRYNEXT*

RETURN Finishes execution of a routine

RETURN is used to finish the execution of a routine. If the routine is a function, the function value is also returned.

Examples

```
errormessage;
Set do1;
   .

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

The *errormessage* procedure is called. When the procedure arrives at the RETURN instruction, program execution returns to the instruction following the procedure call, *Set do1*.

```
FUNC num abs_value(num value)
   IF value<0 THEN
      RETURN -value;
   ELSE
      RETURN value;
   ENDIF
ENDFUNC
```

The function returns the absolute value of a number.

Arguments

RETURN [Return value]

Return value Data type: According to the function declaration

The return value of a function.

The return value must be specified in a RETURN instruction present in a function.

If the instruction is present in a procedure or trap routine, a return value may not be specified.

Program execution

The result of the *RETURN* instruction may vary, depending on the type of routine it is used in:

- Main routine: If a program stop has been ordered at the end of the cycle, the program stops. Otherwise, program execution continues with the first instruction of the main routine.
- Procedure: Program execution continues with the instruction following the procedure call.
- Function: Returns the value of the function.
- Trap routine: Program execution continues from where the interrupt occurred.
- Error handler: In a procedure:
  Program execution continues with the routine that called the routine with the error handler (with the instruction following the procedure call).

  In a function:
  The function value is returned.

Syntax

(EBNF)
RETURN [<expression>]';'

Related information

	Described in:
Functions and Procedures	Basic Characteristics - *Routines*
Trap routines	Basic Characteristics - *Interrupts*
Error handlers	Basic Characteristics - *Error Recovery*

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 active, the robot immediately reads the current position.

This instruction can typically be used when the tool held by the robot is a probe for surface detection. Using the *SearchC* instruction, the outline coordinates of a work object can be obtained.

Examples

SearchC sen1, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10* at a speed of *v100*. When the value of the signal *sen1* changes to active, the position is stored in *sp*.

SearchC \Stop, sen1, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp* and the robot stops immediately.

Arguments

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

[\Stop] Data type: *switch*

The robot movement is stopped, as quickly as possible, without keeping the TCP on the path, 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.

If this argument is omitted, the movement continues (flying search) to the position specified in the *ToPoint* argument.

[\PStop] *(Path Stop)* Data type: *switch*

The robot movement is stopped as quickly as possible, while keeping the TCP on the path, when the value of the search signal changes to active. However, the robot is moved a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\Sup] *(Supervision)* Data type: *switch*

The search instruction is sensitive to signal activation during the complete movement, i.e. even after the first signal change has been reported. If more than one match occurs during a search, program execution stops.

If the argument *\Stop*, *\PStop* or *\Sup* is omitted, the movement continues (flying search) to the position specified in the *ToPoint* argument.

Signal Data type: *signaldi*

The name of the signal to supervise.

SearchPoint Data type: *robtarget*

The position of the TCP and external axes when the search signal has been triggered. The position is specified in the outermost coordinate system, taking the specified tool, work object and active ProgDisp/ExtOffs coordinate system into consideration.

CirPoint Data type: *robtarget*

The circle point of the robot. See the instruction MoveC for a more detailed description of circular movement. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

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

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

Tool Data type: *tooldata*

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

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

The work object (coordinate system) to which the robot positions in the instruction are related.

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

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.

If a flying search is used, i.e. the *\Stop* argument is not specified, the robot movement always continues to the programmed destination point. If a search is made using the switch *\Stop* or *\PStop*, the robot movement stops when the first signal is detected.

The *SearchC* instruction returns the position of the TCP when the value of the digital signal changes to active, as illustrated in Figure 1.

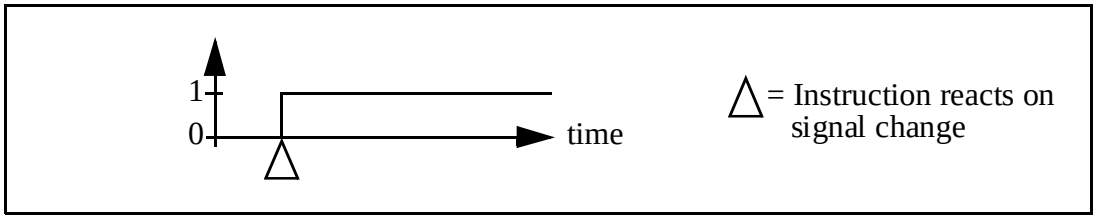


Figure 1 Flank-triggered signal detection (the position is stored when the signal is changed to active the first time only).

In order to get a fast response, use the interrupt-driven sensor signals *sen1*, *sen2* or *sen3* on the system board.

Example

SearchC \Sup, sen1, sp, cirpoint, p10, v100, probe;

The TCP of the *probe* is moved circularly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp*. If the value of the signal changes twice, program execution stops.

Limitations

Zone data for the positioning instruction that precedes *SearchC* must be used carefully.

The start of the search, i.e. when the I/O signal is ready to react, is not, in this case, the programmed destination point of the previous positioning instruction, but a point along the real robot path. Figure 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).

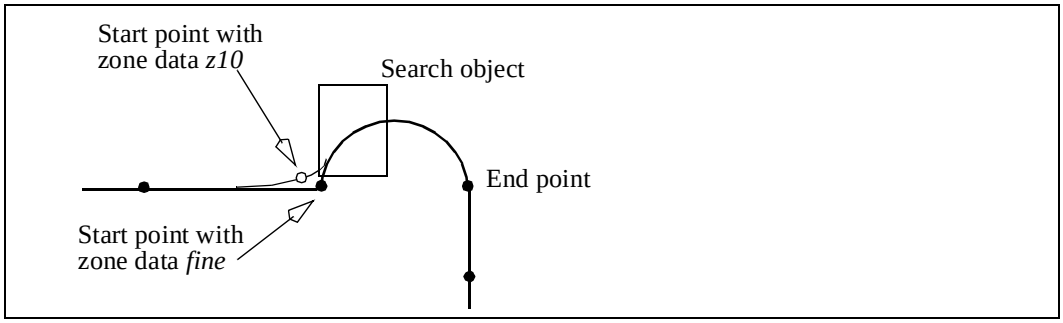


Figure 2 A match is made on the wrong side of the object because the wrong zone data was used.

Typical stop distance using a search velocity of 50 mm/s:

- without TCP on path (switch *Stop*) 1-3 mm
- with TCP on path (switch *PStop*) 12-16 mm

Error handling

An error is reported during a search when:

- no signal detection occurred
- more than one signal detection occurred – this generates an error only if the *Sup* argument is used.

Errors can be handled in different ways depending on the selected running mode:

Continuous forward
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.

Instruction forward
No position is returned and the movement always continues to the programmed destination point. Program execution stops with an error message.

Instruction backward
During backward execution, the instruction just carries out the movement without any signal supervision.

Syntax

SearchC
[’\’ Stop’,’] | [’\’ PStop ’,’] | [’\’ Sup ’,’]
[Signal ’:=’] < variable (**VAR**) of *signaldi* > ’,’
[SearchPoint ’:=’] < var or pers (**INOUT**) of *robtarget* > ’,’
[CirPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[Speed ’:=’] < expression (**IN**) of *speeddata* >
[’\’ V ’:=’ < expression (**IN**) of *num* >]
| [’\’ T ’:=’ < expression (**IN**) of *num* >] ’,’
[Tool ’:=’] < persistent (**PERS**) of *tooldata* >
[’\’ WObj ’:=’ < persistent (**PERS**) of *wobjdata* >] ’;’

Related information

	Described in:
Linear searches	Instructions - *SearchL*
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 active, the robot immediately reads the current position.

This instruction can typically be used when the tool held by the robot is a probe for surface detection. Using the *SearchL* instruction, the outline coordinates of a work object can be obtained.

Examples

SearchL sen1, sp, p10, v100, probe;

The TCP of the *probe* is moved linearly towards the position *p10* at a speed of *v100*. When the value of the signal *sen1* changes to active, the position is stored in *sp*.

SearchL \Stop, sen1, sp, p10, v100, probe;

The TCP of the *probe* is moved linearly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp* and the robot stops immediately.

Arguments

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

[\Stop] Data type: *switch*

The robot movement is stopped as quick as possible, without keeping the TCP on the path, 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.

If this argument is omitted, the movement continues (flying search) to the position specified in the *ToPoint* argument.

[\PStop] *(Path Stop)* Data type: *switch*

The robot movement is stopped as quick as possible, while keeping the TCP on the path, when the value of the search signal changes to active. However, the robot is moved a small distance before it stops and is not moved back to the searched position, i.e. to the position where the signal changed.

[\Sup] *(Supervision)* Data type: *switch*

The search instruction is sensitive to signal activation during the complete movement, i.e. even after the first signal change has been reported. If more than one match occurs during a search, program execution stops.

If the argument *\Stop*, *\PStop* or *\Sup* is omitted, the movement continues (flying search) to the position specified in the *ToPoint* argument.

Signal Data type: *signaldi*

The name of the signal to supervise.

SearchPoint Data type: *robtarget*

The position of the TCP and external axes when the search signal has been triggered. The position is specified in the outermost coordinate system, taking the specified tool, work object and active ProgDisp/ExtOffs coordinate system into consideration.

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

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

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

Tool Data type: *tooldata*

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

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

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

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

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 *\Stop* argument is not specified, the robot movement always continues to the programmed destination point. If a search is made using the switch *\Stop* or *\PStop*, the robot movement stops when the first signal is detected.

The *SearchL* instruction stores the position of the TCP when the value of the digital signal changes to active, as illustrated in Figure 1.

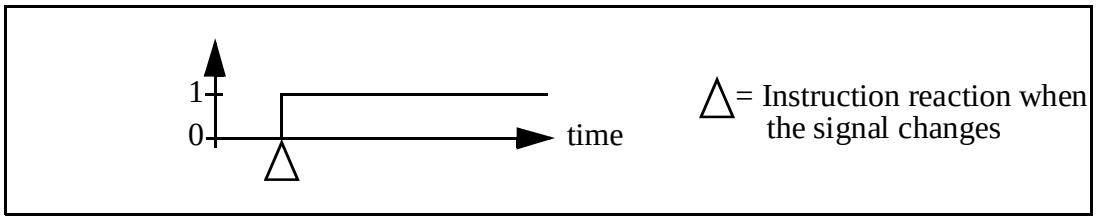


Figure 1 Flank-triggered signal detection (the position is stored when the signal is changed the first time only).

In order to get a fast response, use the interrupt-driven sensor signals *sen1*, *sen2* or *sen3* on the system board.

Examples

```
SearchL \Sup, sen1, sp, p10, v100, probe;
```

The TCP of the *probe* is moved linearly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp*. If the value of the signal changes twice, program execution stops.

```
SearchL \Stop, sen1, sp, p10, v100, tool1;
MoveL sp, v100, fine, tool1;
PDispOn *, tool1;
MoveL p100, v100, z10, tool1;
MoveL p110, v100, z10, tool1;
MoveL p120, v100, z10, tool1;
PDispOff;
```

The TCP of *tool1* is moved linearly towards the position *p10*. When the value of the signal *sen1* changes to active, the position is stored in *sp* and the robot is

moved back to this point. Using program displacement, the robot then moves relative to the searched position, *sp*.

Limitations

Zone data for the positioning instruction that precedes *SearchL* must be used carefully. The start of the search, i.e. when the I/O signal is ready to react, is not, in this case, the programmed destination point of the previous positioning instruction, but a point along the real robot path. Figure 2 to Figure 4 illustrate examples of things that may go wrong when zone data other than *fine* is used.

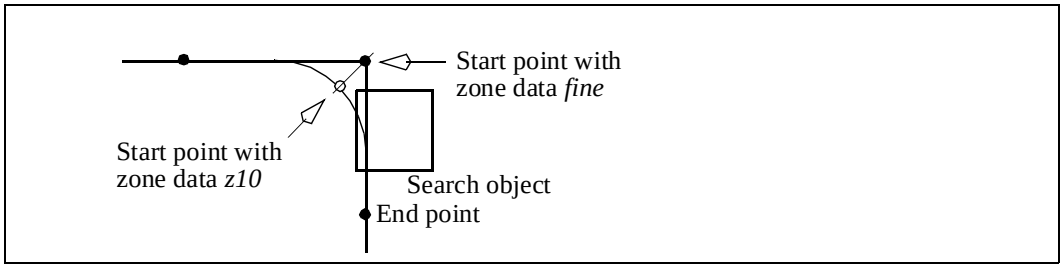


Figure 2 A match is made on the wrong side of the object because the wrong zone data was used.

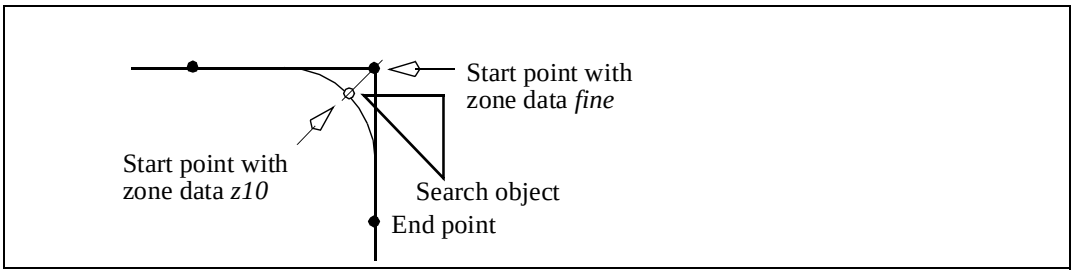


Figure 3 No match detected because the wrong zone data was used.

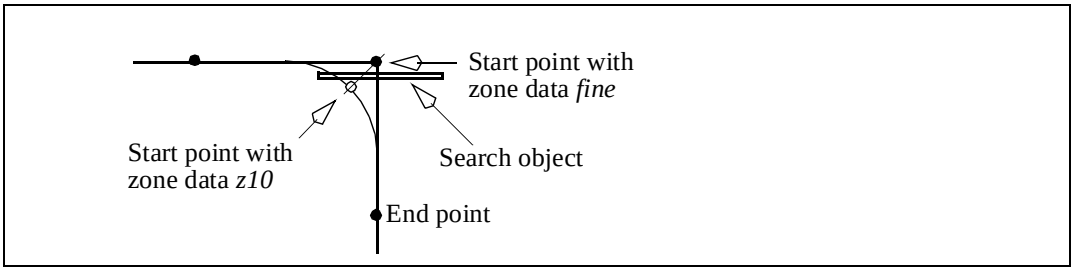


Figure 4 No match detected because the wrong zone data was used.

Typical stop distance using a search velocity of 50 mm/s:

- without TCP on path (switch *Stop*) 1-3 mm
- with TCP on path (switch *PStop*) 12-16 mm

Error handling

An error is reported during a search when:

- no signal detection occurred
- more than one signal detection occurred – this generates an error only if the *\Sup* argument is used.

Errors can be handled in different ways depending on the selected running mode:

Continuous forward
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.

Instruction forward
No position is returned and the movement always continues to the programmed destination point. Program execution stops with an error message.

Instruction backward
During backward execution, the instruction just carries out the movement without any signal supervision.

Example

```
MoveL p10, v100, fine, tool1;
SearchL \Stop, sen1, sp, p20, v100, tool1;
.
ERROR
   IF ERRNO=ERR_WHLSEARCH THEN
      MoveL p10, v100, fine, tool1;
      RETRY;
   ENDIF
```

The robot searches from position *p10* to *p20*. If no signal detection occurs, the robot moves back to *p10* and tries once more.

Syntax

SearchL
[’\’ Stop ’,’] | [’\’ PStop ’,’] | [’\’ Sup ’,’]
[Signal ’:=’] < variable (**VAR**) of *signaldi* > ’,’
[SearchPoint ’:=’] < var or pers (**INOUT**) of *robtarget* > ’,’
[ToPoint ’:=’] < expression (**IN**) of *robtarget* > ’,’
[Speed ’:=’] < expression (**IN**) of *speeddata* >
[’\’ V ’:=’ < expression (**IN**) of *num* >]
| [’\’ T ’:=’ < expression (**IN**) of *num* >] ’,’
[Tool ’:=’] < persistent (**PERS**) of *tooldata* >
[’\’ WObj ’:=’ < persistent (**PERS**) of *wobjdata* >] ’;’

Related information

	Described in:
Circular searches	Instructions - *SearchC*
Linear movement	Motion and I/O Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Using error handlers	RAPID Summary - *Error Recovery*
Motion in general	Motion and I/O Principles

Set Sets a digital output signal

Set is used to set the value of a digital output signal to one.

Examples

Set do15;

The signal *do15* is set to 1.

Set weldon;

The signal *weldon* is set to 1.

Arguments

Set Signal

Signal Data type: *signaldo*

The name of the signal to be set to one.

Program execution

The true value depends on the configuration of the signal. If the signal is inverted in the system parameters, this instruction causes the physical channel to be set to zero.

Syntax

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

Related information

	Described in:
Setting a digital output signal to zero	Instructions - *Reset*
Input/Output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	System Parameters

SetAO Changes the value of an analog output signal

SetAO is used to change the value of an analog output signal.

Example

SetAO ao2, 5.5;

The signal *ao2* is set to *5.5*.

Arguments

SetAO Signal Value

Signal Data type: *signalao*

The name of the analog output signal to be changed.

Value Data type: *num*

The desired value of the signal.

Program execution

The programmed value is scaled (in accordance with the system parameters) before it is sent on the physical channel. See Figure 1.

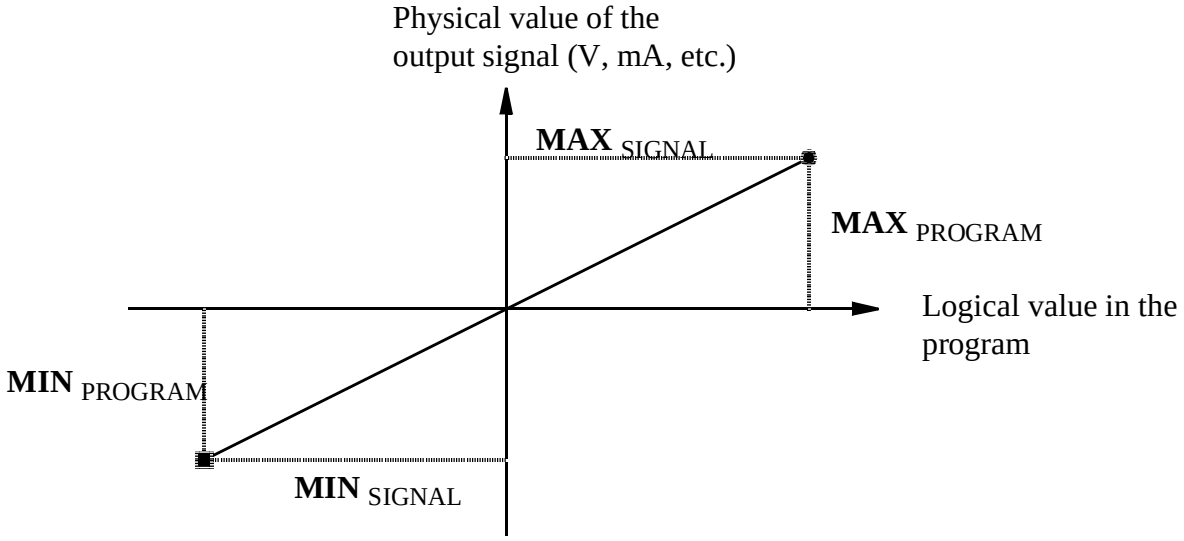


Figure 1 Diagram of how analog signal values are scaled.

Example

```
SetAO weldcurr, curr_outp;
```

The signal *weldcurr* is set to the same value as the current value of the variable *curr_outp*.

Syntax

```
SetAO
   [ Signal ':=' ] < variable (VAR) of signalao > ','
   [ Value ':=' ] < expression (IN) of num > ';'
```

Related information

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

SetDO Changes the value of a digital output signal

SetDO is used to change the value of a digital output signal, with or without a time-delay.

Examples

SetDO do15, 1;

The signal *do15* is set to *1*.

SetDO weld, off;

The signal *weld* is set to *off*.

SetDO \SDelay := 0.2, weld, high;

The signal *weld* is set to *high* with a delay of *0.2* s. Program execution, however, continues with the next instruction.

Arguments

SetDO [\SDelay] Signal Value

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

Delays the change for the amount of time given in seconds (0.1 - 32s). Program execution continues directly with the next instruction. After the given time-delay, the signal is changed without the rest of the program execution being affected.

If the argument is omitted, the value of the signal is changed directly.

Signal Data type: *signaldo*

The name of the signal to be changed.

Value Data type: *dionum*

The desired value of the signal.

The value is specified as 0 or 1.

Program execution

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

Limitations

The length of the pulse has a resolution of 0.01 seconds. Programmed values that differ from this are rounded off.

Syntax

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

Related information

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

SetGO Changes the value of a group of digital output signals

SetGO is used to change the value of a group of digital output signals.

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.

Arguments

SetGO Signal Value

Signal Data type: *signalgo*

The name of the signal group to be changed.

Value Data type: *num*

The desired value of the signal group (a positive integer).

The permitted value is dependent on the number of signals in the group:

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

Program execution

The programmed value is converted to an unsigned binary number. This binary number is sent on the signal group, with the result that individual signals in the group are set to 0 or 1. Due to internal delays, the value of the signal may be undefined for a short period of time.

Syntax

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

Related information

	Described in:
Other input/output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O (system parameters)	System Parameters

SingArea Defines interpolation around singular points

SingArea is used to define how the robot is to move in the proximity of singular points.

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).

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.

Arguments

SingArea [\Wrist] | [\Arm] | [\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).

[\Arm] Data type: *switch*

The tool orientation is allowed to differ somewhat in order to avoid arm singularity. Used when the wrist centre is coincident with the extension of axis 1. (Not implemented in this version).

[\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 in singular points.

Program execution

If one of the arguments *\Wrist* or *\Arm* 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, or if none of the arguments are specified, program execution automatically uses the */Off* argument. This is automatically set

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

Limitations

Only one of the arguments can be specified. This means that arm and wrist singularities cannot be handled at the same time.

Syntax

```
SingArea
   [ '\' Wrist ] | [ '\' Arm ] | [ '\' Off ] ';'
```

Related information

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

-

SoftAct Activating the soft servo

SoftAct (Soft Servo Activate) is used to activate the so called ”soft” servo on any of the robot axes.

Example

SoftAct 3, 20;

Activation of soft servo on axis 3, with softness value 20%.

SoftAct 1, 90 \Ramp:=150;

Activation of the soft servo on axis 1, with softness value 90% and ramp factor 150%.

Arguments

SoftAct Axis Softness [\Ramp]

Axis Data type: *num*

Number of current axis on the robot (1 - 6).

Softness Data type: *num*

Softness value in percent (0 - 100%). 0% denotes min. softness (max. stiffness), and 100% denotes max. softness.

Ramp Data type: *num*

Ramp factor in percent (>= 100%). The ramp factor is used to control the engagement of the soft servo. A factor 100% denotes the normal value; with greater values the soft servo is engaged more slowly (longer ramp). The default value for ramp factor is 100 %.

Program execution

Softness is activated at the value specified for the current axis. The softness value is valid for all movements, until a new softness value is programmed for the current axis, or until the soft servo is deactivated by an instruction.

Limitations

The same axis must not be activated twice, unless there is a moving instruction in between. Thus, the following program sequence should be avoided, otherwise there will be a jerk in the robot movement:

```
SoftAct     n , x ;
SoftAct     n , y ;
```

(n = robot axis n, x and y softness values)

Syntax

SoftAct
[Axis ':='] < expression (**IN**) of *num*> ','
[Softness ':='] < expression (**IN**) of *num*>
['\'Ramp ':=' < expression (**IN**) of *num*>]';'

Related information

	Described in:
Behaviour with the soft servo engaged	Motion and I/O Principles- *Positioning during program execution*

SoftDeact Deactivating the soft servo

SoftDeact (Soft Servo Deactivate) is used to deactivate the so called ”soft” servo on any of the robot axes.

Example

SoftDeact;

Deactivating the soft servo on all axes.

Program execution

The soft servo is deactivated on all axes.

Syntax

SoftDeact’;’

Related information

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

StartMove Restarts robot motion

StartMove is used to resume robot and external axes motion when this has been stopped by the instruction *StopMove*.

Example

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

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

Program execution

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

Error handling

If the robot is too far from the path (more than 10 mm or 20 degrees) to perform a start of the interrupted movement, the system variable *ERRNO* is set to ERR_PATHDIST. This error can then be handled in the error handler.

Syntax

```
StartMove';'
```

Related information

	Described in:
Stopping movements	Instructions - *StopMove*
More examples	Instructions - *StorePath*

-

Stop Stops program execution

Stop is used to temporarily stop program execution.

Program execution can also be stopped using the instruction *EXIT*. This, however, should only be done if a task is complete, or if a fatal error occurs, since program execution cannot be restarted with *EXIT*.

Example

```
TPWrite "The line to the host computer is broken";
Stop;
```

Program execution stops after a message has been written on the teach pendant.

Arguments

Stop [\NoRegain]

[\NoRegain] Data type: *switch*

Specifies for the next program start in manual mode, whether or not the robot and external axes should regain to the stop position. In automatic mode the robot and external axes always regain to the stop position.

If the argument *NoRegain* is set, the robot and external axes will not regain to the stop position (if they have been jogged away from it).

If the argument is omitted and if the robot or external axes have been jogged away from the stop position, the robot displays a question on the teach pendant. The user can then answer, whether or not the robot should regain to the stop position.

Program execution

The instruction stops program execution as soon as the robot and external axes reach the programmed destination point for the movement it is performing at the time. Program execution can then be restarted from the next instruction.

Example

```
MoveL p1, v500, fine, tool1;
TPWrite "Jog the robot to the position for pallet corner 1";
Stop \NoRegain;
```

```
p1_read := CRobT();
MoveL p2, v500, z50, tool1;
```

Program execution stops with the robot at *p1*. The operator jogs the robot to *p1_read*. For the next program start, the robot does not regain to *p1*, so the position *p1_read* can be stored in the program.

Limitations

The movement instruction which precedes this instruction should be terminated with a stop point, in order to be able to restart in this instruction following a power failure.

Syntax

```
Stop
[ '\' NoRegain ]';'
```

Related information

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

StopMove Stops robot motion

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

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

Example

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

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

Program execution

The robot and external axes movements 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.

Examples

```
VAR intnum intno1;
...
CONNECT intno1 WITH go_to_home_pos;
ISignalDI di1,1,intno1;

TRAP go_to_home_pos
   VAR robtarget p10;

   StopMove;
   StorePath;
   p10:=CRobT();
   MoveL home,v500,fine,tool1;
   WaitDI di1,0;
   Move L p10,v500,fine,tool1;
   RestoPath;
   StartMove;
ENDTRAP
```

When the input *di1* is set to 1, an interrupt is activated which in turn activates the interrupt routine *go_to_home_pos*. The current movement is stopped immediately and the robot moves instead to the *home* position. When *di1* is set to 0, the robot returns to the position at which the interrupt occurred and continues to

move along the programmed path.

```
VAR intnum intno1;
...
CONNECT intno1 WITH go_to_home_pos;
ISignalDI di1,1,intno1;

TRAP go_to_home_pos ()
   VAR robtarget p10;

   StorePath;
   p10:=CRobT();
   MoveL home,v500,fine,tool1;
   WaitDI di1,0;
   Move L p10,v500,fine,tool1;
   RestoPath;
   StartMove;
ENDTRAP
```

Similar to the previous example, but the robot does not move to the *home* position until the current movement instruction is finished.

Syntax

```
StopMove';'
```

Related information

	Described in:
Continuing a movement	Instructions - *StartMove*
Interrupts	RAPID Summary - *Interrupts* Basic Characteristics- *Interrupts*

StorePath Stores the path when an interrupt occurs

StorePath is used to store the movement path being executed when an error or interrupt occurs. The error handler or trap routine can then start a new movement and, following this, restart the movement that was stored earlier.

This instruction can be used to go to a service position or to clean the gun, for example, when an error occurs.

Example

```
StorePath;
```

The current movement path is stored for later use.

Program execution

The current movement path of the robot and external axes is saved. After this, another movement can be started in a trap routine or an error handler. When the reason for the error or interrupt has been rectified, the saved movement path can be restarted.

Example

```
TRAP machine_ready;
   VAR robtarget p1;
   StorePath;
   p1 := CRobT();
   MoveL p100, v100, fine, tool1;
   ...
   MoveL p1, v100, fine, tool1;
   RestoPath;
   StartMove;
ENDTRAP
```

When an interrupt occurs that activates the trap routine *machine_ready*, the movement path which the robot is executing at the time is stopped at the end of the instruction (ToPoint) and stored. After this, the robot remedies the interrupt by, for example, replacing a part in the machine and the normal movement is restarted.

Limitations

Only one movement path can be stored at a time.

Syntax

StorePath‘;’

Related information

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

TEST Depending on the value of an expression ...

TEST is used when different instructions are to be executed depending on the value of an expression or data.

If there are not too many alternatives, the *IF..ELSE* instruction can also be used.

Example

```
TEST reg1
CASE 1,2,3 :
   routine1;
CASE 4 :
   routine2;
DEFAULT :
   TPWrite "Illegal choice";
   Stop;
ENDTEST
```

Different instructions are executed depending on the value of *reg1*. If the value is 1-3 *routine1* is executed. If the value is 4, *routine2* is executed. Otherwise, an error message is printed and execution stops.

Arguments

TEST Test data {CASE Test value {, Test value} : ...}
[DEFAULT: ...] ENDTEST

Test data Data type: All

The data or expression with which the test value will be compared.

Test value Data type: Same as test data

The value which the test data must have for the associated instructions to be executed.

Program execution

The test data is compared with the test values in the first CASE condition. If the comparison is true, the associated instructions are executed. After that, program execution continues with the instruction following ENDTEST.

If the first CASE condition is not satisfied, other CASE conditions are tested, and so on. If none of the conditions are satisfied, the instructions associated with DEFAULT are executed (if this is present).

Syntax

(EBNF)
TEST <expression>
{(**CASE** <test value> { ’,’ <test value> } ’:’
    <instruction list>) | **<CSE>** }
[**DEFAULT** ’:’ <instruction list>]
ENDTEST

<test value> ::= <expression>

Related information

	Described in:
Expressions	Basic Characteristics - *Expressions*

TPErase Erases text printed on the teach pendant

TPErase (Teach Pendant Erase) is used to clear the display of the teach pendant.

Example

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

The teach pendant display is cleared before *Execution started* is written.

Program execution

The teach pendant display is completely cleared of all text. The next time text is written, it will be entered on the uppermost line of the display.

Syntax

```
TPErase;
```

Related information

	Described in:
Writing on the teach pendant	RAPID Summary - *Communication*

TPReadFK Reads function keys

TPReadFK (Teach Pendant Read Function Key) is used to write text above the functions keys and to find out which key is depressed.

Example

```
TPReadFK reg1, "More ?", "", "", "", "Yes", "No";
```

The text *More ?* is written on the teach pendant display and the function keys 4 and 5 are activated by means of the text strings *Yes* and *No* respectively (see Figure 1). 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.

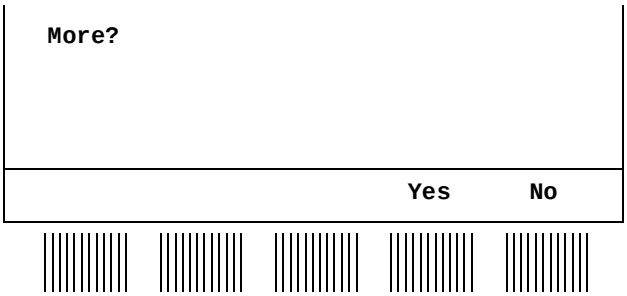


Figure 1 The operator can input information via the function keys.

Arguments

TPReadFK Answer Text FK1 FK2 FK3 FK4 FK5 [\MaxTime] [\DIBreak] [\BreakFlag]

Answer Data type: *num*

The variable for which, depending on which key is pressed, the numeric value 1..5 is returned. If the function key 1 is pressed, 1 is returned, and so on.

Text Data type: *string*

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

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

The text to be written as a prompt for the appropriate function key (a maximum of 7 characters). FK1 is the left-most key.

Function keys without prompts are specified by an empty string "".

[\MaxTime] Data type: *num*

The maximum amount of time [s] that program execution waits. If no function key is depressed within this time, the program continues to execute in the error handler unless the BreakFlag is used (see below). The constant ERR_TP_MAXTIME can be used to test whether or not the maximum time has elapsed.

[\DIBreak] *(Digital Input Break)* Data type: *signaldi*

The digital signal that may interrupt the operator dialog. If no function key is depressed when the signal is set to 1 (or is already 1), the program continues to execute in the error handler, unless the BreakFlag is used (see below). The constant ERR_TP_DIBREAK can be used to test whether or not this has occurred.

[\BreakFlag] Data type: *errnum*

A variable that will hold the error code if maxtime or dibreak is used. If this optional variable is omitted, the error handler will be executed. The constants ERR_TP_MAXTIME and ERR_TP_ DIBREAK can be used to select the reason.

Program execution

The information text is always written on a new line. If the display is full of text, this body of text is moved up one line first. Strings longer than the width of the teach pendant (40 characters) are split into two lines.

Prompts are written above the appropriate function keys. Keys without prompts are deactivated.

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

Example

```
VAR errnum errvar;
...
TPReadFK reg1, "Go to service position?", "", "", "", "Yes", "No" \MaxTime:= 600
   \DIBreak:= di5\BreakFlag:= errvar;
IF reg1 = 4 or OR errvar = ERR_TP_DIBREAK THEN
   MoveL service, v500, fine, tool1;
   Stop;
END IF
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.

Syntax

TPReadFK
[Answer':='] <var or pers (**INOUT**) of *num*>','
[Text':='] <expression (**IN**) of *string*>','
[FK1 ':='] <expression (**IN**) of *string*>','
[FK2 ':='] <expression (**IN**) of *string*>','
[FK3 ':='] <expression (**IN**) of *string*>','
[FK4 ':='] <expression (**IN**) of *string*>','
[FK5 ':='] <expression (**IN**) of *string*>
['\'MaxTime ':=' <expression (**IN**) of *num*>]
['\'DIBreak ':=' <variable (**VAR**) of *signaldi*>]
['\'BreakFlag ':=' <var or pers (**INOUT**) of *errnum*>]';'

Related information

	Described in:
Writing to and reading from the teach pendant	RAPID Summary - *Communication*
Replying via the teach pendant	Running Production

TPReadNum Reads a number from the teach pendant

TPReadNum (Teach Pendant Read Numerical) is used to read a number from the teach pendant.

Example

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

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

Arguments

TPReadNum Answer String [\MaxTime] [\DIBreak] [\BreakFlag]

Answer Data type: *num*

The variable for which the number input via the teach pendant is returned.

String Data type: *string*

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

[\MaxTime] Data type: *num*

The maximum amount of time that program execution waits. If no number is input within this time, the program continues to execute in the error handler unless the BreakFlag is used (see below). The constant ERR_TP_MAXTIME can be used to test whether or not the maximum time has elapsed.

[\DIBreak] *(Digital Input Break)* Data type: *signaldi*

The digital signal that may interrupt the operator dialog. If no number is input when the signal is set to 1 (or is already 1), the program continues to execute in the error handler unless the BreakFlag is used (see below). The constant ERR_TP_DIBREAK can be used to test whether or not this has occurred.

[\BreakFlag] Data type: *errnum*

A variable that will hold the error code if maxtime or dibreak is used. If this optional variable is omitted, the error handler will be executed.The constants ERR_TP_MAXTIME and ERR_TP_ DIBREAK can be used to select the reason.

Program execution

The information text is always written on a new line. If the display is full of text, this body of text is moved up one line first. Strings longer than the width of the teach pendant (40 characters) are split into two lines.

Program execution waits until a number is typed on the numeric keyboard (followed by Enter or ***OK***).

Example

```
TPReadNum reg1, “How many units should be produced?“;
FOR i FROM 1 TO reg1 DO
   produce_part;
ENDFOR
```

The text *How many units should be produced?* is written on the teach pendant display. The routine *produce_part* is then repeated the number of times that is input via the teach pendant.

Syntax

```
TPReadNum
   [Answer’:=’] <var or pers (INOUT) of num>’,’
   [String’:=’] <expression (IN) of string>
   [’\’MaxTime ’:=’ <expression (IN) of num>]
   [’\’DIBreak ’:=’ <variable (VAR) of signaldi>]
   [’\’BreakFlag ’:=’ <var or pers (INOUT) of errnum>] ’;’
```

Related information

	Described in:
Writing to and reading from the teach pendant	RAPID Summary - *Communication*
Entering a number on the teach pendant	Production Running
Examples of how to use the arguments MaxTime, DIBreak and BreakFlag	Instructions - *TPReadFK*

TPWrite Writes on the teach pendant

TPWrite (Teach Pendant Write) is used to write text on the teach pendant. The value of certain data can be written as well as text.

Examples

TPWrite "Execution started";

The text *Execution started* is written on the teach pendant.

TPWrite "No of produced parts="\Num:=reg1;

If, for example, the answer to *No of produced parts=5*, enter 5 instead of *reg1* on the teach pendant.

Arguments

TPWrite String [\Num] | [\Bool] | [\Pos] | [\Orient]

String Data type: *string*

The text string to be written (a maximum of 80 characters).

[\Num] *(Numeric)* Data type: *num*

The data whose numeric value is to be written after the text string.

[\Bool] *(Boolean)* Data type: *bool*

The data whose logical value is to be written after the text string.

[\Pos] *(Position)* Data type: *pos*

The data whose position is to be written after the text string.

[\Orient] *(Orientation)* Data type: *orient*

The data whose orientation is to be written after the text string.

Program execution

Text written on the teach pendant always begins on a new line. When the display is full of text, this text is moved up one line first. Strings that are longer than the width of the teach pendant (40 characters) are divided up into two lines.

If one of the arguments *\Num*, *\Bool*, *\Pos* or *\Orient* is used, its value is first converted

to a text string before it is added to the first string. The conversion from value to text string takes place as follows:

Argument	Value	Text string
\Num	23	"23"
\Num	1.141367	"1.14137"
\Bool	TRUE	"TRUE"
\Pos	[1817.3,905.17,879.11]	"[1817.3,905.17,879.11]"
\Orient	[0.96593,0,0.25882,0]	"[0.96593,0,0.25882,0]"

Decimals are rounded off to 5 places of decimals, but there may not be more than 6 digits in the text string. If the decimal part is less than 0.000005 or greater than 0.999995, the number is rounded to an integer.

Limitations

The arguments *\Num*, *\Bool*, *\Pos* and *\Orient* are mutually exclusive and thus cannot be used simultaneously in the same instruction.

Syntax

```
TPWrite
   [String':='] <expression (IN) of string>
   ['\'Num':=' <expression (IN) of num> ]
   | ['\'Bool':=' <expression (IN) of bool> ]
   | ['\'Pos':=' <expression (IN) of pos> ]
   | ['\'Orient':=' <expression (IN) of orient> ]';'
```

Related information

	Described in:
Clearing and reading the teach pendant	RAPID Summary - *Communication*

-

TriggC Circular robot movement with events

TriggC (Trigg Circular) is used to set output signals and/or run interrupt routines at fixed positions, at the same time as the robot is moving on a circular path.

One or more (max. 4) events can be defined using the instructions *TriggIO*, *TriggEquip* or *TriggInt* and afterwards these definitions are referred to in the instruction *TriggC*.

Examples

```
VAR triggdata gunon;

TriggIO gunon, 0 \Start \DOp:=gun, on;

MoveL p1, v500, z50, gun1;
TriggC p2, p3, v500, gunon, fine, gun1;
```

The digital output signal *gun* is set when the robot's TCP passes the midpoint of the corner path of the point *p1*.

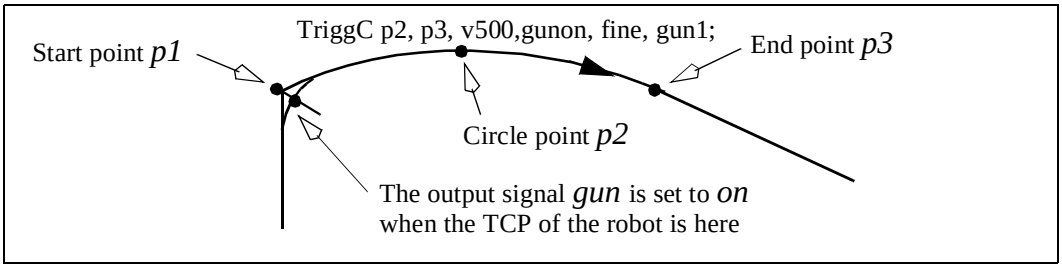


Figure 1 Example of fixed-position IO event.

Arguments

TriggC [\Conc] CirPoint ToPoint Speed [\T]
Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj]

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

Subsequent logical instructions are executed while the robot is moving. This argument is used to shorten the cycle time when, for example, communicating with external equipment, and synchronisation is not required. It can also be used to tune the execution of the robot path, to avoid warning 50024 Corner path failure, or error 40082 Deceleration limit.

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

If this argument is omitted, the subsequent instruction is only executed after the robot has reached the specified stop point or 100 ms before the specified zone.

CirPoint Data type: *robtarget*

The circle point of the robot. See the instruction *MoveC* for a more detailed description of circular movement. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

Trigg_1 Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T2] (*Trigg 2*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T3] (*Trigg 3*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

[\T4] (*Trigg 4)* Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO, TriggEquip* or *TriggInt*.

Zone Data type: *zonedata*

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

Tool Data type: *tooldata*

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

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

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

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

Program execution

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

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 0.1 \Time , intno1;
...
TriggC p1, p2, v500, trigg1, fine, gun1;
TriggC p3, p4, v500, trigg1, fine, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the work point is at a position *0.1* s before the point *p2* or *p4* respectively.

Limitations

If the current start point deviates from the usual, so that the total positioning length of the instruction *TriggC* is shorter than usual, it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried out will be undefined. The program logic in the user program may not be based on a normal sequence of trigger activities

for an ”incomplete movement”.

The instruction *TriggC* should never be started from the beginning with the robot in position after the circle point. Otherwise the robot will not take the programmed path (positioning around the circular path in another direction compared with that programmed).

Syntax

```
TriggC
   ['\' Conc ',']
   [ CirPoint ':=' ] < expression (IN) of robtarget > ','
   [ ToPoint ':=' ] < expression (IN) of robtarget > ','
   [ Speed ':=' ] < expression (IN) of speeddata >
      [ '\' T ':=' < expression (IN) of num > ] ','
   [Trigg_1 ':=' ] < variable (VAR) of triggdata >
   [ '\' T2 ':=' < variable (VAR) of triggdata > ]
   [ '\' T3 ':=' < variable (VAR) of triggdata > ]
   [ '\' T4 ':=' < variable (VAR) of triggdata > ] ','
   [Zone ':=' ] < expression (IN) of zonedata > ','
   [ Tool ':=' ] < persistent (PERS) of tooldata >
   [ '\' WObj ':=' < persistent (PERS) of wobjdata > ] ';'
```

Related information

	Described in:
Linear movement with triggers	Instructions - *TriggL*
Joint movement with triggers	Instructions - *TriggJ*
Definition of triggers	Instructions - *TriggIO*, *TriggEquip* *TriggInt*
Circular movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion Principles

TriggEquip Defines a fixed position-time I/O event

TriggEquip (Trigg Equipment) is used to define conditions and actions for setting a digital, a group of digital, or an analog output signal at a fixed position along the robot's movement path with possibility to do time compensation for the lag in the external equipment.

The data defined is used for implementation in one or more subsequent *TriggL, TriggC* or *TriggJ* instructions.

Examples

VAR triggdata gunon;

TriggEquip gunon, 10, 0.1 \DOp:=gun, 1;

TriggL p1, v500, gunon, z50, gun1;

The tool *gun1* opens in point p2, when the TCP is *10* mm before the point *p1*. To reach this, the digital output signal *gun* is set to the value *1*, when TCP is *0.1* s before the point p2. The gun is full open when TCP reach point p2.

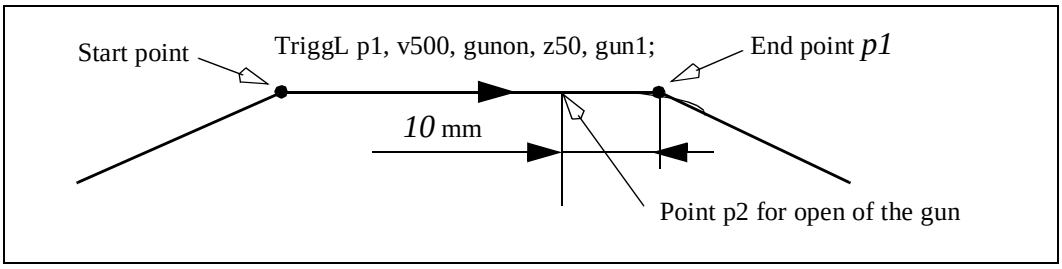


Figure 1 Example of fixed position-time I/O event.

Arguments

TriggEquip TriggData Distance [\Start] EquipLag [\DOp] | [\GOp] | [\AOp] 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.

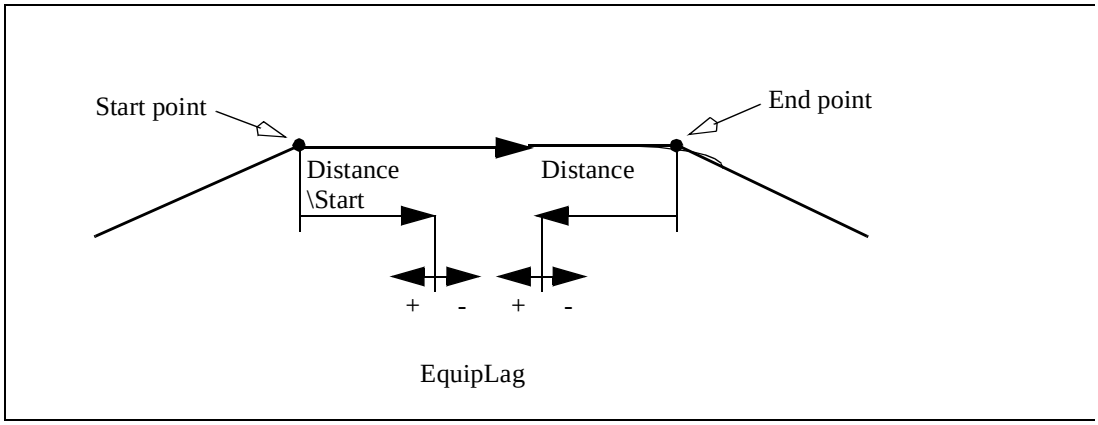


Figure 2 Use of argument EquipLag.

[\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.

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).

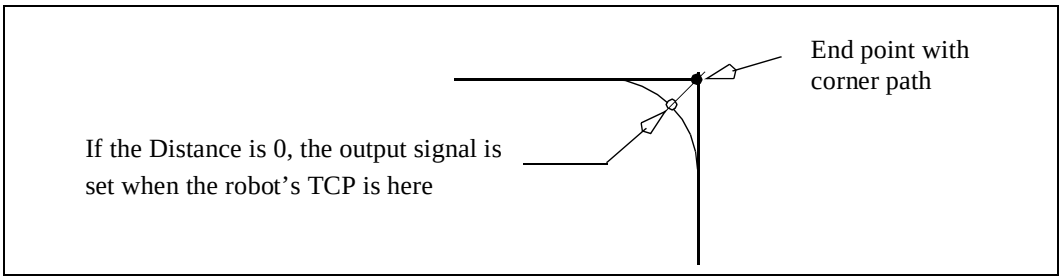


Figure 3 Fixed position-time I/O on a corner path.

The position-time related event will be generated when the start point (end point) is passed, if the specified distance from the end point (start point) is not within the length of movement of the current instruction (*Trigg...*). With use of argument *EquipLag* with negative time (delay), the I/O signal can be set after the end point.

Examples

```
VAR triggdata glueflow;

TriggEquip glueflow, 1 \Start, 0.05 \AOp:=glue, 5.3;

MoveJ p1, v1000, z50, tool1;
TriggL p2, v500, glueflow, z50, tool1;
```

The analog output signal *glue* is set to the value *5.3* when the TCP passes a point located *1* mm after the start point *p1* with compensation for equipment lag *0.05* s.

```
...
```

TriggL p3, v500, glueflow, z50, tool1;

The analog output signal *glue* is set once more to the value *5.3* when the TCP passes a point located *1* mm after the start point *p2*.

Limitations

Regarding accuracy with *TriggEquip*, the following is applicable when setting I/O at a specified distance from the start point/end point in the instructions *TriggL* or *TriggC*:

- Better accuracy is obtained by using corner paths than with stop points.
- When setting I/O in advance, the best accuracy is obtained when *EquipLag* < 60 ms, equivalent to the lag in the robot's servo. For *EquipLag* > 60 ms, an approximate method is used in which the dynamic limitations of the robot are taken into consideration. For *EquipLag* > 60 ms, SingArea must be used in the vicinity of a singular point in order to achieve an acceptable accuracy.
- When using delayed I/O setting by specifying a negative *EquipLag*, the resolution obtained is better for digital signals (1 ms) than for analog signals (10 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*>] ','
[SetValue ':='] < expression (**IN**) of *num*>
['\' Inhibit ':=' < persistent (**PERS**) of *bool*>] ','

Related information

	Described in:
Use of triggers	Instructions - *TriggL, TriggC, TriggJ*
Definition of other triggs	Instruction - *TriggIO, TriggInt*
More examples	Data Types - *triggdata*
Set of I/O	Instructions - *SetDO, SetGO, SetAO*

TriggInt Defines a position related interrupt

TriggInt is used to define conditions and actions for running an interrupt routine at a position on the robot's movement path.

The data defined is used for implementation in one or more subsequent *TriggL, TriggC* or *TriggJ* instructions.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 5, intno1;
...
TriggL p1, v500, trigg1, z50, gun1;
TriggL p2, v500, trigg1, z50, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the TCP is at a position *5* mm before the point *p1* or *p2* respectively.

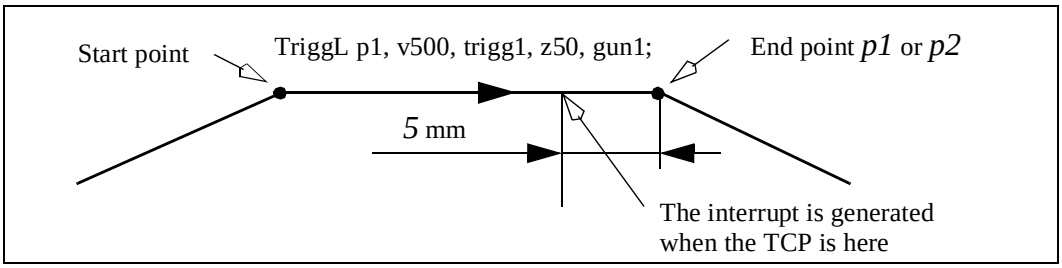


Figure 1 Example position related interrupt.

Arguments

TriggInt TriggData Distance [\Start] | [\Time] Interrupt

TriggData Data type: *triggdata*

Variable for storing the *triggdata* returned from this instruction. These *triggdata* are then used in the subsequent *TriggL, TriggC* or *TriggJ* instructions.

Distance Data type: *num*

Defines the position on the path where the interrupt shall be generated.

Specified as the distance in mm (positive value) from the end point of the movement path (applicable if the argument *Start* or *Time* is not set).

See the section entitled Program execution for further details.

[\Start] Data type: *switch*

Used when the distance for the argument *Distance* starts at the movement start point instead of the end point.

[\Time] Data type: *switch*

Used when the value specified for the argument *Distance* is in fact a time in seconds (positive value) instead of a distance.

Position related interrupts in time can only be used for short times (< 0.5 s) before the robot reaches the end point of the instruction. See the section entitled Limitations for more details.

Interrupt Data type: *intnum*

Variable used to identify an interrupt.

Program execution

When running the instruction *TriggInt*, data is stored in a specified variable for the argument *TriggData* and the interrupt that is specified in the variable for the argument *Interrupt* is activated.

Afterwards, when one of the instructions *TriggL*, *TriggC* or *TriggJ* is executed, the following are applicable, with regard to the definitions in *TriggInt*:

The distance specified in the argument *Distance*:

Linear movement	The straight line distance
Circular movement	The circle arc length
Non-linear movement	The approximate arc length along the path (to obtain adequate accuracy, the distance should not exceed one half of the arc length).

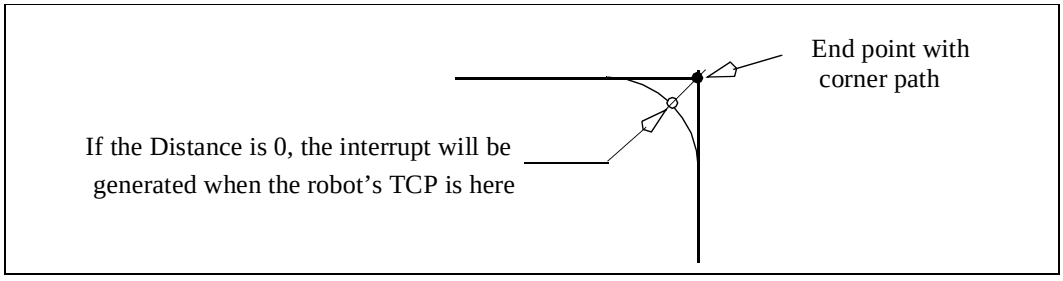


Figure 2 Position related interrupt on a corner path.

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 in distance (without the argument *Time*) give better accuracy when using flying points compared with stop points for the end point (or start point). I/O events in time (with the argument *Time*) give better accuracy when using stop points compared with flying points.

Interrupt events in time (with the argument *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 interrupt 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.

To obtain the best accuracy when setting an output at a fixed position along the robot's path, you should use the instruction *TriggIO* or *TriggEquip* in preference to the instructions *TriggInt* with *SetDO/SetGO/SetAO* in an interrupt routine.

Syntax

TriggInt
[TriggData ':='] < variable (**VAR**) of *triggdata*> ','
[Distance ':='] < expression (**IN**) of *num*>
['\' Start] | ['\' Time] ','
[Interrupt ':='] < variable (**VAR**) of *intnum*> ';'

Related information

	Described in:
Use of triggers	Instructions - *TriggL, TriggC, TriggJ*
Definition of position fix I/O	Instruction - *TriggIO, TriggEquip*
More examples	Data Types - *triggdata*
Interrupts	Basic Characteristics - *Interrupts*

TriggIO Defines a fixed position I/O event

TriggIO is used to define conditions and actions for setting a digital, a group of digital, or an analog output signal at a fixed position along the robot's movement path.

To obtain a fixed position I/O event, *TriggIO* compensates for the lag in the control system (lag between robot and servo) but not for any lag in the external equipment. For compensation of both lags use *TriggEquip*.

The data defined is used for implementation in one or more subsequent *TriggL*, *TriggC* or *TriggJ* instructions.

Examples

```
VAR triggdata gunon;

TriggIO gunon, 10 \DOp:=gun, 1;

TriggL p1, v500, gunon, z50, gun1;
```

The digital output signal *gun* is set to the value *1* when the TCP is *10* mm before the point *p1*.

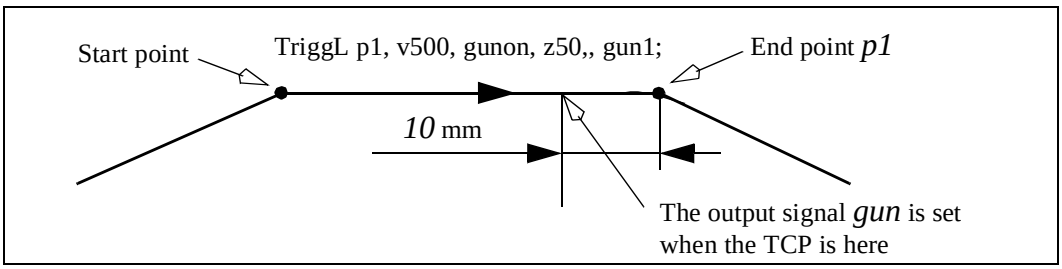


Figure 1 Example of fixed-position IO event.

Arguments

TriggIO TriggData Distance [\Start] | [\Time] [\DOp] | [\GOp] | [\AOp] 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.

SetValue Data type: *num*

Desired value of output signal (within the allowed range for the current signal).

[\DODelay] (*Digital Output Delay)* Data type: *num*

Time delay in seconds (positive value) for a digital output signal or group of digital output signals.

Only used to delay setting digital output signals, after the robot has reached the specified position. There will be no delay if the argument is omitted.

The delay is not synchronised with the movement.

Program execution

When running the instruction *TriggIO*, the trigger condition is stored in a specified variable for the argument *TriggData*.

Afterwards, when one of the instructions *TriggL*, *TriggC* or *TriggJ* is executed, the following are applicable, with regard to the definitions in *TriggIO*:

The distance specified in the argument *Distance*:

Linear movement	The straight line distance
Circular movement	The circle arc length
Non-linear movement	The approximate arc length along the path (to obtain adequate accuracy, the distance should not exceed one half of the arc length).

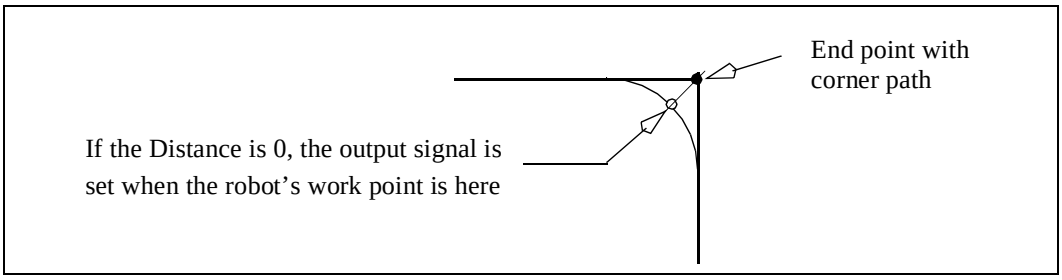


Figure 2 Fixed position I/O on a corner path.

The fixed position I/O will be generated when the start point (end point) is passed, if the specified distance from the end point (start point) is not within the length of movement of the current instruction (*Trigg...*).

Examples

```
VAR triggdata glueflow;

TriggIO glueflow, 1 \Start \AOp:=glue, 5.3;

MoveJ p1, v1000, z50, tool1;
TriggL p2, v500, glueflow, z50, tool1;
```

The analog output signal *glue* is set to the value *5.3* when the work point passes a point located *1* mm after the start point *p1*.

```
...
TriggL p3, v500, glueflow, z50, tool1;
```

The analog output signal *glue* is set once more to the value *5.3* when the work point passes a point located *1* mm after the start point *p2*.

Limitations

I/O events in distance (without the argument *Time*) give better accuracy when using flying points compared with stop points for the end point (or start point). I/O events in time (with the argument *Time*) gives better accuracy when using stop points compared with flying points.

I/O events in time (with the argument *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.

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*>] ','
[SetValue ':='] < expression (**IN**) of *num*>
['\' DODelay ':=' < expression (**IN**) of *num*>] ';'

Related information

	Described in:
Use of triggers	Instructions - *TriggL, TriggC, TriggJ*
Definition of position-time I/O event	Instruction - *TriggEquip*
Definition of position related interrupts	Instruction - *TriggInt*
More examples	Data Types - *triggdata*
Set of I/O	Instructions - *SetDO, SetGO, SetAO*

TriggJ Axis-wise robot movements with events

TriggJ (Trigg Joint) is used to set output signals and/or run interrupt routines at fixed positions, at the same time as the robot is moving on a circular path.

One or more (max. 4) events can be defined using the instructions *TriggIO*, *TriggEquip* or *TriggInt* and afterwards these definitions are referred to in the instruction *TriggJ*.

Examples

```
VAR triggdata gunon;

TriggIO gunon, 0 \Start \DOp:=gun, on;

MoveL p1, v500, z50, gun1;
TriggJ p2, v500, gunon, fine, gun1;
```

The digital output signal *gun* is set when the robot's TCP passes the midpoint of the corner path of the point *p1*.

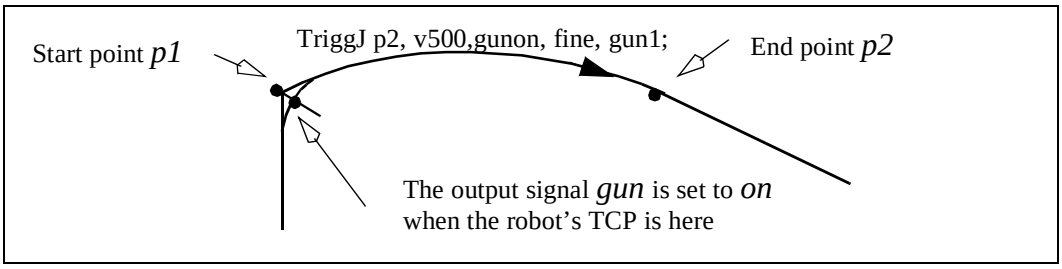


Figure 1 Example of fixed-position IO event.

Arguments

TriggJ [\Conc] ToPoint Speed [\T] Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj]

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

Subsequent logical instructions are executed while the robot is moving. This argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required. It can also be used to tune the execution of the robot path, to avoid warning 50024 Corner path failure or error 40082 Deceleration limit.

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

If this argument is omitted, the subsequent instruction is only executed after the robot has reached the specified stop point or 100 ms before the specified zone.

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

Trigg_1 Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

[\T2] (*Trigg 2*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

[\T3] (*Trigg 3*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

[\T4] (*Trigg 4)* Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

Zone Data type: *zonedata*

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

Tool Data type: *tooldata*

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

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

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

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

axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

Program execution

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

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 0.1 \Time , intno1;
...
TriggJ p1, v500, trigg1, fine, gun1;
TriggJ p2, v500, trigg1, fine, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the work point is at a position *0.1* s before the point *p1* or *p2* respectively.

Limitations

If the current start point deviates from the usual, so that the total positioning length of the instruction *TriggJ* is shorter than usual (e.g. at the start of *TriggJ* with the robot position at the end point), it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried will be undefined. The program logic in the user program may not be based on a normal sequences of trigger activities for an ”incomplete movement”.

Syntax

```
TriggJ
   ['\' Conc ',']
   [ ToPoint ':=' ] < expression (IN) of robtarget > ','
   [ Speed ':=' ] < expression (IN) of speeddata >
      [ '\' T ':=' < expression (IN) of num > ] ','
   [Trigg_1 ':=' ] < variable (VAR) of triggdata >
   [ '\' T2 ':=' < variable (VAR) of triggdata > ]
   [ '\' T3 ':=' < variable (VAR) of triggdata > ]
   [ '\' T4 ':=' < variable (VAR) of triggdata > ] ','
   [Zone ':=' ] < expression (IN) of zonedata > ','
   [ Tool ':=' ] < persistent (PERS) of tooldata >
   [ '\' WObj ':=' < persistent (PERS) of wobjdata > ] ';'
```

Related information

	Described in:
Linear movement with triggs	Instructions - *TriggL*
Circular movement with triggers	Instructions - *TriggC*
Definition of triggers	Instructions - *TriggIO*, *TriggEquip* or *TriggInt*
Joint movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion Principles

TriggL Linear robot movements with events

TriggL (Trigg Linear) is used to set output signals and/or run interrupt routines at fixed positions, at the same time as the robot is making a linear movement.

One or more (max. 4) events can be defined using the instructions *TriggIO*, *TriggEquip* or *TriggInt* and afterwards these definitions are referred to in the instruction *TriggL*.

Examples

```
VAR triggdata gunon;

TriggIO gunon, 0 \Start \DOp:=gun, on;

MoveJ p1, v500, z50, gun1;
TriggL p2, v500, gunon, fine, gun1;
```

The digital output signal *gun* is set when the robot's TCP passes the midpoint of the corner path of the point *p1*.

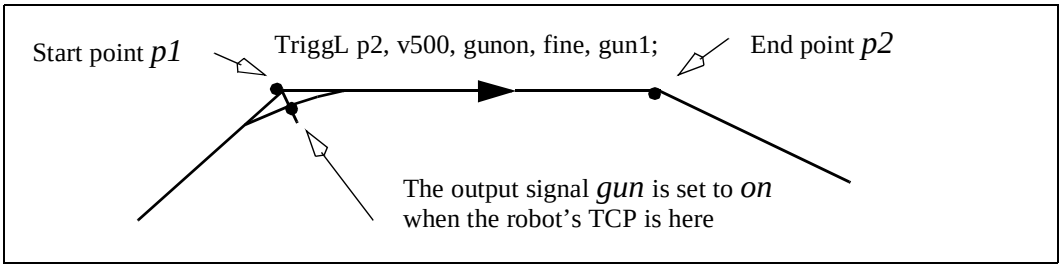


Figure 1 Example of fixed-position IO event.

Arguments

TriggL [\Conc] ToPoint Speed [\T] Trigg_1 [\T2] [\T3] [\T4] Zone Tool [\WObj]

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

Subsequent logical instructions are executed while the robot is moving. This argument is used to shorten the cycle time when, for example, communicating with external equipment, if synchronisation is not required. It can also be used to tune the execution of the robot path, to avoid warning 50024 Corner path failure or error 40082 Deceleration limit.

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

If this argument is omitted, the subsequent instruction is only executed after the robot has reached the specified stop point or 100 ms before the specified zone.

ToPoint Data type: *robtarget*

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

Speed Data type: *speeddata*

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

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

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

Trigg_1 Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

[\T2] (*Trigg 2*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

[\T3] (*Trigg 3*) Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

[\T4] (*Trigg 4)* Data type: *triggdata*

Variable that refers to trigger conditions and trigger activity, defined earlier in the program using the instructions *TriggIO*, *TriggEquip* or *TriggInt*.

Zone Data type: *zonedata*

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

Tool Data type: *tooldata*

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

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

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

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

axes are used, this argument must be specified for a linear movement relative to the work object to be performed.

Program execution

See the instruction *MoveL* for information about linear movement.

As the trigger conditions are fulfilled when the robot is positioned closer and closer to the end point, the defined trigger activities are carried out. The trigger conditions are fulfilled either at a certain distance before the end point of the instruction, or at a certain distance after the start point of the instruction, or at a certain point in time (limited to a short time) before the end point of the instruction.

During stepping execution forwards, the I/O activities are carried out but the interrupt routines are not run. During stepping execution backwards, no trigger activities at all are carried out.

Examples

```
VAR intnum intno1;
VAR triggdata trigg1;
...
CONNECT intno1 WITH trap1;
TriggInt trigg1, 0.1 \Time , intno1;
...
TriggL p1, v500, trigg1, fine, gun1;
TriggL p2, v500, trigg1, fine, gun1;
...
IDelete intno1;
```

The interrupt routine *trap1* is run when the work point is at a position *0.1* s before the point *p1* or *p2* respectively.

Limitations

If the current start point deviates from the usual, so that the total positioning length of the instruction *TriggL* is shorter than usual (e.g. at the start of *TriggL* with the robot position at the end point), it may happen that several or all of the trigger conditions are fulfilled immediately and at the same position. In such cases, the sequence in which the trigger activities are carried out will be undefined. The program logic in the user program may not be based on a normal sequence of trigger activities for an ”incomplete movement”.

Syntax

```
TriggL
   ['\' Conc ',']
   [ ToPoint ':=' ] < expression (IN) of robtarget > ','
   [ Speed ':=' ] < expression (IN) of speeddata >
      [ '\' T ':=' < expression (IN) of num > ] ','
   [Trigg_1 ':=' ] < variable (VAR) of triggdata >
   [ '\' T2 ':=' < variable (VAR) of triggdata > ]
   [ '\' T3 ':=' < variable (VAR) of triggdata > ]
   [ '\' T4 ':=' < variable (VAR) of triggdata > ] ','
   [Zone ':=' ] < expression (IN) of zonedata > ','
   [ Tool ':=' ] < persistent (PERS) of tooldata >
   [ '\' WObj ':=' < persistent (PERS) of wobjdata > ] ';'
```

Related information

	Described in:
Circular movement with triggers	Instructions - *TriggC*
Joint movement with triggers	Instructions - *TriggJ*
Definition of triggers	Instructions - *TriggIO*, *TriggEquip* or *TriggInt*
Linear movement	Motion Principles - *Positioning during Program Execution*
Definition of velocity	Data Types - *speeddata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Motion in general	Motion Principles

TRYNEXT Jumps over an instruction which has caused an error

TRYNEXT is used to jump over an instruction which has caused an error. Instead, the next instruction is run.

Example

```
reg2 := reg3/reg4;
   .
ERROR
   IF ERRNO = ERR_DIVZERO THEN
      reg2:=0;
      TRYNEXT;
   ENDIF
```

An attempt is made to divide *reg3* by *reg4*. If reg4 is equal to 0 (division by zero), a jump is made to the error handler, where *reg2* is set to 0. The *TRYNEXT* instruction is then used to continue with the next instruction.

Program execution

Program execution continues with the instruction subsequent to the instruction that caused the error.

Limitations

The instruction can only exist in a routine's error handler.

Syntax

TRYNEXT';'

Related information

	Described in:
Error handlers	Basic Characteristics- *Error Recovery*

TuneReset Resetting servo tuning

TuneReset is used to reset the dynamic behaviour of all robot axes and external mechanical units to their normal values.

Example

TuneReset;

Resetting tuning values for all axes to 100%.

Program execution

The tuning values for all axes are reset to 100%.

Syntax

TuneReset ';'

Related information

	Described in:
Tuning servos	Instructions - *TuneServo*

TuneServo Tuning servos

TuneServo is used to tune the dynamic behaviour 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*.

Note. To obtain optimal tuning it is essential that the correct load data is used. Check on this before using *TuneServo*.

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 behaviour. 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 alterna-

tive method to the use of *TuneServo*.

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 speed (< 500 mm/s).

Recommended values: tune_kv 100 - 180%, tune_ti 50 - 100%. Tune_kp should not be used for robot axes. Too high or low values of tune_kv/tune_ti cause vibrations or oscillations.

Never use these tune types at high speed or when the required path accuracy is fulfilled.

Example

TuneServo IRB, 2, 90;

Activating of tuning type *TUNE_DF* with the tuning value *90*% on axis *2* in the mechanical unit *IRB*.

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 - 200). 100% is the normal value.

[\Type] Data type: *tunetype*

Type of servo tuning. Available types are *TUNE_DF*, *TUNE_KP*, *TUNE_KV* and *TUNE_TI*. 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*.

Syntax

TuneServo
[MecUnit ':='] < variable (**VAR**) of *mecunit*> ','
[Axis ':='] < expression (**IN**) of *num*> ','
[TuneValue ':='] < expression (**IN**) of *num*>
['\' Type ':=' <expression (**IN**) of *tunetype*>]';'

Related information

	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*
-	

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.*

Example

UnLoad ram1disk \File:="PART_A.MOD";

UnLoad the program module *PART_A.MOD* from the program memory, that previously was loaded into the program memory with *Load*. (See instructions *Load*). (*ram1disk* is a predefined string constant "ram1disk:").

Arguments

UnLoad FilePath [\File]

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* 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* instruction.

Program execution

To be able to execute a *UnLoad* instruction in the program, a *Load* instruction with the same file path and name must have been executed earlier in the program.

The program execution waits for the program module to be finish unloading before the execution proceeds with the next instruction.

After that the program module is unloaded, the rest of the program modules will be linked.

For more information see the instruction called *Load*.

Examples

UnLoad "ram1disk:DOORDIR/DOOR1.MOD";

UnLoad the program module *DOOR1.MOD* from the program memory, that previously was loaded into the program memory with *Load*. (See instructions *Load*).

UnLoad "ram1disk:DOORDIR" \File:="DOOR1.MOD";

Same as above but another syntax.

Limitations

It is not allowed to unload a program module that is executing.

TRAP routines, system I/O events and other program tasks cannot execute during the unloading.

Avoid ongoing robot movements during the unloading.

Program stop during execution of *UnLoad* instruction results in guard stop with motors off and error message "20025 Stop order timeout" on the Teach Pendant.

Error handling

If the file in the *UnLoad* instruction cannot be unloaded, because of ongoing execution within the module or wrong path (module not loaded with *Load*), then the system variable ERRNO is set to ERR_UNLOAD (see “Data types - errnum”). This error can then be handled in the error handler.

Syntax

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

Related information

	Described in:
Load a program module	Instructions - *Load*
Accept unresolved references	System Parameters - *Controller* System Parameters - *Tasks* System Parameters - *BindRef*

VelSet Changes the programmed velocity

VelSet is used to increase or decrease the programmed velocity of all subsequent positioning instructions. This instruction is also used to maximize the velocity.

Example

VelSet 50, 800;

All the programmed velocities are decreased to 50% of the value in the instruction. The TCP velocity is not, however, permitted to exceed 800 mm/s.

Arguments

VelSet Override Max

Override Data type: *num*

Desired velocity as a percentage of programmed velocity. 100% corresponds to the programmed velocity.

Max Data type: *num*

Maximum TCP velocity in mm/s.

Program execution

The programmed velocity of all subsequent positioning instructions is affected until a new *VelSet* instruction is executed.

The argument *Override* affects:

- All velocity components (TCP, orientation, rotating and linear external axes) in *speeddata*.
- The programmed velocity override in the positioning instruction (the argument *\V*).
- Timed movements.

The argument *Override* does not affect:

- The welding speed in *welddata*.
- The heating and filling speed in *seamdata*.

The argument *Max* only affects the velocity of the TCP.

The default values for *Override* and *Max* are 100% and 5000 mm/s respectively. These values are automatically set

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

Example

```
VelSet 50, 800;
MoveL p1, v1000, z10, tool1;
MoveL p2, v2000, z10, tool1;
MoveL p3, v1000\T:=5, z10, tool1;
```

The speed is *500* mm/s to point *p1* and *800* mm/s to *p2*. It takes *10* seconds to move from p2 to *p3*.

Limitations

The maximum speed is not taken into consideration when the time is specified in the positioning instruction.

Syntax

```
VelSet
   [ Override ':=' ] < expression (IN) of num > ','
   [ Max ':=' ] < expression (IN) of num > ';'
```

Related information

	Described in:
Definition of velocity	Data Types - *speeddata*
Positioning instructions	RAPID Summary - *Motion*

WaitDI Waits until a digital input signal is set

WaitDI (Wait Digital Input) is used to wait until a digital input is set.

Example

WaitDI di4, 1;

Program execution continues only after the *di4* input has been set.

WaitDI grip_status, 0;

Program execution continues only after the *grip_status* input has been reset.

Arguments

WaitDI Signal Value [\MaxTime] [\TimeFlag]

Signal Data type: *signaldi*

The name of the signal.

Value Data type: *dionum*

The desired value of the signal.

[\MaxTime] *(Maximum Time)* Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the condition is met, the error handler will be called, if there is one, with the error code ERR_WAIT_MAXTIME. If there is no error handler, the execution will be stopped.

[\TimeFlag] *(Timeout Flag)* Data type: *bool*

The output parameter that contains the value TRUE if the maximum permitted waiting time runs out before the condition is met. If this parameter is included in the instruction, it is not considered to be an error if the max. time runs out. This argument is ignored if the *MaxTime* argument is not included in the instruction.

Program Running

If the value of the signal is correct, when the instruction is executed, the program simply continues with the following instruction.

If the signal value is not correct, the robot enters a waiting state and when the signal changes to the correct value, the program continues. The change is detected with an interrupt, which gives a fast response (not polled).

When the robot is waiting, the time is supervised, and if it exceeds the max time value, the program will continue if a Time Flag is specified, or raise an error if it's not. If a Time Flag is specified, this will be set to true if the time is exceeded, otherwise it will be set to false.

Syntax

WaitDI
[Signal ':='] < variable (**VAR**) of *signaldi* > ','
[Value ':='] < expression (**IN**) of *dionum* >
['\'MaxTime ':='<expression (**IN**) of *num*>]
['\'TimeFlag':='<variable (**VAR**) of *bool*>] ';'

Related information

	Described in:
Waiting until a condition is satisfied	Instructions - *WaitUntil*
Waiting for a specified period of time	Instructions - *WaitTime*

WaitDO Waits until a digital output signal is set

WaitDO (Wait Digital Output) is used to wait until a digital output is set.

Example

WaitDO do4, 1;

Program execution continues only after the *do4* output has been set.

WaitDO grip_status, 0;

Program execution continues only after the *grip_status* output has been reset.

Arguments

WaitDO Signal Value [\MaxTime] [\TimeFlag]

Signal Data type: *signaldo*

The name of the signal.

Value Data type: *dionum*

The desired value of the signal.

[\MaxTime] *(Maximum Time)* Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the condition is met, the error handler will be called, if there is one, with the error code ERR_WAIT_MAXTIME. If there is no error handler, the execution will be stopped.

[\TimeFlag] *(Timeout Flag)* Data type: *bool*

The output parameter that contains the value TRUE if the maximum permitted waiting time runs out before the condition is met. If this parameter is included in the instruction, it is not considered to be an error if the max. time runs out. This argument is ignored if the *MaxTime* argument is not included in the instruction.

Program Running

If the value of the signal is correct, when the instruction is executed, the program simply continues with the following instruction.

If the signal value is not correct, the robot enters a waiting state and when the signal changes to the correct value, the program continues. The change is detected with an interrupt, which gives a fast response (not polled).

When the robot is waiting, the time is supervised, and if it exceeds the max time value, the program will continue if a Time Flag is specified, or raise an error if its not. If a Time Flag is specified, this will be set to true if the time is exceeded, otherwise it will be set to false.

Syntax

WaitDI
[Signal ':='] < variable (**VAR**) of *signaldo* > ','
[Value ':='] < expression (**IN**) of *dionum* >
['\'MaxTime ':='<expression (**IN**) of *num*>]
['\'TimeFlag':='<variable (**VAR**) of *bool*>] ';'

Related information

	Described in:
Waiting until a condition is satisfied	Instructions - *WaitUntil*
Waiting for a specified period of time	Instructions - *WaitTime*

WaitTime Waits a given amount of time

WaitTime is used to wait a given amount of time. This instruction can also be used to wait until the robot and external axes have come to a standstill.

Example

WaitTime 0.5;

Program execution waits 0.5 seconds.

Arguments

WaitTime [\InPos] Time

[\InPos] Data type: *switch*

If this argument is used, the robot and external axes must have come to a standstill before the waiting time starts to be counted.

Time Data type: *num*

The time, expressed in seconds, that program execution is to wait.

Program execution

Program execution temporarily stops for the given amount of time. Interrupt handling and other similar functions, nevertheless, are still active.

Example

WaitTime \InPos,0;

Program execution waits until the robot and the external axes have come to a standstill.

Limitations

If the argument *Inpos* is used, the movement instruction which precedes this instruction should be terminated with a stop point, in order to be able to restart in this instruction following a power failure.

Syntax

WaitTime
[’\’InPos’,’]
[Time ’:=’] <expression (**IN**) of *num*>’;’

Related information

	Described in:
Waiting until a condition is met	Instructions - *WaitUntil*
Waiting until an I/O is set/reset	Instruction - *WaitDI*

WaitUntil Waits until a condition is met

WaitUntil is used to wait until a logical condition is met; for example, it can wait until one or several inputs have been set.

Example

WaitUntil di4 = 1;

Program execution continues only after the *di4* input has been set.

Arguments

WaitUntil [\InPos] Cond [\MaxTime] [\TimeFlag]

[\InPos] Data type: *switch*

If this argument is used, the robot and external axes must have stopped moving before the condition starts being evaluated.

Cond Data type: *bool*

The logical expression that is to be waited for.

[\MaxTime] Data type: *num*

The maximum period of waiting time permitted, expressed in seconds. If this time runs out before the condition is set, the error handler will be called, if there is one, with the error code ERR_WAIT_MAXTIME. If there is no error handler, the execution will be stopped.

[\TimeFlag] *(Timeout Flag)* Data type: *bool*

The output parameter that contains the value TRUE if the maximum permitted waiting time runs out before the condition is met. If this parameter is included in the instruction, it is not considered to be an error if the max. time runs out. This argument is ignored if the *MaxTime* argument is not included in the instruction.

Program execution

If the programmed condition is not met on execution of a *WaitUntil* instruction, the condition is checked again every 100 ms.

When the robot is waiting, the time is supervised, and if it exceeds the max time value, the program will continue if a *TimeFlag* is specified, or raise an error if it’s not. If a *TimeFlag* is specified, this will be set to TRUE if the time is exceeded, otherwise it will be set to false.

Examples

```
VAR bool timeout;
WaitUntil start_input = 1 AND grip_status = 1\MaxTime := 60
          \TimeFlag := timeout;
IF timeout THEN
   TPWrite "No start order received within expected time";
ELSE
   start_next_cycle;
ENDIF
```

If the two input conditions are not met within *60* seconds, an error message will be written on the display of the teach pendant.

```
WaitUntil \Inpos, di4 = 1;
```

Program execution waits until the robot has come to a standstill and the *di4* input has been set.

Limitation

If the argument *\Inpos is used*, the movement instruction which precedes this instruction should be terminated with a stop point, in order to be able to restart in this instruction following a power failure.

Syntax

```
WaitUntil
   ['\'InPos',']
   [Cond ':='] <expression (IN) of bool>
   ['\'MaxTime ':='<expression (IN) of num>]
   ['\'TimeFlag':='<variable (VAR) of bool>] ';'
```

Related information

	Described in:
Waiting until an input is set/reset	Instructions - *WaitDI*
Waiting a given amount of time	Instructions - *WaitTime*
Expressions	Basic Characteristics - *Expressions*

WHILE Repeats as long as ...

WHILE is used when a number of instructions are to be repeated as long as a given condition is met.

If it is possible to determine the number of repetitions in advance, the *FOR* instruction can be used.

Example

```
WHILE reg1 < reg2 DO
   ...
   reg1 := reg1 +1;
ENDWHILE
```

Repeats the instructions in the WHILE loop as long as *reg1 < reg2*.

Arguments

WHILE Condition DO ... ENDWHILE

Condition Data type: *bool*

The condition that must be met for the instructions in the WHILE loop to be executed.

Program execution

1. The condition is calculated. If the condition is not met, the WHILE loop terminates and program execution continues with the instruction following ENDWHILE.
2. The instructions in the WHILE loop are executed.
3. The WHILE loop is repeated, starting from point 1.

Syntax

(EBNF)
WHILE <conditional expression> **DO**
   <instruction list>
ENDWHILE

Related information

	Described in:
Expressions	Basic Characteristics - *Expressions*

Write Writes to a character-based file or serial channel

Write is used to write to a character-based file or serial channel. The value of certain data can be written as well as text.

Examples

Write logfile, "Execution started";

The text *Execution started* is written to the file with reference name *logfile*.

Write logfile, "No of produced parts="\Num:=reg1;

The text *No of produced parts=5*, for example, is written to the file with the reference name *logfile* (assuming that the contents of *reg1* is 5).

Arguments

Write IODevice String [\Num] | [\Bool] | [\Pos] | [\Orient] [\NoNewLine]

IODevice Data type: *iodev*

The name (reference) of the current file or serial channel.

String Data type: *string*

The text to be written.

[\Num] *(Numeric)* Data type: *num*

The data whose numeric values are to be written after the text string.

[\Bool] *(Boolean)* Data type: *bool*

The data whose logical values are to be written after the text string.

[\Pos] *(Position)* Data type: *pos*

The data whose position is to be written after the text string.

[\Orient] *(Orientation)* Data type: *orient*

The data whose orientation is to be written after the text string.

[\NoNewLine] Data type: *switch*

Omits the line-feed character that normally indicates the end of the text.

Program execution

The text string is written to a specified file or serial channel. If the argument *\NoNewLine* is not used, a line-feed character (LF) is also written.

If one of the arguments *\Num*, *\Bool*, *\Pos* or *\Orient* is used, its value is first converted to a text string before being added to the first string. The conversion from value to text string takes place as follows:

Argument	Value	Text string
\Num	23	"23"
\Num	1.141367	"1.14137"
\Bool	TRUE	"TRUE"
\Pos	[1817.3,905.17,879.11]	"[1817.3,905.17,879.11]"
\Orient	[0.96593,0,0.25882,0]	"[0.96593,0,0.25882,0]"

Decimals are rounded off to 5 places of decimals, but there may be no more than 6 digits in the text string. If the decimal part is less than 0.000005 or greater than 0.999995, the number is rounded to an integer.

Example

```
VAR iodev printer;
.
Open "sio1:", printer\Write;
WHILE DInput(stopprod)=0 DO
    produce_part;
    Write printer, "Produced part="\Num:=reg1\NoNewLine;
    Write printer, "          "\NoNewLine;
    Write printer, CTime();
ENDWHILE
Close printer;
```

A line, including the number of the produced part and the time, is output to a printer each cycle. The printer is connected to serial channel *sio1:*. The printed message could look like this:

```
Produced part=473          09:47:15
```

Limitations

The arguments *\Num*, *\Bool*, *\Pos* and *\Orient* are mutually exclusive and thus cannot be used simultaneously in the same instruction.

This instruction can only be used for files or serial channels that have been opened for writing.

Error handling

If an error occurs during writing, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Syntax

```
Write
    [IODevice':='] <variable (VAR) of iodev>','
    [String':='] <expression (IN) of string>
    ['\'Num':=' <expression (IN) of num> ]
    | ['\'Bool':=' <expression (IN) of bool> ]
    | ['\'Pos':=' <expression (IN) of pos> ]
    | ['\'Orient':=' <expression (IN) of orient> ]
    ['\'NoNewLine]';'
```

Related information

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

WriteBin Writes to a binary serial channel

WriteBin is used to write a number of bytes to a binary serial channel.

Example

WriteBin channel2, text_buffer, 10;

10 characters from the *text_buffer* list are written to the channel referred to by *channel2*.

Arguments

WriteBin IODevice Buffer NChar

IODevice Data type: *iodev*

Name (reference) of the current serial channel.

Buffer Data type: *array of num*

The list (array) containing the numbers (characters) to be written.

NChar *(Number of Characters)* Data type*: num*

The number of characters to be written from the *Buffer*.

Program execution

The specified number of numbers (characters) in the list is written to the serial channel.

Limitations

This instruction can only be used for serial channels that have been opened for binary reading and writing.

Error handling

If an error occurs during writing, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Example

```
VAR iodev channel;
VAR num out_buffer{20};
VAR num input;
VAR num nchar;
Open "sio1:", channel\Bin;

out_buffer{1} := 5;                          ( enq )
WriteBin channel, out_buffer, 1;
input := ReadBin (channel \Time:= 0.1);

IF input = 6 THEN                            ( ack )
   out_buffer{1} := 2;                       ( stx )
   out_buffer{2} := 72;                      ( 'H' )
   out_buffer{3} := 101;                     ( 'e' )
   out_buffer{4} := 108;                     ( 'l' )
   out_buffer{5} := 108;                     ( 'l' )
   out_buffer{6} := 111;                     ( 'o' )
   out_buffer{7} := 32;                      ( ' ' )
   out_buffer{8} := 119;                     ( 'w' )
   out_buffer{9} := 111;                     ( 'o' )
   out_buffer{10} := 114;                    ( 'r' )
   out_buffer{11} := 108;                    ( 'l' )
   out_buffer{12} := 100;                    ( '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.

Syntax

```
WriteBin
   [IODevice':='] <variable (VAR) of iodev>','
   [Buffer':='] <array {*} (IN) of num>','
   [NChar':='] <expression (IN) of num>';'
```

Related information

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

Abs Gets the absolute value

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

Example

reg1 := Abs(reg2);

Reg1 is assigned the absolute value of *reg2*.

Return value

Data type: *num*

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

e.g.	Input value	Returned value
	3	3
	-3	3
	-2.53	2.53

Arguments

Abs (Input)

Input

Data type: *num*

The input value.

Example

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

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

Syntax

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

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

Related information

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

ACos Calculates the arc cosine value

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

Example

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

Return value

Data type: *num*

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

Arguments

ACos (Value)

Value Data type: *num*

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

Syntax

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

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

Related information

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

ASin Calculates the arc sine value

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

Example

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

Return value

Data type: *num*

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

Arguments

ASin (Value)

Value Data type: *num*

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

Syntax

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

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

Related information

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

ATan Calculates the arc tangent value

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

Example

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

Return value

Data type: *num*

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

Arguments

ATan (Value)

Value Data type: *num*

The argument value.

Syntax

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

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

Related information

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

ATan2 Calculates the arc tangent2 value

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

Example

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

Return value

Data type: *num*

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

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

Arguments

ATan2 (Y X)

Y Data type: *num*

The numerator argument value.

X Data type: *num*

The denominator argument value.

Syntax

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

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

Related information

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

CDate Reads the current date as a string

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

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

Example

```
VAR string date;

date := CDate();
```

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

Return value

Data type: *string*

The current date in a string.

The standard date format is "year-month-day", e.g. "93-05-16".

Example

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

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

Syntax

CDate ’(’ ’)’

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

Related Information

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

CJointT Reads the current joint angles

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

Example

VAR jointtarget joints;

joints := CJointT();

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

Return value

Data type: *jointtarget*

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

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

The returned values are related to the calibration position.

Syntax

CJointT'(')'

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

Related information

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

ClkRead Reads a clock used for timing

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

Example

reg1:=ClkRead(clock1);

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

Return value

Data type: *num*

The time in seconds stored in the clock.

Argument

ClkRead (Clock)

Clock Data type: *clock*

The name of the clock to read.

Program execution

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

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

If the clock has overflowed, program execution is stopped with an error message.

Syntax

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

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

Related Information

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

Cos Calculates the cosine value

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

Example

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

Return value

Data type: *num*

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

Arguments

Cos (Angle)

Angle Data type: *num*

The angle value, expressed in degrees.

Syntax

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

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

Related information

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

CPos Reads the current position (pos) data

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

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

Example

```
VAR pos pos1;

pos1 := CPos(\Tool:=tool1 \WObj:=wobj0);
```

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

Return value

Data type: *pos*

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

Arguments

CPos ([\Tool] [\WObj])

[\Tool] Data type: *tooldata*

The tool used for calculation of the current robot position.

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

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

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

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

When programming, it is very sensible to always specify arguments \Tool and \WObj. The function will always then return the wanted position, although some other tool or work object has been activated manually.

Program execution

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

Example

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

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

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

Syntax

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

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

Related information

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

CRobT Reads the current position (robtarget) data

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

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

Example

```
VAR robtarget p1;

p1 := CRobT(\Tool:=tool1 \WObj:=wobj0);
```

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

Return value

Data type: *robtarget*

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

Arguments

CRobT ([\Tool] [\WObj])

[\Tool] Data type: *tooldata*

The tool used for calculation of the current robot position.

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

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

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

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

When programming, it is very sensible to always specify arguments \Tool and \WObj. The function will always then return the wanted position, although some other tool or work object has been activated manually.

Program execution

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

Example

```
VAR robtarget p2;

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

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

Syntax

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

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

Related information

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

CTime Reads the current time as a string

CTime is used to read the current system time.

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

Example

```
VAR string time;

time := CTime();
```

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

Return value

Data type: *string*

The current time in a string.

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

Example

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

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

Syntax

```
CTime ’(’ ’)’
```

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

Related Information

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

CTool Reads the current tool data

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

Example

PERS tooldata temp_tool;

temp_tool := CTool();

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

Return value

Data type: *tooldata*

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

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

Syntax

CTool'('')'

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

Related information

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

CWObj Reads the current work object data

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

Example

```
PERS wobjdata temp_wobj;

temp_wobj := CWObj();
```

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

Return value

Data type: *wobjdata*

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

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

Syntax

```
CWObj'('')'
```

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

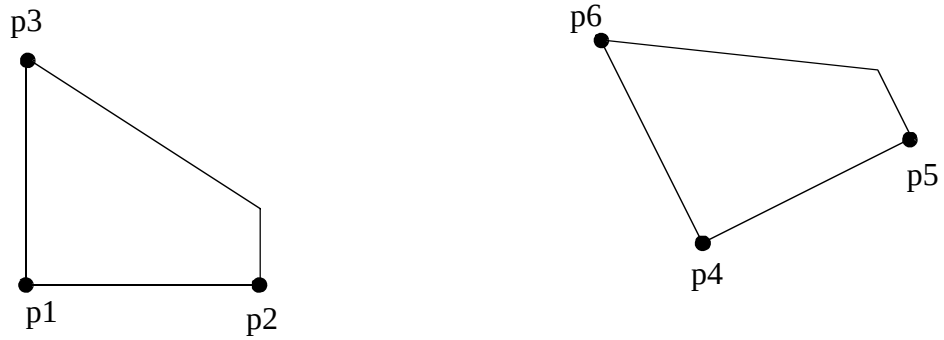
Related information

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

DefDFrame Define a displacement frame

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

Example



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

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

Return value

Data type: *pose*

The displacement frame.

Arguments

DefDFrame (OldP1 OldP2 OldP3 NewP1 NewP2 NewP3)

OldP1 Data type: *robtarget*

The first original position.

OldP2 Data type: *robtarget*

The second original position.

OldP3 Data type: *robtarget*

The third original position.

NewP1 Data type: *robtarget*

The first displaced position. This position must be measured and determined with great accuracy.

NewP2 Data type: *robtarget*

The second displaced position. It should be noted that this position can be measured and determined with less accuracy in one direction, e.g. this position must be placed on a line describing the new direction of *p1* to *p2*.

NewP3 Data type: *robtarget*

The third displaced position. This position can be measured and determined with less accuracy in two directions, e.g. it has to be placed in a plane describing the new plane of *p1*, *p2* and *p3*.

Syntax

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

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

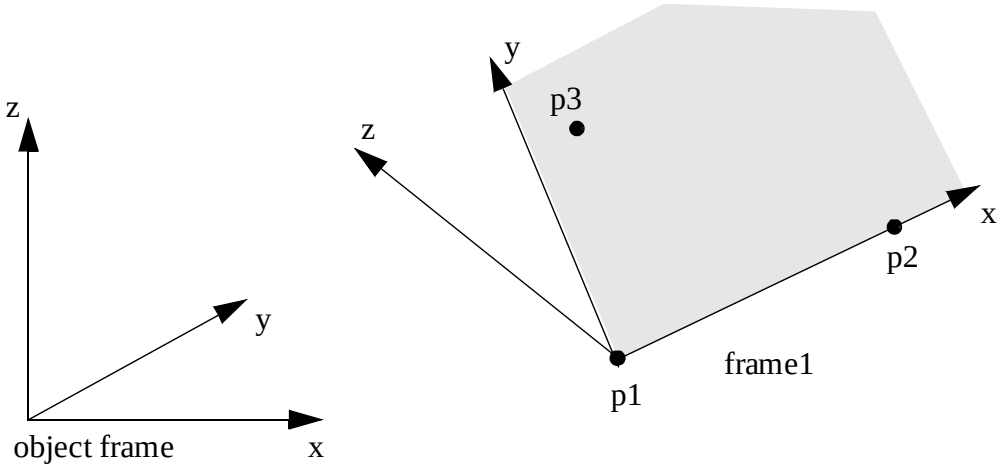
Related information

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

DefFrame Define a frame

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

Example



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

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

Return value

Data type: *pose*

The calculated frame.

The calculation is related to the active object coordinate system.

Arguments

DefFrame (NewP1 NewP2 NewP3 [\Origin])

NewP1 Data type: *robtarget*

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

NewP2 Data type: *robtarget*

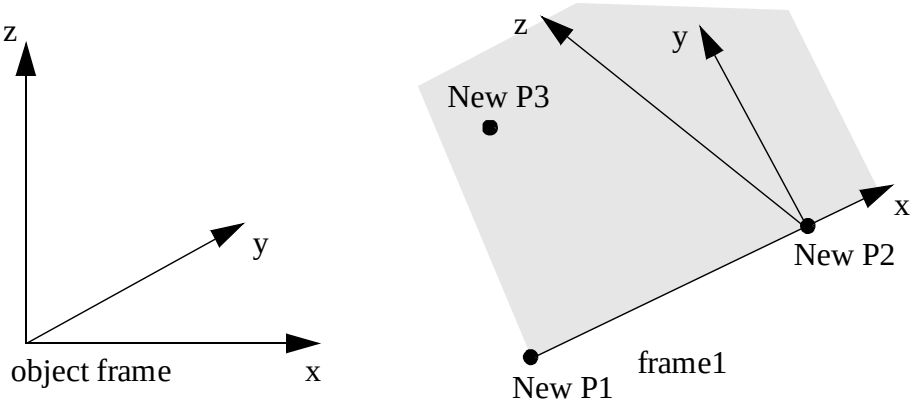
The second position, which will define the direction of the x-axis of the new frame.

NewP3 Data type: *robtarget*

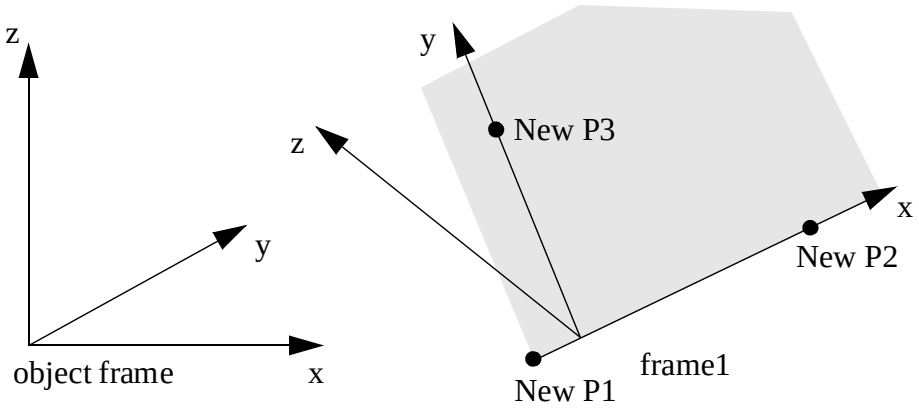
The third position, which will define the xy-plane of the new frame. The position of point 3 will be on the positive y side, see the figure above.

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

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



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



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

Syntax

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

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

Related information

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

Dim Obtains the size of an array

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

Example

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

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

ENDPROC
```

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

Return value

Data type: *num*

The number of array elements of the specified dimension.

Arguments

Dim (ArrPar DimNo)

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

The name of the array.

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

The desired array dimension: 1 = first dimension
2 = second dimension
3 = third dimension

Example

```
PROC add_matrix(VAR num array1{*,*,*}, num array2{*,*,*})

   IF Dim(array1,1) <> Dim(array2,1) OR Dim(array1,2) <> Dim(array2,2) OR
      Dim(array1,3) <> Dim(array2,3) THEN
      TPWrite "The size of the matrices are not the same";
      Stop;
   ELSE
      FOR i1 FROM 1 TO Dim(array1, 1) DO
         FOR i2 FROM 1 TO Dim(array1, 2) DO
            FOR i3 FROM 1 TO Dim(array1, 3) DO
               array1{i1,i2,i3} := array1{i1,i2,i3} + array2{i1,i2,i3};
            ENDFOR
         ENDFOR
      ENDFOR
   ENDIF
   RETURN;

ENDPROC
```

Two matrices are added. If the size of the matrices differs, the program stops and an error message appears.
This procedure can take any three-dimensional arrays of data type *num* as an input.

Syntax

```
Dim '('
   [ArrPar':='] <reference (REF) of any type> ','
   [DimNo':='] <expression (IN) of num> ')'
```

A REF parameter requires that the corresponding argument be either a constant, a variable or an entire persistent. The argument could also be an IN parameter, a VAR parameter or an entire PERS parameter.

A function with a return value of the data type *num*.

Related information

	Described in:
Array parameters	Basic Characteristics - *Routines*
Array declaration	Basic Characteristics - *Data*

DOutput Reads the value of a digital output signal

DOutput is used to read the current value of a digital output signal.

Example

IF DOutput(do2) = 1 THEN . . .

If the current value of the signal *do2* is equal to *1*, then . . .

Return value

Data type: *dionum*

The current value of the signal (0 or 1).

Arguments

DOutput (Signal)

Signal Data type: *signaldo*

The name of the signal to be read.

Program execution

The value read depends on the configuration of the signal. If the signal is inverted in the system parameters, the value returned by this function is the opposite of the true value of the physical channel.

Example

IF DOutput(auto_on) <> active THEN . . .

If the current value of the system signal *auto_on* is *not active*, then ..., i.e. if the robot is in the manual operating mode, then ... Note that the signal must first be defined as a system output in the system parameters.

Syntax

DOutput '('
[Signal ':='] < variable (**VAR**) of *signaldo* > ')'

A function with a return value of the data type *dionum*.

Related information

	Described in:
Input/Output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	User's Guide - *System Parameters*

EulerZYX Gets Euler angles from orient

EulerZYX (Euler ZYX rotations) is used to get an Euler angle component from an orient type variable.

Example

```
VAR num anglex;
VAR num angley;
VAR num anglez;
VAR pose object;
.
.
anglex := GetEuler(\X, object.rot);
angley := GetEuler(\Y, object.rot);
anglez := GetEuler(\Z, object.rot);
```

Return value

Data type: *num*

The corresponding Euler angle, expressed in degrees, range [-180, 180].

Arguments

EulerZYX ([\X] | [\Y] | [\Z] Rotation)

The arguments \X, \Y and \Z are mutually exclusive. If none of these are specified, a run-time error is generated.

[\X] Data type: *switch*

Gets the rotation around the X axis.

[**Y**] Data type: *switch*

Gets the rotation around the Y axis.

[**Z**] Data type: *switch*

Gets the rotation around the Z axis.

Rotation Data type: *orient*

The rotation in its quaternion representation.

Syntax

```
EulerZYX'('
    ['\'X ',’] | ['\'Y ',’] | ['\'Z ',’]
    Rotation ':='] <expression (IN) of orient>
    ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Exp Calculates the exponential value

Exp (Exponential) is used to calculate the exponential value, e^x.

Example

```
VAR num x;
VAR num value;
.
.
value:= Exp( x);
```

Return value

Data type: *num*

The exponential value e^x .

Arguments

Exp (Exponent)

Exponent — Data type: *num*

The exponent argument value.

Syntax

Exp'('
[Exponent ':='] <expression (**IN**) of *num*>
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

GetTime Reads the current time as a numeric value

GetTime is used to read a specified component of the current system time as a numeric value.

GetTime can be used to :

- have the program perform an action at a certain time,
- perform certain activities on a weekday,
- abstain from performing certain activities on the weekend,
- respond to errors differently depending on the time of day.

Example

```
hour := GetTime(\Hour);
```

The current hour is stored in the variable *hour*.

Return value

Data type: *num*

One of the four time components specified below.

Argument

GetTime ([\WDay] | [\Hour] | [\Min] | [\Sec])

[\WDay] Data type: *switch*

Return the current weekday.
Range: 1 to 7 (Monday to Sunday).

[\Hour] Data type: *switch*

Return the current hour.
Range: 0 to 23.

[\Min] Data type: *switch*

Return the current minute.
Range: 0 to 59.

[\Sec] Data type: *switch*

Return the current second.
Range: 0 to 59.

One of the arguments must be specified, otherwise program execution stops with an error message.

Example

```
weekday := GetTime(\WDay);
hour := GetTime(\Hour);
IF weekday < 6 AND hour >6 AND hour < 16 THEN
   production;
ELSE
   maintenance;
ENDIF
```

If it is a weekday and the time is between 7:00 and 15:59 the robot performs production. At all other times, the robot is in the maintenance mode.

Syntax

```
GetTime '('
   ['\' WDay ]
   | [ '\' Hour ]
   | [ '\' Min ]
   | [ '\' Sec ] ')'
```

A function with a return value of the type *num*.

Related Information

	Described in:
Time and date instructions	RAPID Summary - *System & Time*
Setting the system clock	User's Guide - *System Parameters*

GOutput Reads the value of a group of digital output signals

G*Output* is used to read the current value of a group of digital output signals.

Example

IF GOutput(go2) = 5 THEN ...

If the current value of the signal *go2* is equal to *5*, then ...

Return value

Data type: *num*

The current value of the signal (a positive integer).

The values of each signal in the group are read and interpreted as an unsigned binary number. This binary number is then converted to an integer.

The value returned lies within a range that is dependent on the number of signals in the group.

No. of signals	Return value	No. of signals	Return value
1	0 - 1	9	0 - 511
2	0 - 3	10	0 - 1023
3	0 - 7	11	0 - 2047
4	0 - 15	12	0 - 4095
5	0 - 31	13	0 - 8191
6	0 - 63	14	0 - 16383
7	0 - 127	15	0 - 32767
8	0 - 255	16	0 - 65535

Arguments

GOutput (Signal)

Signal Data type: *signalgo*

The name of the signal group to be read.

Syntax

GOutput ’(’
[Signal ’:=’] < variable (**VAR**) of *signalgo* > ’)’

A function with a return value of data type *num*.

Related information

	Described in:
Input/Output instructions	RAPID Summary - *Input and Output Signals*
Input/Output functionality in general	Motion and I/O Principles - *I/O Principles*
Configuration of I/O	User’s Guide - *System Parameters*

IndInpos Independent In position status

IndInpos is used to test whether an independent axis has reached the selected position.

Example

```
IndAMove Station_A,1\ToAbsNum:=90,20;
WaitUntil IndInpos(Station_A,2) = TRUE;
WaitTime 0.2;
```

Wait until axis 1 of *Station_A* is in the 90 degrees position.

Return value

Data type: *bool*

The return values from *IndInpos* are:

Return value	Axis status
TRUE	In position and has zero speed.
FALSE	Not in position and/or has not zero speed

Arguments

IndInpos MecUnit Axis

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).

Limitations

An independent axis executed with the instruction *IndCMove* always returns the value FALSE, even when the speed is set to zero.

A wait period of 0.2 seconds should be added after the instruction, to ensure that the correct status has been achieved. This time period should be longer for external axes with poor performance.

Error handling

If the axis is not in independent mode, the system variable ERRNO will be set to ERR_AXIS_IND.

Syntax

IndInpos ’(’
[MecUnit’:=’] < variable (**VAR**) of *mecunit*> ’,’
[Axis’:=’] < expression (**IN**) of *num*>’)’

A function with a return value of the data type *bool*.

Related information

	Described in:
Independent axes in general	Motion and I/O Principles - *Program execution*
Check the speed status for independent axes	Functions - *IndSpeed*

IndSpeed Independent Speed status

IndSpeed is used to test whether an independent axis has reached the selected speed.

Example

```
IndCMove Station_A, 2, 3.4;
WaitUntil IndSpeed(Station_A,2 \InSpeed) = TRUE;
WaitTime 0.2;
```

Wait until axis *2* of *Station_A* has reached the speed 3.4 degrees/s.

Return value

Data type: *bool*

The return values from *IndSpeed* are:

Return value	Axis status
option *\InSpeed*	
TRUE	Has reached the selected speed.
FALSE	Has not reached the selected speed.
option *\ZeroSpeed*	
TRUE	Zero speed.
FALSE	Not zero speed

Arguments

IndSpeed MecUnit Axis [\InSpeed] | [\ZeroSpeed]

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).

[\InSpeed] Data type: *switch*

IndSpeed returns value TRUE if the axis has reached the selected speed otherwise FALSE.

[\ZeroSpeed] Data type: *switch*

IndSpeed returns value TRUE if the axis has zero speed otherwise FALSE.

If both the arguments *\InSpeed* and *\ZeroSpeed* are omitted, an error message will be displayed.

Limitation

The function *IndSpeed\InSpeed* will always return the value FALSE in the following situations:

- The robot is in manual mode with reduced speed.
- The speed is reduced using the *VelSet* instruction.
- The speed is reduced from the production window.

A wait period of 0.2 seconds should be added after the instruction, to ensure that the correct status is obtained. This time period should be longer for external axes with poor performance.

Error handling

If the axis is not in independent mode, the system variable ERRNO will be set to ERR_AXIS_IND.

Syntax

IndSpeed ’(’
[MecUnit’:=’] < variable (**VAR**) of *mecunit*> ’,’
[Axis’:=’] < expression (**IN**) of *num*>
[’\’ InSpeed] | [’\’ ZeroSpeed] ’)’

A function with a return value of the data type *bool*.

Related information

	Described in:
Independent axes in general	Motion and I/O Principles - *Program execution*
More examples	Instructions - *IndCMove*
Check the position status for independent axes	Functions - *IndInpos*

IsPers Is Persistent

IsPers is used to test if a data object is a persistent variable or not.

Example

```
PROC procedure1 (INOUT num parameter1)
   IF IsVar(parameter1) THEN
      ! For this call reference to a variable
      ...
   ELSEIF IsPers(parameter1) THEN
      ! For this call reference to a persistent variable
      ...
   ELSE
      ! Should not happen
      EXIT;
   ENDIF
ENDPROC
```

The procedure *procedure1* will take different actions depending on whether the actual parameter *parameter1* is a variable or a persistent variable.

Return value

Data type: *bool*

TRUE if the tested actual INOUT parameter is a persistent variable.
FALSE if the tested actual INOUT parameter is not a persistent variable.

Arguments

IsPers (DatObj)

DatObj (*Data Object*) Data type: any type

The name of the formal INOUT parameter.

Syntax

```
IsPers'('
   [ DatObj ':=' ] < var or pers (INOUT) of any type > ')'
```

A function with a return value of the data type *bool*.

Related information

	Described in:
Test if variable	Function - *IsVar*
Types of parameters (access modes)	RAPID Characteristics - *Routines*

IsVar Is Variable

IsVar is used to test whether a data object is a variable or not.

Example

```
PROC procedure1 (INOUT num parameter1)
   IF IsVAR(parameter1) THEN
      ! For this call reference to a variable
      ...
   ELSEIF IsPers(parameter1) THEN
      ! For this call reference to a persistent variable
      ...
   ELSE
      ! Should not happen
      EXIT;
   ENDIF
ENDPROC
```

The procedure *procedure1* will take different actions, depending on whether the actual parameter *parameter1* is a variable or a persistent variable.

Return value

Data type: *bool*

TRUE if the tested actual INOUT parameter is a variable.
FALSE if the tested actual INOUT parameter is not a variable.

Arguments

IsVar (DatObj)

DatObj (*Data Object*) Data type: any type

The name of the formal INOUT parameter.

Syntax

```
IsVar'('
   [ DatObj ':=' ] < var or pers (INOUT) of any type > ')'
```

A function with a return value of the data type *bool*.

Related information

	Described in:
Test if persistent	Function - *IsPers*
Types of parameters (access modes)	RAPID Characteristics - *Routines*

MirPos Mirroring of a position

MirPos (Mirror Position) is used to mirror the translation and rotation parts of a position.

Example

```
CONST robtarget p1;
VAR robtarget p2;
PERS wobjdata mirror;
.
.
p2 := MirPos(p1, mirror);
```

p1 is a robtarget storing a position of the robot and an orientation of the tool. This position is mirrored in the xy-plane of the frame defined by *mirror*, relative to the world coordinate system. The result is new robtarget data, which is stored in *p2*.

Return value

Data type: *robtarget*

The new position which is the mirrored position of the input position.

Arguments

MirPos (Point MirPlane [\WObj] [\MirY])

Point Data type: *robtarget*

The input robot position. The orientation part of this position defines the current orientation of the tool coordinate system.

MirPlane *(Mirror Plane)* Data type: *wobjdata*

The work object data defining the mirror plane. The mirror plane is the xy-plane of the object frame defined in *MirPlane*. The location of the object frame is defined relative to the user frame, also defined in *MirPlane*, which in turn is defined relative to the world frame.

[**WObj**] (*Work Object)* Data type: *wobjdata*

The work object data defining the object frame, and user frame, relative to which the input position, *Point*, is defined. If this argument is left out, the position is defined relative to the World coordinate system.
Note. If the position is created with a work object active, this work object must be referred to in the argument.

[**MirY**] *(Mirror Y)* Data type: *switch*

If this switch is left out, which is the default rule, the tool frame will be mirrored as regards the x-axis and the z-axis. If the switch is specified, the tool frame will be mirrored as regards the y-axis and the z-axis.

Limitations

No recalculation is done of the robot configuration part of the input robtarget data.

Syntax

MirPos'('
[Point ':='] < expression (**IN**) of *robtarget*>','
[MirPlane ':='] <expression (**IN**) of *wobjdata*> ','
['\'WObj ':=' <expression (**IN**) of *wobjdata*>]
['\'MirY]')'

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

NumToStr Converts numeric value to string

NumToStr (Numeric To String) is used to convert a numeric value to a string.

Example

VAR string str;

str := NumToStr(0.38521,3);

The variable *str* is given the value "0.385".

reg1 := 0.38521

str := NumToStr(reg1, 2\Exp);

The variable *str* is given the value "3.85E-01".

Return value

Data type: *string*

The numeric value converted to a string with the specified number of decimals, with exponent if so requested. The numeric value is rounded if necessary. The decimal point is suppressed if no decimals are included.

Arguments

NumToStr (Val Dec [\Exp])

Val (*Value*) Data type: *num*

The numeric value to be converted.

Dec (*Decimals*) Data type: *num*

Number of decimals. The number of decimals must not be negative or greater than the available precision for numeric values.

[\Exp] (*Exponent*) Data type: *switch*

To use exponent.

Syntax

NumToStr'('
[Val ':='] <expression (**IN**) of *num*> ','
[Dec ':='] <expression (**IN**) of *num*>
[\Exp]
')'

A function with a return value of the data type *string*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

Offs Displaces a robot position

Offs is used to add an offset to a robot position.

Examples

MoveL Offs(p2, 0, 0, 10), v1000, z50, tool1;

The robot is moved to a point *10* mm from the position *p2* (in the z-direction).

p1 := Offs (p1, 5, 10, 15);

The robot position *p1* is displaced *5* mm in the x-direction, *10* mm in the y-direction and *15* mm in the z-direction.

Return value

Data type: *robtarget*

The displaced position data.

Arguments

Offs (Point XOffset YOffset ZOffset)

Point Data type: *robtarget*

The position data to be displaced.

XOffset Data type: *num*

The displacement in the x-direction.

YOffset Data type: *num*

The displacement in the y-direction.

ZOffset Data type: *num*

The displacement in the z-direction.

Example

```
PROC pallet (num row, num column, num distance, PERS tooldata tool,
             PERS wobjdata wobj)

  VAR robtarget palletpos:=[[0, 0, 0], [1, 0, 0, 0], [0, 0, 0, 0],
                            [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]];

  palettpos := Offs (palettpos, (row-1)*distance, (column-1)*distance, 0);
  MoveL palettpos, v100, fine, tool\WObj:=wobj;

ENDPROC
```

A routine for picking parts from a pallet is made. Each pallet is defined as a work object (see Figure 1). The part to be picked (row and column) and the distance between the parts are given as input parameters.
Incrementing the row and column index is performed outside the routine.

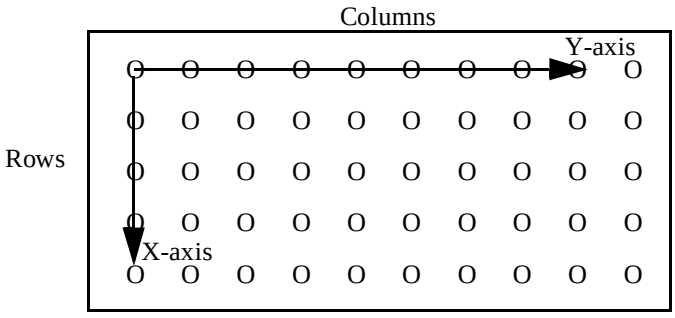


Figure 1 The position and orientation of the pallet is specified by defining a work object.

Syntax

```
Offs '('
   [Point ':='] <expression (IN) of robtarget> ','
   [XOffset ':='] <expression (IN) of num> ','
   [YOffset ':='] <expression (IN) of num> ','
   [ZOffset ':='] <expression (IN) of num> ')'
```

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Position data	Data Types - *robtarget*

OpMode Read the operating mode

OpMode (Operating Mode) is used to read the current operating mode of the system.

Example

```
TEST OpMode()
CASE OP_AUTO:
    ...
CASE OP_MAN_PROG:
    ...
CASE OP_MAN_TEST:
    ...
DEFAULT:
    ...
ENDTEST
```

Different program sections are executed depending on the current operating mode.

Return value

Data type: *symnum*

The current operating mode as defined in the table below.

Return value	Symbolic constant	Comment
0	OP_UNDEF	Undefined operating mode
1	OP_AUTO	Automatic operating mode
2	OP_MAN_PROG	Manual operating mode max. 250 mm/s
3	OP_MAN_TEST	Manual operating mode full speed, 100 %

Syntax

OpMode'(' ')'

A function with a return value of the data type *symnum*.

Related information

	Described in:
Different operating modes	User's Guide - *Starting up*
Reading running mode	Functions - *RunMode*

OrientZYX Builds an orient from Euler angles

OrientZYX (Orient from Euler ZYX angles) is used to build an orient type variable out of Euler angles.

Example

```
VAR num anglex;
VAR num angley;
VAR num anglez;
VAR pose object;
.
object.rot := OrientZYX(anglez, angley, anglex)
```

Return value

Data type: *orient*

The orientation made from the Euler angles.

The rotations will be performed in the following order:
-rotation around the z axis,
-rotation around the new y axis
-rotation around the new x axis.

Arguments

OrientZYX (ZAngle YAngle XAngle)

ZAngle Data type: *num*

The rotation, in degrees, around the Z axis.

YAngle Data type: *num*

The rotation, in degrees, around the Y axis.

XAngle Data type: *num*

The rotation, in degrees, around the X axis.

The rotations will be performed in the following order:
-rotation around the z axis,
-rotation around the new y axis
-rotation around the new x axis.

Syntax

OrientZYX’(’
[ZAngle ’:=’] <expression (**IN**) of *num*> ’,’
[YAngle ’:=’] <expression (**IN**) of *num*> ’,’
[XAngle ’:=’] <expression (**IN**) of *num*>
’)’

A function with a return value of the data type *orient*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

ORobT Removes a program displacement from a position

ORobT (Object Robot Target) is used to transform a robot position from the program displacement coordinate system to the object coordinate system and/or to remove an offset for the external axes.

Example

```
VAR robtarget p10;
VAR robtarget p11;

p10 := CRobT();
p11 := ORobT(p10);
```

The current positions of the robot and the external axes are stored in *p10* and *p11*. The values stored in *p10* are related to the ProgDisp/ExtOffs coordinate system. The values stored in *p11* are related to the object coordinate system without any offset on the external axes.

Return value

Data type: *robtarget*

The transformed position data.

Arguments

ORobT (OrgPoint [\InPDisp] | [\InEOffs])

OrgPoint *(Original Point)* Data type: *robtarget*

The original point to be transformed.

[\InPDisp] *(In Program Displacement)* Data type: *switch*

Returns the TCP position in the ProgDisp coordinate system, i.e. removes external axes offset only.

[\InEOffs] *(In External Offset)* Data type: *switch*

Returns the external axes in the offset coordinate system, i.e. removes program displacement for the robot only.

Examples

```
p10 := ORobT(p10 \InEOffs );
```

The ORobT function will remove any program displacement that is active, leaving the TCP position relative to the object coordinate system. The external axes will remain in the offset coordinate system.

```
p10 := ORobT(p10 \InPDisp );
```

The ORobT function will remove any offset of the external axes. The TCP position will remain in the ProgDisp coordinate system.

Syntax

```
ORobT '('
   [ OrgPoint ':=' ] < expression (IN) of robtarget>
   ['\'InPDisp] | ['\'InEOffs]')'
```

A function with a return value of the data type *robtarget*.

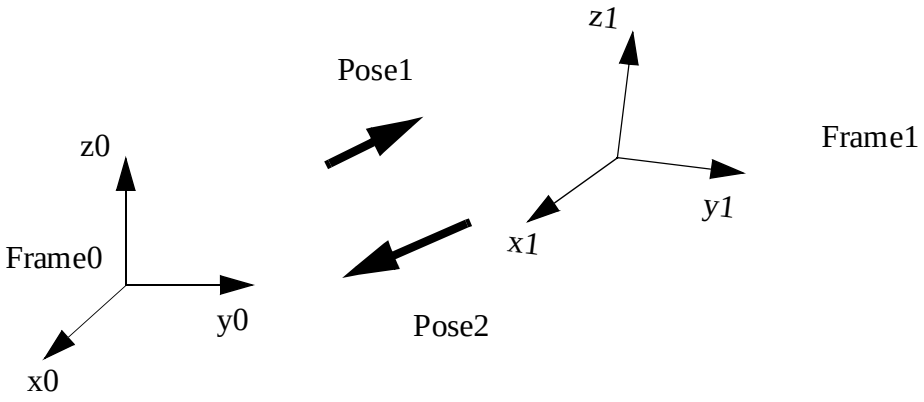
Related information

	Described in:
Definition of program displacement for the robot	Instructions - *PDispOn, PDispSet*
Definition of offset for external axes	Instructions - *EOffsOn, EOffsSet*
Coordinate systems	Motion and I/O Principles - *Coordinate Systems*

PoseInv Inverts the pose

PoseInv (Pose Invert) calculates the reverse transformation of a pose.

Example



Pose1 represents the coordinates of Frame1 related to Frame0.
The transformation giving the coordinates of Frame0 related to Frame1 is obtained by the reverse transformation:

```
VAR pose pose1;
VAR pose pose2;
.
.
pose2 := PoseInv(pose1);
```

Return value

Data type: *pose*

The value of the reverse pose.

Arguments

PoseInv (Pose)

Pose Data type: *pose*

The pose to invert.

Syntax

PoseInv'('
[Pose ':='] <expression (**IN**) of *pose*>
')'

A function with a return value of the data type *pose*.

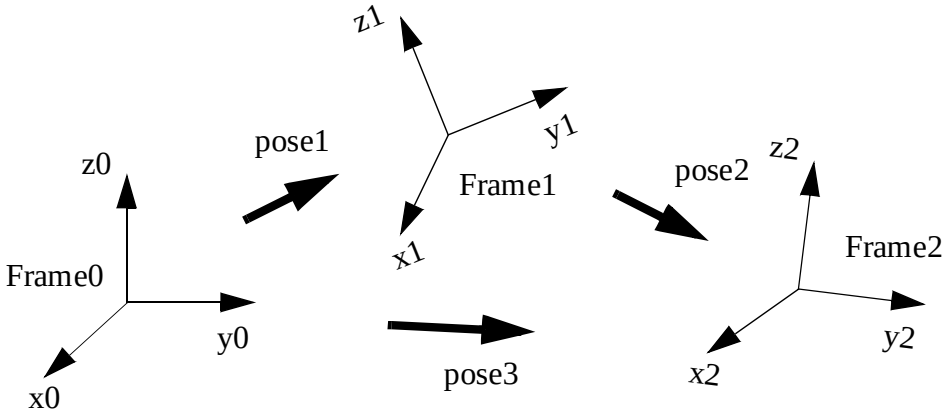
Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

PoseMult Multiplies pose data

PoseMult (Pose Multiply) is used to calculate the product of two frame transformations. A typical use is to calculate a new frame as the result of a displacement acting on an original frame.

Example



pose1 represents the coordinates of Frame1 related to Frame0.
pose2 represents the coordinates of Frame2 related to Frame1.

The transformation giving pose3, the coordinates of Frame2 related to Frame0, is obtained by the product of the two transformations:

```
VAR pose pose1;
VAR pose pose2;
VAR pose pose3;
.
.
pose3 := PoseMult(pose1, pose2);
```

Return value

Data type: *pose*

The value of the product of the two poses.

Arguments

PoseMult (Pose1 Pose2)

Pose1 Data type: *pose*

The first pose.

Pose2 Data type: *pose*

The second pose.

Syntax

PoseMult'('
[Pose1 ':='] <expression (**IN**) of *pose*> ','
[Pose2 ':='] <expression (**IN**) of *pose*>
')'

A function with a return value of the data type *pose*.

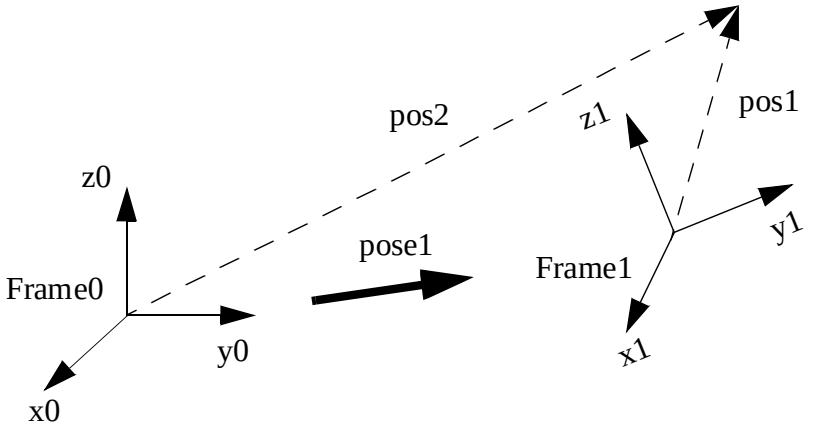
Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

PoseVect Applies a transformation to a vector

PoseVect (Pose Vector) is used to calculate the product of a pose and a vector.
It is typically used to calculate a vector as the result of the effect of a displacement on an original vector.

Example



pose1 represents the coordinates of Frame1 related to Frame0.
pos1 is a vector related to Frame1.

The corresponding vector related to Frame0 is obtained by the product:

```
VAR pose pose1;
VAR pos pos1;
VAR pos pos2;
.
.
pos2:= PoseVect(pose1, pos1);
```

Return value

Data type: *pos*

The value of the product of the pose and the original pos.

Arguments

PoseVect (Pose Pos)

Pose Data type: *pose*

The transformation to be applied.

Pos Data type: *pos*

The pos to be transformed.

Syntax

PoseVect'('
[Pose ':='] <expression (**IN**) of *pose*> ','
[Pos ':='] <expression (**IN**) of *pos*>
')'

A function with a return value of the data type *pos*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Pow Calculates the power of a value

Pow (Power) is used to calculate the exponential value in any base.

Example

```
VAR num x;
VAR num y
VAR num reg1;
.
reg1:= Pow(x, y);
```

reg1 is assigned the value x^y.

Return value

Data type: *num*

The value of the base x raised to the power of the exponent y (x^y).

Arguments

Pow (Base Exponent)

Base Data type: *num*

The base argument value.

Exponent Data type: *num*

The exponent argument value.

Limitations

The execution of the function x^y will give an error if:

. x < 0 and y is not an integer;
. x = 0 and y ≤ 0.

Syntax

```
Pow'('
   [Base ':='] <expression (IN) of num> ','
   [Exponent ':='] <expression (IN) of num>
   ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Present Tests if an optional parameter is used

Present is used to test if an optional argument has been used when calling a routine.

An optional parameter may not be used if it was not specified when calling the routine. This function can be used to test if a parameter has been specified, in order to prevent errors from occurring.

Example

```
PROC feeder (\switch on | \switch off)

  IF Present (on) Set do1;
  IF Present (off) Reset do1;

ENDPROC
```

The output *do1*, which controls a feeder, is set or reset depending on the argument used when calling the routine.

Return value

Data type: *bool*

TRUE = The parameter value or a switch has been defined when calling the routine.

FALSE = The parameter value or a switch has not been defined.

Arguments

Present (OptPar)

OptPar *(Optional Parameter)* Data type: Any type

The name of the optional parameter to be tested.

Example

```
PROC glue (\switch on, num glueflow, robtarget topoint, speeddata speed,
        zonedata zone, PERS tooldata tool, \PERS wobjdata wobj)

  IF Present (on) PulseDO glue_on;
  SetAO gluesignal, glueflow;
  IF Present (wobj) THEN
    MoveL topoint, speed, zone, tool \WObj=wobj;
  ELSE
    MoveL topoint, speed, zone, tool;
  ENDIF

ENDPROC
```

A glue routine is made. If the argument *\on* is specified when calling the routine, a pulse is generated on the signal *glue_on*. The robot then sets an analog output *gluesignal*, which controls the glue gun, and moves to the end position. As the wobj parameter is optional, different MoveL instructions are used depending on whether this argument is used or not.

Syntax

```
Present '('
  [OptPar':='] <reference (REF) of any type> ')'
```

A REF parameter requires, in this case, the optional parameter name.

A function with a return value of the data type *bool*.

Related information

	Described in:
Routine parameters	Basic Characteristics - *Routines*

ReadBin Reads from a binary serial channel

ReadBin (Read Binary) is used to read a byte (8 bits) from a binary serial channel.

Example

```
VAR iodev inchannel;
.
Open "sio1:", inchannel\Bin;
character := ReadBin(inchannel);
```

A byte is read from the binary channel *inchannel*.

Return value

Data type: *num*

A byte (8 bits) is read from a specified serial channel. This byte is converted to the corresponding positive numeric value. If the file is empty (end of file), the number -1 is returned.

Arguments

ReadBin (IODevice [\Time])

IODevice Data type: *iodev*

The name (reference) of the current serial channel.

[\Time] Data type: *num*

The max. time for the reading operation (timeout) in seconds. If this argument is not specified, the max. time is set to 60 seconds.

If this time runs out before the reading operation is finished, the error handler will be called with the error code ERR_DEV_MAXTIME. If there is no error handler, the execution will be stopped.

The timeout function is in use also during program stop and will be noticed in RAPID program at program start.

Program execution

Program execution waits until a byte (8 bits) can be read from the binary serial channel.

Limitations

The function can only be used for channels that have been opened for binary reading and writing.

Error handling

If an error occurs during reading, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Syntax

ReadBin'('
[IODevice ':='] <variable (**VAR**) of *iodev*>
['\'Time':=' <expression (**IN**) of *num*>]')'

A function with a return value of the type *num*.

Related information

	Described in:
Opening (etc.) serial channels	RAPID Summary - *Communication*

ReadMotor Reads the current motor angles

ReadMotor is used to read the current angles of the different motors of the robot and external axes. The primary use of this function is in the calibration procedure of the robot.

Example

```
VAR num motor_angle2;

motor_angle2 := ReadMotor(2);
```

The current motor angle of the second axis of the robot is stored in *motor_angle2*.

Return value

Data type: *num*

The current motor angle in radians of the stated axis of the robot or external axes.

Arguments

ReadMotor [\MecUnit] Axis

MecUnit *(Mechanical Unit)* Data type: *mecunit*

The name of the mechanical unit for which an axis is to be read. If this argument is omitted, the axis for the robot is read. (Note, in this release only robot is permitted for this argument).

Axis Data type: *num*

The number of the axis to be read (1 - 6).

Program execution

The motor angle returned represents the current position in radians for the motor and independently of any calibration offset. The value is not related to a fix position of the robot, only to the resolver internal zero position, i.e. normally the resolver zero position closest to the calibration position (the difference between the resolver zero position and the calibration position is the calibration offset value). The value represents the full movement of each axis, although this may be several turns.

Example

```
VAR num motor_angle3;

motor_angle3 := ReadMotor(\MecUnit:=robot, 3);
```

The current motor angle of the third axis of the robot is stored in *motor_angle3*.

Syntax

```
ReadMotor'('
   ['\'MecUnit ':=' < variable (VAR) of mecunit>',']
   [Axis ':=' ] < expression (IN) of num>
   ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Reading the current joint angle	Functions - *CJointT*

ReadNum Reads a number from a file or the serial channel

ReadNum (Read Numeric) is used to read a number from a character-based file or the serial channel.

Example

```
VAR iodev infile;
.
Open "flp1:file.doc", infile\Read;
reg1 := ReadNum(infile);
```

Reg1 is assigned a number read from the file *file.doc* on the diskette.

Return value

Data type: *num*

The numeric value read from a specified file. If the file is empty (end of file), the number 9.999E36 is returned.

Arguments

ReadNum (IODevice [\Time])

IODevice Data type: *iodev*

The name (reference) of the file to be read.

[\Time] Data type: *num*

The max. time for the reading operation (timeout) in seconds. If this argument is not specified, the max. time is set to 60 seconds.

If this time runs out before the read operation is finished, the error handler will be called with the error code ERR_DEV_MAXTIME. If there is no error handler, the execution will be stopped.

The timeout function is also in use during program stop and will be noticed in RAPID program at program start.

Program execution

The function reads a line from a file, i.e. reads everything up to and including the next line-feed character (LF), but not more than 80 characters. If the line exceeds 80 characters, the remainder of the characters will be read on the next reading.

The string that is read is then converted to a numeric value; e.g. “234.4” is converted to the numeric value 234.4. If all the characters read are not digits, 0 is returned.

Example

```
reg1 := ReadNum(infile);
IF reg1 > EOF_NUM THEN
   TPWrite "The file is empty"
   ..
```

Before using the number read from the file, a check is performed to make sure that the file is not empty.

Limitations

The function can only be used for files that have been opened for reading.

Error handling

If an access error occurs during reading, the system variable ERRNO is set to ERR_FILEACC. If there is an attempt to read non numeric data, the system variable ERRNO is set to ERR_RCVDATA. These errors can then be dealt with by the error handler.

Predefined data

The constant *EOF_NUM* can be used to stop reading, at the end of the file.

```
CONST num EOF_NUM := 9.998E36;
```

Syntax

```
ReadNum ’(’
   [IODevice ’:=’] <variable (VAR) of iodev>
   [’\’Time’:=’ <expression (IN) of num>]’)’
```

A function with a return value of the type *num*.

Related information

	Described in:
Opening (etc.) serial channels	RAPID Summary - *Communication*

ReadStr Reads a string from a file or serial channel

ReadStr (Read String) is used to read text from a character-based file or from the serial channel.

Example

```
VAR iodev infile;
.
Open "flp1:file.doc", infile\Read;
text := ReadStr(infile);
```

Text is assigned a text string read from the file *file.doc* on the diskette.

Return value

Data type: *string*

The text string read from the specified file. If the file is empty (end of file), the string "EOF" is returned.

Arguments

ReadStr (IODevice [\Time])

IODevice Data type: *iodev*

The name (reference) of the file to be read.

[\Time] Data type: *num*

The max. time for the reading operation (timeout) in seconds. If this argument is not specified, the max. time is set to 60 seconds.

If this time runs out before the read operation is finished, the error handler will be called with the error code ERR_DEV_MAXTIME. If there is no error handler, the execution will be stopped.

Program execution

The function reads a line from a file, i.e. reads everything up to and including the next line-feed character (LF), but not more than 80 characters. If the line exceeds 80 characters, the remainder of the characters will be read on the next reading.

Example

```
text := ReadStr(infile);
IF text = EOF THEN
   TPWrite "The file is empty";
   .
```

Before using the string read from the file, a check is performed to make sure that the file is not empty.

Limitations

The function can only be used for files that have been opened for reading.

Error handling

If an error occurs during reading, the system variable ERRNO is set to ERR_FILEACC. This error can then be handled in the error handler.

Predefined data

The constant *EOF* can be used to check if the file was empty when trying to read from the file or to stop reading at the end of the file.

```
CONST string EOF := "EOF";
```

Syntax

```
ReadStr '('
   [IODevice ':='] <variable (VAR) of iodev>
   ['\'Time':=' <expression (IN) of num>]')'
```

A function with a return value of the type *string*.

Related information

	Described in:
Opening (etc.) serial channels	RAPID Summary - *Communication*

RelTool Make a displacement relative to the tool

RelTool (Relative Tool) is used to add a displacement and/or a rotation, expressed in the tool coordinate system, to a robot position.

Example

```
MoveL RelTool (p1, 0, 0, 100), v100, fine, tool1;
```

The robot is moved to a position that is 100 mm from p1 in the direction of the tool.

```
MoveL RelTool (p1, 0, 0, 0 \Rz:= 25), v100, fine, tool1;
```

The tool is rotated 25° around its z-axis.

Return value

Data type: *robtarget*

The new position with the addition of a displacement and/or a rotation, if any, relative to the active tool.

Arguments

RelTool (Point Dx Dy Dz [\Rx] [\Ry] [\Rz])

Point Data type: *robtarget*

The input robot position. The orientation part of this position defines the current orientation of the tool coordinate system.

Dx Data type: *num*

The displacement in mm in the x direction of the tool coordinate system.

Dy Data type: *num*

The displacement in mm in the y direction of the tool coordinate system.

Dz Data type: *num*

The displacement in mm in the z direction of the tool coordinate system.

[\Rx] Data type: *num*

The rotation in degrees around the x axis of the tool coordinate system.

[\Ry] Data type: *num*

The rotation in degrees around the y axis of the tool coordinate system.

[\Rz] Data type: *num*

The rotation in degrees around the z axis of the tool coordinate system.

In the event that two or three rotations are specified at the same time, these will be performed first around the x-axis, then around the new y-axis, and then around the new z-axis.

Syntax

RelTool'('
[Point ':='] < expression (**IN**) of *robtarget*>','
[Dx ':='] <expression (**IN**) of *num*> ','
[Dy ':='] <expression (**IN**) of *num*> ','
[Dz ':='] <expression (**IN**) of *num*>
['\'Rx ':=' <expression (**IN**) of *num*>]
['\'Ry ':=' <expression (**IN**) of *num*>]
['\'Rz ':=' <expression (**IN**) of *num*>]')'

A function with a return value of the data type *robtarget*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Positioning instructions	RAPID Summary - *Motion*

Round Round is a numeric value

Round is used to round a numeric value to a specified number of decimals or to an integer value.

Example

VAR num val;

val := Round(0.38521\Dec:=3);

The variable *val* is given the value 0.385.

val := Round(0.38521\Dec:=1);

The variable *val* is given the value 0.4.

val := Round(0.38521);

The variable *val* is given the value 0.

Return value

Data type: *num*

The numeric value rounded to the specified number of decimals.

Arguments

Round (Val [\Dec])

Val (*Value*) Data type: *num*

The numeric value to be rounded.

[\Dec] (*Decimals* Data type: *num*

Number of decimals.

If the specified number of decimals is 0 or if the argument is omitted, the value is rounded to an integer.

The number of decimals must not be negative or greater than the available precision for numeric values.

Syntax

Round’(’
[Val ’:=’] <expression (**IN**) of *num*>
[\Dec ’:=’ <expression (**IN**) of *num*>]
’)’

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Truncating a value	Functions - *Trunc*

RunMode Read the running mode

RunMode (Running Mode) is used to read the current running mode of the system.

Example

```
IF RunMode() = RUN_CONT_CYCLE THEN
.
.
ENDIF
```

The program section is executed only for continuous or cycle running.

Return value

Data type: *symnum*

The current running mode as defined in the table below.

Return value	Symbolic constant	Comment
0	RUN_UNDEF	Undefined running mode
1	RUN_CONT_CYCLE	Continuous or cycle running mode
2	RUN_INSTR_FWD	Instruction forward running mode
3	RUN_INSTR_BWD	Instruction backward running mode
4	RUN_SIM	Simulated running mode

Syntax

RunMode’(’ ’)’

A function with a return value of the data type *symnum*.

Related information

	Described in:
Reading operating mode	Functions - *OpMode*

Sin Calculates the sine value

Sin (Sine) is used to calculate the sine value from an angle value.

Example

```
VAR num angle;
VAR num value;
.
.
value := Sin(angle);
```

Return value

Data type: *num*

The sine value, range [-1, 1] .

Arguments

Sin (Angle)

Angle Data type: *num*

The angle value, expressed in degrees.

Syntax

```
Sin'('
   [Angle':='] <expression (IN) of num>
   ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

Sqrt Calculates the square root value

Sqrt (Square root) is used to calculate the square root value.

Example

```
VAR num x_value;
VAR num y_value;
.
.
y_value := Sqrt( x_value);
```

Return value

Data type: *num*

The square root value.

Arguments

Sqrt (Value)

Value Data type: *num*

The argument value for square root ($\sqrt{\ }$); it has to be ≥ 0.

Syntax

```
Sqrt'('
   [Value':='] <expression (IN) of num>
   ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*

StrFind Searches for a character in a string

StrFind (String Find) is used to search in a string, starting at a specified position, for a character that belongs to a specified set of characters.

Example

```
VAR num found;

found := StrFind("Robotics",1,"aeiou");
```

The variable *found* is given the value 2.

```
found := StrFind("Robotics",1,"aeiou"\NotInSet);
```

The variable *found* is given the value 1.

```
found := StrFind("IRB 6400",1,STR_DIGIT);
```

The variable *found* is given the value 5.

```
found := StrFind("IRB 6400",1,STR_WHITE);
```

The variable *found* is given the value 4.

Return value

Data type: *num*

The character position of the first character, at or past the specified position, that belongs to the specified set. If no such character is found, String length +1 is returned.

Arguments

StrFind (Str ChPos Set [\NotInSet])

Str (*String*) Data type: *string*

The string to search in.

ChPos (*Character Position*) Data type: *num*

Start character position. A runtime error is generated if the position is outside the string.

Set Data type: *string*

Set of characters to test against.

[\NotInSet] Data type: *switch*

Search for a character not in the set of characters.

Syntax

StrFind'('
[Str ':='] <expression (**IN**) of *string*> ','
[ChPos ':='] <expression (**IN**) of *num*> ','
[Set':='] <expression (**IN**) of *string*>
[\NotInSet]
')'

A function with a return value of the data type *num*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrLen Gets the string length

StrLen (String Length) is used to find the current length of a string.

Example

```
VAR num len;

len := StrLen("Robotics");
```

The variable *len* is given the value 8.

Return value

Data type: *num*

The number of characters in the string (>=0).

Arguments

StrLen (Str)

Str (*String*) Data type: *string*

The string in which the number of characters is to be counted.

Syntax

```
StrLen'('
   [ Str ':=' ] <expression (IN) of string>
   ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrMap Maps a string

StrMap (String Mapping) is used to create a copy of a string in which all characters are translated according to a specified mapping.

Example

```
VAR string str;

str := StrMap("Robotics","aeiou","AEIOU");
```

The variable *str* is given the value "RObOtIcs".

```
str := StrMap("Robotics",STR_LOWER, STR_UPPER);
```

The variable *str* is given the value "ROBOTICS".

Return value

Data type: *string*

The string created by translating the characters in the specified string, as specified by the "from" and "to" strings. Each character, from the specified string, that is found in the "from" string is replaced by the character at the corresponding position in the "to" string. Characters for which no mapping is defined are copied unchanged to the resulting string.

Arguments

StrMap (Str FromMap ToMap)

Str (*String*) Data type: *string*

The string to translate.

FromMap Data type: *string*

Index part of mapping.

ToMap Data type: *string*

Value part of mapping.

Syntax

StrMap’(’
[Str ’:=’] <expression (**IN**) of *string*> ’,’
[FromMap’:=’] <expression (**IN**) of *string*> ’,’
[ToMap’:=’] <expression (**IN**) of *string*>
’)’

A function with a return value of the data type *string*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrMatch Search for pattern in string

StrMatch (String Match) is used to search in a string, starting at a specified position, for a specified pattern.

Example

```
VAR num found;

found := StrMatch("Robotics",1,"bo");
```

The variable *found* is given the value 3.

Return value

Data type: *num*

The character position of the first substring, at or past the specified position, that is equal to the specified pattern string. If no such substring is found, string length +1 is returned.

Arguments

StrMatch (Str ChPos Pattern)

Str (*String*) Data type: *string*

The string to search in.

ChPos (*Character Position*) Data type: *num*

Start character position. A runtime error is generated if the position is outside the string.

Pattern Data type: *string*

Pattern string to search for.

Syntax

```
StrMatch'('
    [ Str ':=' ] <expression (IN) of string> ','
    [ ChPos ':=' ] <expression (IN) of num> ','
    [ Pattern':=' ] <expression (IN) of string>
    ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrMemb Checks if a character belongs to a set

StrMemb (String Member) is used to check whether a specified character in a string belongs to a specified set of characters.

Example

```
VAR bool memb;

memb := StrMemb("Robotics",2,"aeiou");
```

The variable *memb* is given the value TRUE, as o is a member of the set "aeiou".

```
memb := StrMemb("Robotics",3,"aeiou");
```

The variable *memb* is given the value FALSE, as b is not a member of the set "aeiou".

```
memb := StrMemb("S-721 68  VÄSTERÅS",3,STR_DIGIT);
```

The variable *memb* is given the value TRUE.

Return value

Data type: *bool*

TRUE if the character at the specified position in the specified string belongs to the specified set of characters.

Arguments

StrMemb (Str ChPos Set)

Str (*String*) Data type: *string*

The string to check in.

ChPos (*Character Position*) Data type: *num*

The character position to check. A runtime error is generated if the position is outside the string.

Set Data type: *string*

Set of characters to test against.

Syntax

StrMemb’(’
[Str ’:=’] <expression (**IN**) of *string*> ’,’
[ChPos ’:=’] <expression (**IN**) of *num*> ’,’
[Set’:=’] <expression (**IN**) of *string*>
’)’

A function with a return value of the data type *bool*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrOrder Checks if strings are ordered

StrOrder (String Order) is used to check whether two strings are in order, according to a specified character ordering sequence.

Example

```
VAR bool le;

le := StrOrder("FIRST","SECOND",STR_UPPER);
```

The variable *le* is given the value TRUE, because "FIRST" comes before "SECOND" in the character ordering sequence STR_UPPER.

Return value

Data type: *bool*

TRUE if the first string comes before the second string (Str1 <= Str2) when characters are ordered as specified.

Characters that are not included in the defined ordering are all assumed to follow the present ones.

Arguments

StrOrder (Str1 Str2 Order)

Str1 (*String 1*) Data type: *string*

First string value.

Str2 (*String 2*) Data type: *string*

Second string value.

Order Data type: *string*

Sequence of characters that define the ordering.

Syntax

```
StrOrder'('
    [ Str1 ':=' ] <expression (IN) of string> ','
    [ Str2 ':=' ] <expression (IN) of string> ','
    [ Order ':=' ] <expression (IN) of string>
    ')'
```

A function with a return value of the data type *bool*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrPart Finds a part of a string

StrPart (String Part) is used to find a part of a string, as a new string.

Example

```
VAR string part;

part := StrPart("Robotics",1,5);
```

The variable *part* is given the value "Robot".

Return value

Data type: *string*

The substring of the specified string, which has the specified length and starts at the specified character position.

Arguments

StrPart (Str ChPos Len)

Str (*String*) Data type: *string*

The string in which a part is to be found.

ChPos (*Character Position*) Data type: *num*

Start character position. A runtime error is generated if the position is outside the string.

Len (*Length*) Data type: *num*

Length of string part. A runtime error is generated if the length is negative or greater than the length of the string, or if the substring is (partially) outside the string.

Syntax

```
StrPart'('
   [ Str ':=' ] <expression (IN) of string> ','
   [ ChPos ':=' ] <expression (IN) of num> ','
   [ Len':=' ] <expression (IN) of num>
   ')'
```

A function with a return value of the data type *string*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

StrToVal Converts a string to a value

StrToVal (String To Value) is used to convert a string to a value of any data type.

Example

```
VAR bool ok;
VAR num nval;

ok := StrToVal("3.85",nval);
```

The variable *ok* is given the value TRUE and *nval* is given the value *3.85*.

Return value

Data type: *bool*

TRUE if the requested conversion succeeded, FALSE otherwise.

Arguments

StrToVal (Str Val)

Str (*String*) Data type: *string*

A string value containing literal data with format corresponding to the data type used in argument *Val*. Valid format as for RAPID literal aggregates.

Val (*Value*) Data type: *ANYTYPE*

Name of the variable or persistent of any data type for storage of the result from the conversion. The data is unchanged if the requested conversion failed.

Example

```
VAR string 15 := “[600, 500, 225.3]”;
VAR bool ok;
VAR pos pos15;

ok := StrToVal(str15,pos15);
```

The variable *ok* is given the value TRUE and the variable *p15* is given the value that are specified in the string *str15*.

Syntax

StrToVal’(’
[Str ’:=’] <expression (**IN**) of *string*> ’,’
[Val ’:=’] <var or pers (**INOUT**) of *ANYTYPE*>
’)’

A function with a return value of the data type *bool*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

Tan Calculates the tangent value

Tan (Tangent) is used to calculate the tangent value from an angle value.

Example

```
VAR num angle;
VAR num value;
.
.
value := Tan(angle);
```

Return value

Data type: *num*

The tangent value.

Arguments

Tan (Angle)

Angle Data type: *num*

The angle value, expressed in degrees.

Syntax

```
Tan'('
   [Angle ':='] <expression (IN) of num>
   ')'
```

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Arc tangent with return value in the range [-180, 180]	Functions - *ATan2*

TestDI Tests if a digital input is set

TestDI is used to test whether a digital input is set.

Examples

IF TestDI (di2) THEN . . .

If the current value of the signal *di2* is equal to 1, then . . .

IF NOT TestDI (di2) THEN . . .

If the current value of the signal *di2* is equal to 0, then . . .

WaitUntil TestDI(di1) AND TestDI(di2);

Program execution continues only after both the *di1* input and the *di2* input have been set.

Return value

Data type: *bool*

TRUE = The current value of the signal is equal to 1.

FALSE = The current value of the signal is equal to 0.

Arguments

TestDI (Signal)

Signal Data type: *signaldi*

The name of the signal to be tested.

Syntax

TestDI '('
[Signal ':='] < variable (**VAR**) of *signaldi* > ')'

A function with a return value of the data type *bool*.

Related information

	Described in:
Reading the value of a digital input signal	Functions - *DInput*
Input/Output instructions	RAPID Summary - *Input and Output Signals*

Trunc Truncates a numeric value

Trunc (Truncate) is used to truncate a numeric value to a specified number of decimals or to an integer value.

Example

VAR num val;

val := Trunc(0.38521\Dec:=3);

The variable *val* is given the value 0.385.

reg1 := 0.38521

val := Trunc(reg1\Dec:=1);

The variable *val* is given the value 0.3.

val := Trunc(0.38521);

The variable *val* is given the value 0.

Return value

Data type: *num*

The numeric value truncated to the specified number of decimals.

Arguments

Trunc (Val [\Dec])

Val (*Value*) Data type: *num*

The numeric value to be truncated.

[\Dec] (*Decimals*) Data type: *num*

Number of decimals.

If the specified number of decimals is 0 or if the argument is omitted, the value is truncated to an integer.

The number of decimals must not be negative or greater than the available precision for numeric values.

Syntax

Trunc'('
[Val ':='] <expression (**IN**) of *num*>
[\Dec ':=' <expression (**IN**) of *num*>]
')'

A function with a return value of the data type *num*.

Related information

	Described in:
Mathematical instructions and functions	RAPID Summary - *Mathematics*
Rounding a value	Functions - *Round*

ValToStr Converts a value to a string

ValToStr (Value To String) is used to convert a value of any data type to a string.

Example

```
VAR string str;
VAR pos p := [100,200,300];

str := ValToStr(1.234567);
```

The variable *str* is given the value "1.23457".

```
str := ValToStr(TRUE);
```

The variable *str* is given the value "TRUE".

```
str := ValToStr(p);
```

The variable *str* is given the value "[100,200,300]".

Return value

Data type: *string*

The value converted to a string with standard RAPID format. This means in principle that decimal numbers are rounded off to 5 places of decimals, but not with more than 6 digits.

A runtime error is generated if the resulting string is too long.

Arguments

ValToStr (Val)

Val (*Value*) Data type: *ANYTYPE*

A value of any data type.

Syntax

```
ValToStr'('
   [ Val ':=' ] <expression (IN) of ANYTYPE>
   ')'
```

A function with a return value of the data type *string*.

Related information

	Described in:
String functions	RAPID Summary - *String Functions*
Definition of string	Data Types - *string*
String values	Basic Characteristics - *Basic Elements*

CONTENTS

Page

1 System Module *User*

In order to facilitate programming, predefined data is supplied with the robot. This data does not have to be created and, consequently, can be used directly.

If this data is used, initial programming is made easier. It is, however, usually better to give your own names to the data you use, since this makes the program easier for you to read.

1.1 Contents

User comprises five numerical data (registers), one work object data, one clock and two symbolic values for digital signals.

Name	Data type	Declaration
reg1	num	VAR num reg1:=0
reg2	.	.
reg3	.	.
reg4	.	.
reg5	num	VAR num reg5:=0
wobj1	wobjdata	PERS wobjdata wobj1:=wobj0
clock1	clock	VAR clock clock1
high	dionum	CONST dionum high:=1
low	dionum	CONST dionum low:=0

User is a system module, which means that it is always present in the memory of the robot regardless of which program is loaded.

1.2 Creating new data in this module

This module can be used to create such data and routines that must always be present in the program memory regardless of which program is loaded, e.g. tools and service routines.

- Choose **View: Modules** from the Program window.
- Select the system module *User* and press Enter [↵].
- Change, create data and routines in the normal way (see *Programming and Testing).*

1.3 Deleting this data

Warning: If the Module is deleted, the CallByVar instruction will not work.

To delete all data (i.e. the entire module)

- Choose **View: Modules** from the Program window.
- Select the module *User*.
- Press Delete ⌫.

To change or delete individual data

- Choose **View: Data** from the Program window.
- Choose **Data: In All Modules**.
- Select the desired data. If this is not shown, press the ***Types*** function key to select the correct data type.
- Change or delete in the normal way (see *Programming and Testing).*

CONTENTS

1 Programming Off-line

RAPID programs can easily be created, maintained and stored in an ordinary office computer. All information can be read and changed directly using a normal text editor. This chapter explains the working procedure for doing this. In addition to off-line programming, you can use the computer tool, QuickTeach.

1.1 File format

The robot stores and reads RAPID programs in TXT format (ASCII) and can handle both DOS and UNIX text formats. If you use a word-processor to edit programs, these must be saved in TXT format (ASCII) before they are used in the robot.

1.2 Editing

When a program is created or changed in a word-processor, all information will be handled in the form of text. This means that information about data and routines will differ somewhat from what is displayed on the teach pendant.

Note that the value of a stored position is only displayed as an *on the teach pendant, whereas the text file will contain the actual position value (x, y, z, etc.).

In order to minimise the risk of errors in the syntax (faulty programs), you should use a template. A template can take the form of a program that was created previously on the robot or using QuickTeach. These programs can be read directly to a word-processor without having to be converted.

1.3 Syntax check

Programs must be syntactically correct before they are loaded into the robot. By this is meant that the text must follow the fundamental rules of the RAPID language. One of the following methods should be used to detect errors in the text:

- Save the file on diskette and try to open it in the robot. If there are any syntactical errors, the program will not be accepted and an error message will be displayed. To obtain information about the type of error, the robot stores a log called PGMCPL1.LOG on the internal RAM disk. Copy this log to a diskette using the robot's File Manager. Open the log in a word-processor and you will be able to read which lines were incorrect and receive a description of the error.
- Open the file in QuickTeach or ProgramMaker.
- Use a RAPID syntax check program for the PC.

When the program is syntactically correct, it can be checked and edited in the robot.

To make sure that all references to routines and data are correct, use the command **File: Check Program**. If the program has been changed in the robot, it can be stored on diskette again and processed or stored in a PC.

1.4 Examples

The following shows examples of what routines look like in text format.

```
%%%
    VERSION: 1
     LANGUAGE: ENGLISH
%%%
MODULE main
VAR intnum process_int ;
! Demo of RAPID program
PROC main()
    MoveL p1, v200, fine, gun1;
ENDPROC


TRAP InvertDo12
! Trap routine for TriggInt
    TEST INTNO
       CASE process_int:
       InvertDO do12;
       DEFAULT:
       TPWrite “Unknown trap , number=”\Num:=INTNO;
    ENDTEST
ENDTRAP

LOCAL FUNC num MaxNum(num t1, num t2)
    IF t1 > t2 THEN
       RETURN t1;
    ELSE
       RETURN t2;
    ENDIF
ENDFUNC
ENDMODULE
```

1.5 Making your own instructions

In order to make programming easier, you can customize your own instructions. These are created in the form of normal routines, but, when programming and test-running, function as instructions:

- They can be taken from the instruction pick list and programmed as normal instructions.
- The complete routine will be run during step-by-step execution.

- Create a new system module where you can place your routines that will function as instructions. Alternatively, you can place them in the USER system module.
- Create a routine in this system module with the name that you want your new instruction to be called. The arguments of the instruction are defined in the form of routine parameters. Note that the name of the parameters will be displayed in the window during programming and should therefore be given names that the user will understand.
- Place the routine in one of the Most Common pick lists.
- If the instruction is to behave in a certain way during backward program execution, this can be done in the form of a backward handler. If there is no such handler, it will not be possible to get past the instruction during backward program execution (see Chapter 13 in this manual - Basic Characteristics). A backward handler can be entered using the command **Routine: Add Backward Handler** from the *Program Routines* window.
- Test the routine thoroughly so that it works with different types of input data (arguments).
- Change the module attribute to NOSTEPIN. The complete routine will then be run during step-by-step execution. This attribute, however, must be entered off-line.

Example: To make the gripper easier to handle, two new instructions are made, *GripOpen* and *GripClose*. The output signal's name is given to the instruction's argument, e.g. *GripOpen gripper1*.

```
MODULE My_instr (SYSMODULE, NOSTEPIN)
PROC GripOpen (VAR signaldo Gripper)
    Set Gripper;
    WaitTime 0.2;
ENDPROC
PROC GripClose (VAR signaldo Gripper)
    Reset Gripper;
    WaitTime 0.2;
ENDPROC
ENDMODULE
```

seamdata Seam data

Seamdata is used to control the start and end of the weld. *Seamdata* is also used if the process is restarted after a welding operation has been interrupted.

The actual weld phase is controlled using *welddata*.

Description

Seam data describes data, the values of which, as a rule, can be maintained unaltered when welding a complete run and often also when welding several seams. Seam data is used when preparing for the welding operation, when igniting the arc, when heating after the ignition and also when ending the weld.

Seam data is included in all arc welding instructions to facilitate controlled end and start phases irrespective of where the interrupts or restarts occur.

Note. Some of the components of seam data depend on the configuration of the robot. If a given feature is omitted, the corresponding component is left out of the seam data. The conditions that must be met for components to exist are described in the system parameters.

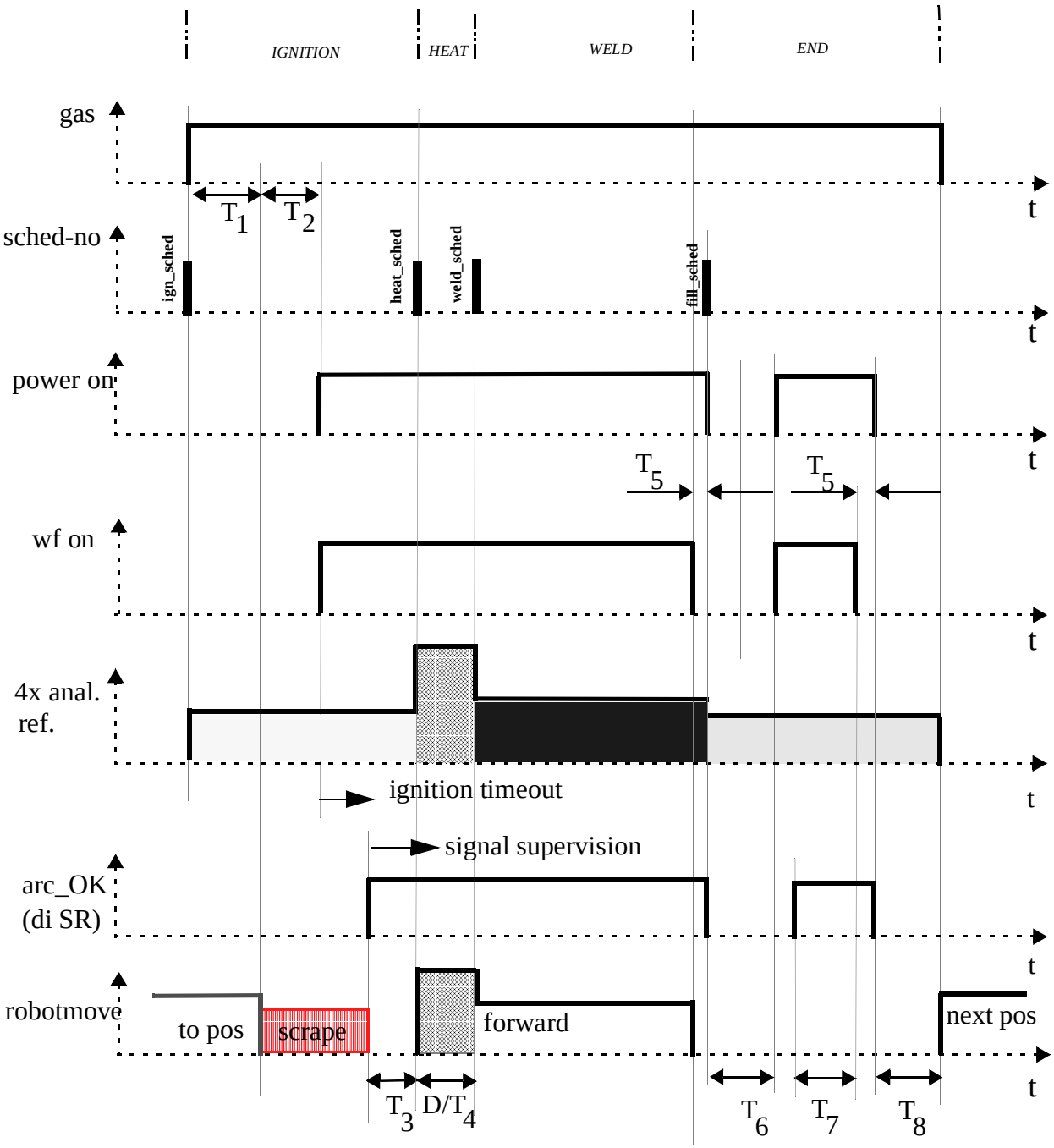
All voltages can be expressed in two ways (determined by the welding equipment):

- As absolute values (only positive values are used in this case).
- As corrections of values set in the process equipment (both positive and negative values are used in this case).

Feeding the weld electrode in this section refers to MIG/MAG welding. In the case of TIG welding:

- A *cold* wire is supplied to the *wire feed*.
- The necessary *welding current* reference value can be connected to any of the three analog outputs that are not used. (The *Welding voltage* reference is not used.)

The welding sequence



T1: max. gas_purge/arc_preset time
T2: gas_preflow time
T3: ignition_movement_delay time
D/T4: heating distance/time
T5: burnback time
T6: max cooling/arc_preset time
T7: filling time
T8: max cooling/gas_postflow time

Components

Component group: Ignition

purge_time Data type: *num*

The time (in seconds) it takes to fill gas lines and the welding gun with protective gas, so-called “gas purging”.

If the first weld instruction contains the argument *On* (flying start), the gas flow is activated at the specified gas purge time before the programmed position is reached.

If the positioning time to the start position of the weld is shorter than the gas purge time, or if the *On* argument is not used, the robot waits in the weld start position until the gas purge time has expired.

preflow_time Data type: *num*

The time (in seconds) it takes to preflow the weld object with protective gas, so-called “gas preflowing”.

The robot is stationary in position during this time before the arc is ignited.

ign_sched *(ignition schedule)* Data type: *num*

The identity (expressed as a number) of a weld program in connected welding equipment. It is sent to the welding equipment to be used during ignition of the arc.

See System Parameter *Arc Welding - Equipment - schedport_type.*

ign_voltage Data type: *num*

The welding voltage (in volts) during ignition of the arc.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

ign_wirefeed Data type: *num*

The feed speed of the weld electrode during ignition of the arc.

The unit is defined in the system parameter *Arc Welding - Units - unit_feed* and, as a rule, is m/minute or inches per minute.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

ign_volt_adj *(ignition voltage adjustment)* Data type: *num*

The welding voltage adjustment during ignition of the arc.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

ign_curr_adj *(ignition current adjustment)* Data type: *num*

The current adjustment during ignition of the arc.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

ign_move_delay *(ignition movement delay)* Data type: *num*

The delay (in seconds) from the time the arc is considered stable at ignition until the heating phase is started. The ignition references remain valid during the ignition movement delay.

scrape_start *(scrape start type)* Data type: *num*

Type of scrape at weld start. Scrape type at restart will not be affected (it will always be Weaving scrape).

Scrape types:
0 No scrape. No scrape will occur at weld start.
1 Weaving scrape.
2 Fast scrape. The robot does not wait for the arc OK signal at the start point. However, the ignition is considered incorrect if the ignition timeout has been exceeded.

Component group: Heat

heat_speed Data type: *num*

The welding speed during heating at the start of the weld phase.

The unit is defined in the system parameter *Arc Welding - Units- velocity_unit* and, as a rule, is mm/s or inches per minute.

heat_time Data type: *num*

The heating time (in seconds) at the start of the weld phase.

Heat_time is only used during timed positioning and when *heat_distance* or *heat_speed* equal zero.

heat_distance Data type: *num*

The distance along which heat data must be active at the start of the weld.

The unit is defined in *System Parameters - Arc Welding - Units - length_unit* and as a rule, is mm or inches.

heat_sched *(heating schedule)* Data type: *num*

The identity (expressed as a number) of a weld program in connected welding equipment. It is sent to the welding equipment when the arc has been ignited and is used during heating.

See System Parameter *Arc Welding - Equipment - schedport_type.*

heat_voltage Data type: *num*

The welding voltage (in volts) during heating.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

heat_wirefeed Data type: *num*

The feed speed of the weld electrode during heating.

The unit is defined in the system parameter *Arc Welding - Units - unit_feed* and, as a rule, is m/minute or inches per minute.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

heat_volt_adj *(heating voltage adjustment)* Data type: *num*

The voltage adjustment for heating.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

heat_curr_adj *(heating current adjustment)* Data type: *num*

The current adjustment during heating.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

Component group: End

cool_time *(cooling time)* Data type: *num*

The time (in seconds) during which the process is closed before other terminating activities (filling) take place.

fill_time Data type: *num*

The crater-filling time (in seconds) at the end phase of the weld.

bback_time *(burnback time)* Data type: *num*

The time (in seconds) during which the weld electrode is burnt back when electrode feeding has stopped. This to prevent the electrode getting stuck to the hardening weld when a MIG/MAG process is switched off.

Burnback time is used twice in the end phase; first when the weld phase is being finished, the second time after crater-filling.

rback_time *(rollback time)* Data type: *num*

The time (in seconds) during which a cold wire is rolled back after the power source has been switched off. This to prevent the wire getting stuck to the hardening weld when a TIG process is switched off.

The functions *burnback* and *rollback* are mutually exclusive.

postflow_time Data type: *num*

The time (in seconds) required for purging with protective gas after the end of a process. The purpose of gas postflow is to prevent the weld electrode and the seam from oxidizing during cooling.

fill_sched *(finish schedule)* Data type: *num*

The identity (expressed as a number) of a weld program in connected welding equipment. It is sent to the welding equipment when the weld phase is completed and is used when crater-filling.

See System Parameter *Arc Welding - Equipment - schedport_type.*

fill_voltage *(crater-filling voltage)* Data type: *num*

The welding voltage (in volts) during crater-filling at the end phase of a process.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

fill_wirefeed *(crater-filling wirefeed)* Data type: *num*

The feed speed of the weld electrode when crater-filling.

The unit is defined in the system parameter *Arc Welding - Units - unit_feed* and, as a rule, is m/minute or inches per minute.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

fill_volt_adj *(filling voltage adjustment)* Data type: *num*

The voltage adjustment during crater-filling.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

fill_curr_adj *(filling current adjustment)* Data type: *num*

The current adjustment during crater-filling.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

Structure

<data object of *seamdata*>
<gas_purge_time of *num*>
<gas_preflow_time of *num*>
<ign_sched of *num*>
<ign_voltage of *num*>
<ign_wirefeed of *num*>
<ign_volt_adj of *num*>
<ign_curr_adj of *num*>
<ign_move_delay of *num*>
<heat_speed of *num*>
<heat_time of *num*>
<heat_distance of *num*>
<heat_sched of *num*>
<heat_voltage of *num*>
<heat_wirefeed of *num*>
<heat_volt_adj of *num*>
<heat_curr_adj of *num*>
<cool_time of *num*>
<fill_time of *num*>
<bback_time of *num*>
<rback_time of *num*>
<gas_postflow_time of *num*>
<fill_sched of *num*>

<fill_voltage of *num*>
<fill_wirefeed of *num*>
<fill_volt_adj of *num*>
<fill_curr_adj of *num*>

Note that the structure changes depending on the configuration of the robot.

Related information

	Described in:
Weld data	Data Types - *welddata*
Installation parameters for welding equipment and functions	System Parameters - *Arc Welding*
Process phases and time diagrams	RAPID Summary- *Arc Welding*
Arc welding instructions	Instructions - *ArcL, ArcC*

weavedata Weave data

Weavedata is used to define any weaving carried out during arc welding.

Weaving can be used during the heat and weld phases of a seam.

Description

Weaving is a movement, superimposed on the basic path of the process.

There are three types of weaving pattern to choose from: zigzag, V-shaped and triangular weaving. These are illustrated in Figure 1 to Figure 3.

All weave data components apply to both the heat phase and the weld phase.

The unit for weave data components that specify a distance is defined in the parameter *Arc Welding - Units - length_unit* and, as a rule, is mm or inches. (These components are *weave_length, _width, _height, _bias* and *dwell* lengths.)

Note. Some of the components of weave data depend on the configuration of the robot. If a given feature is omitted, the corresponding component is left out of the weave data. The conditions that must be met for components to exist are described in the system parameters.

Components

weave_shape *(weld weave shape)* Data type: *num*

The shape of the weaving pattern in the weld phase.

Specified value	Weaving pattern
0	No weaving.
1	Zigzag weaving as illustrated in Figure 1.

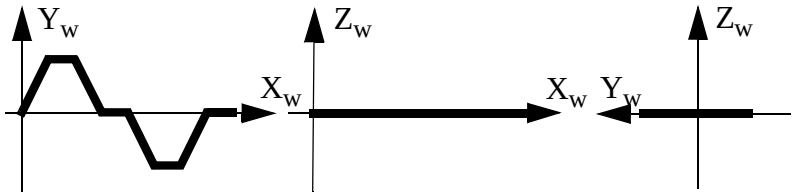


Figure 1 Zig-zag weaving results in weaving horizontal to the seam.

Specified value	Weaving type
2	V-shaped weaving as illustrated in Figure 2.

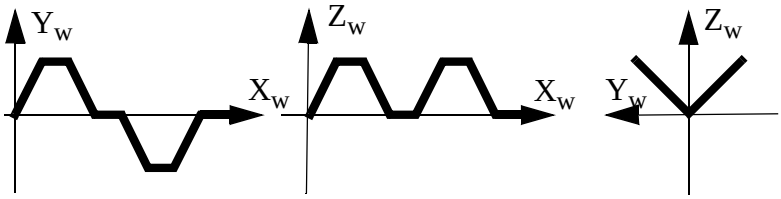


Figure 2 V-shaped weaving results in weaving in the shape of a "V", vertical to the seam.

3	Triangular weaving as illustrated in Figure 3.

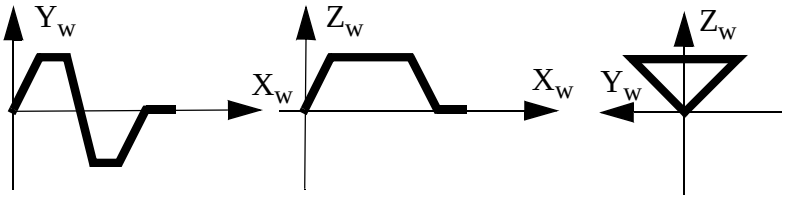


Figure 3 Triangular weaving results in a triangular shape, vertical to the seam.

weave_type (*weld weave interpolation type*) Data type: *num*

The type of weaving in the weld phase.

Specified value	Weaving type
0	Geometric weaving. All axes are used during weaving.
1	Wrist weaving.

weave_length Data type: *num*

The length of the weaving cycle in the weld phase (see Figure 4).

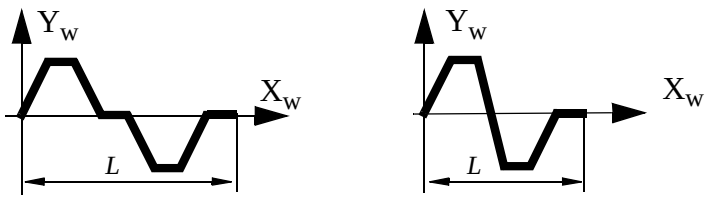


Figure 4 The length (L) of the weaving cycle for zig-zag, V-shaped and triangular weaving.

weave_width Data type: *num*

The width of the weaving pattern in the weld phase (see Figure 5).

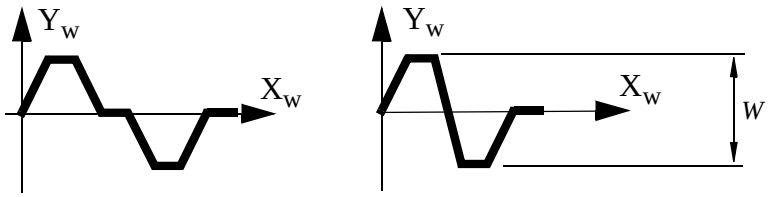


Figure 5 The width (W) of the weaving pattern for all weaving patterns.

weave_height Data type: *num*

The height of the weaving pattern during V-shaped and triangular weaving (see Figure 6).

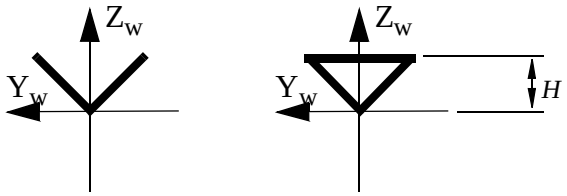


Figure 6 The height (H) of the weaving pattern for V-shaped and triangular weaving.

dwell_left Data type: *num*

The length of the dwell used to force the TCP to move only in the direction of the seam at the left turning point of the weave (see Figure 7).

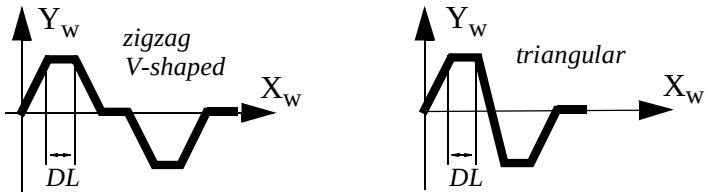


Figure 7 The length of the left dwell (DL) for different types of weaving patterns.

dwell_center Data type: *num*

The length of the dwell used to force the TCP to move only in the direction of the seam at the centre point of the weave (see Figure 8).

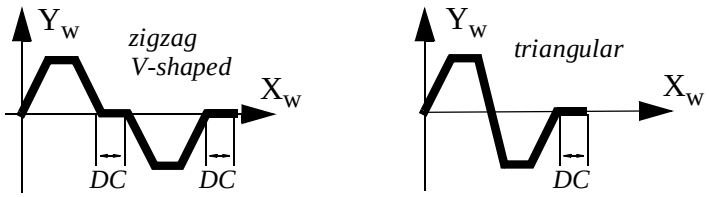


Figure 8 The length of the central dwell (DC) for different types of weaving patterns.

dwell_right Data type: *num*

The length of the dwell used to force the TCP to move only in the direction of the seam at the right turning point of the weave (see Figure 9).

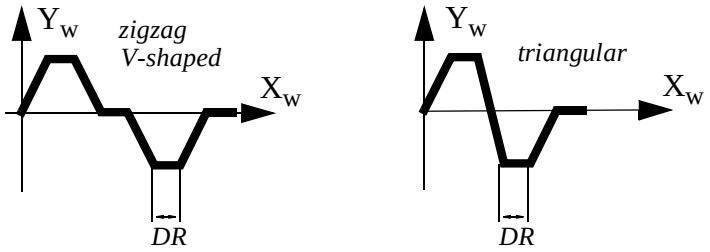


Figure 9 The length of the right dwell (DR) for different types of weaving patterns.

weave_dir *(weave direction angle)* Data type: *num*

The weave direction angle horizontal to the seam (see Figure 10). An angle of zero degrees results in a weave vertical to the seam.

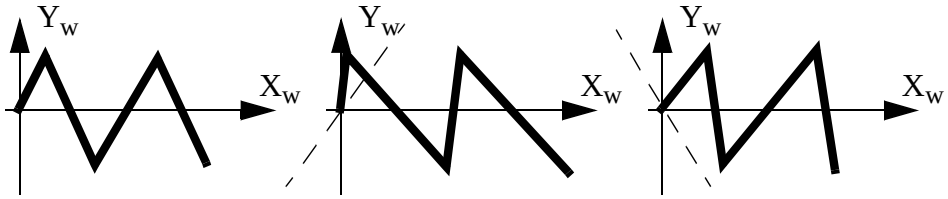


Figure 10 The shape of the weaving pattern at 0 degrees and at a positive and negative angle.

weave_tilt *(weave tilt angle)* Data type: *num*

The weave tilt angle, vertical to the seam (see Figure 11). An angle of zero degrees results in a weave which is vertical to the seam.

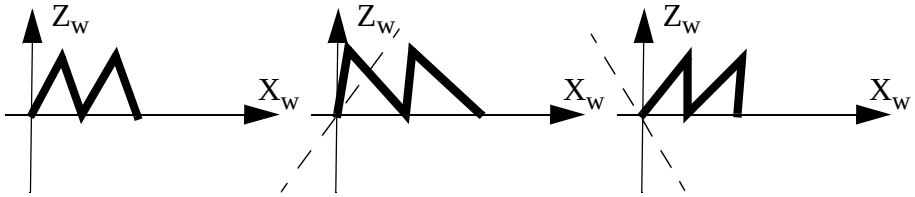


Figure 11 V-weaving at 0 degrees and at a positive and negative angle.

weave_ori *(weave orientation angle)* Data type: *num*

The weave orientation angle, horizontal-vertical to the seam (see Figure 12). An angle of zero degrees results in symmetrical weaving.

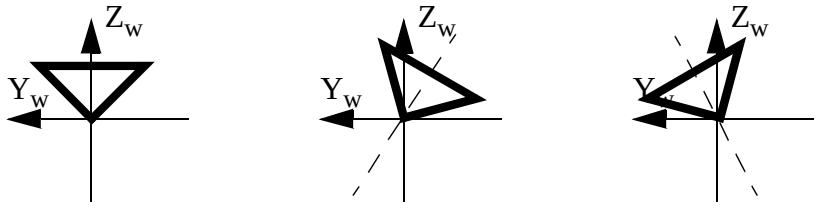


Figure 12 Triangular weaving at 0 degrees and at a positive and negative angle.

weave_bias *(weave centre bias)* Data type: *num*

The bias horizontal to the weaving pattern (see Figure 13). The bias can only be specified for zig-zag weaving and may not be greater than half the width of the weave.

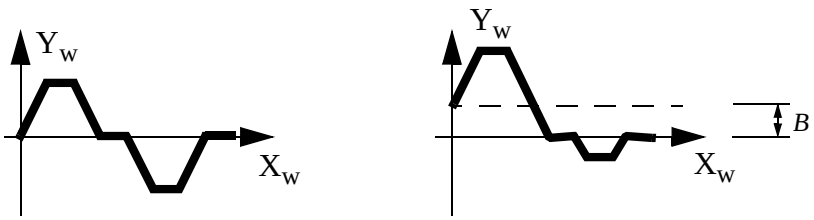


Figure 13 Zig-zag weaving with and without bias (B).

weave_sync_left Data type: *num*

The coordination position to the left of the weaving pattern. It is specified as a percentage of the width on the left of the weaving centre. When weaving is carried out beyond this point, a digital output signal is automatically set to one, as illustrated in Figure 14. This type of coordination is intended for seam tracking using WeldGuide.

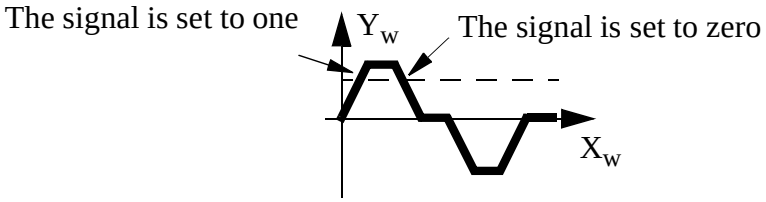


Figure 14 When WeldGuide is used, a sync. signal is required.

weave_sync_right Data type: *num*

The coordination position to the right of the weaving pattern. It is specified as a percentage of the width on the right of the weaving centre. When weaving is carried out beyond this point, a digital output signal is automatically set to one, as illustrated in Figure 15. This type of coordination is intended for seam tracking using WeldGuide.

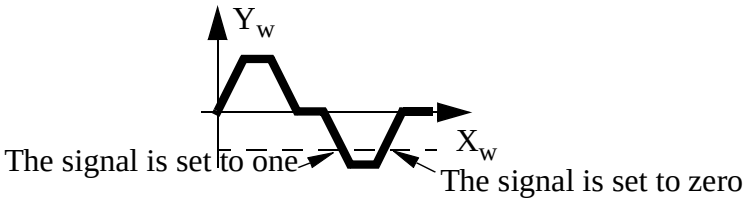


Figure 15 When WeldGuide is used, a sync. signal is required.

wg_track_on Data type: *num*

Activate the weldguide seam tracker.

Limitations

The maximum weaving frequency is 2 Hz.

The inclination of the weaving pattern must not exceed the ratio 1:10 (84 degrees). (See Figure 16).

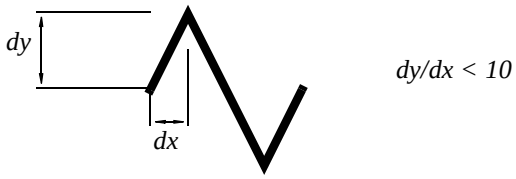


Figure 16 The weaving pattern may not be inclined more than in the ratio 1:10.

Structure

<data object of *weavedata*>
<weave_shape of *num*>
<weave_type of *num*>
<weave_length of *num*>
<weave_width of *num*>
<weave_height of *num*>
<dwell_left of *num*>
<dwell_center of *num*>
<dwell_right of *num*>
<weave_dir of *num*>
<weave_tilt of *num*>
<weave_ori of *num*>
<weave_bias of *num*>
<weave_sync_left of *num*>
<weave_sync_right of *num*>
<wg_track_on of *num*>

Related information

	Described in:
Installation parameters for welding equipment and functions	System Parameters - *Arc Welding*
Process phases and timing schedules	RAPID Summary - *Arc Welding*
Arc-welding instructions	Instructions - *ArcL*, *ArcC*

welddata Weld data

Welddata is used to control the weld during the weld phase, that is, from when the arc is established until the weld is completed.

Other phases, such as the start and end phases, are controlled using *seamdata*.

Description

Weld data describes data that is often changed along a seam. Weld data used in a given instruction along a path affects the weld until the specified position is reached. Using instructions with different weld data, it is thus possible to achieve optimum control of the welding equipment along an entire seam.

Weld data affects the weld when fusion has been established (after heating) at the start of a process.

In the case of a *flying start*, the arc is not ignited until the destination position of the arc welding instruction with the *\On* argument is reached, which means that weld data does not have any effect on the weld in this instruction.

If one arc welding instruction is exchanged for another during a weld, new weld data will occur in the middle of the corner path.

Note. Some of the components of weld data depend on the configuration of the robot. If a given feature is omitted, the corresponding component is left out of the weld data. The conditions that must be met for components to exist are described in the system parameters.

All voltages can be expressed in two ways (determined by the welding equipment):

- As absolute values (only positive values are used in this case).
- As corrections of values set in the process equipment (both positive and negative values are used in this case).

Feeding the weld electrode in this section refers to MIG/MAG welding. In the case of TIG welding:

- A *cold* wire is supplied to the *wire feed*.
- The necessary *welding current* reference value can be connected to any of the three analog outputs that are not used. (The *Welding voltage* reference is not used.)

Example

```
MoveJ   p1, v100, z10, gun1;
MoveJ   p2, v100, fine, gun1;
ArcL    \On, p3, v100, seam1, weld1, weave1, fine, gun1;
ArcL    p4, v100, seam1, weld2, weave1, z10, gun1;
ArcL   \Off, p5, v100, seam1,weld3, weave3, fine, gun1;
MoveJ  p6, v100, z10, gun1;
```

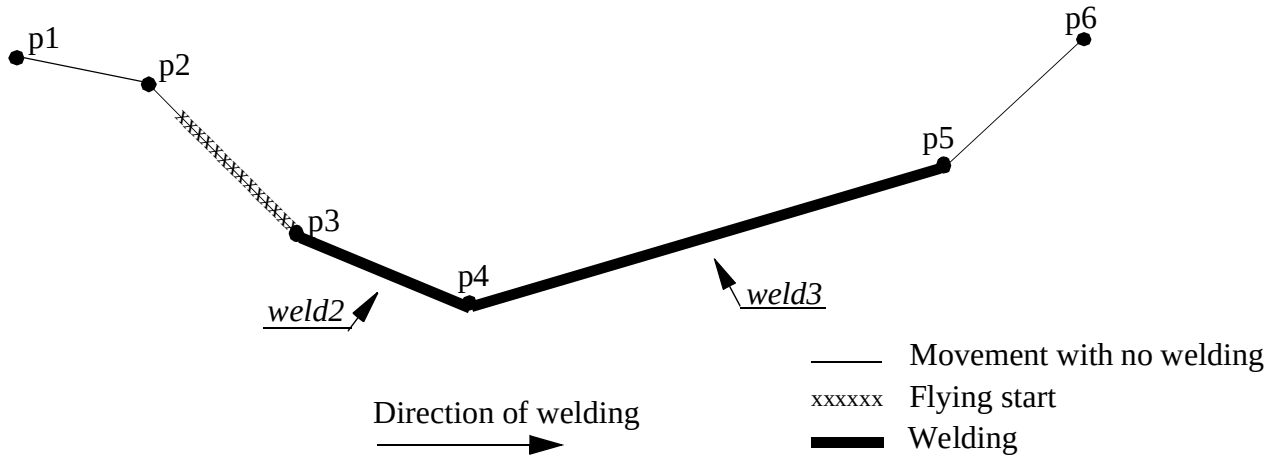
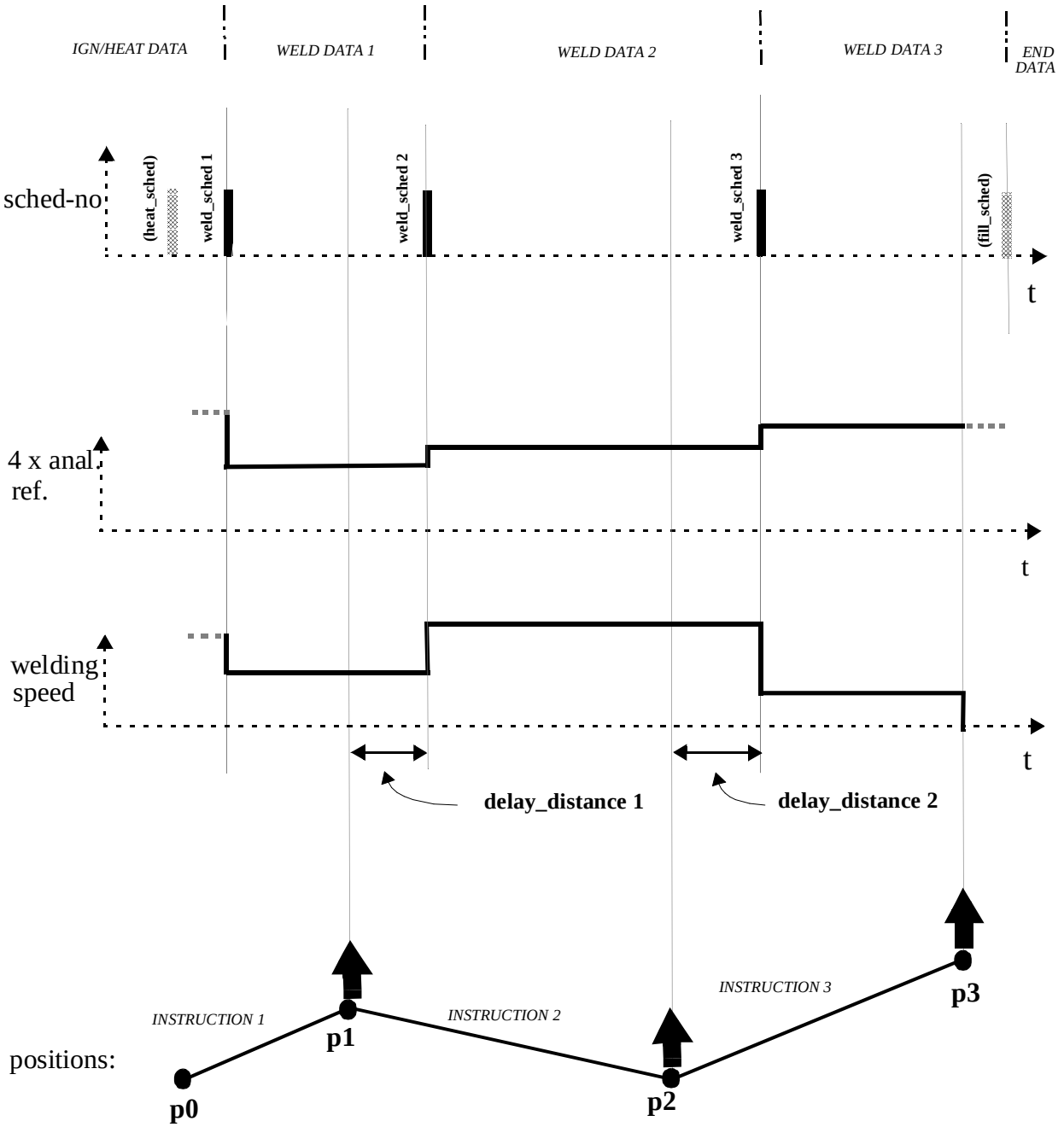


Figure 1 Weld data, such as speed and welding voltage, can be changed at each position.

Weld data is changed in every instruction. As the argument *\On* is used in the first instruction, the first weld data is never used.

The welding sequence



Components

weld_sched (*weld schedule)* Data type: *num*

The identity (expressed as a number) of weld programs to send to the welding equipment.

See System Parameter *Arc Welding - Equipment- schedport_type.*

weld_speed Data type: *num*

The desired welding speed.

The unit is defined in the system parameter *Arc Welding - Units- velocity_unit* and, as a rule, is mm/s or inches per minute.

If the movements of external axes are coordinated, the welding speed is the relative speed between the tool and the object.

If the movements of external axes are not coordinated, the welding speed is the TCP speed. The speed of the external axes is then described in the instruction's speed data. The slowest axis determines the speed to enable all axes to reach the destination position at the same time.

weld_voltage Data type: *num*

The welding voltage (in volts) during the weld phase.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

weld_wirefeed Data type: *num*

The feed speed of the weld electrode during the weld phase.

The unit is defined in the system parameter *Arc Welding - Units - unit_feed* and, as a rule, is metres per minute or inches per minute.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

delay_distance Data type: *num*

The delay distance (after the *destination* position) for a changeover to new weld data in the next arc welding instruction.

The unit is defined in *System Parameters - Arc Welding - Units - length_unit* and as a rule, is mm or inches.

Usually, when changing from one arc welding instruction to another, a fly-by point is used. This results in a changeover point in the middle of the corner path. By using delay distance, the new weld data starts to take effect somewhat later (see Figure 2).

In a weld *end* instruction the delay distance will have no effect.

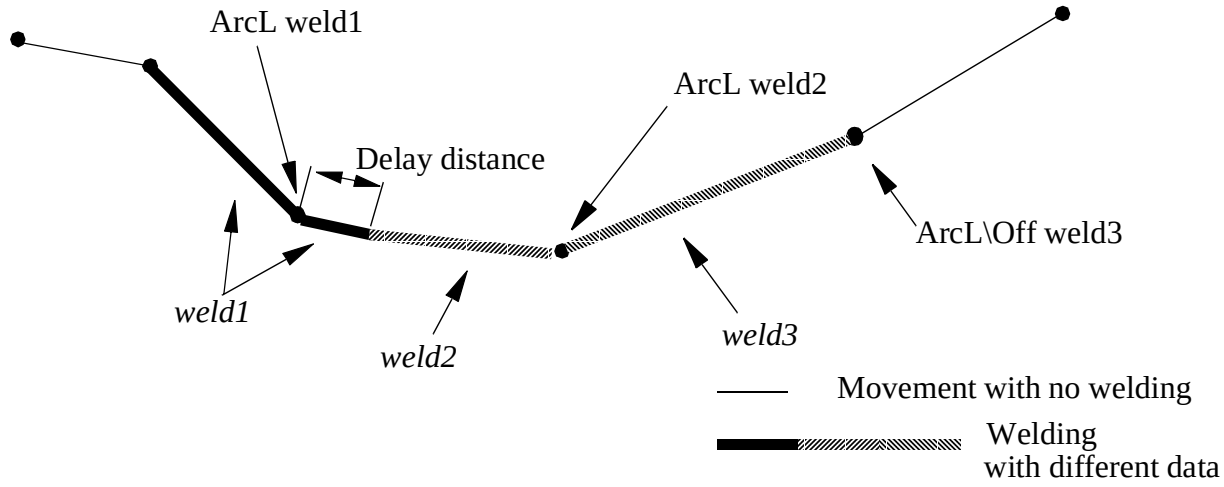


Figure 2 In the above example, the weld data changeover from weld1 to weld2 is delayed and weld2 has a delay_distance=0. The delay_distance in weld3 will thus have no effect.

Delay_distance can, for example, be used in *ArcC* instructions to move the changeover of weld data without reprogramming the circle positions.

weld_volt_adj *(welding voltage adjustment)* Data type: *num*

The voltage adjustment during the weld phase.

The value specified is scaled and sent to the corresponding analog output, in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

weld_curr_adj *(weld current adjustment)* Data type: *num*

The current adjustment during the weld phase.

The specified value is scaled and sent in accordance with the setting in the system parameters for analog signals.

This signal can be used for arbitrary purposes when something needs to be controlled using an analog output signal.

Examples

The type of weld shown in Figure 3 is desired, with a welding voltage of 30 V and a wire feed speed of 15 m/min. The welding speed is 20 mm/s.

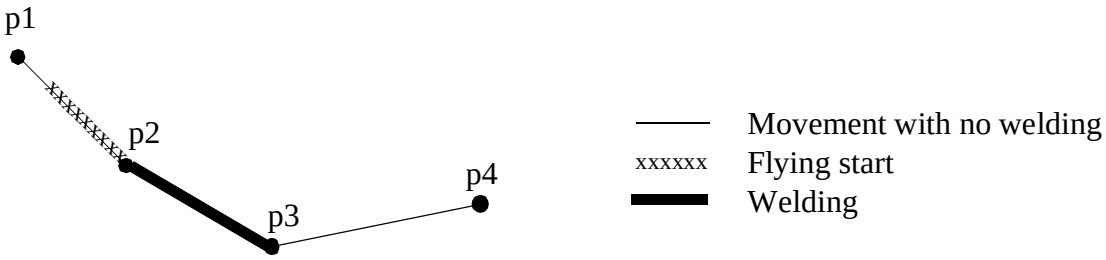


Figure 3 Welding between two points.

```
PERS welddata weld1 := [20,30,15,0];

MoveJ p1, v100, z20, gun1;
ArcL \On, p2, v100, seam1, weld1, noweave, fine, gun1;
ArcL \Off, p3, v100, seam1, weld1, noweave, fine, gun1;
MoveJ p4, v100, z20, gun1;
```

The weld data values for a weld such as the one in Figure 3 are as follows:

Component	weld1	
weld_speed	20 mm/s	Speed in relation to the seam
weld_voltage	30 V	Sent to an analog output signal
weld_wirefeed	15 m/min.	Sent to an analog output signal
delay_distance	0 mm	No delay

The weld schedule identity, weld voltage adjustment and weld current adjustment components are not active in this example.

The weld data argument does not have any effect in the ArcL *\On* instruction.

The type of weld shown in Figure 4 is required. The first section is to be welded using a voltage of 50 V and a wire feed speed of 20 m/min. After a specified distance on the circular arc, the voltage is to be increased to 55 V. The welding speed is 30 mm/s in each section.

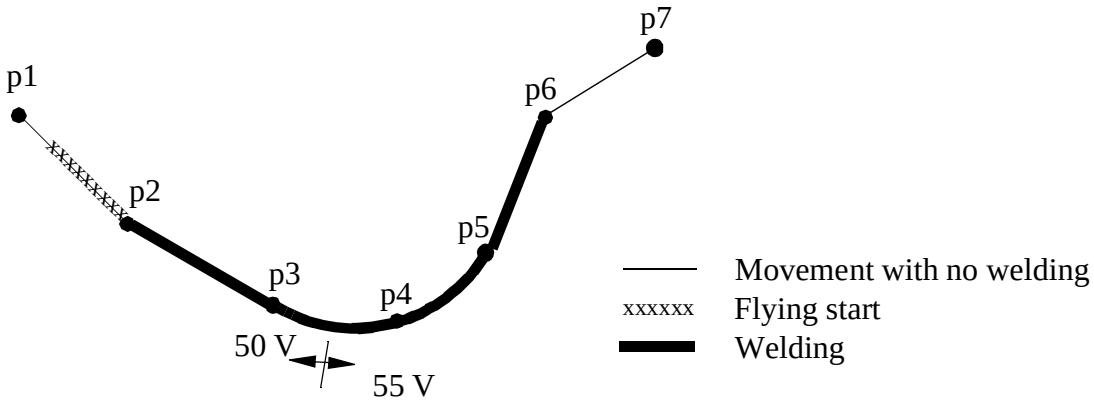


Figure 4 The weld data is changed after a specified distance on the circular path.

```
PERS welddata weld1 := [10,30,50,20,0];
PERS welddata weld2 := [10,30,55,20,17];

MoveJ p1, v100, z20, gun1;
ArcL \On, p2, v100, seam1, weld1, noweave, fine, gun1;
ArcL p3, v100, seam1, weld1, noweave, z10, gun1;
ArcC p4, p5, v100, seam1, weld2, noweave, z10, gun1;
ArcL \Off, p6, v100, seam1, weld2, noweave, fine, gun1;
MoveJ p7, v100, z20, gun1;
```

The weld data values for a weld such as the one in Figure 4 are as follows:

Component	weld1	weld2	
weld_sched	10	10	Identity sent to the welding equipment
weld_speed	30 mm/s	30 mm/s	
weld_voltage	50 V	55 V	
weld_wirefeed	20 m/min.	20 m/min.	
delay_distance	0 mm	17 mm	weld2 is delayed 17 mm

The weld voltage adjustment and weld current adjustment components are not active in this example.

The weld data argument does not have any effect in the ArcL *\On* instruction.

Structure

<data object of *welddata*>
  <weld_sched of *num*>
  <weld_speed of *num*>
  <weld_voltage of *num*>
  <weld_wirefeed of *num*>
  <delay_distance of *num*>
  <weld_volt_adj of *num*>
  <weld_curr_adj of *num*>

Note that the structure changes depending on the configuration of the robot.

Related information

	Described in:
Seam data	Data Types - *seamdata*
Installation parameters for welding equipment and functions	System Parameters - *Arc Welding*
Process phases	RAPID Summary- *Arc Welding*
Arc welding instructions	Instructions - *ArcL, ArcC*

ArcC ArcC1 ArcC2 Arc welding with circular motion

ArcC (Arc Circular) is used to weld along a circular path. The instruction controls and monitors the entire welding process as follows:

- The tool centre point is moved in a circle to the specified destination position.
- All phases, such as the start and end phases, of the welding process are controlled.
- The welding process is monitored continuously.

The only difference between *ArcC, ArcC1* and *ArcC2* is that they are connected to different process systems configured in the System Parameters. Although *ArcC* is used in the examples, *ArcC1* or *ArcC2* could equally well be used.

Example

```
MoveL . . . .
ArcL   \On, p1, v100, seam1, weld5, noweave, fine, gun1;
ArcC   \Off, p2, p3, v100, seam1, weld5, noweave, fine, gun1;
MoveL . . . .
```

This welds a circular seam between points *p1* and *p3* (via point *p2*) as illustrated in Figure 1.

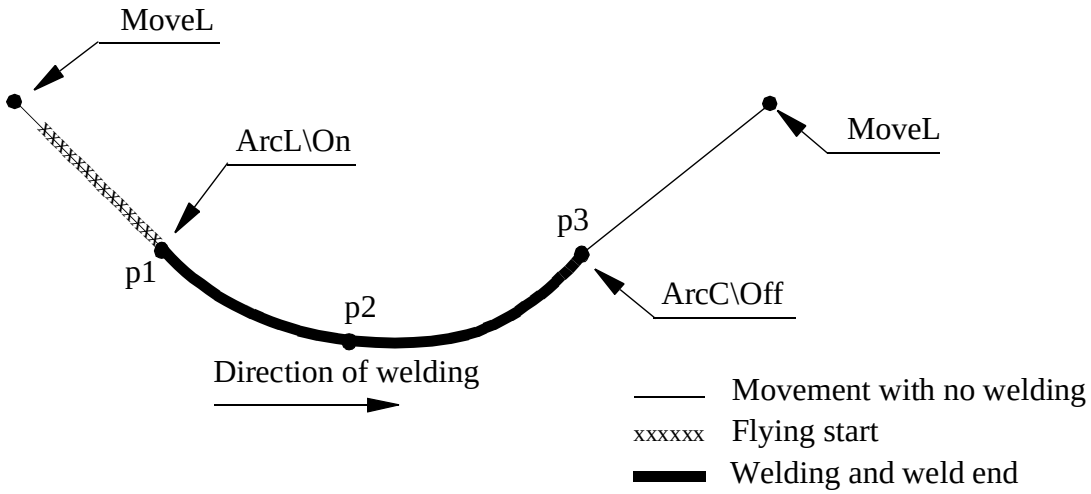


Figure 1 Welding with flying start.

On the way to p1, preparations for the weld start, such as gas preflowing, are carried out. The process and the actual weld movement then start at position p1 and end at p3. The start and end processes are determined by *seam1* and the welding process by *weld5*. Weaving data is carried out according to *noweave*. (No weaving if the *weave_shape* component value is zero.)

V100 specifies the speed attained during the flying start to p1.

Arguments

ArcC [\On] | [\Off] CirPoint ToPoint Speed [\T] Seam Weld Weave Zone [\Z] Tool [\WObj]

[\On] Data type: *switch*

The argument *\On* is used to get a *flying start* (see Figure 1) which, in turn, results in shorter cycle times.

The argument *\On* may only be used in the first of the arc welding instructions to result in a seam. As the end instructions cannot include the argument *\On*, welding with a flying start must include at least two instructions.

The start preparations at a flying start, e.g. gas purging, are carried out on the way to the weld start position.

When the argument *\On* is not used, the weld starts at the position before the *ArcC* instruction (see Figure 2) and the robot remains stationary at the previous position whilst *all* weld start activities are in progress.

Whether or not a flying start is used, the start position for the weld will always be a stop point – regardless of what is specified in the *Zone* argument for that position.

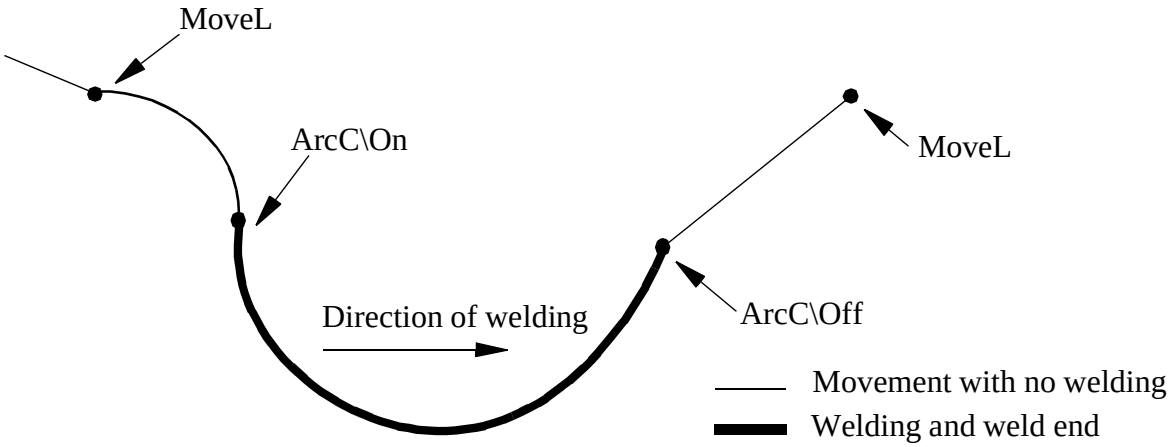


Figure 2 If welding is started without the argument \On, the weld is begun at the previous position.

[\Off] Data type: *switch*

If the argument *\Off* is used, welding ends when the robot reaches the destination position. Regardless of what is specified in the *Zone* argument, the destination position will be a stop point.

If an ArcC instruction without the argument *\Off* is followed by *MoveJ*, for example, welding will end, but in an uncontrolled fashion. Logical instructions, such as *Set do1*, however, can be used between two arc welding instructions without ending the welding process.

CirPoint Data type: *robtarget*

The circle point of the robot. The circle point is a position on the circle between the start point and the destination point. To obtain the best accuracy, it should be placed about halfway between the start and destination points. If it is placed too close to the start or destination point, the robot may give a warning. The circle point is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

ToPoint Data type: *robtarget*

The destination position of the robot and external axes. This is either defined as a named position or stored directly in the instruction (indicated by an * in the instruction).

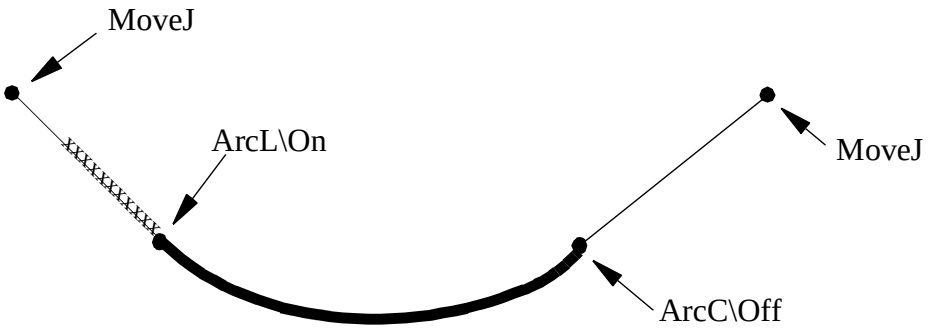
Speed Data type: *speeddata*

The speed of the TCP is controlled by the argument *Speed* in the following cases:

- When the argument *\On* is used (weld start preparations at a flying start).
- When the program is run instruction-by-instruction (no welding).

The speed of the TCP *during welding* is the same as for the arguments *Seam* and *Weld* (see Figure 3).

Speed data also describes the speed of the tool's reorientation and the speed of any uncoordinated external axes.



——— Movement with no welding; speed in accordance with *Speed*
xxxxxx Flying start; speed in accordance with *Speed*
▬▬ Welding; speed in accordance with the *Seam* and *Weld* arguments (specified in the heat and weld speed components).

Figure 3 The speed at different phases of the welding process.

[\T] *(Time)* Data type: *num*

The argument \T is used to specify the total time of movement in seconds, directly in the instruction. Time is thus substituted for the speed specified in the arguments *Speed*, *Seam* and *Weld*.

This argument can be used when, for example, one or more uncoordinated external axes participate in the movement. Uncoordinated external axes should be avoided because when they are used, the program becomes more difficult to adjust. Use coordinated external axes instead. Weaving is deactivated during execution of ArcX instructions with \T arguments.

Seam Data type: *seamdata*

Seam data describes the start and end phases of a welding process.

The argument *Seam* is included in all arc welding instructions so that, regardless of the position of the robot when the process is interrupted, a proper weld end and restart are achieved.

Normally the same seam data is used in all instructions of a seam.

Weld Data type: *welddata*

Weld data describes the weld phase of the welding process.

Weld data is often changed from one instruction to the next, along a seam.

Weave Data type: *weavedata*

Weave data describes the weaving that is to take place during the heat and weld phases. Welding without weaving is obtained by specifying, for example, the weave data *noweave*. (No weaving if the *weave_shape* component value is zero.)

Zone Data type: *zonedata*

Zone data defines how close the axes must be to the programmed position before they can start moving towards the next position.

In the case of a fly-by point, a corner path is generated past that position. In the case of a stop point (*fine*), the movement is interrupted until all axes have reached the programmed point.

A stop point is always generated automatically at the start position of a weld (even in the case of a *flying start*) and at a *controlled* weld end position. Fly-by points, such as *z10*, should be used for all other weld positions.

Weld data changes over to the next arc welding instruction at the centre point of the corner path (if not delayed by the *delay_distance* component in the *Weld* argument).

[\Z] *(Zone)* Data type: *num*

This argument is used to specify the positional accuracy of the robot's TCP directly in the instruction. The size of the zone is specified in mm and is thus substituted in the corresponding zone specified in the zone data. The *\Z* argument is also useful when trimming individual corner paths.

Tool Data type: *tooldata*

The tool used in the movement. The TCP of the tool is the point moved to the specified destination position. The z-axis of the tool should be parallel with the torch.

[\WObj] *(Work Object)* Data type: *wobjdata*

The work object (coordinate system) to which the instruction's robot position is referenced.

When this argument is omitted, the robot position is referenced to the world coordinate system. It must, however, be specified if a stationary TCP or coordinated external axes are used.

\WObj can be used if a coordinate system is defined for either the object in question or the weld seam.

Program execution

Controlling process equipment

The process equipment is controlled by the robot in such a way that the entire process and each of its phases are coordinated with the robot's movements.

Motion

Robot and external axes are moved to the destination position as follows:

- The TCP of the tool is moved circularly at a constant programmed speed. When coordinated axes are used, they are moved circularly at constant programmed speed relative to the work object.
- The tool is reorientated at even intervals throughout the entire course.
- Uncoordinated external axes are executed at a constant speed which means that they reach their destination at the same time as the robot axes.

If the programmed speed of reorientation or of the external axes is exceeded, these speeds will be limited, thereby reducing the speed of the TCP.

The destination position is referenced to:

- the specified object coordinate system if the argument *\WObj* is used;
- the world coordinate system if the argument *\WObj* is not used.

Limitations

When weaving, the distance between the programmed positions should be longer than the periodic time of weaving. If the distance is shorter and if there is a significant

change of angle in the path, the weaving pattern will be distorted.

The instruction *ArcC* 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).

Error management

The process is supervised by a number of signal inputs. If anything abnormal is detected, program execution will stop. If, however, an error handler is programmed, the errors defined below can be remedied without stopping production. See the example in the *RestoPath* instruction.

Error constant (*ERRNO* value)	Description
AW_START_ERR	Start condition error; torch, gas or water supervision
AW_IGNI_ERR	Ignition error; arc supervision
AW_WELD_ERR	Weld error; arc supervision
AW_EQIP_ERR	Weld equipment error; voltage, current, water or gas supervision during welding
AW_WIRE_ERR	Wire stick error; wire stick supervision
AW_STOP_ERR	Welding interrupted with the stop process input

The process supervision is determined as a part of the process equipment configuration.

At the *start* of the process the robot checks that the following *preconditions* have been met:

- stop_process
- water_OK
- gas_OK
- torch_OK

If, after the start command is given, no approved start profile is indicated on the digital input, *arc_OK*, within a predetermined time period, the process start will be interrupted.

When the process is started, all supervision inputs selected are monitored continuously:

- stop_process, water_OK, gas_OK, arc_OK, volt_OK, curr_OK, feed_OK.

The wirestick_err supervision is checked at the end of the weld.

Example

```
MoveL ...
ArcL    \On, *,v100, seam1, weld5, weave1, fine, gun1\Wobj:=wobj1;
ArcC    *, *, v100, seam1,weld5, weave1, z10, gun1\Wobj:=wobj1;
ArcL    *, v100, seam1,weld5, weave1, z10, gun1\Wobj:=wobj1;
ArcC    \Off, *, *, v100, seam1,weld3, weave3, fine, gun1\Wobj:=wobj1;
MoveL...
```

In this example, a weld is performed in which weld data and weave data are changed in the final part of the weld, which is illustrated in Figure 4. Note that an arc welding instruction must be used to change the direction of the path despite the fact that no weld data is changed.

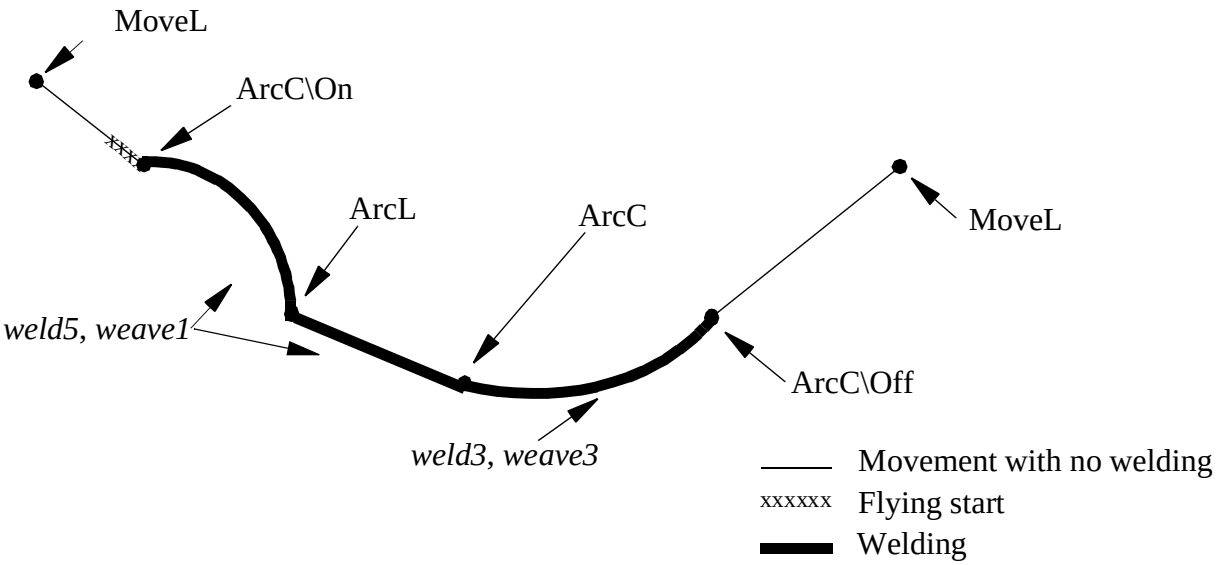


Figure 4 The direction and weld data can be changed by programming several arc welding instructions.

It is assumed, in this example, that a coordinated external axis is used in the movement. In this case, the *wobj1* work object must be specified in the instruction.

Syntax

```
ArcC
   [ '\'On',' ] | [ '\'Off',' ]
   [ CirPoint ':=' ] < expression (IN) of robtarget > ','
   [ ToPoint ':=' ] < expression (IN) of robtarget > ','
   [ Speed ':=' ] < expression (IN) of speeddata >
      [ ( '\' T ':=' < expression (IN) of num > ) ] ','
   [ Seam ':=' ] < persistent (PERS) of seamdata > ','
   [ Weld ':=' ] < persistent (PERS) of welddata > ','
   [ Weave ':=' ] < persistent (PERS) of weavedata > ','
   [Zone ':=' ] < expression (IN) of zonedata >
      [ '\' Z ':=' < expression (IN) of num > ] ','
   [ Tool ':=' ] < persistent (PERS) of tooldata >
   [ '\' WObj ':=' < persistent (PERS) of wobjdata > ] ';'
```

Related information

	Described in:
Performing a linear weld	Instructions - *ArcL*
Other positioning instructions	RAPID Summary - *Motion*
Definition of speed	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Definition of seam data	Data Types - *seamdata*
Definition of weld data	Data Types - *welddata*
Definition of weave data	Data Types - *weavedata*
Installation parameters for welding equipment and welding functions	System Parameters - *Arc Welding*
Movements in general	Motion Principles
Coordinate systems	Motion Principles - *Coordinate Systems*
Process phases and sub-activities	RAPID Summary - *Arc Welding*

ArcL ArcL1 ArcL2 Arc welding with linear motion

ArcL (Arc Linear) is used to weld along a straight seam. The instruction controls and monitors the entire welding process as follows:

- The tool centre point is moved linearly to the specified destination position.
- All phases of the welding process, such as the start and end phases, are controlled.
- The welding process is monitored continuously.

The only difference between *ArcL*, *ArcL1* and *ArcL2* is that they are connected to different process systems configured in the System Parameters. Although *ArcL* is used in the examples, *ArcL1* or *ArcL2* could equally well be used.

Example

```
MoveJ . . . .
ArcL   \On, p1, v100, seam1, weld5, noweave, fine, gun1;
ArcL   \Off, p2, v100, seam1, weld5, noweave, fine, gun1;
MoveJ . . . .
```

This welds a straight seam between points *p1* and *p2*, as illustrated in Figure 1.

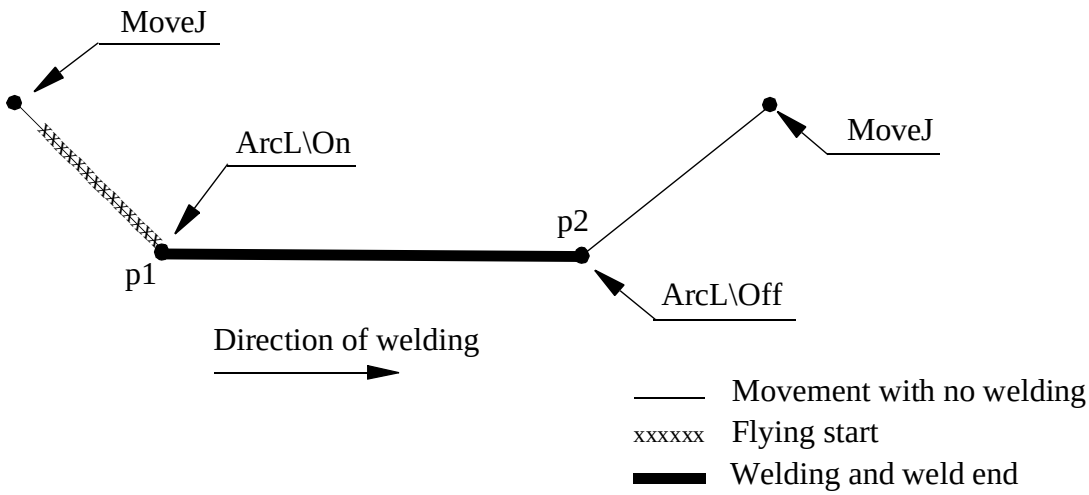


Figure 1 Welding with flying start.

On the way to p1, preparations for the weld start, such as gas preflowing, are carried out. The process and the actual weld movement then start at position p1 and end at p2. The start and end processes are determined by *seam1* and the welding process by *weld5*. Weaving data is carried out according to *noweave*. (No weaving if the *weave_shape* component value is zero.)

V100 specifies the speed attained during the flying start to p1.

Arguments

ArcL [\On] | [\Off] ToPoint Speed [\T] Seam Weld Weave Zone [\Z] Tool [\WObj]

[\On] Data type: *switch*

The argument *\On* is used to obtain a *flying start* (see Figure 1) which, in turn, results in shorter cycle times.

The argument *\On* may only be used in the first of the arc welding instructions to result in a seam. As the end instructions cannot include the argument *\On*, welding with a flying start must include at least two instructions.

The start preparations at a flying start, e.g. gas purging, are carried out on the way to the weld start position.

When the argument *\On* is not used, the weld starts at the position before the *ArcL* instruction (see Figure 2) and the robot remains stationary at the previous position whilst *all* weld start activities are in progress.

Whether or not a flying start is used, the start position for the weld will always be a stop point – regardless of what is specified in the *Zone* argument for that position.

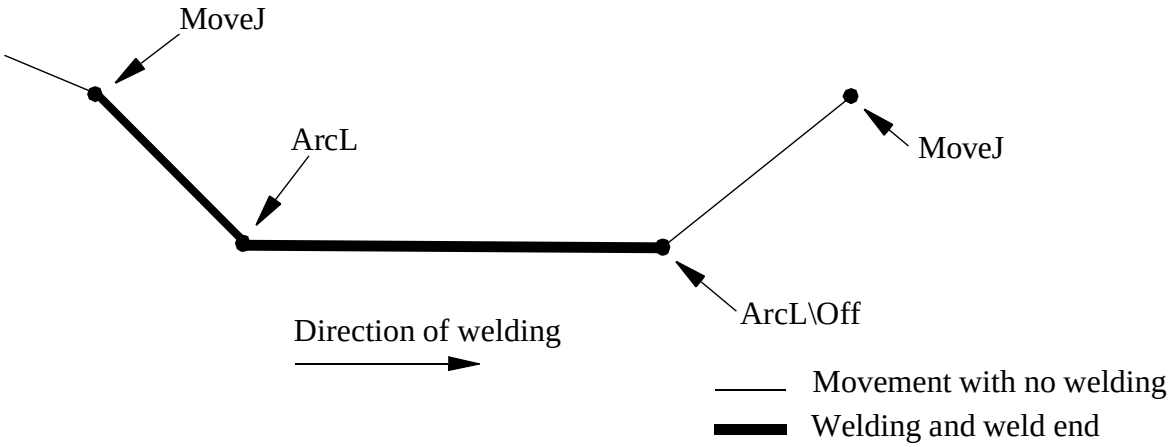


Figure 2 If welding is started without the argument \On, the weld is begun at the previous position.

[\Off] Data type: *switch*

If the argument *\Off* is used, welding ends when the robot reaches the destination position. Regardless of what is specified in the *Zone* argument, the destination position will be a stop point.

If an ArcL instruction without the argument *\Off* is followed by *MoveJ*, for example, welding will end, but in an uncontrolled fashion. Logical instructions, such as *Set do1*, however, can be used between two arc welding instructions without ending the welding process.

ToPoint Data type: *robtarget*

The destination position of the robot and external axes. This is either defined as a named position or stored directly in the instruction (indicated by an * in the instruction).

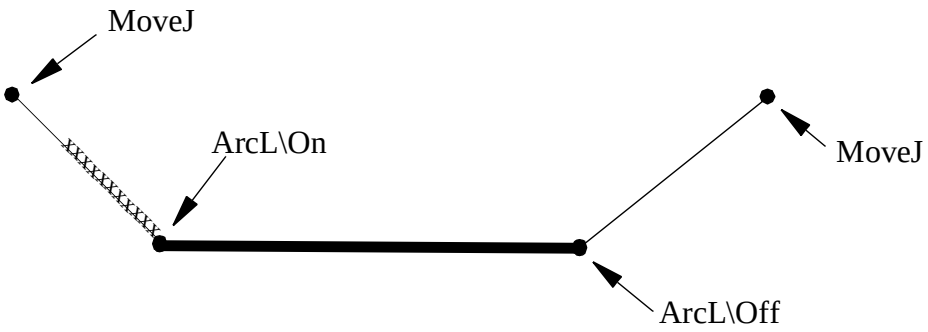
Speed Data type: *speeddata*

The speed of the TCP is controlled by the argument *Speed* in the following cases:

- When the argument *On* is used (weld start preparations at a flying start).
- When the program is run instruction-by-instruction (no welding).

The speed of the TCP *during welding* is the same as for the arguments *Seam* and *Weld.* (See Figure 3)

Speed data also describes the speed of the tool's reorientation and the speed of any uncoordinated external axes.



Movement with no welding; speed in accordance with *Speed*

xxxxxx Flying start; speed in accordance with *Speed*

Welding; speed in accordance with the *Seam* and *Weld* arguments (specified in the heat and weld speed components).

Figure 3 The speed at different phases of the welding process.

[\T] *(Time)* Data type: *num*

The argument \T is used to specify the total time of movement in seconds directly in the instruction. Time is thus substituted for the speed specified in the arguments *Speed*, *Seam* and *Weld*.

This argument can be used when, for example, one or more uncoordinated external axes participate in the movement. Uncoordinated external axes should, however, be avoided since, if used, the program becomes more difficult to adjust. Use coordinated external axes instead. Weaving is deactivated during execution of ArcX instructions with \T arguments.

Seam Data type: *seamdata*

Seam data describes the start and end phases of a welding process.

The argument *Seam* is included in all arc welding instructions so that, regardless of the position of the robot when the process is interrupted, a proper weld end and restart is achieved.

Normally the same seam data is used in all instructions of a seam.

Weld Data type: *welddata*

Weld data describes the weld phase of the welding process.

Weld data is often changed from one instruction to the next along a seam.

Weave Data type: *weavedata*

Weave data describes the weaving that is to take place during the heat and weld phases. Welding without weaving is obtained by specifying, for example, the weave data *noweave.* (No weaving if the *weave_shape* component value is zero.)

Zone Data type: *zonedata*

Zone data defines how close the axes must be to the programmed position before they can start moving towards the next position.

In the case of a fly-by point, a corner path is generated past that position. In the case of a stop point (*fine*), the movement is interrupted until all axes have reached the programmed point.

A stop point is always generated automatically at the start position of a weld (even in the case of a *flying start*) and at a *controlled* weld end position. Fly-by points, such as *z10*, should be used for all other weld positions.

Weld data changes over to the next arc welding instruction at the centre point of the corner path (if not delayed by the *delay_distance* component in the *Weld* argument).

[\Z] *(Zone)* Data type: *num*

This argument is used to specify the positional accuracy of the robot's TCP directly in the instruction. The size of the zone is specified in mm and is thus substituted in the corresponding zone specified in the zone data. The *\Z* argument is also useful when trimming individual corner paths.

Tool Data type: *tooldata*

The tool used in the movement. The TCP of the tool is the point moved to the specified destination position. The z-axis of the tool should be parallel with the torch.

[\WObj] *(Work Object)* Data type: *wobjdata*

The work object (coordinate system) to which the instruction's robot position is referenced.

When this argument is omitted, the robot position is referenced to the world coordinate system. It must, however, be specified if a stationary TCP or coordinated external axes are used.

\WObj can be used if a coordinate system is defined for either the object in question or the weld seam.

Program execution

Controlling process equipment

The process equipment is controlled by the robot in such a way that the entire process and each of its phases are coordinated with the robot's movements.

Motion

Robot and external axes are moved to the destination position as follows:

- The TCP of the tool is moved linearly at a constant programmed speed. When coordinated axes are used, they are moved linearly at constant programmed speed relative to the work object.
- The tool is reorientated at even intervals throughout the entire course.
- Uncoordinated external axes are executed at a constant speed which means that they reach their destination at the same time as the robot axes.

If the programmed speed of reorientation or of the external axes is exceeded, these speeds will be limited, thereby reducing the speed of the TCP.

The destination position is referenced to:

- the specified object coordinate system if the argument *\WObj* is used;
- the world coordinate system if the argument *\WObj* is not used.

Limitations

When weaving, the distance between the programmed positions should be longer than the periodic time of weaving. If the distance is shorter and if there is a significant change of angle in the path, the weaving pattern will be distorted.

Fault management

The process is supervised by a number of signal inputs. If anything abnormal is detected, program execution will stop. If, however, an error handler is programmed, the errors defined below can be remedied without stopping production. See the exam-

ple in the *RestoPath* instruction.

Error constant (*ERRNO* value)	Description
AW_START_ERR	Start condition error; torch, gas or water supervision
AW_IGNI_ERR	Ignition error; arc supervision
AW_WELD_ERR	Weld error; arc supervision
AW_EQIP_ERR	Weld equipment error; voltage, current, water or gas supervision during welding
AW_WIRE_ERR	Wire stick error; wire stick supervision
AW_STOP_ERR	Welding interrupted using the stop process input

The process supervision is determined as a part of the process equipment configuration.

At the *start* of the process the robot checks that the following *preconditions* have been met:

- stop_process
- water_OK
- gas_OK
- torch_OK

If, after the start command is given, no approved start profile is indicated on the digital input, *arc_OK*, within a predetermined time period, the process start will be interrupted.

When the process is started, all supervision inputs selected are monitored continuously:

- stop_process, water_OK, gas_OK, arc_OK, volt_OK, curr_OK, feed_OK

The wirestick_err supervision is checked at the end of the weld.

Example

```
MoveL ...
ArcL   \On, *, v100, seam1, weld5, weave1, fine, gun1\Wobj:=wobj1;
ArcL   *, v100, seam1,weld5, weave1, z10, gun1\Wobj:=wobj1;
ArcL   *, v100, seam1,weld5, weave1, z10, gun1\Wobj:=wobj1;
ArcL   \Off, *, v100, seam1,weld3, weave3, fine, gun1\Wobj:=wobj1;
MoveL ...
```

In this example, a weld is performed in which weld data and weave data are changed in the final part of the weld, which is illustrated in Figure 4. Note that an arc welding instruction must be used to change the direction of the path despite the fact that no weld data is changed.

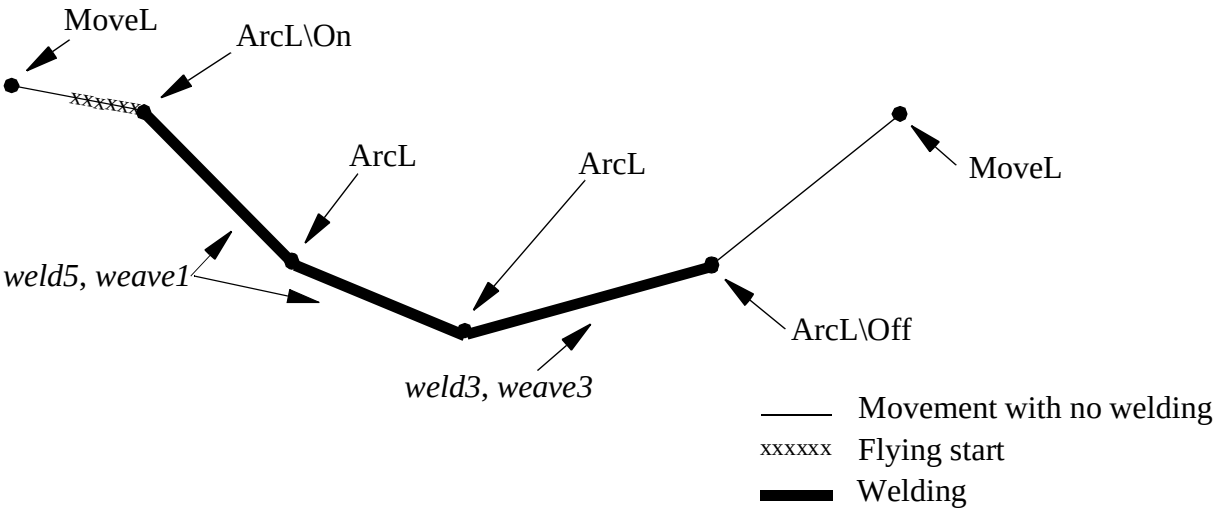


Figure 4 The direction and weld data can be changed by programming several ArcL instructions.

It is assumed, in this example, that a coordinated external axis is used in the movement. In this case, the *wobj1* work object must be specified in the instruction.

Syntax

```
ArcL
   [ ’\’On’,’ ] | [ ’\’Off’,’ ]
   [ ToPoint ’:=’ ] < expression (IN) of robtarget > ’,’
   [ Speed ’:=’ ] < expression (IN) of speeddata >
      [ ( ’\’ T ’:=’ < expression (IN) of num > ) ] ’,’
   [ Seam ’:=’ ] < persistent (PERS) of seamdata > ’,’
   [ Weld ’:=’ ] < persistent (PERS) of welddata > ’,’
   [ Weave ’:=’ ] < persistent (PERS) of weavedata > ’,’
   [Zone ’:=’ ] < expression (IN) of zonedata >
      [ ’\’ Z ’:=’ < expression (IN) of num > ] ’,’
   [ Tool ’:=’ ] < persistent (PERS) of tooldata >
   [ ’\’ WObj ’:=’ < persistent (PERS) of wobjdata > ’;’
```

Related information

	Described in:
Performing a circular weld	Instructions - *ArcC*
Other positioning instructions	RAPID Summary - *Motion*
Definition of speed	Data Types - *speeddata*
Definition of zone data	Data Types - *zonedata*
Definition of tools	Data Types - *tooldata*
Definition of work objects	Data Types - *wobjdata*
Definition of seam data	Data Types - *seamdata*
Definition of weld data	Data Types - *welddata*
Definition of weave data	Data Types - *weavedata*
Installation parameters for welding equipment and welding functions	System Parameters - *Arc Welding*
Movements in general	Motion Principles
Coordinate systems	Motion Principles - *Coordinate Systems*
Process phases and sub-activities	RAPID Summary - *Arc Welding*

gundata Spot weld gun data

Gundata is used to define spot weld gun specific data, to control the gun in an optimal way in the weld process.

Description

Gundata is used in spot weld instructions and has the following structure:

- Number of electrode pairs.
- Number of pressure levels.
- Flag to indicate if a signal is required to test accurate gun closure.
- Flag to indicate if a signal is required to test accurate gun opening.
- Counter for the number of welds done and maximum allowed number (one counter and maximum value for each pair).
- Closing times.
- Opening times.

Components

nof_tips *(number of tips)* Data type: *num*

Number of electrode pairs (1 or 2)

nof_plevels *(number of pressure levels)* Data type: *num*

Number of pressure levels (1 - 4)

close_request *(closure request)* Data type: *bool*

If the flag is TRUE, the ordered pressure level is tested before the weld may start.

open_request *(opening request)* Data type: *bool*

If the flag is TRUE, the gun open signal is tested before the next motion is released. The time out is defined by *sw_go_timeout*.
(See Predefined Data and Programs - *System Module SWUSER*)

Note. The **open_time** must elapse before the test is done (see below).

If the flag is FALSE, the next motion is always released after **open_time** (see below).

tip1_counter Data type: *num*

Counter for the number of welds done with the first pair of electrode tips. The counter is automatically incremented. Use of counter is optional. The zeroing shall be handled by the user program.

tip2_counter Data type: *num*

Counter for the number of welds done with the second pair of electrode tips. The counter is automatically incremented. Use of counter is optional. The zeroing shall be handled by the user program.

tip1_max [1] Data type: *num*

Maximum number of welds to be made by the first pair of electrode tips before tip service is required. This parameter may be used for automatic tip dressing etc.

tip2_max [1] Data type: *num*

Maximum value for number of welds allowed with the second pair of electrode tips before tip service shall be performed. To be used if desired by the user program.

close_time1 Data type: *num*

Time [s] to close the gun if pressure p1 is activated.

close_time2 Data type: *num*

Time [s] to close the gun if pressure p2 is activated.

close_time3 Data type: *num*

Time [s] to close the gun if pressure p3 is activated.

close_time4 Data type: *num*

Time [s] to close the gun if pressure p4 is activated.

build_up_p1[1] Data type: *num*

Time [s] elapsed to build up the pressure from the moment when the gun has just about closed. The pressure p1 has to be set in advance. This parameter may be used to measure the closing time automatically.

build_up_p2[1] Data type: *num*

Time [s] elapsed to build up the pressure from the moment when the gun has just about closed. The pressure p2 has to be set in advance.

1. To be defined only if used by the user program.

build_up_p3[1] Data type: *num*

Time [s] elapsed to build up the pressure from the moment when the gun has just about closed. The pressure p3 has to be set in advance.

build_up_p4[1] Data type: *num*

Time [s] elapsed to build up the pressure from the moment when the gun has just about closed. The pressure p4 has to be set in advance.

open_time Data type: *num*

The time [s] that always elapses between the gun opening order and the release of the next motion or the test of gun open (see above).

Structure

```
< dataobject of gundata>
   <nof_tips of num>
   <nof_plevels of num>
   <close_request of bool>
   <open_request of bool>
   <tip1_counter of num>
   <tip2_counter of num>
   <tip1_max of num>
   <tip2_max of num>
   <close_time1 of num>
   <close_time2 of num>
   <close_time3 of num>
   <close_time4 of num>
   <build_up_p1 of num>
   <build_up_p2 of num>
   <build_up_p3 of num>
   <build_up_p4 of num>
   <open_time of num>
```

.Related information

	Described in:
Spot weld instruction	Instructions - *SpotL*
Definition of tool TCP, weight etc.	Data types - *tooldata*

1. To be defined only if used by the user program.

spotdata Spot weld data

Spotdata is used to define the parameters that control a weld timer and weld gun for welding a certain spot.

Description

Spotdata is referred to spot weld instructions and contains data which controls the welding in the actual instruction.

Spotdata has the following structure:

- Program number for the program in the weld timer to be used.
- Electrode tip to be activated (in case of a double gun).
- Desired gun pressure (if connected).
- Weld timer number (if more than one).

Components

prog_no *(program number)* Data type: *num*

Defines the internal program in the weld timer to be used for the welding.

The allowed range is 0..**sw_prog_max** (defined in the system module *SWUSER*).

tip_no *(electrode pair number)* Data type: *num*

Defines the electrode pair to be activated. The following alternatives are available:

1:	Electrode pair 1
2:	Electrode pair 2
12:	Electrode pairs 1 and 2. Both pairs are closing at the same time. The welding is done in sequence, pair 1 first and pair 2 second. Then, both electrodes are opening together.
21:	Electrode pairs 2 and 1. Both pairs are closing at the same time. The welding is done in sequence, pair 2 first and pair 1 second. Then, both electrodes are opening together.

For guns with only one pair of electrodes the value shall be 1.

gun_pressure *(gun pressure)* Data type: *num*

Defines the gun pressure to use.

The following alternatives are available: Gun pressure 1-4

timer_no *(weld timer number)* Data type: *num*

Defines the weld timer to be used. Only used for certain type of weld timers.

If only one timer is used (normal case) the value shall be 1.

Example

PERS spotdata spot1:= [16, 1,4,1];

The spotdata spot1 is programmed for an equipment containing one single gun and one weld timer. When spot1 is used the following occurs:

- The program number 16 is controlling the welding.
- The gun pressure must reach level 4 before the weld is allowed to start.

Structure

<dataobject of *spotdata*>
<*prog_no* of *num*>
<*tip_no* of *num*>
<*gun_pressure* of *num*>
<*timer_no* of *num*>

Related information

	Described in:
The Spot weld instruction	Instructions - *SpotL*
Gundata	Data Types - *Gundata*

SpotL Spot Welding with motion

SpotL (SpotLinear) is used in spotwelding to control the motion, gun closure/opening and the welding process. *SpotL* moves the TCP linearly to the end position.

Example

SpotL p100, vmax, spot10, gun7, tool7;

This is the only instruction needed to implement a complete welding operation.

The TCP for *tool7* is moved on a linear path to the position *p100* with the speed given in *vmax*. The weld position is always a stop position since the welding is always performed while the robot is standing still. The gun closes in advance on it's way to the position. The welding is started and supervised until finished and the gun is reopening.

Note. The program continues to execute after the weld has started and is not blocked until the next order that contains a robot motion. This may be inhibited by the switch \NoConc (see below).

Spotdata spot10 contains parameters to the welding equipment.

Gundata gun7 contains gun specific weld data.

Arguments

SpotL ToPoint Speed Spot [\InPos] [\NoConc] [\Retract] Gun Tool [\WObj]

ToPoint Data type: *robtarget*

The destination point of the robot and external axes. It is defined as a named position or stored directly in the instruction (marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity for the tool centre point, the tool reorientation and external axes.

Spot Data type: *spotdata*

Spot data that is associated with the weld process equipment.

[\InPos] Data type: *switch*

The optional argument \InPos inhibits the preclosing of the gun. The gun is closed first when the robot has reached the end position. This argument will increase the execution time but is useful in narrow situations.

[\NoConc] Data type: *switch*

The optional argument \NoConc prevents the program from continuing the execution until the actual weld is finished. It should be used when the next instruction is a logical instruction.

[\Retract] Data type: *switch*

The optional argument \Retract will make the gun open to its large gap (retract) after the weld. If the argument is omitted the gun opens to its small gap (work).

Gun Data type: *gundata*

Weld specific tool data for the gun in use.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination position, and should be the position of the tips when the gun is closed.

[\WObj] 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 in order to perform a linear movement relative to the work object.

Customising

The SpotWare package gives the user plenty of scope for customising the *SpotL* instruction:

- by user defined data which affects the internal behaviour in *SpotL*. (See Predefined Data and Programs - *System Module SWUSER, SWUSRF, SWUSRC*).
- by user defined routines which are called at predefined spots in the internal sequence. (See Predefined Data and Programs - *System Module SWUSER, SWUSRF, SWUSRC*).
- by changing the I/O configuration. (See System Parameters - *Spot Welding*).

However, the main subject of this *SpotL* instruction description is the default setup.

Program execution

Internal sequence in a *SpotL* instruction:

- The gun starts to move towards the position.
- The weld program number is set for the external weld timer.
- The gun pressure is set.
- The preweld supervision is done (SpotWare Plus).
- The gun starts to close at a defined time before the position (if argument \InPos is not used).
- The gun is closed when it has reached the position.
- The weld counter is incremented.
- The OK signal for pressure reached is expected.
- The preweld supervision signals are tested (SpotWare).
- The movement to the next position is blocked.
- The start signal is sent to the weld timer.
- The program execution continues to the next movement instruction (unless argument \NoConc is used).
- When the ready signal from the timer is reached, the gun is opened.
- The motion is released on a release signal or a certain time after the gun opening.

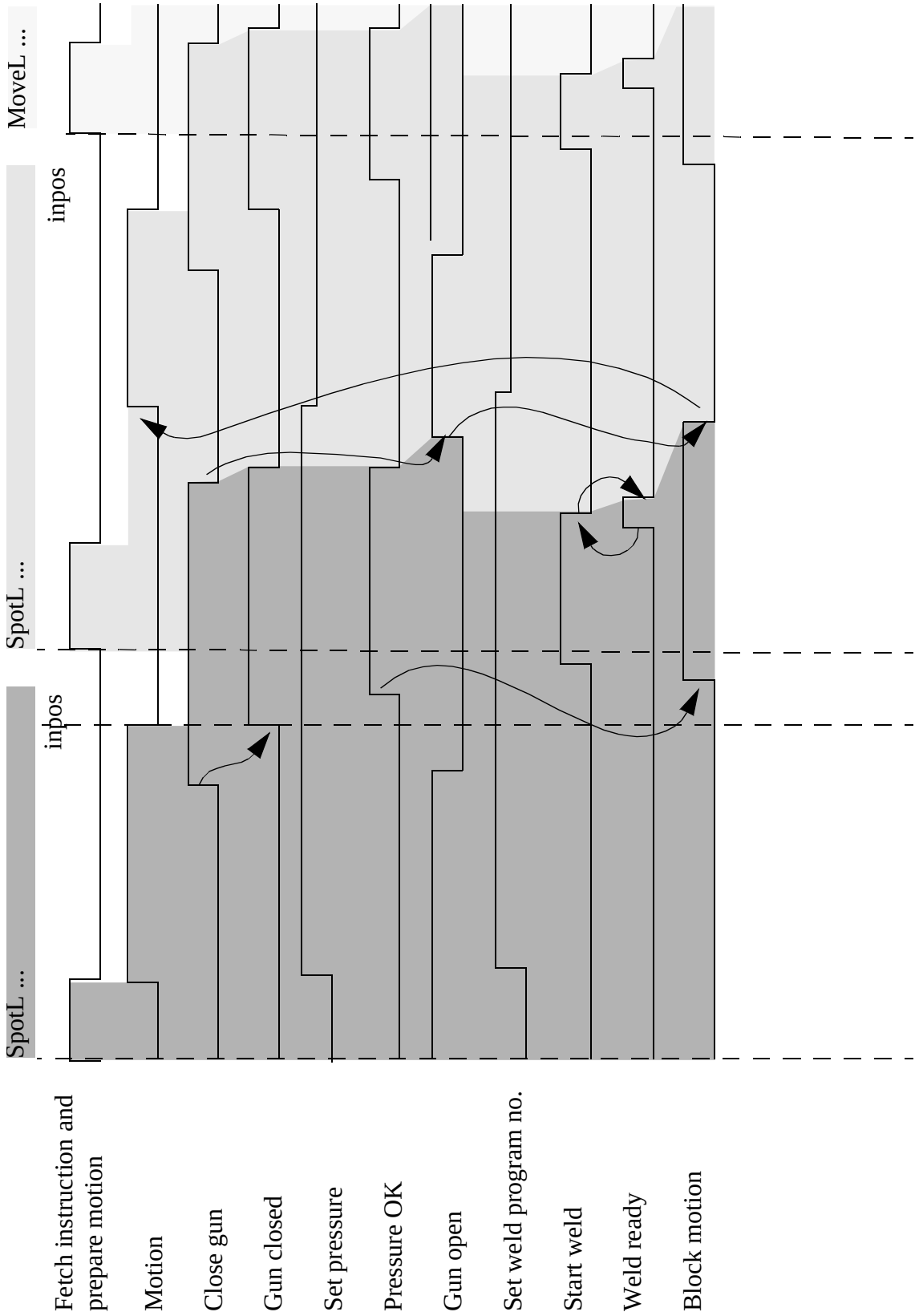


Figure 1 Spot weld sequence.

Gun closure

The gun closure is activated at a defined time before the weld position, irrespective of the actual speed. Closure shall be finished when the gun arrives at the end position. The close time is defined in the gun data. It relates to the time for closing the gun from the work stroke and is dependent on the chosen gun pressure (in spot data).

A maximum of four discrete gun pressures can be chosen in spot data. The gun pressure is set as soon as the motion towards the position starts. A check is made on whether or not the gun pressure has been attained, if so demanded by gun data.

When the gun is closing, the *work_select* is set to 1, if desired. See Predefined Data and Programs.

Welding

The start signal is sent to the timer as soon as the robot has reached the end position. Before the start signal is sent, a number of supervisions must also be acknowledged. These are user defined routines. See Predefined Data and Programs.

The start signal is high during the entire welding period. It is reset either after weld ready or weld timeout.

Gun opening

The gun opens to a small or large stroke, after the welding has finished, depending on the parameter \Retract. The opening is supervised in such a way that a gun open signal is expected. This will release the next motion.

The gun is also opened after a weld error and other error situations.

Motion

To gain time the program pointer goes on to the next motion instruction while the welding is still preforming. This makes it possible to perform the next motion immediately on a release order.

This means that instructions after *SpotL*, other than motional, such as logical instructions, will be performed concurrently with the welding. If this is not the intention the parameter *NoConc* shall be used.

A move enable for the next motion can be configured to be a signal from the gun or a certain time after weld ready.

The end position is related to the used object coordinate system in *WObj*.

Additional functionality in SpotWare Plus

SpotWare Plus opens up a wider range of customising the SpotWare packet by offering more routines to the user. These are straight forward routines hooked into the base software. The following functions can be freely programmed by the user:

Gun closure - sw_close_gun

Gun opening - sw_open_gun

Pressure setting - sw_set_pressure

Preclose time calculation - sw_close_time

Note that the default code gives the same behaviour as described above for SpotWare.

See Predefined Data and Programs - *System Modules SWUSRC*.

Internal sequence: The start signal can be sent to the timer immediately - if so configured - after the preweld supervision has been acknowledged. See Predefined Data and Programs - *System Module SWUSRC*.

Program stop and restart

Stop during the motion and restart

The robot stops on the path. If the signal to close the gun has already been sent the gun reopens at the stop.

On restart, the robot continues towards the programmed position, closes the gun again and the sequence in *SpotL* carries on as normal.

Stop during welding and restart

The welding is finished. The program pointer is in the next motion instruction. Validation of the weld is not done (ready/timeout).

On restart, the instruction is resumed by opening the gun, weld validation and then the normal sequence.

If SpotWare Plus is used, validation is done after the stop and after the gun has opened.

Instruction by instruction execution

Forwards

The motion and the welding are done.

Backwards

The gun is set to work or retract stroke depending on \Retract.
The motion is performed backwards.

Simulated welding

Full simulation of a timer.

Activated by setting *sw_sim_weld* TRUE. This will inhibit the start signal to the timer. The simulation time is defined in *sw_sim_time*. No preweld supervision is performed.

Simulation in the timer

Activated by setting *sw_inhib_weld* TRUE. This will set the *current_enable* signal low at the next weld. No preweld supervisions performed.

Error handling

Events in an error situation

When *SpotL* is stopped by a supervision, the following take place:

- The signal *process_error* is set.
- An error message is displayed.
- The error message is logged.

Error situations

The following error situations can occur:

- Instruction parameter error.
- Supervision error before welding.
- Weld error.
- Supervision error after welding and before the next motion.
- Supervision error for gun pressure.
- Supervision error for gun closure (SpotWare Plus)
- Supervision error for gun opening.

The supervision activities before and after welding can be modified by the user.

In SpotWare Plus all the supervision behaviour described above, except weld error and parameter error, can be modified by the user.

To stop execution with an error message, simply put the desired message in the assigned error string and it will show on the teach pendant.

The behaviour below is programmed as default.

See also Predefined Data and Programs.

Instruction parameter error

The error occurs when *SpotL* is called with faulty parameters. The program stops.

The parameter must be changed and the current instruction restarted from the beginning.

Supervision before welding

The supervisions in *sw_preweld_sup* are in progress. See Predefined Data and Programs - *System Module SWUSER* or *SWUSRC*.

Weld error

A weld error occurs if the ready signal from the weld timer has not been set to a certain time (*sw_wr_timeout*). *SpotL* can be configured to automatically reweld a certain number of times before the error is displayed and the execution stops, waiting for a manual action.

- The start signal is set low.
- The gun opens.
- The process error signal is set high.
- The following manual choices are available:

  Automatic mode: ***Service*** / ***Reweld***

  Manual mode: ***Service*** / ***Skip*** / ***Reweld*** (see the dialog box in Figure 2).

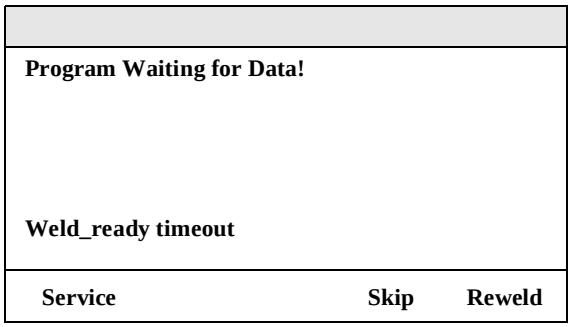


Figure 2 Dialog box for weld error.

Restart by choice Service

(If SpotWare Plus is used, this is not available in execution mode “stepwise forward”)

- The process error signal is reset.
- The user defined routine *sw_service_wf* is executing, for instance moving to a home position.
- The robot moves back to the spot welding point.
- Back to the dialog as shown in Figure 2.

Restart by choice Skip

- The *reset_fault* signal is pulsed.
- The process error signal is reset.
- The program execution is resumed but omitting the faulty weld.

Restart by manual action SwRunProc (SpotWare Plus only)

- Same result as *Reweld*.

Restart by choice Reweld

- The *reset_fault* signal is pulsed.
- The process error signal is reset.
- The gun closes.
- The start signal is set with a time delay of *sw_reset_time2* and the program execution is resumed.

Supervision after the weld

Supervisions in the user defined routine *sw_postweld_sup* are executing. See Predefined Data and Programs - *System Module SWUSER* or *SWUSRC*.

Gun closure error

The error occurs if the chosen pressure is not reached after a certain time. The signal *p1_ok* is supervised if *gun_pressure*=1 i spotdata etc.

- The process error signal is set high.
- The following manual choice is available: ***Service*** / ***Retry***.
- Choice ***Service***: the user defined *routine sw_service_cg* is executing.
- Back to the manual choice: ***Service*** / ***Retry***.
- Choice ***Retry***: The gun is closing and the pressure is tested again.

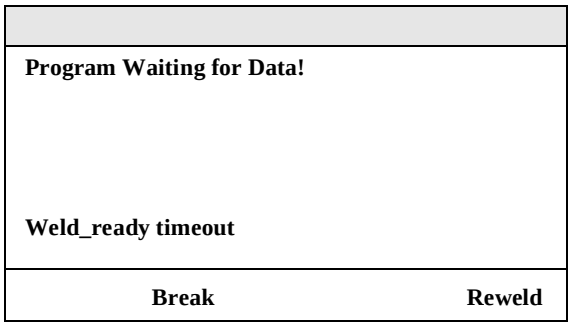
Gun opening error

The error occurs if the gun has not opened after a certain time. The signal *tip1_open* or *tip1_open* and *tip2_open* if *nof_tips*=2 in gundata.

- The process error signal is set high.
- The following manual choice is available: ***Service*** / ***Retry***
- Choice ***Service***: the user defined routine *sw_service_og* is executing.
- Back to the dialog shown in Figure 2.
- Choice ***Retry***: The gun is opening and the signals are tested again.

Error recovery when program instance 0 is stopped (SpotWare Plus)

When the program task 0 is stopped (user program stopped / stepwise execution / manual action) the service action is not a possible choice since it contains robot movements. Instead a “Break” choice is possible, which aborts the current process.



Manual actions (SpotWare Plus only)

Manual actions allow the execution of spot weld functions without having to program a SpotL-instruction. It can be used as a tool to test user defined code before programming the line program.

The manual actions execute the same process code as when running SpotL, i.e. full error recovery etc. is provided.

Manual actions execute in any system state except when a SpotL-instruction is in progress. Manual actions are activated by assigned virtual outputs.

See System Parameters - *Spot Welding*.

Communication

SpotL communicates with its equipment using parallel signals.

For a complete description of the I/O configuration, see System Parameters - *Spot Welding*.

Some weld timers with serial interface are supported. In those cases the SpotL parallel interface is still valid and the serial communication messages are mapped internally. See separate documentation.

Power failure handling

At system restart after power failure:

- All spotweld output signals are set to the old status, as well as gun close signals and weld start signals.

At program restart after power failure:

- The robot returns to the path and the program execution which was interrupted is continued.
- If a power failure occurred when a weld process was active, the current spot is automatically rewelded.

Syntax

SpotL
[ToPoint ‘:=’] < expression (**IN**) of *robtarget* > ‘,’
[Speed ‘:=’] < expression (**IN**) of *speeddata* > ‘,’
[Spot‘:=’] < persistent (**PERS**) of *spotdata* >
[‘\’ InPos]
[‘\’ NoConc]
[‘\’ Retract] ‘,’
[Gun‘:=’] < persistent (**PERS**) of *gundata*> ‘,’
[Tool ‘:=’] < persistent (**PERS**) of *tooldata* >
[’\’ WObj ’:=’ < persistent (**PERS**) of *wobjdata* >] ’;’

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data types - *speeddata*
Definition of zone data	Data types - *zonedata*
Definition of tool	Data types - *tooldata*
Definition of work objects	Data types - *wobjdata*
Definition of spot data	Data types - *spotdata*
Definition of gun data	Data types - *gundata*
Overview Spot welding	RAPID Summary - *Spot Welding*
Customising tools	Predefined data and programs - *System Module SwUser, SwUsrF, SwUsrC*
I/O configuration	System parameters - *Spot Welding*
Motion in general	Motion and I/O Principles
Coordinate systems	Motion and I/O Principles - *Coordinate systems*

System Module *SWUSER*

The system module SWUSER contains data and routines aimed at customising the behaviour of the SpotWare application.

The names are predefined and used internally when a *SpotL* instruction is used. They must therefore not be changed.

When standard SpotWare Plus is used, see the system modules SWUSRF and SWUSRC.

Contents

Data

The following global data are predefined:

Name	Declaration	Description
sw_weld_counter	PERS num sw_weld_counter := 0	Counter for welded spots. The counter is automatically incremented in the SpotL instruction.
sw_gp_timeout	CONST num sw_gp_timeout := 2	Gun pressure timeout [s]gh
sw_wr_timeout	CONST num sw_wr_timeout := 2	Weld ready timeout [s]
sw_go_timeout	CONST num sw_go_timeout := 2	Gun open timeout [s]
sw_prog_max	CONST num sw_prog_max := 63	Max value for prog_no in spotdata.
sw_inhib_weld	CONST bool sw_inhib_weld := FALSE	Flag to set the weld timer in simulation mode. If TRUE: The signal current_enable is cleared when next SpotL instruction is executed. If FALSE: The signal current_enable is set when next SpotL instruction is executed.

Name	Declaration	Description
sw_sim_weld	CONST bool sw_sim_weld := FALSE	Flag to internally simulate the weld timer. If TRUE: Weld start signal is not activated.
sw_sim_time	CONST num sw_sim_time := 0.5	Simulated weld time if sw_sim_weld = TRUE.
sw_aut_reweld	CONST num sw_aut_reweld := 0	Number of automatic tries to reweld after weld_ready timeout.
sw_parity	CONST num sw_parity:= 2	0 = none, 1 = odd, 2 = even
sw_reset_time1	CONST num sw_reset_time1 := 0.5	Reset pulse [s]
sw_reset_time2	CONST num sw_reset_time2 := 0.5	Wait time after the reset pulse and after each time current_enable is changed [s].
sw_inpos	CONST bool sw_inpos := FALSE	If TRUE: Gun preclosing is deactivated
sw_close_corr	VAR num sw_close_corr := 100	Correction factor used when the next gun preclose time is calculated [%]. Is automatically reset to 100 after each preclose calculation. Preferably this data can be changed in a separate assignment instruction just before the influenced SpotL instruction.
sw_aut_work_sel	CONST bool sw_aut_work_sel := TRUE	If TRUE: The work_select output is set to 1 when the gun is closed. If FALSE: The work_select output is not influenced when the gun is closed.
sw_start_type	CONST num sw_start_type:=0	Tells the system how to start the weld process in the timer: 0: The weld process start is triggered by a specific start signal (e.g. start1). The program number is set in advance. 1: The weld process is triggered by the program number outputs. No presetting of the program number

Normally it is suitable to use the module SWUSER to store variables of the type spotdata, gundata and tooldata, used for the application. The following variables are predefined but their names and values can be changed freely.

Name	Declaration
gun1	PERS gundata gun1 := [1, 4, TRUE, TRUE, 0, 0, 20000, 20000, 0.050, 0.050, 0.050, 0.050, 0, 0, 0, 0, 0]
toolg1	PERS tooldata toolg1 := [TRUE, [[0, 0, 0], [1, 0, 0, 0]], [-1, [0, 0, 0],[1, 0, 0, 0], 0, 0, 0]]
spot1	PERS spotdata spot1 := [1, 1, 1, 1]

Routines

There are two groups of predefined routines installed with the application. They are all used by the *SpotL* instruction.

These routines have a default functionality but can easily be changed.

User defined supervision routines:

sw_preweld_sup

The routine is executed before each start of the weld process.

If an error text is assigned to the text string errtext, then the system is automatically stopped with the error text on the display and with a possibility to restart the supervision routine and continue the program execution.

Default functionality:
If the simulate weld or inhibit weld functions are not activated, then the following digital input signals are tested:

timer_ready
flow_ok
temp_ok
current_ok

In the event of an error, a suitable error text is assigned to the text string errtext.

If timer_ready = 0, then one attempt to reset the timer is made and a new check of the input is made before an error is indicated.

sw_postweld_sup

The routine is executed after the weld process and before movement to the next position.

If an error text is assigned to the text string errtext, then the system is automatically stopped with the error text on the display and with a possibility to restart the supervision routine and continue the program execution.

Default functionality:
The work_select signal is set or reset according to the Retract switch in the instruction.

sw_bwd_sup

The routine is executed when the SpotL instruction is executed backwards, and before the movement to the previous position.

Default functionality:
The work_select signal is set or reset according to the Retract switch in the instruction.

User defined service routines:

After execution of a service routine, the robot moves to the interrupted weld position with reduced speed. The text “Service routine ready” and the old function keys are shown.

sw_service_cg (spotweld service close gun)

The routine is executed when the gun error “Gun pressure not reached” occurs and the function key “Service” is pressed.

By pressing Retry back from the service routine, a new attempt to close the gun is made and then program execution continues.

Default functionality:
No functions, only an information text and a “Return” key are shown. The service routine is ended when “Return” is pressed.

sw_service_og (spotweld service open gun)

The routine is executed when the gun error “Gun open timeout” occurs and the function key “Service” is pressed.

By pressing Retry back from the service, a new test will be made if the gun is open, and then program execution continues.

Default functionality:
No functions, only an information text and a “Return” key are shown. The service routine is ended when “Return” is pressed.

sw_service_wf (spotweld service weld fault)

The routine is executed when the error “Weld ready timeout” occurs and the function key “Service” is pressed.

By pressing Reweld back from the service, a new attempt to weld the spot is made and then program execution is continued.

Default functionality:
No functions, only an information text and a “Return” key are shown. The service routine is ended when “Return” is pressed.

System Module *SWUSRC*

The system module SWUSRC contains data and routines aimed at customising the behaviour of the SpotWare Plus application. They are used and executed commonly by all RAPID tasks of SpotWare.

NB. Any change of routines in SWUSRC requires the following three steps to affect the application (see also System Parameters Multitasking):

- **save to ramdisk**
- **touch the multitasking configuration**
- **WARM START to affect the application**

The names are predefined and used internally when a *SpotL* instruction is used. They must therefore not be changed.

If standard SpotWare is used, see the system module SWUSER.

Contents

Data

The following global data are predefined:

Name	Declaration	Description
sw_weld_counter	PERS num sw_weld_counter := 0	Counter for welded spots. The counter is automatically incremented in the SpotL instruction.
sw_gp_timeout	PERS num sw_gp_timeout := 2	Gun pressure timeout [s]
sw_wr_timeout	PERS num sw_wr_timeout := 2	Weld ready timeout [s]
sw_go_timeout	PERS num sw_go_timeout := 2	Gun open timeout [s]
sw_prog_max	PERS num sw_prog_max := 63	Max. value for prog_no in spotdata.

Name	Declaration	Description
sw_inhib_weld	PERS bool sw_inhib_weld := FALSE	Flag to set the weld timer in simulation mode. If TRUE: The signal current_enable is cleared when the next SpotL instruction is executed. If FALSE: The signal current_enable is set when the next SpotL instruction is executed.
sw_sim_weld	PERS bool sw_sim_weld := FALSE	Flag to internally simulate the weld timer. If TRUE: Weld start signal is not activated.
sw_sim_time	PERS num sw_sim_time := 0.5	Simulated weld time if sw_sim_weld = TRUE.
sw_aut_reweld	PERS num sw_aut_reweld := 0	Number of automatic tries to reweld after weld_ready timeout.
sw_parity	PERS num sw_parity:= 0	0 = none, 1 = odd, 2 = even
sw_reset_time1	PERS num sw_reset_time1 := 0.5	Reset pulse [s]
sw_reset_time2	PERS num sw_reset_time2 := 0.5	Wait time after the reset pulse and after each time current_enable is changed [s].
sw_async_weld	PERS num sw_async_weld := FALSE	TRUE means the start of the weld is set independently of inpos, i.e. immediately after closing the gun
sw_servo_corr	PERS num sw_servo_corr := 0.061	Correction time for internal delay. This value affects the preclosing time. If the dynamic step resolution is increased this value shall be decreased.
sw_start_type	PERS num sw_start_type:=0	Tells the system how to start the weld process in the timer: 0: The weld process start is triggered by a specific start signal (e.g. start1). The program number is set in advance. 1: The weld process is triggered by the program number outputs. No presetting of the program number

The following predefined data is used by the manual actions to operate on:

Name	Declaration
sw_man_gun	PERS gundata gun1 := [1, 4, TRUE, TRUE, 0, 0, 20000, 20000, 0.050, 0.050, 0.050, 0.050, 0, 0, 0, 0, 0]
sw_man_spot	PERS spotdata spot1 := [1, 1, 1, 1];
sw_man_retr	PERS num sw_man_retr := 0;

Routines

The following predefined routines are installed with the application. They are all used by the *SpotL* instruction.

These routines have a default functionality but can easily be changed.

User defined supervision routines for SpotL:

The following routines are called by the SpotL sequence.

PROC sw_preweld_sup ()

spotweld preweld supervision

The routine is executed before each start of the weld process.

If an error text is assigned to the text string errtext, then the system is automatically stopped with the error text on the display and with a possibility to restart the supervision routine and continue the program execution.

Default functionality:
If the simulate weld or inhibit weld functions are not activated, then the following digital input signals are tested:

timer_ready
flow_ok
temp_ok
current_ok

In the event of an error, a suitable error text is assigned to the text string errtext.

If timer_ready = 0, then one attempt to reset the timer is made and a new check of the input is made before an error is indicated.

PROC sw_postweld_sup ()

spotweld postweld supervision

The routine is executed after the weld process before movement to the next position.

If an error text is assigned to the text string errtext, then the system is automatically stopped with the error text on the display and with a possibility to restart the supervision routine and continue the program execution.

Default functionality:
The work_select signal is set or reset according to the Retract switch in the instruction.

User defined independent supervision routines:

The following routines are called independently of the SpotL-sequence.

PROC sw_sup_init ()

spotweld supervision init

The routine is executed at warm start. Here the user interrupt numbers used by sw_sup_trap should be initialised.

Default functionality:
Connection and activation of signals and interrupts used by *sw_sup_trap*.

PROC sw_motor_on ()

spotweld motor on

Routine called by the trap *sw_sup_trap*.

Default functionality:
Connected to interrupt number imotor_on. Set weld_power and pulse water_on outputs.

PROC sw_motor_off ()

spotweld motor off

Routine called by the trap *sw_sup_trap*.

Default functionality:
Connected to interrupt number imotor_off. Reset weld_power output.

PROC sw_proc_ok ()

spotweld process ok

Routine called by the trap *sw_sup_trap*.

Default functionality:
Connected to interrupt number iproc_ok. Set weld_power and pulse water_on outputs.

PROC sw_proc_error ()

spotweld process error

Routine called by the trap *sw_sup_trap*.

Default functionality:
Connected to interrupt number iproc_error. Reset weld_power output.

PROC sw_sup_curren ()

spotweld supervise current enable

Routine called by the trap *sw_sup_trap*.

Default functionality:
Connected to interrupt number icurr_enable. Set weld_power and pulse water_on outputs.

PROC sw_sup_currdis ()

spotweld supervise current disable

Routine called by the trap *sw_sup_trap*.

Default functionality:
Connected to interrupt number icurr_disable. Reset weld_power output.

User defined trap routines:

The interrupts in these trap routines will always be alert to execute independently of the actual system state.

TRAP sw_sup_trap ()

spotweld supervision trap

Trap routine connected to the supervision.

Default functionality:
Control of the output signals water_start and weld_power depending on the system state. See System Parameters - *Spotwelding*.

User defined process routines for SpotL:

The following routines are called by the SpotL sequence.

PROC sw_close_gun ()

spotweld close gun

The routine is executed each time a closing of the gun is ordered. N.B. also the manual action.

If an error text is assigned to the text string errtext_close, then the system is automatically stopped with the error text on the display and with a possibility to restart the supervision routine and continue the program execution.

Default functionality:
See Instructions - *SpotL*.

PROC sw_open_gun (num context)

spotweld open gun

The routine is executed each time an opening of the gun is ordered. N.B. also the manual action.

Parameters:

- **context**: reason for opening. A negative value is used to leave out the gun open tests and to ignore the errtext_open when a quick open is needed in an error situation.

If an error text is assigned to the text string errtext_open, then the system is automatically stopped with the error text on the display and with a possibility to restart the supervision routine and continue the program execution.

Default functionality:
See Instructions - *SpotL*.

PROC sw_set_pressure ()

spotweld set pressure

The routine is executed in the preparation phase of the spot weld process.

Default functionality:
Sets the pressure output group according to gun_pressure. See Instructions - *SpotL*.

System Module SWUSRF

The system module SWUSRF contains data and routines aimed at customising the behaviour of the SpotWare Plus application. They are used and executed by the RAPID foreground task of SpotWare.

The routine names are predefined and used internally when a *SpotL* instruction is used. They must therefore not be changed.

If standard SpotWare is used, see the system module SWUSER.

Contents

Data

The following global data are predefined:

Name	Declaration	Description
sw_inpos	PERS bool sw_inpos := FALSE	If TRUE: Gun preclosing is deactivated.
sw_close_corr	VAR num sw_close_corr := 100	Correction factor used when the next gun preclose time is calculated [%]. Is automatically reset to 100 after each preclose calculation. Preferably this data can be changed in a separate assignment instruction just before the influenced SpotL instruction.

Normally it is suitable to use the module SWUSRF to store variables of the type spotdata, gundata and tooldata, used for the application. The following variables are predefined but their names and values can be changed freely.

Name	Declaration
gun1	PERS gundata gun1 := [1, 4, TRUE, TRUE, 0, 0, 20000, 20000, 0.050, 0.050, 0.050, 0.050, 0, 0, 0, 0, 0]
toolg1	PERS tooldata toolg1 := [TRUE, [[0, 0, 0], [1, 0, 0, 0]], [-1, [0, 0, 0],[1, 0, 0, 0], 0, 0, 0]]
spot1	PERS spotdata spot1 := [1, 1, 1, 1]

Routines

The following predefined routines are installed with the application. They are all used by the *SpotL* instruction.

These routines have a default functionality but can easily be changed.

User defined service routines:

After execution of a service routine, the robot moves to the interrupted weld position with reduced speed. The text “Service routine ready” and the old function keys are shown.

PROC sw_service_cg ();

spotweld service close gun

The routine is executed when the gun error “Gun pressure not reached” occurs and the function key “Service” is pressed.

By pressing Retry back from the service routine, a new attempt to close the gun is made and then program execution continues.

Default functionality:
No functions, only an information text and a “Return” key are shown. The service routine is ended when “Return” is pressed.

PROC sw_service_og ();

spotweld service open gun

The routine is executed when the gun error “Gun open timeout” occurs and the function key “Service” is pressed.

By pressing Retry back from the service, a new test will be made if the gun is open, and then program execution continues.

Default functionality:
No functions, only an information text and a “Return” key are shown. The service routine is ended when “Return” is pressed.

PROC sw_service_wf ();

spotweld service weld fault

The routine is executed when the error “Weld ready timeout” occurs and the function key “Service” is pressed.

By pressing Reweld back from the service, a new attempt to weld the spot is made and then program execution is continued.

Default functionality:
No functions, only an information text and a “Return” key are shown. The service routine is ended when “Return” is pressed.

General user routines and functions:

The following routines are connected to different entries in the SpotWare RAPID base software.

FUNC num sw_close_time (spotdata spot, gundata gun);
spotweld close time calculation

The function is executed during motion preparation. It returns the preclosing time for the gun.

Input parameters:
spot of type spotdata: actual spot given in the SpotL-instruction in preparation
gun of type gundata: actual gun given in the SpotL-instruction in preparation

Default functionality:
The resulting time is dependent on the four pressure levels in gundata.

System Module *SWTOOL*

The system module SWTOOL contains data functions and routines. The module is declared NOVIEW and contains utilities to be used as a toolbox when customising SpotWare Plus. This module is accessible commonly by all RAPID tasks of SpotWare Plus.

Contents

Routines

The following predefined routines are installed with the application. They are used to influence *SpotL* instruction.

PROC SwSetCurrSpot (spotdata spot)

SpotWeldSetCurrentSpot.

The function changes the spotdata parameter of the running spotware process.

PROC SwSetCurrGun (gundata gun)

SpotWeldSetCurrentGun

The function changes the gundata parameter of the running spotware process.

PROC SwSetCurrRetr (num retr)

SpotWeldSetCurrentRetract

The function changes the retract switch parameter of the running spotware process.

retr 0: work stroke

retr 1: retract stroke

PROC SwGetCurrNoConc (bool noconc)

SpotWeldSetCurrentNoConcurrency

The function changes the NoConc switch parameter of the running spotware process.

noconc TRUE: NoConc is present

noconc FALSE: NoConc is not present

Functions

The following predefined functions are installed with the application.

FUNC spotdata SwGetCurrSpot

SpotWeldGetCurrentSpot.

The function returns the content of the spotdata parameter of the running spotware process.

FUNC gundata SwGetCurrGun

SpotWeldGetCurrentGun

The function returns the content of the gundata parameter of the running spotware process.

FUNC num SwGetCurrRetr

SpotWeldGetCurrentRetract

The function returns the content of the retract switch parameter of the running spotware process.

Return value 0: work stroke

Return value 1: retract stroke

FUNC bool SwGetCurrNoConc

SpotWeldGetCurrentNoConcurrency

The function returns the content of the NoConc switch parameter of the running spotware process.

Return value TRUE: NoConc is present

Return value FALSE: NoConc is not present

FUNC num SwWaitInput (VAR signaldi input, num value \num MaxWait)

SpotWeldWaitInput

Waits until the signal *input* has reached *value*. A max. time *MaxWait* can be added as an optional parameter. When running a user routine called by the spot weld process, in which the process is supposed to wait for an input signal, this function is used to let the system abandon the current process.

Return value SW_OK: Signal was set to output *value*.

Return value SW_TIMEOUT: The time specified in MaxWait has been exceeded.

Return value SW_CANCEL: The SpotWare process has received an abort and wants to cancel the current process. It shall cause a return from the current user routine.

FUNC num SwWaitOutput (VAR signaldo output, num value \num MaxWait)

SpotWeldWaitOutput

Waits until the signal output has reached *value*. A maximum time MaxWait can be added as an optional parameter. When running a user routine called by the spotweld process, in which the process is supposed to wait for an output signal, this function is used to let the system abandon the current process.

Return value SW_OK: Signal was set to *value*.

Return value SW_TIMEOUT: The time specified in MaxWait has been exceeded.

Return value SW_CANCEL: The SpotWare process has received an abort and wants to cancel the current process. It shall cause a return from the current user routine.

ggundata Gluing gun data

Ggundata is used to define gluing gun specific data, which is then to be used to control the gun in an optimal way during the gluing process.

Note that the TCP and weight of the gluing gun are defined in *tooldata*.

Description

Ggundata is used in glue instructions and has the following structure:

- Which gun is to be used (1 or 2).
- Time for preopening and preclosing the gun.
- Type of flow1 and flow2.
- Times for presetting of flow1 and flow2 on.
- Times for presetting flow1 and flow2 changes.
- Times for presetting of flow1 and flow2 off.
- Times for lag in glue gun for speed dips
- Value for the speed, at which the logical maximum value for the analog outputs shall be set.

Components

ggun_no *(number of gun to use)* Data type: *num*

Number to define if this ggundata is for gun 1 or gun 2.

gl_on_time *(preopen time for the gun)* Data type: *num*

Time in s needed for the gun to open.

gl_off_time *(preclose time for the gun)* Data type: *num*

Time in s needed for the gun to close.

fl1_type *(type of flow1)* Data type: *num*

Type of flow1 signal, set to none, fixed or speed proportional, where none = 0, fixed = 1 and speed proportional = 2.

fl1_on_time *(preset on time flow1)* Data type: *num*

Time in s needed to set up the flow1 in the gun when the instruction is programmed with the *\On* argument.

fl1_time *(preset change time flow1)* Data type: *num*

Time in s needed to set up the flow1 in the gun when the instruction is programmed without the *\On* and *\Off* argument (i.e. time needed to change the flow from one specific value to another).

fl1_off_time *(preset off time flow1)* Data type: *num*

Time in s needed to reset the flow1 in the gun when the instruction is programmed with the *\Off* argument.

fl1_delay *(lag delay flow1)* Data type: *num*

Time in s to compensate for the lag in the gluegun for TCP speed dips.

fl1_refspeed *(glue reference speed)* Data type: *num*

Glue reference speed in mm/s. Normally the max. used glue speed for this gun. Used in calculation of the flow1 value for speed proportional signals. This value must be > 0 also when flow type=fixed.

fl2_type *(type of flow2)* Data type: *num*

Type of flow2 signal, set to none, fixed or speed proportional, where none = 0, fixed = 1 and speed proportional = 2.

fl2_on_time *(preset on time flow2)* Data type: *num*

Time in s needed to set up the flow2 in the gun when the instruction is programmed with the *\On* argument.

fl2_time *(preset change time flow2)* Data type: *num*

Time in s needed to set up the flow1 in the gun when the instruction is programmed without the *\On* and *\Off* argument (i.e. time needed to change the flow from one specific value to another).

fl2_off_time *(preset off time flow2)* Data type: *num*

Time in s needed to reset the flow2 in the gun when the instruction is programmed with the *\Off* argument.

fl2_delay *(lag delay flow2)* Data type: *num*

Time in s to compensate for the lag in the gluegun for TCP speed dips.

fl2_refspeed *(glue reference speed)* Data type: *num*

Glue reference speed in mm/s. Normally the tax. used glue speed for this gun. Used in calculation of the flow2 value for speed proportional signals. This value must be > 0 also when flow type=fixed.

Limitations

Max. number of guns: 2

Max. real lag compensation (*fl1_delay* and *fl2_delay*): 60 - 80 ms

The values for the different times within the dataset must be between 0 to 1 seconds.

The value for the reference speed has to be > 0.

Predefined data

The predefined data *ggun1* defines the use of gun no *1*, a flow1 and flow2 of type *2*, i.e. proportional flow values are used, all belonging times are set to zero and the reference speed is *1000* mm/s for each flow.

PERS ggun1 := [1,0,0,2,0,0,0,0,1000,2,0,0,0,0,1000];

Structure

< dataobject of *ggundata*>
  <*ggun_no* of *num*>
  <*gl_on_time* of *num*>
  <*gl_off_time* of *num*>
  <*fl1_type* of *num*>
  <*fl1_on_time* of *num*>
  <*fl1_time* of *num*>
  <*fl1_off_time* of *num*>
  <*fl1_delay* of *num*>
  <*fl1_refspeed* of *num*>
  <*fl2_type* of *num*>
  <*fl2_on_time* of *num*>
  <*fl2_time* of *num*>
  <*fl2_off_time* of *num*>
  <*fl2_delay* of *num*>
  <*fl2_refspeed* of *num*>

Related information

	Described in:
Glue instruction	Instructions - *GlueL / GlueC*

GlueC Gluing with circular motion

GlueC (GlueCircular) is used in gluing to control the motion, gun opening and the gluing process. *GlueC* moves the TCP on a circular path to the end position.

Example 1

```
GlueL \On, p1, v250, ggun1 \F1:=100 \F2:=80, z30, tool7;
GlueC p2, p3, v250, ggun1 \F1:=90 \F2:=70, z30, tool7;
GlueL \Off, p4, v250, ggun1, z30, tool7;
```

1. The TCP for *tool7* is moved on a linear path to the position *p1* with the speed given in *v250*. Due to the *\On* argument the gun opens and the glue flow is started according to the data given in *ggundata ggun1* in advance on its way to *p1*. The glue flow is started with the percentage values given by the *\F1:=100* and *\F2:=80* parameters.

2. The TCP is then moved from *p1* towards *p3* with the flow values given by the preceding glue instruction. Before *p3* is reached, the flow values are changed to 90% and 70% respectively. The time when that is performed is specified in *ggun1*.

3. The TCP is then moved from *p3* towards *p4* with the flow values given by the preceding glue instruction. Due to the *\Off* argument the outputs will be reset according to the times given in *ggun1* before *p3* is reached.

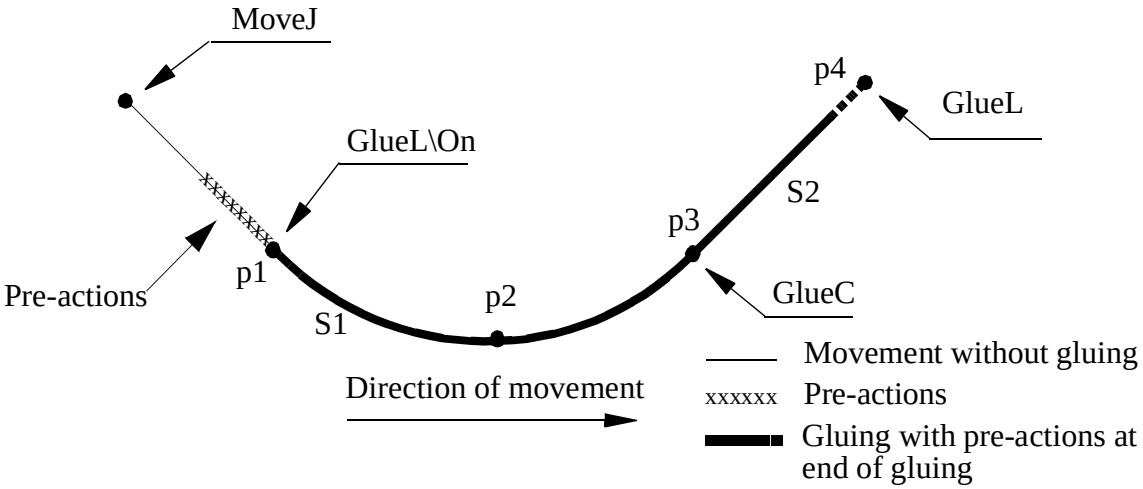


Figure 1 Glue example 1

S1: Flow1=100, Flow2=80, i.e. the glue values given by the first instruction are active.
S2: Flow1=90, Flow2=70, i.e. the glue values given by the second instruction are active.

Arguments

GlueC [\On|]\[Off] [\Conc] CirPoint ToPoint Speed Gluegun [\F1] [\F2] [\D] Zone Tool [\WObj]

[\On] Data type: *switch*

The argument *On* is used in the first Glue instruction to start the glue process (see Figure 1).

The argument may only be used in the first glue instruction to perform the necessary gun opening and setting of the flow in advance. Executing two consecutive instructions with *On* argument will result in an error message.

As the instruction cannot contain the arguments *On* and *Off* together, the gluing path must have at least 2 instructions, one containing the *On* and one containing the *Off* argument.

The pre-actions, which are performed on the path to the programmed position, are setting of the gun opening output and setting of the analog glue outputs.

[\Off] Data type: *switch*

The argument *Off* is used in the last gluing instruction to terminate the gluing when the programmed position is reached. On the way to the end position the output for opening the gun as well as the flow outputs will be reset according to the given time within the specified ggundata.

So it is not possible to terminate gluing without using the *Off* argument during movements.

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed while the robot is moving. This argument is used in the same situations as corresponding argument in other Move instructions but for gluing it is also useful to permit higher speeds when consecutive glue instructions are close to each other.

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 zone.

CirPoint Data type: *robtarget*

The circle point of the robot. The circle point is a point on the circle between the start point and the destination point. To obtain the best accuracy, it should be placed about halfway between the start and destination points. If it is placed too close to the start or end point, the robot may give a warning. The circle point is defined as a named position or stored directly in the instruction (if marked with

an * in the instruction).

ToPoint Data type: *robtarget*

The destination point of the robot and the external axes. It is defined as a named position or stored directly in the instruction (if marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity for the tool centre point, the tool reorientation and external axes.

GlueGun Data type: *ggundata*

Gun specific data, e.g. the reaction time for setting a flow etc., for the gun in use.

[\F1] *(Flow1)* Data type: *num*

The argument *F1* gives a value in percent to adjust the flow1 for the next part of the path. If no value is programmed the same value as in previous glue instruction is used. If no value is programmed in a glue instruction with *On* argument the value 0 is used.

[\F2] *(Flow2)* Data type: *num*

The argument *F2* gives a value in percent to adjust the flow2 for the next part of the path. If no value is programmed the same value as in previous glue instruction is used. If no value is programmed in a glue instruction with *On* argument the value 0 is used.

[\D] *(Distance)* Data type: *num*

The optional argument *D* gives the possibility to perform all pre-actions during the instruction the given distance (mm) in advance of the programmed position.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination position, and should be the position of the tip when the gun is open.

[\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 in order to perform a linear movement relative to the work object.

Customizing

The GlueWare package provides opportunities for the user to customize the *GlueL* instruction:

- by user defined data which affects the internal behaviour in *GlueL*. (See Predefined Data and Programs - *System Module GLUSER*).
- by changing the I/O configuration. (See System Parameters - *Gluing*).

However, the main subject of this *GlueC* instruction description is the default setup.

Program execution

Internal sequence in a *GlueC* instruction:

- The gun starts to move towards the position.
- If the argument *\On* is used the gun opening output *DO_Ggun1* or *DO_Ggun2* is set at the specified time before the position is reached. If the argument *\D* is used the output is set at the specified distance plus the specified times before the position is reached.
- The values for flow1 and flow2 are set at the analog outputs *AO_G1Flow1* and AO_G1F*low2,* and *AO_G2Flow1* and AO_G2F*low2* respectively at the specified time before the position is reached. If the argument *\D* is used the outputs are set at the specified distance plus the specified times before the position is reached.
- When the programmed position is reached, the program execution continues with the next instruction.

Calculation of the glue flow values

See corresponding description of GlueL

Program stop and restart

Stop during the glue process

If a stop occurs when the glue process is active, the gluing outputs are cleared and an error message is displayed. When the restart is ordered, the remaining instructions in the current set of glue instructions are performed as normal positioning instructions. The gluing is restarted with the next glue instruction with an *\On* argument.

Instruction by instruction execution

Forwards

The gun is closed and motion without gluing is done.

Backwards

The gun is closed and the motion is performed backwards without gluing.

Simulated gluing

Activated by setting the variable *gl_sim_glue* to TRUE. This will inhibit the gun opening and the flow signals. (See Predefined Data and Programs - *System Module GLUSER*)

Limitations

It is not possible to restart the current glue sequence after a stop. The gluing is restarted with the next glue instruction with an *On* argument.

Error handling

Error situations

The following error situations are handled:

- Instruction argument error.
- Wrong *ggundata* values.
- Start without *On* argument.
- Start with two instructions with *On* argument.
- End with *Off* argument without having started with *On* argument.
- Stop during execution of glue instructions.

The faulty instruction or data must be changed and the current set of glue instructions must be restarted from the beginning.

Syntax

```
GlueC
   [[‘\’On ]|[‘\’Off ]
   [’\’ Conc]’,’]
   [CirPoint‘:=’]<expression (IN) of robtarget>‘,’
   [ToPoint‘:=’]<expression (IN) of robtarget>‘,’
   [Speed‘:=’]<expression (IN) of speeddata>’,’
   [Gluegun‘:=’]<persistent (PERS) of ggundata>
                  [’\’F1’:=’<expression (IN) of num>]
                  [’\’F2’:=’<expression (IN) of num>]
                  [’\’D’:=’<expression (IN) of num>]’,’
   [Zone‘:=’]<expression (IN) of zonedata>’,’
   [Tool‘:=’]<persistent (PERS) of tooldata>
   [’\’WObj’:=’<persistent (PERS) of wobjdata>]’;’
```

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data types - *speeddata*
Definition of zone data	Data types - *zonedata*
Definition of tool	Data types - *tooldata*
Definition of work objects	Data types - *wobjdata*
Definition of gluegun data	Data types - *ggundata*
Overview Gluing	RAPID Summary - *GlueWare*
Customizing tools	Predefined data and programs - System Module *GLUSER*
I/O configuration	System parameters - *GlueWare*
Motion in general	Motion and I/O Principles

GlueL Gluing with linear motion

GlueL (GlueLinear) is used in gluing to control the motion, gun opening and the gluing process. *GlueL* moves the TCP linearly to the end position.

Example 1

```
GlueL   \On, p1, v250, ggun1 \F1:=100 \F2:=80, z30, tool7;
GlueL   p2, v250, ggun1 \F1:=90 \F2:=70, z30, tool7;
GlueL   \Off, p3, v250, ggun1, z30, tool7;
```

1. The TCP for *tool7* is moved on a linear path to the position *p1* with the speed given in *v250*. Due to the *\On* argument the gun opens and the glue flow is started according to the data given in *ggundata ggun1* in advance on its way to *p1*. The glue flow is started with the percentage values given by the *\F1:=100* and *\F2:=80* parameters.

2. The TCP then is moved from *p1* towards *p2* with the flow values given by the preceding glue instruction. Before *p2* is reached, the flow values are changed to 90% and 70% respectively. The time when that is performed is specified in *ggun1*.

3. The TCP then is moved from *p2* towards *p3* with the flow values given by the preceding glue instruction. Due to the *\Off* argument the outputs will be reset according to the times given in *ggun1* before *p3* is reached.

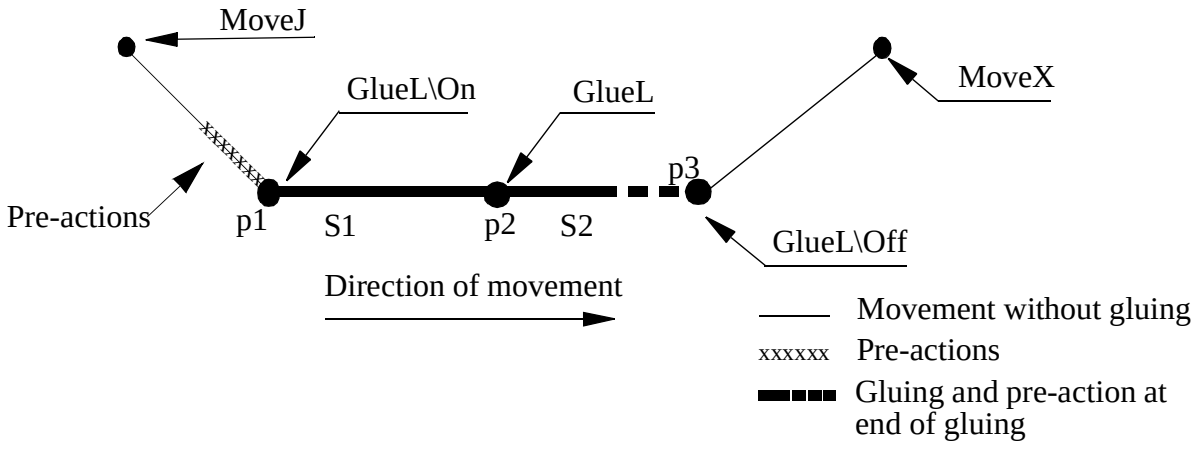


Figure 1 Glue example 1

S1: Flow1=100, Flow2=80, i.e. the glue values given by the first instruction are active.
S2: Flow1=90, Flow2=70, i.e. the glue values given by the second instruction are active.

Arguments

GlueL [\On]|[\Off] [\Conc] ToPoint Speed GlueGun [\F1] [\F2] [\D] Zone Tool [\WObj]

[\On] Data type: *switch*

The argument *On* is used in the first Glue instruction to start the glue process (see Figure 1).

The argument may only be used in the first glue instruction to perform the necessary gun opening and setting of the flow in advance. Executing two consecutive instructions with *On* argument will result in an error message.

As the instruction cannot contain the arguments *On* and *Off* together, the gluing path must have at least 2 instructions, one containing the *On* and one containing the *Off* argument.

The pre-actions, which are performed on the path to the programmed position, are setting of the gun opening output and setting of the analog glue outputs.

[\Off] Data type: *switch*

The argument *Off* is used in the last gluing instruction to terminate the gluing when the programmed position is reached. On the way to the end position the output for opening the gun as well as the flow outputs will be reset according to the given time within the specified ggundata.

So it is not possible to terminate gluing without using the *Off* argument during movements.

[\Conc] *(Concurrent)* Data type: *switch*

Subsequent instructions are executed while the robot is moving. This argument is used in the same situations as corresponding argument in other Move instructions but for gluing it is also useful to permit higher speeds when consecutive glue instructions are close to each other.

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 zone.

ToPoint Data type: *robtarget*

The destination point of the robot and the external axes. It is defined as a named position or stored directly in the instruction (if marked with an * in the instruction).

Speed Data type: *speeddata*

The speed data that applies to movements. Speed data defines the velocity for the tool centre point, the tool reorientation and external axes.

GlueGun Data type: *ggundata*

Gun specific data, e.g. the reaction time for setting a flow etc., for the gun in use (see *Datatypes - Ggundata*).

[\F1] *(Flow1)* Data type: *num*

The argument *\F1* gives a value in percent to adjust the flow1 for the next part of the path. If no value is programmed the same value as in previous glue instruction is used. If no value is programmed in a glue instruction with *\On* argument the value 0 is used.

[\F2] *(Flow2)* Data type: *num*

The argument *\F2* gives a value in percent to adjust the flow2 for the next part of the path. If no value is programmed the same value as in previous glue instruction is used. If no value is programmed in a glue instruction with *\On* argument the value 0 is used.

[\D] *(Distance)* Data type: *num*

The optional argument *\D* gives the possibility to perform all pre-actions given the distance (mm) in advance of the programmed position.

Zone Data type: *zonedata*

Zone data for the movement. Zone data describes the size of the generated corner path.

Tool Data type: *tooldata*

The tool in use when the robot moves. The tool centre point is the point moved to the specified destination position, and should be the position of the tip when the gun is open.

[\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 in order to perform a linear movement relative to the work object.

Customizing

The GlueWare package provides opportunities for the user to customize the *GlueL* instruction:

- by user defined data which affects the internal behaviour in *GlueL*. (See Predefined Data and Programs - *System Module GLUSER*).
- by changing the I/O configuration. (See System Parameters - *Gluing*).

However, the main subject of this *GlueL* instruction description is the default setup.

Program execution

Internal sequence in a *GlueL* instruction:

- The gun starts to move towards the position.
- If the argument *\On* is used the gun opening output *DO_Ggun1* or *DO_Ggun2* is set at the specified time before the position is reached. If the argument *\D* is used the output is set at the specified distance plus the specified times before the position is reached.
- The values for flow1 and flow2 are set at the analog outputs *AO_G1Flow1* and AO_G1F*low2,* and *AO_G2Flow1* and AO_G2F*low2* respectively at the specified time before the position is reached. If the argument *\D* is used the outputs are set at the specified distance plus the specified times before the position is reached.
- When the programmed position is reached, the program execution continues with the next instruction.

Calculation of the glue flow values

The following data is used when the logical flow1 value is calculated (The logical flow2 is calculated in a similar way):

F1	instruction parameter (0-100)
gl_fl1_ovr	global override (Default: 100). (See Predefined Data and Programs - *System Module GLUSER*)
gl_fl1_ref	logical max. value (Default: 1000). (See Predefined Data and Programs - *System Module GLUSER*)
current speed	current robot speed (mm/s).
fl1_refspeed	glue reference speed (mm/s). (See Data Types- *ggundata*)

Calculation of logical flow1 when fl1_type = 1 (fixed):

logical flow1 = F1 * gl_fl1_ovr * gl_fl1_ref/ 10000

Calculation of logical flow1 when fl1_type = 2 (proportional):

logical flow1 = (F1 * gl_fl1_ovr * gl_fl1_ref/ 10000) * current speed/fl1_refspeed

This means: With the default values above and with the default setup for logical max. and min. for the analog glue outputs (See System Parameters - *GlueWare*) we get following result:

If flow1_type = 1 (fixed): Physical max. value is activated if F1 = 100 (%) in the instruction.

If flow1_type = 2 (proportional): Physical max. value is activated if F1 = 100 (%) in the instruction and the actual speed is the same as fl1_refspeed in current ggundata..

Example 2

```
GlueL   \On, p1, v250, ggun1 \F1:=100 \F2:=80 \D:=50, z30, tool7;
GlueL   p2, v250, ggun1 \F1:=90 \F2:=70 \D:=50, z30, tool7;
GlueL   \Off, p3, v250, ggun1 \D:=50, z30, tool7;
```

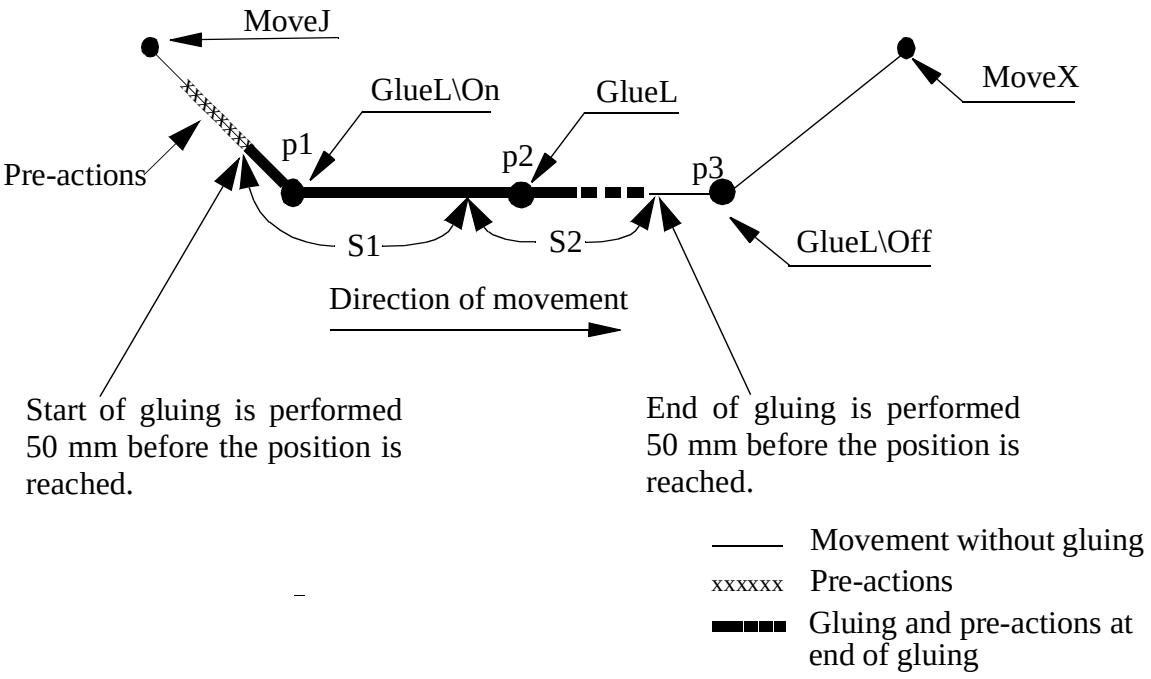


Figure 2 Glue example 2

In this example everything is the same as in example 1 except that all glue specific actions are executed an additional distance (50 mm) before the programmed positions are reached.

Assume that the logical maximum value for the signals is 10. When fl1_type=1 (fixed) and gl_fl1_our=100 (normal value for glue flow override), the values 10 on stretch S1 and 9 on stretch S2 are obtained for flow 1. When gl_fl1.our is changed to 50, the values 500 and 450 respectively will be obtained. When fl2_type=2 (proportional) and gl_fl2_our=100, the values 800 on stretch S1 and 700 on stretch S2 are obtained for flow 2, if the current speed is the same as the reference speed (fl2_refspeed).

When the speed is reduced, e.g. in a corner path , the flow will also be reduced correspondingly. The physical values of the signals are also determined by how the analog signals are configured in the system parameters (relationship between physical and logical values).

Program stop and restart

Stop during the glue process

If a stop occurs when the glue process is active, the gluing outputs are cleared and an error message is displayed. When the restart is ordered, the remaining instructions in the current set of glue instructions are performed as normal positioning instructions.

The gluing is restarted with the next glue instruction with an *\On* argument.

Instruction by instruction execution

Forwards

The gun is closed and motion without gluing is done.

Backwards

The gun is closed and the motion is performed backwards without gluing.

Simulated gluing

Activated by setting the variable *gl_sim_glue* to TRUE. This will inhibit the gun opening and the flow signals. (See Predefined Data and Programs - *System Module GLUSER*)

Limitations

It is not possible to restart the current glue sequence after a stop. The gluing is restarted with the next glue instruction with an *\On* argument.

Error handling

Error situations

The following error situations are handled:

- Instruction argument error.
- Wrong *ggundata* values.
- Start without *\On* argument.
- Start with two instructions with *\On* argument.
- End with *\Off* argument without having started with *\On* argument.
- Stop during execution of glue instructions.

The faulty instruction or data must be changed and the current set of glue instructions must be restarted from the beginning.

Syntax

GlueL
[[‘\’On]|[‘\’Off]
[’\’ Conc]’,’]
[ToPoint‘:=’]<expression (**IN**) of *robtarget*>‘,’
[Speed‘:=’]<expression (**IN**) of *speeddata*>’,’
[Gluegun‘:=’]<persistent (**PERS**) of *ggundata*>
[’\’F1’:=’<expression (**IN**) of *num*>]
[’\’F2’:=’<expression (**IN**) of *num*>]
[’\’D’:=’<expression (**IN**) of *num* >]’,’
[Zone‘:=’]<expression (**IN**) of *zonedata*>’,’
[Tool‘:=’]<persistent (**PERS**) of *tooldata*>
[’\’WObj’:=’<persistent (**PERS**) of *wobjdata*>]’;’

Related information

	Described in:
Other positioning instructions	RAPID Summary - *Motion*
Definition of velocity	Data types - *speeddata*
Definition of zone data	Data types - *zonedata*
Definition of tool	Data types - *tooldata*
Definition of work objects	Data types - *wobjdata*
Definition of gluegun data	Data types - *ggundata*
Overview Gluing	RAPID Summary - *GlueWare*
Customizing tools	Predefined data and programs - System Module *GLUSER*
I/O configuration	System parameters - *GlueWare*
Motion in general	Motion and I/O Principles

System Module *GLUSER*

The system module GLUSER contains data and routines aimed to customising the behaviour of the GlueWare application.

The names are predefined and used internally when a *GlueL* or *GlueC* instruction is used. The names must therefore not be changed.

Contents

Data

The following global data are predefined:

Name	Declaration	Description
gl_fl1_ovr	CONST num gl_fl1_ovr := 100	Global override for flow1 signal Range: 0-200%
gl_fl2_ovr	CONST num gl_fl2_ovr := 100	Global override for flow2 signal Range: 0-200%
gl_fl1_ref	CONST num gl_fl1_ref := 1000	Reference value used in calculation of glue flow1. Normally the same value as Logical Max for the analog output signal for flow 1.
gl_fl2_ref	CONST num gl_fl2_ref := 1000	Reference value used in calculation of glue flow2. Normally the same value as Logical Max for the analog output signal for flow 2.
gl_sim_glue	CONST bool gl_sim_glue := FALSE	Flag to simulate the gluing. If TRUE: No glue signals are activated.
ggun1	PERS ggundata ggun1 := [1,0,0,2,0,0,0,0,1000,2,0,0,0,0,1000]	Predefined ggundata with default values.

Routines

There are some predefined routines installed with the application. These routines have no default functionality, but can be changed to customise the behaviour of GlueWare.

routine gl_err_actions

This routine is executed when an error detected of the GlueWare occurs.

routine gl_power_on

This routine is executed each time the system is switched on.

routine gl_start

The routine is executed each time the execution of a program is started from the beginning. (From Main)

routine gl_restart

The routine is executed each time the execution of a stopped program is continued.

routine gl_stop

This routine is executed when a normal stop during program execution is performed.

routine gl_qstop

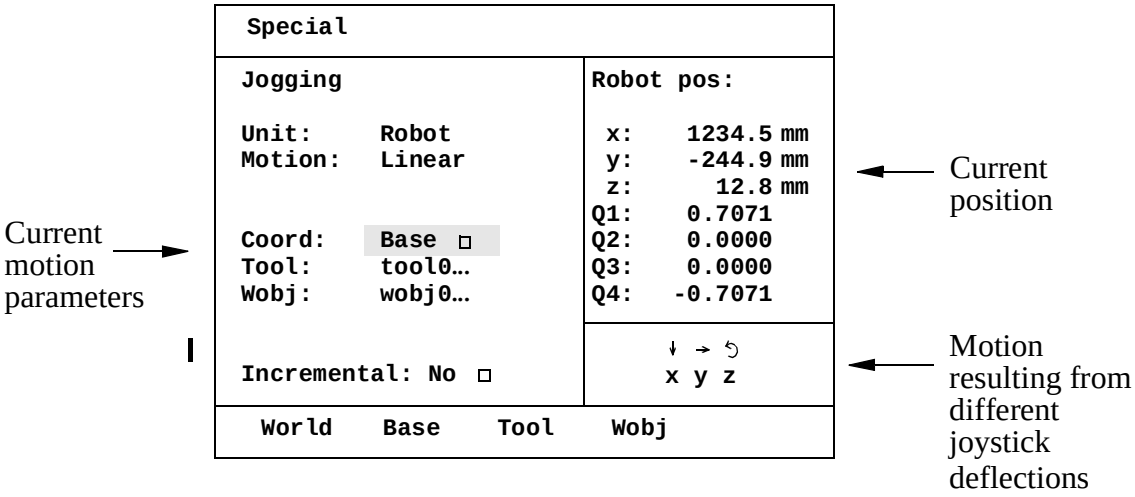
This routine is executed when an emergency stop during program execution is performed.

CONTENTS

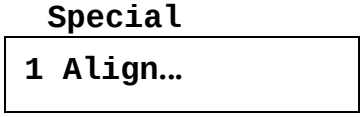
Page

1 The Jogging Window

1.1 Window: Jogging



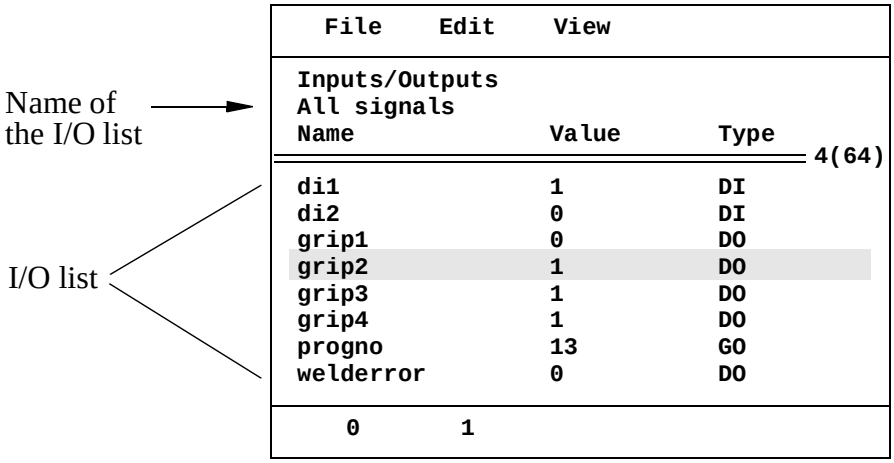
1.1.1 Menu: Special



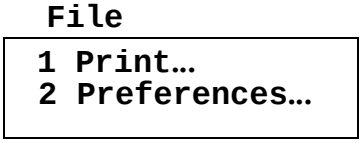
Command	Used to:
Align	Align the tool

2 The Inputs/Outputs Window

2.1 Window: Inputs/Outputs



2.1.1 Menu: File



Command	Used to:
Print	print the current I/O list
Preferences	make preferences in the Inputs/Outputs window

2.1.2 Menu: Edit

```
  Edit
1 Goto…
2 Goto Top
3 Goto Bottom
```

Command:	Used to:
Goto	go to a specific line in the list
Goto Top	go to the first line in the list
Goto Bottom	go to the last line in the list

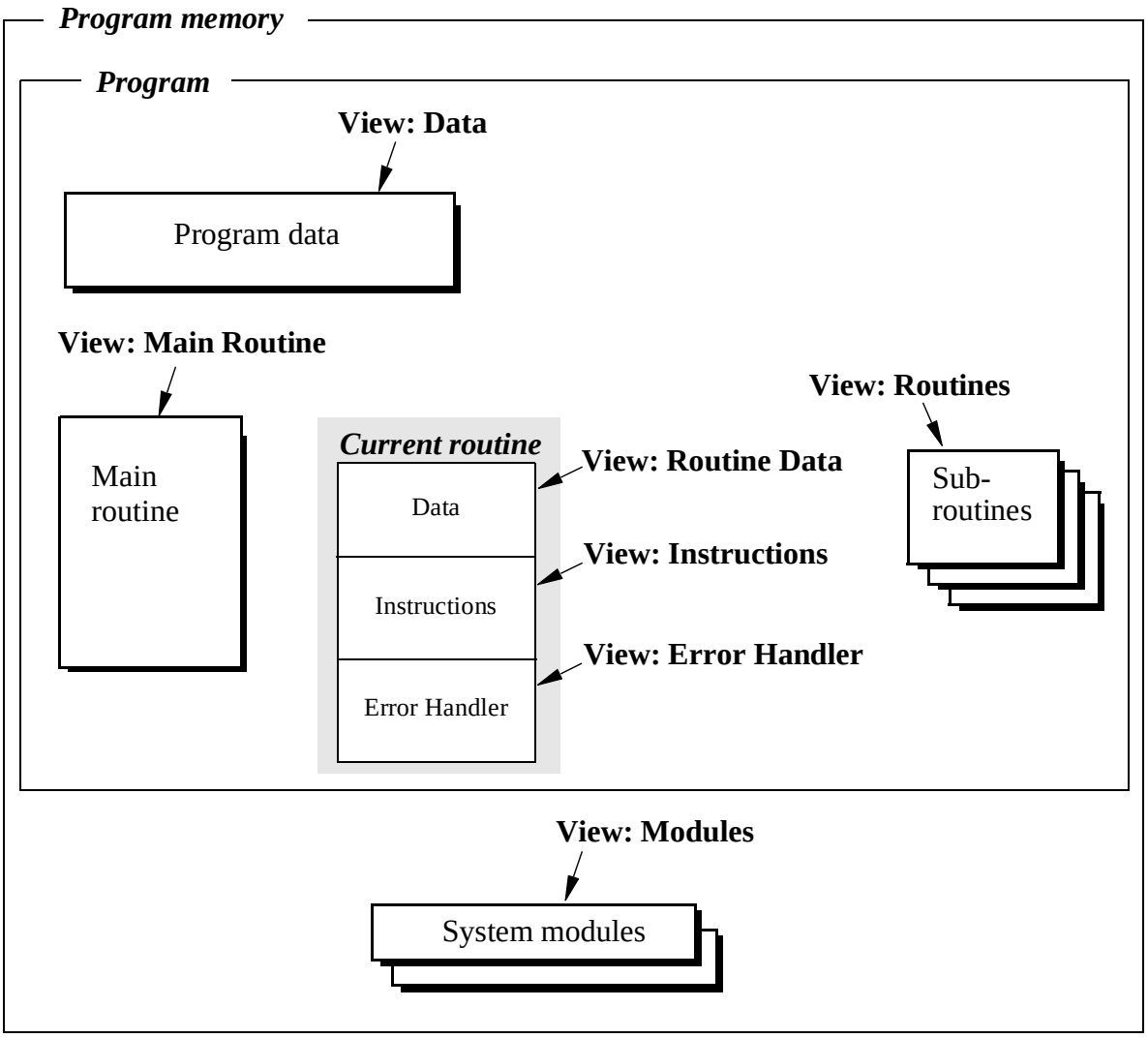
2.1.3 Menu: View

```
  View
1 Most Common
2 All Signals
3 Digital In
4 Digital Out
5 Analog
6 Groups
7 Safety
8 Boards
```

Command:	Used to view:
Most Common	the most common list
All Signals	all user signals
Digital In	all digital inputs
Digital Out	all digital outputs
Analog	all analog signals
Groups	all groups of digital signals
Safety	all safety signals
Boards	all I/O boards

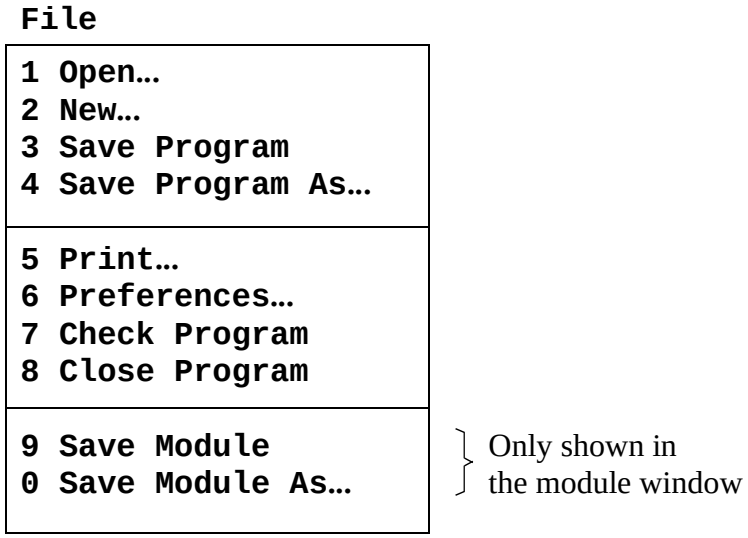
3 The Program Window

3.1 Moving between different parts of the program



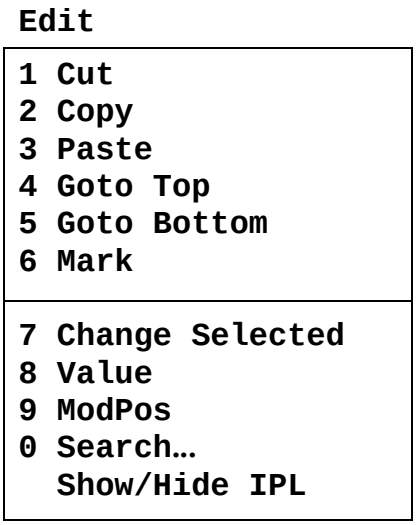
3.2 General menus

3.2.1 Menu: File



Command:	Used to:
Open	read programs from mass storage
New	create new programs
Save Program	save programs on mass storage
Save Program As	save programs on mass storage with new names
Print	print the program
Preferences	make preferences in the Program window
Check Program	check that the program is correct
Close Program	erase the program from the program memory
Save Module	save a module on mass storage
Save Module As	save a module on mass storage with a new name

3.2.2 Menu: Edit



Command	Used to:
Cut	cut selected lines to the clipboard buffer
Copy	copy selected lines to the clipboard buffer
Paste	paste the contents of the clipboard buffer into a program
Goto Top	go to the first line
Goto Bottom	go to the last line
Mark	select several lines
Change Selected	change an instruction argument
Value	show the current value (for the selected argument)
ModPos	modify a position
Search	search for/replace a specific argument
Show/Hide IPL	show/hide an instruction pick list

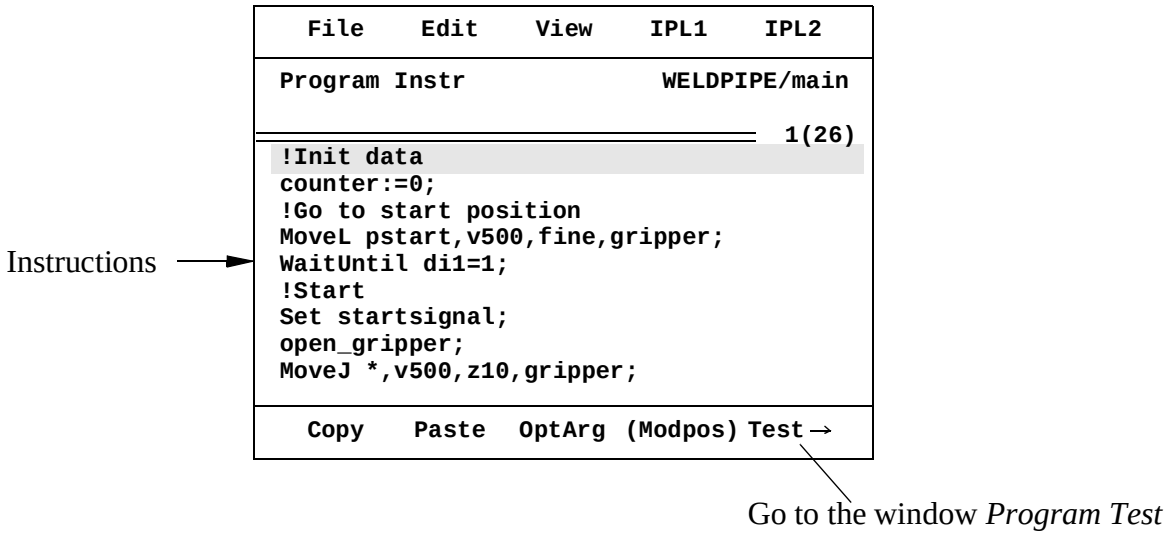
3.2.3 Menu: View

View

```
1 Instr."<latest routine>"
2 Routines
3 Data "<latest type>"
4 Data Types
5 Test
6 Modules
-----------------------
7 Main Routine
8 Selected Routine
-----------------------
9 Error Handler
```

Command	Used to view:
Instr.	instructions for the current routine – *Program Instruction* window
Routines	all routines – *Program Routines* window
Data	program data – *Program Data* window
Data Types	all data types – *Program Data Types* window
Test	the *Program Test* window
Modules	all modules – *Program Modules* window
Main Routine	instructions for the main routine
Selected Routine	instructions for the selected routine
Error Handler	error handler of the current routine

3.3 Window: Program Instr



3.3.1 Menu: IPL1 (shows different instruction pick lists)

IPL1

```
1 Common
2 Prog. Flow
3 Various
4 Motion Settings
5 Motion&Process
6 IO
7 Communicate
8 Interrupts
9 Error Recovery
0 System&Time
  Mathematics
```

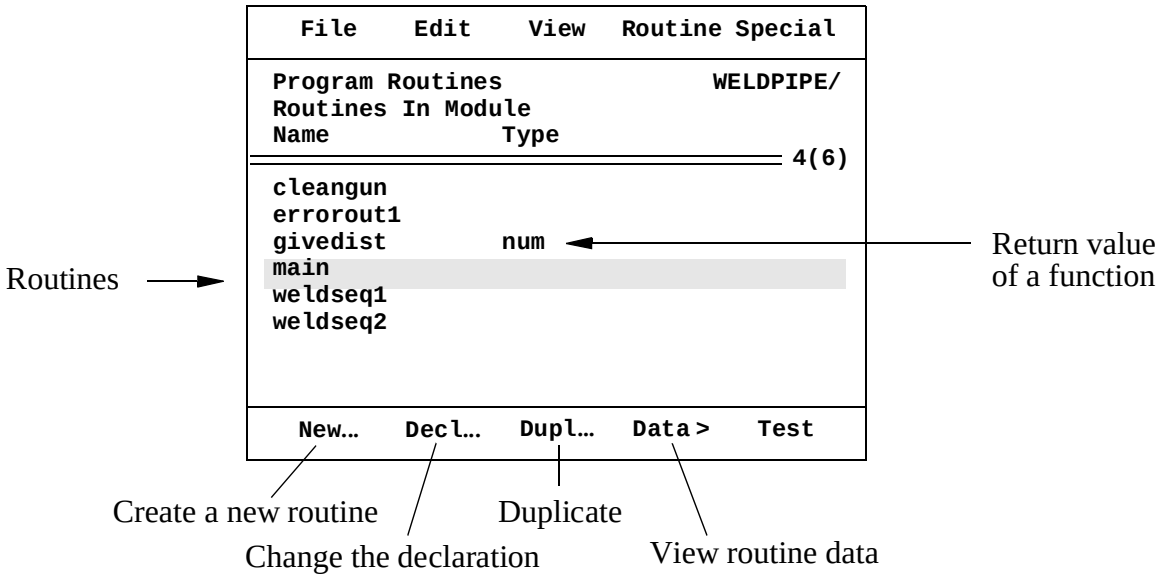
3.3.2 Menu: IPL2 (shows different instruction pick lists)

IPL2

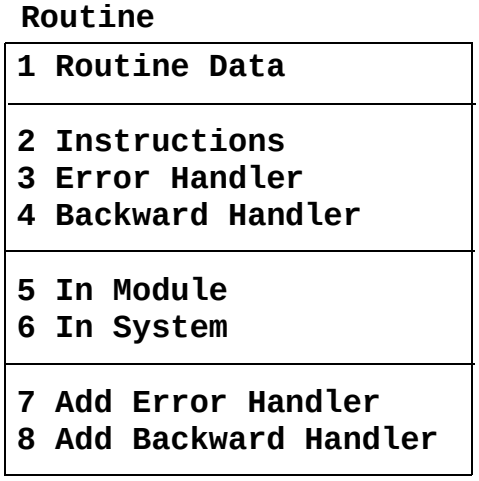
```
1 Most Common 1
2 Most Common 2
3 Most Common 3
```

```
4 Motion Set Adv
5 Motion Adv
6 Ext. Computer
7 Multi Tasking
8 RAPID Support
9 Service
```

3.4 Window: Program Routines

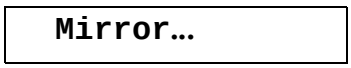


3.4.1 Menu: Routine



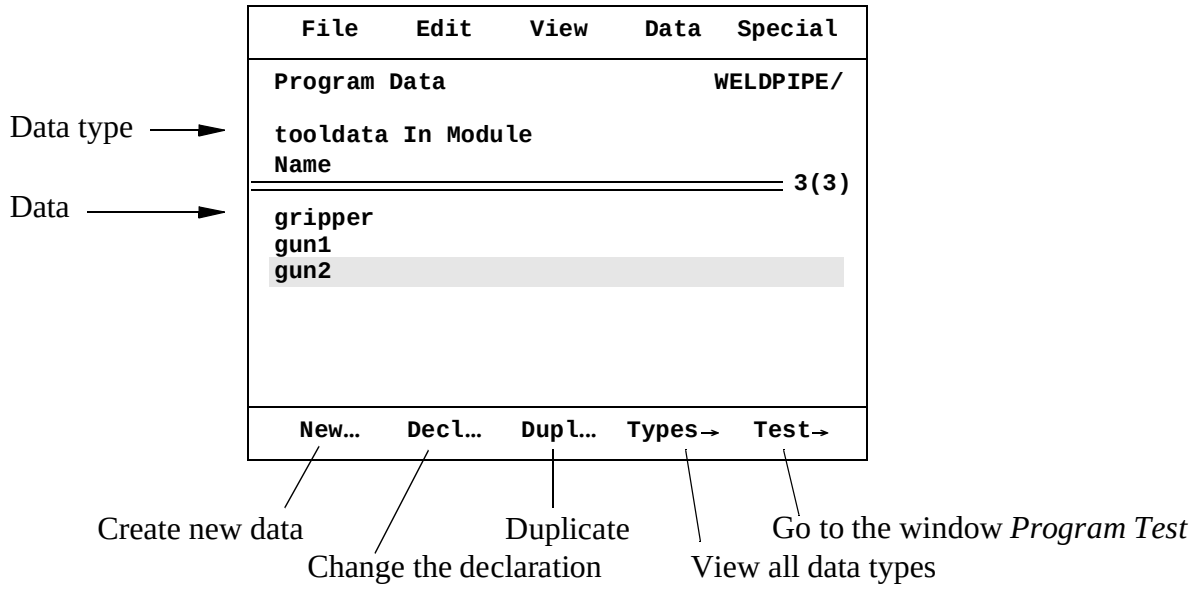
Command:	Used to:
Routine Data	create a new routine
Instructions	view the instructions of the selected routine
Error Handler	view the error handler of the selected routine
Backward Handler	view the backward handler of the selected routine
In Module	view only the routines in the current module
In System	view all routines in all modules
Add/Remove Error Handler	add/remove an error handler to the selected routine
Add/Remove Backward Handler	add/remove a backward handler to the selected routine

3.4.2 Menu: Special



Command:	Used to:
Mirror	mirror a routine or a module

3.5 Window: Program Data



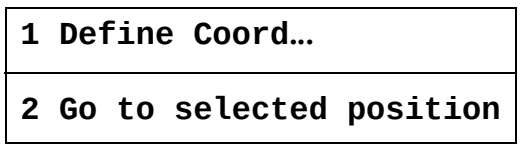
3.5.1 Menu: Data

Data

1 Value **2 Types**
3 In Module **4 In System** **5 In Routine “routine name”**

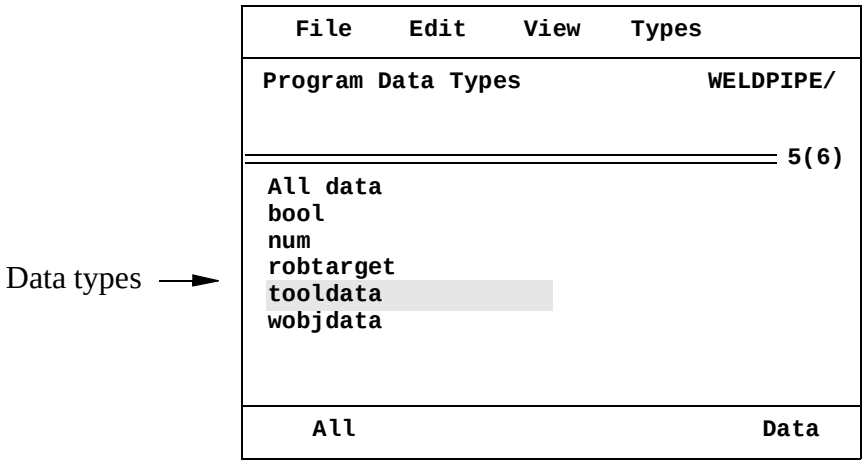
Command:	Used to:
Value	read or change the current value of selected data
Types	call up the list with all data types
In Module	call up only the data in the current module
In System	create new data
In Routine	call up all routine data

3.5.2 Menu: Special

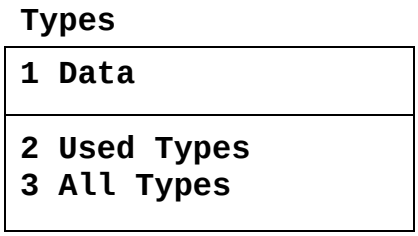


Command:	Used to:
Define Coord	define a tool, work object or program displacement
Go to selected position	go to a selected position

3.6 Window: Program Data Types

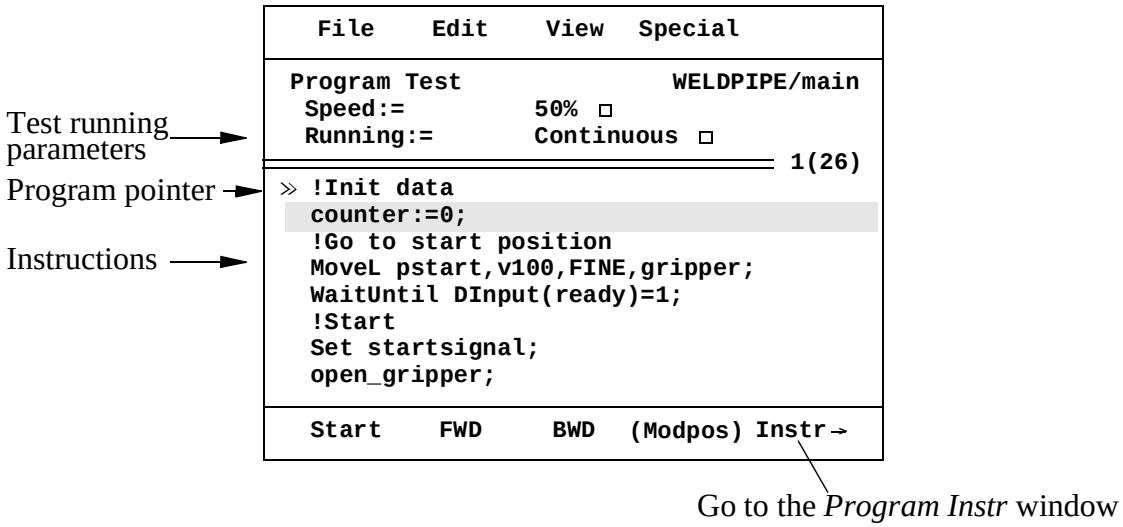


3.6.1 Menu: Types

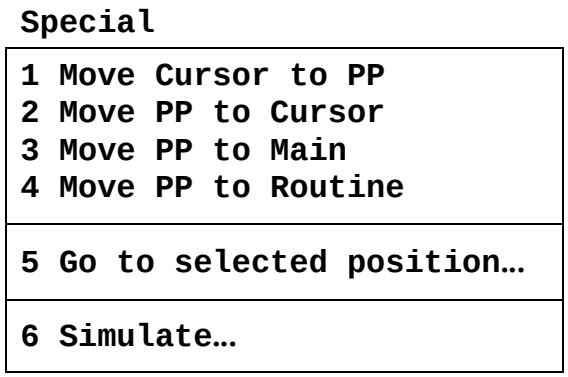


Command:	Used to:
Data	call up all data of a selected type
Used Types	call up only those data types that are used
All Types	call up all data types

3.7 Window: Program Test

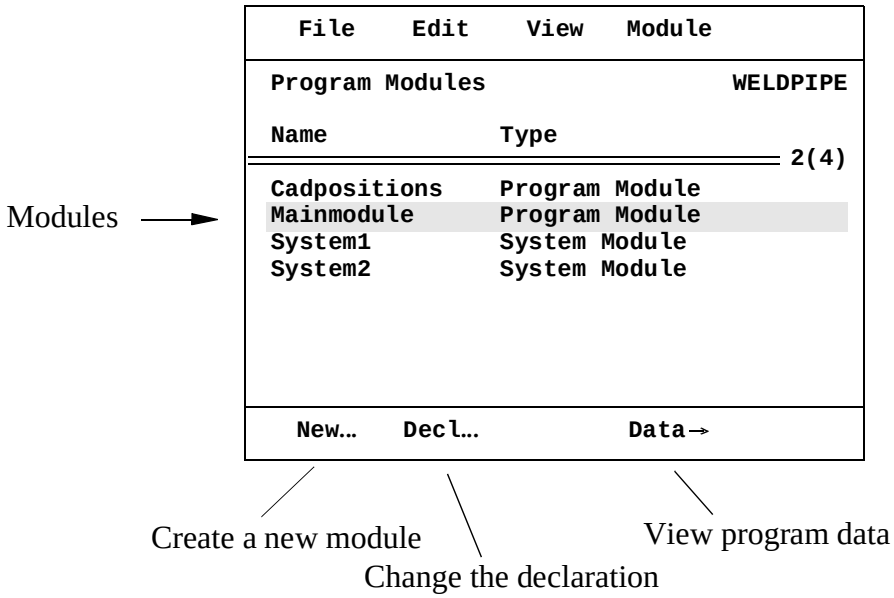


3.7.1 Menu: Test

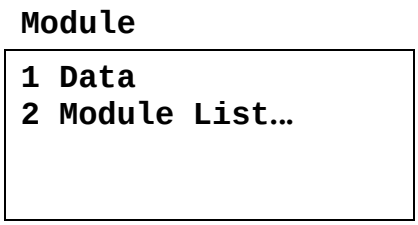


Command:	Used to:
Move Cursor to PP	start from the latest stopped instruction
Move PP to Cursor	start from the selected instruction
Move PP to Main	start from the main routine
Move PP to Routine	start from any routine
Go to selected position	go to a selected position
Simulate	allow program execution in MOTORS OFF mode

3.8 Window: Program Modules



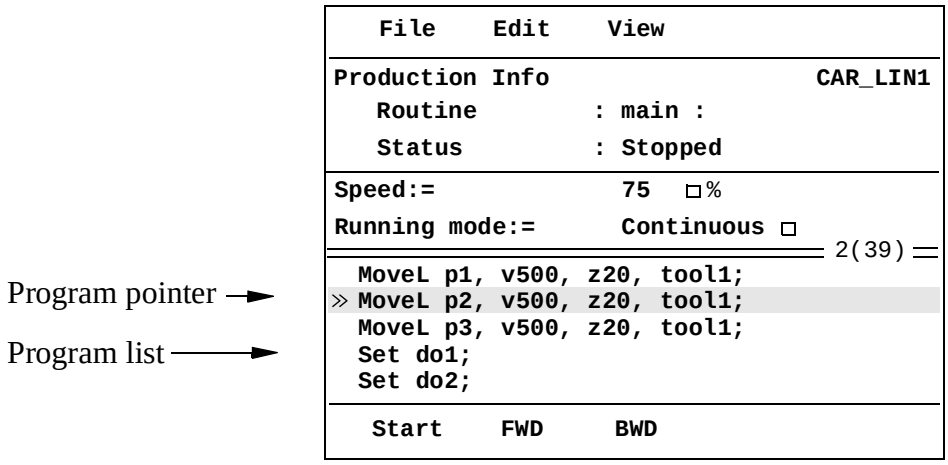
3.8.1 Menu: Module



Command:	Used to:
Data	view program data
Module List	view the complete module in a list

4 The Production Window

4.1 Window: Production



4.1.1 Menu: File

File

```
1 Load Program…
```

Command	Used to:
Load Program	load a program

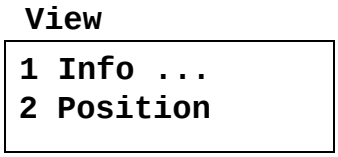
4.1.2 Menu: Edit

Edit

```
1 Goto…
2 Start from Beginning
```

Command	Used to:
Goto	go to a specific instruction
Start from Beginning	go to the first instruction in the program

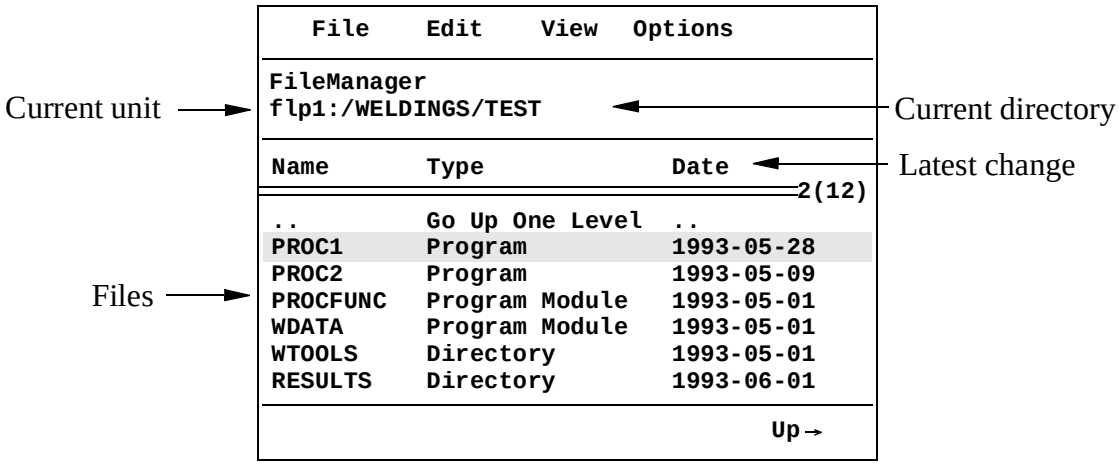
4.1.3 Menu: View



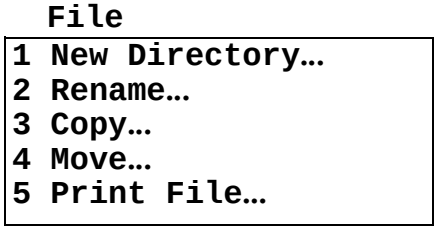
Command	Used to:
Info	display the program in the lower part of the window
Position	tune a position

5 The FileManager

5.1 Window: FileManager

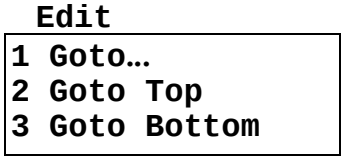


5.1.1 Menu: File



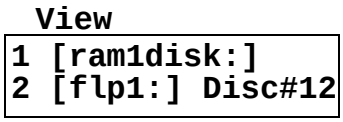
Command:	Used to:
New Directory	create a new directory
Rename	change the name of a selected file
Copy	copy a selected file or directory to another mass memory or directory
Move	move a selected file or directory to another mass memory or directory
Print File	print a file on a printer

5.1.2 Menu: Edit



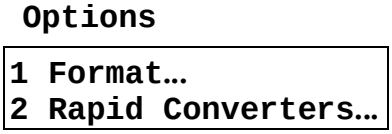
Command:	Used to:
Goto	go to a specific line in a list
Goto Top	go to the first file in a list
Goto Bottom	go to the last file in a list

5.1.3 Menu: View



Command:	Used to view:
ram1disk:	the files on the RAM disk
flp1:	the files on the diskette

5.1.4 Menu: Options

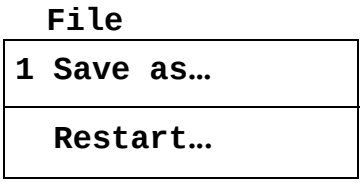


Command:	Used to:
Format	format a diskette
Rapid Converters	convert old program versions

6 The Service Window

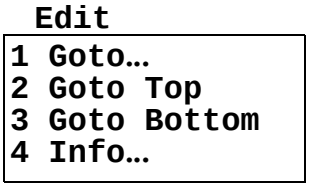
6.1 General menus

6.1.1 Menu: File



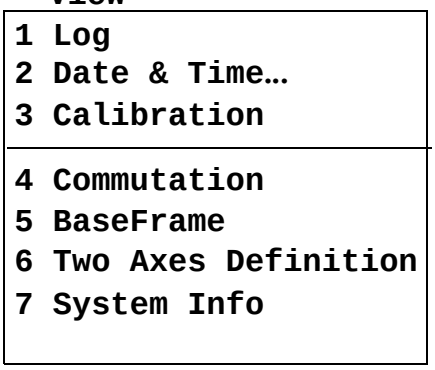
Command	Used to:
Save as	save logs on a diskette or other mass memory
Restart	restart the robot

6.1.2 Menu: Edit



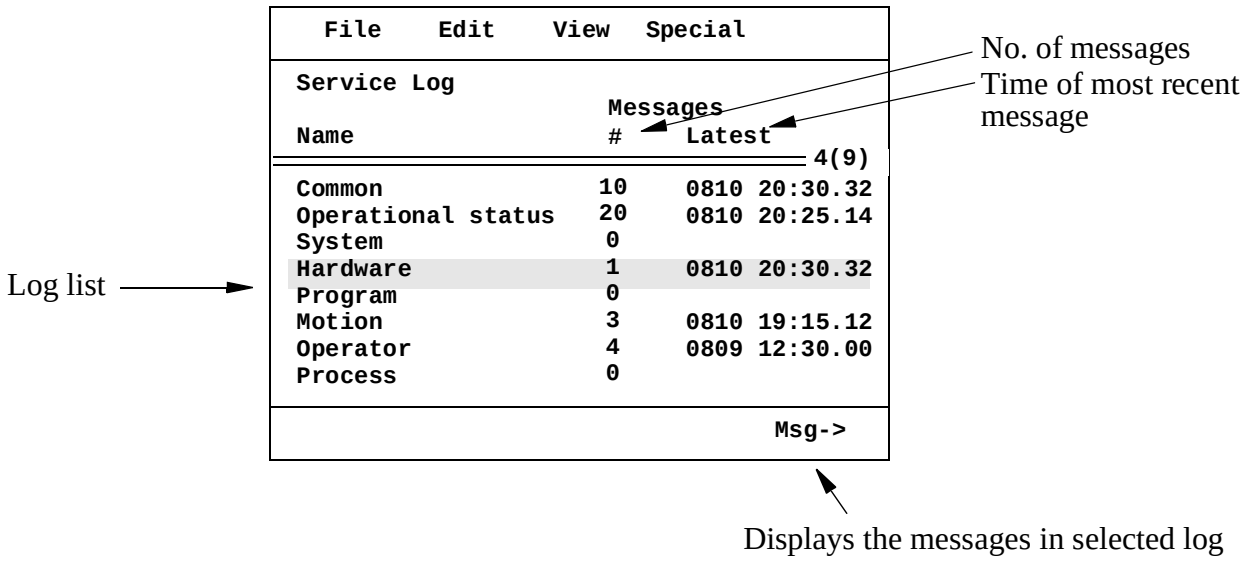
Command	Used to:
Goto	go to a specific line in a list
Goto Top	go to the first line in a list
Goto Bottom	go to the last line in a list
Info	view information about selected log messages

6.1.3 Menu: View

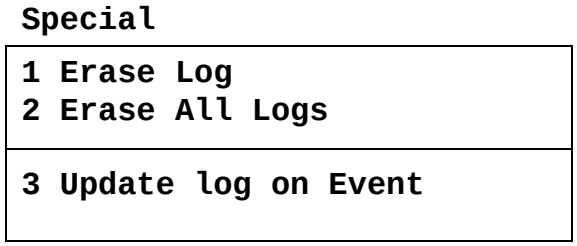


Command	Used to:
Log	display the different logs
Date & Time	set the date and time
Calibration	calibrate the robot
Commutation	commutate the motors (see The Product Manual/ *Repairs*)
BaseFrame	calibrate the base coordinate system
Two Axes Definition	calibrate the base coordinate system for a two axes manipulator
System Info	display system information

6.2 Window Service Log

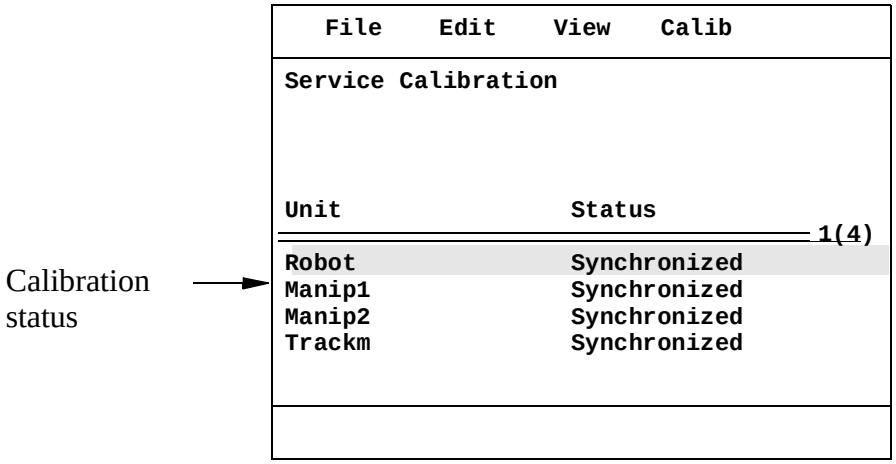


6.2.1 Menu: Special

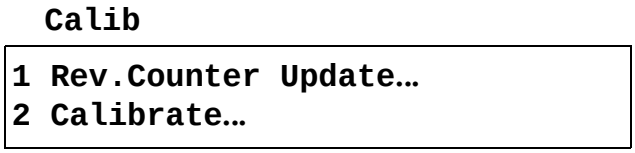


Command:	Used to:
Erase Log	erase contents in selected log
Erase All Logs	erase contents in all logs
Update log on Event	update the log directly when a message is sent – the command is changed to **"Update log on Command"** when selected, which means that the log is not updated until the function key **Update** is pressed

6.3 Window Service Calibration

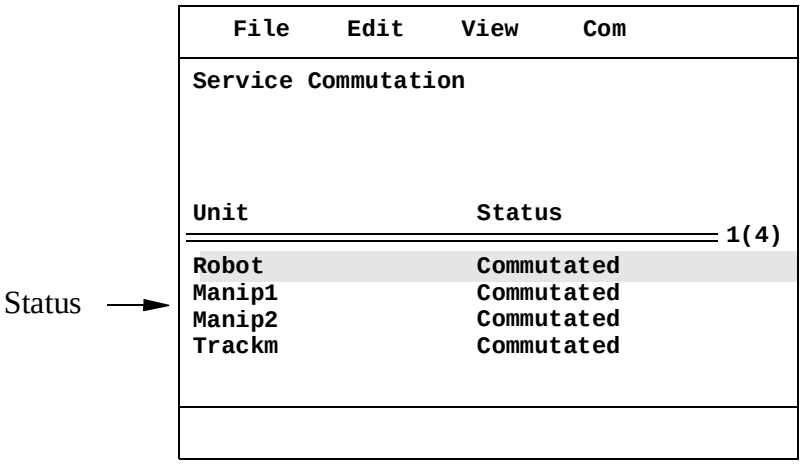


6.3.1 Menu: Calib

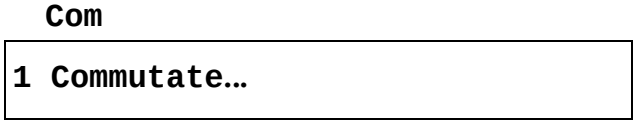


Command:	Used to:
Rev.Counter Update	update the counter
Calibrate	calibrate using the measurement system

6.4 Window Service Commutation



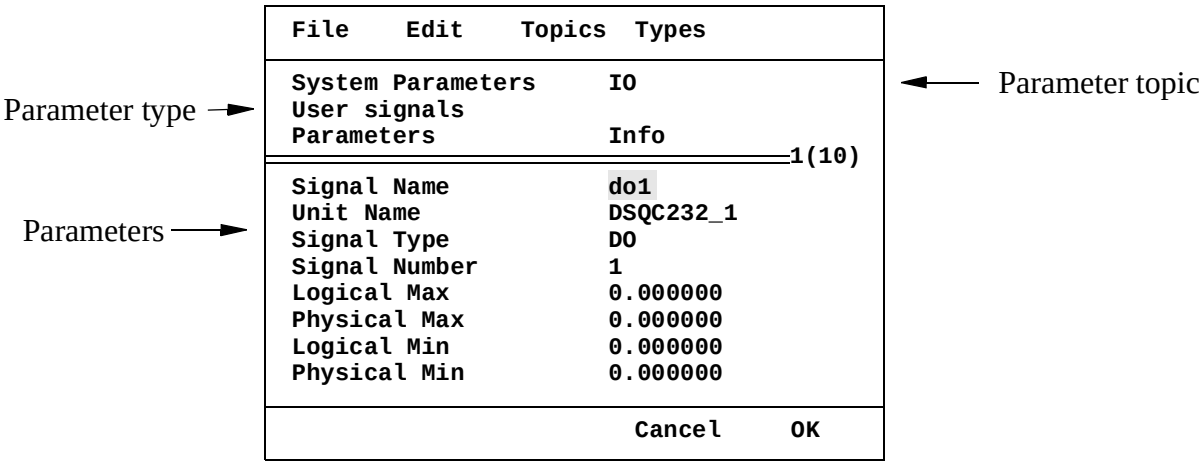
6.4.1 Menu: Com



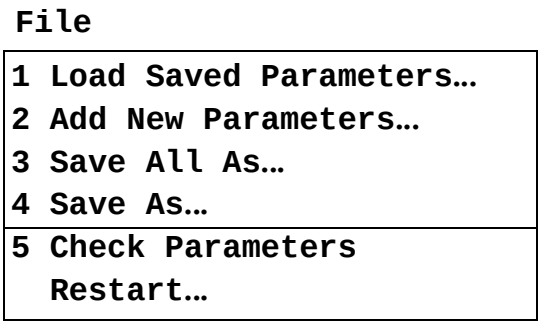
Command:	Used to:
Commutate	commutate using the measurement system

7 The System Parameters

7.1 Window: System Parameters

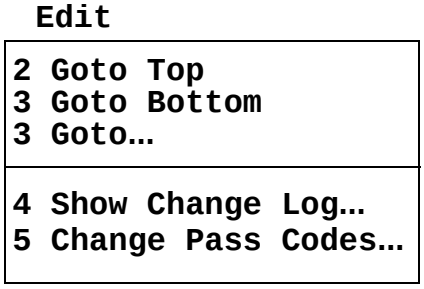


7.1.1 Menu: File



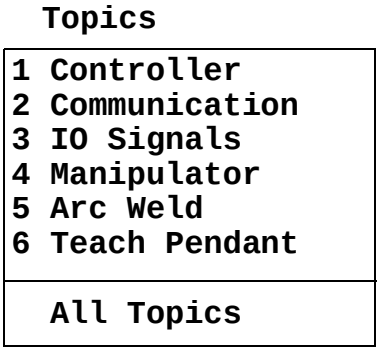
Command:	Used to:
Load Saved Parameters	load parameters from mass storage
Add New Parameters	add parameters from mass storage
Save All As	save all parameters on mass storage
Save As	save parameters on mass storage
Check Parameters	check parameters before restart
Restart	restart the robot

7.1.2 Menu: Edit



Command:	Used to:
Goto Top	go to the first line in a list
Goto Bottom	go to the last line in a list
Goto	go to a specific line in a list
Show Change Log	view information about the latest modifications made
Change Pass Codes	change pass codes

7.1.3 Menu: Topics



Command:	Used to view:
Controller	the parameter of the Controller topic
Communication	the parameter of the Communication topic
IO Signals	the parameters of the IO topic
Manipulator	the parameters of the Manipulator topic
Arc Weld	the parameters of the Arc Weld topic
Teach Pendant	the parameters of the Teach Pendant topic
All Topics	all topics

The contents under the chapter

“Special Functionality in this Robot”

is depending on how the robot is equipped,

and is therefore not included in the On-line Manual.

Click on the Main menu button below to continue to the front page.

Main menu