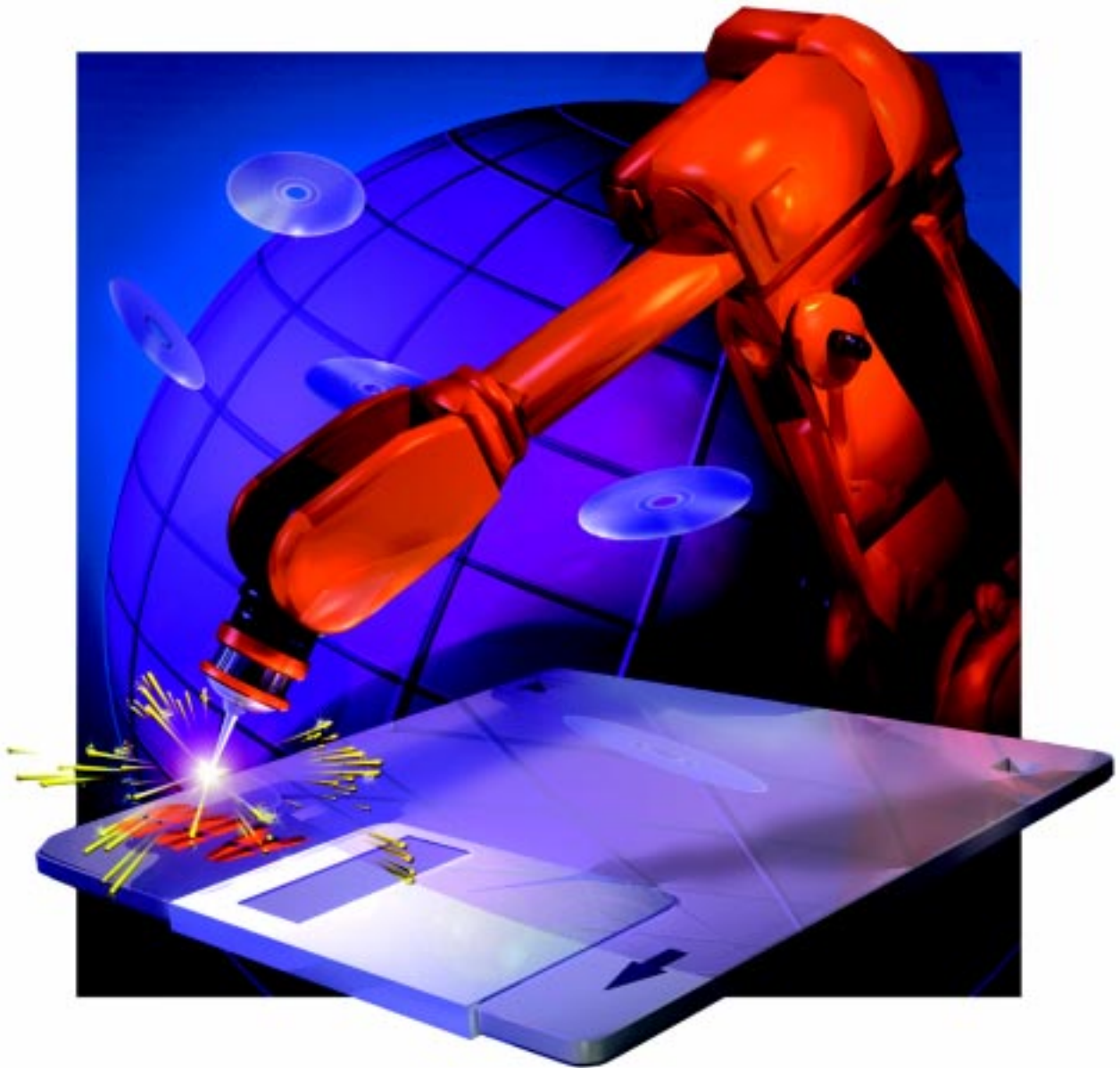


# RAPID Reference Manual

*RAPID Overview On-line*



**ABB Flexible Automation**





---

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



3HAC 0966-50  
For BaseWare OS 3.1 Rev.1

# RAPID Overview

*Table of Contents*

*Introduction*

*RAPID Summary*

*Basic Characteristics*

*Motion and I/O Principles*

*Programming Off-line*

*Predefined Data and Programs*

*Index, Glossary*

The information in this document is subject to change without notice and should not be construed as a commitment by ABB Robotics Products AB. ABB Robotics Products AB assumes no responsibility for any errors that may appear in this document.

In no event shall ABB Robotics Products AB be liable for incidental or consequential damages arising from use of this document or of the software and hardware described in this document.

This document and parts thereof must not be reproduced or copied without ABB Robotics Products AB's written permission, and contents thereof must not be imparted to a third party nor be used for any unauthorized purpose. Contravention will be prosecuted.

Additional copies of this document may be obtained from ABB Robotics Products AB at its then current charge.

© ABB Robotics Products AB

Article number: 3HAC 0966-50  
Issue: For BaseWare OS 3.1 Rev.1

ABB Robotics Products AB  
S-721 68 Västerås  
Sweden

# CONTENTS

	Page
<b>1 Table of Contents .....</b>	<b>1-1</b>
<b>2 Introduction .....</b>	<b>2-1</b>
<b>1 Other Manuals .....</b>	<b>2-1</b>
<b>2 How to Read this Manual.....</b>	<b>2-1</b>
2.1 Typographic conventions .....	2-2
2.2 Syntax rules .....	2-2
2.3 Formal syntax .....	2-3
<b>3 RAPID Summary.....</b>	<b>3-1</b>
<b>1 The Structure of the Language.....</b>	<b>3-1</b>
<b>2 Controlling the Program Flow.....</b>	<b>3-3</b>
2.1 Programming principles .....	3-3
2.2 Calling another routine.....	3-3
2.3 Program control within the routine .....	3-3
2.4 Stopping program execution .....	3-4
2.5 Stop current cycle.....	3-4
<b>3 Various Instructions.....</b>	<b>3-5</b>
3.1 Assigning a value to data .....	3-5
3.2 Wait .....	3-5
3.3 Comments .....	3-5
3.4 Loading program modules .....	3-6
3.5 Various functions.....	3-6
3.6 Basic data .....	3-6
3.7 Conversion function .....	3-6
<b>4 Motion Settings .....</b>	<b>3-7</b>
4.1 Programming principles .....	3-7
4.2 Defining velocity.....	3-7
4.3 Defining acceleration .....	3-8
4.4 Defining configuration management .....	3-8
4.5 Defining the payload .....	3-8
4.6 Defining the behaviour near singular points .....	3-8
4.7 Displacing a program .....	3-9
4.8 Soft servo .....	3-9
4.9 Adjust the robot tuning values .....	3-9
4.10 World Zones .....	3-10
4.11 Data for motion settings .....	3-10
<b>5 Motion .....</b>	<b>3-11</b>
5.1 Programming principles .....	3-11

	Page
5.2 Positioning instructions.....	3-12
5.3 Searching.....	3-12
5.4 Activating outputs or interrupts at specific positions .....	3-12
5.5 Motion control if an error/interrupt takes place.....	3-13
5.6 Controlling external axes .....	3-13
5.7 Independent axes.....	3-13
5.8 Position functions.....	3-14
5.9 Motion data .....	3-15
5.10 Basic data for movements.....	3-15
<b>6 Input and Output Signals.....</b>	<b>3-17</b>
6.1 Programming principles.....	3-17
6.2 Changing the value of a signal.....	3-17
6.3 Reading the value of an input signal.....	3-17
6.4 Reading the value of an output signal.....	3-17
6.5 Testing input on output signals .....	3-18
6.6 Disabling and enabling I/O modules.....	3-18
6.7 Defining input and output signals .....	3-18
<b>7 Communication.....</b>	<b>3-19</b>
7.1 Programming principles.....	3-19
7.2 Communicating using the teach pendant .....	3-19
7.3 Reading from or writing to a character-based serial channel/file .....	3-20
7.4 Communicating using binary serial channels/files .....	3-20
7.5 Data for serial channels.....	3-20
<b>8 Interrupts .....</b>	<b>3-21</b>
8.1 Programming principles.....	3-21
8.2 Connecting interrupts to trap routines.....	3-21
8.3 Ordering interrupts.....	3-22
8.4 Cancelling interrupts.....	3-22
8.5 Enabling/disabling interrupts .....	3-22
8.6 Data type of interrupts .....	3-22
<b>9 Error Recovery .....</b>	<b>3-23</b>
9.1 Programming principles.....	3-23
9.2 Creating an error situation from within the program .....	3-23
9.3 Restarting/returning from the error handler.....	3-24
9.4 Data for error handling.....	3-24
<b>10 System &amp; Time.....</b>	<b>3-25</b>
10.1 Programming principles.....	3-25
10.2 Using a clock to time an event.....	3-25

10.3	Reading current time and date .....	3-25
10.4	Retrieve time information from file .....	3-26
<b>11</b>	<b>Mathematics .....</b>	<b>3-27</b>
11.1	Programming principles .....	3-27
11.2	Simple calculations on numeric data.....	3-27
11.3	More advanced calculations .....	3-27
11.4	Arithmetic functions.....	3-28
<b>12</b>	<b>Spot Welding .....</b>	<b>3-29</b>
12.1	Spot welding features .....	3-29
12.2	Principles of SpotWare.....	3-30
12.3	Programming principles .....	3-31
12.4	Spot welding instructions.....	3-31
12.5	Spot welding data.....	3-31
<b>13</b>	<b>Arc Welding.....</b>	<b>3-33</b>
13.1	Programming principles .....	3-33
13.2	Arc welding instructions .....	3-33
13.3	Arc welding data .....	3-34
<b>14</b>	<b>GlueWare .....</b>	<b>3-35</b>
14.1	Glueing features .....	3-35
14.2	Programming principles .....	3-35
14.3	Glue instructions .....	3-35
14.4	Glue data .....	3-36
<b>15</b>	<b>External Computer Communication .....</b>	<b>3-37</b>
15.1	Programming principles .....	3-37
15.2	Sending a program-controlled message from the robot to a computer .....	3-37
<b>16</b>	<b>Service Instructions .....</b>	<b>3-39</b>
16.1	Directing a value to the robot's test signal .....	3-39
<b>17</b>	<b>String Functions .....</b>	<b>3-41</b>
17.1	Basic Operations .....	3-41
17.2	Comparison and Searching .....	3-41
17.3	Conversion .....	3-42
<b>18</b>	<b>Multitasking .....</b>	<b>3-43</b>
18.1	Basics .....	3-43
18.2	Resource access Protection .....	3-43
<b>19</b>	<b>Syntax Summary .....</b>	<b>3-45</b>
19.1	Instructions.....	3-45
19.2	Functions .....	3-49

<b>4 Basic Characteristics .....</b>	<b>Page 4-1</b>
<b>1 Basic Elements .....</b>	<b>4-1</b>
1.1 Identifiers .....	4-1
1.2 Spaces and new-line characters .....	4-2
1.3 Numeric values .....	4-2
1.4 Logical values .....	4-2
1.5 String values.....	4-2
1.6 Comments .....	4-3
1.7 Placeholders .....	4-3
1.8 File header.....	4-3
1.9 Syntax .....	4-4
<b>2 Modules.....</b>	<b>4-7</b>
2.1 Program modules .....	4-7
2.2 System modules .....	4-8
2.3 Module declarations.....	4-8
2.4 Syntax .....	4-8
<b>3 Routines .....</b>	<b>4-11</b>
3.1 Routine scope.....	4-11
3.2 Parameters.....	4-12
3.3 Routine termination .....	4-13
3.4 Routine declarations.....	4-13
3.5 Procedure call.....	4-14
3.6 Syntax .....	4-15
<b>4 Data Types .....</b>	<b>4-19</b>
4.1 Non-value data types.....	4-19
4.2 Equal (alias) data types .....	4-19
4.3 Syntax .....	4-20
<b>5 Data.....</b>	<b>4-21</b>
5.1 Data scope.....	4-21
5.2 Variable declaration .....	4-22
5.3 Persistent declaration .....	4-22
5.4 Constant declaration.....	4-23
5.5 Initiating data .....	4-23
5.6 Syntax .....	4-24
<b>6 Instructions.....</b>	<b>4-27</b>
6.1 Syntax .....	4-27
<b>7 Expressions.....</b>	<b>4-29</b>
7.1 Arithmetic expressions.....	4-29



7.2 Logical expressions .....	4-30
7.3 String expressions .....	4-30
7.4 Using data in expressions .....	4-31
7.5 Using aggregates in expressions .....	4-32
7.6 Using function calls in expressions .....	4-32
7.7 Priority between operators .....	4-33
7.8 Syntax .....	4-34
<b>8 Error Recovery .....</b>	<b>4-37</b>
8.1 Error handlers .....	4-37
<b>9 Interrupts .....</b>	<b>4-39</b>
9.1 Interrupt manipulation .....	4-39
9.2 Trap routines .....	4-40
<b>10 Backward execution .....</b>	<b>4-41</b>
1.1 Backward handlers .....	4-41
1.2 Limitation of move instructions in the backward handler .....	4-42
<b>11 Multitasking .....</b>	<b>4-43</b>
11.1 Synchronising the tasks .....	4-43
11.1 Synchronising using polling .....	4-44
11.1 Synchronising using an interrupt .....	4-44
11.2 Intertask communication .....	4-45
11.3 Type of task .....	4-46
11.4 Priorities .....	4-46
11.5 Trust Level .....	4-47
11.6 Task sizes .....	4-48
11.7 Something to think about .....	4-48
11.8 Programming scheme .....	4-48
11.8 The first time .....	4-48
11.8 Iteration phase .....	4-49
11.8 Finnish phase .....	4-49
<b>5 Motion and I/O Principles .....</b>	<b>5-1</b>
<b>1 Coordinate Systems .....</b>	<b>5-1</b>
1.1 The robot's tool centre point (TCP) .....	5-1
1.2 Coordinate systems used to determine the position of the TCP .....	5-1
1.3 Coordinate systems used to determine the direction of the tool .....	5-6
1.4 Related information .....	5-10
<b>2 Positioning during Program Execution .....</b>	<b>5-11</b>
2.1 General .....	5-11
2.2 Interpolation of the position and orientation of the tool .....	5-11

	Page
2.3 Interpolation of corner paths .....	5-14
2.4 Independent axes.....	5-20
2.5 Soft Servo.....	5-22
2.6 Stop and restart .....	5-23
2.7 Related information .....	5-24
<b>3 Synchronisation with logical instructions.....</b>	<b>5-25</b>
3.1 Sequential program execution at stop points .....	5-25
3.2 Sequential program execution at fly-by points .....	5-25
3.3 Concurrent program execution .....	5-26
3.4 Path synchronisation .....	5-29
3.5 Related information .....	5-30
<b>4 Robot Configuration.....</b>	<b>5-31</b>
4.1 Robot configuration data for 6400C .....	5-33
4.2 Related information .....	5-34
<b>5 Robot kinematic models .....</b>	<b>5-35</b>
5.1 Robot kinematics .....	5-35
5.2 General kinematics.....	5-37
5.3 Related information .....	5-39
<b>6 Motion Supervision/Collision Detection .....</b>	<b>5-41</b>
6.1 Introduction.....	5-41
6.2 Tuning of Collision Detection levels .....	5-41
6.3 Motion supervision dialogue box.....	5-41
6.4 Digital outputs.....	5-43
6.5 Limitations .....	5-43
6.6 Related information .....	5-44
<b>7 Singularities.....</b>	<b>5-45</b>
7.1 Singularity points/IRB 6400C.....	5-46
7.2 Program execution through singularities .....	5-46
7.3 Jogging through singularities.....	5-46
7.4 Related information .....	5-46
<b>8 World Zones .....</b>	<b>5-47</b>
8.1 Using global zones.....	5-47
8.2 Using World Zones .....	5-47
8.3 Definition of World Zones in the world coordinate system.....	5-47
8.4 Supervision of the Robot TCP .....	5-48
8.5 Actions .....	5-49
8.6 Minimum size of World Zones. ....	5-50
8.7 Maximum number of World Zones.....	5-50

8.8 Power failure, restart, and run on .....	5-50
8.9 Related information.....	5-51
<b>9 I/O Principles.....</b>	<b>5-53</b>
9.1 Signal characteristics.....	5-53
9.2 System signals.....	5-54
9.3 Cross connections .....	5-54
9.4 Limitations .....	5-55
9.5 Related information.....	5-55
<b>6 Programming Off-line.....</b>	<b>6-1</b>
1 Programming .....	6-1
1.1 File format.....	6-1
1.2 Editing.....	6-1
1.3 Syntax check .....	6-1
1.4 Examples.....	6-2
1.5 Making your own instructions .....	6-2
<b>7 Predefined Data and Programs .....</b>	<b>7-1</b>
1 System Module User .....	7-1
1.1 Contents .....	7-1
1.2 Creating new data in this module.....	7-1
1.3 Deleting this data .....	7-2
<b>8 Index, Glossary .....</b>	<b>8-3</b>



# 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 *RAPID Overview 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.

*RAPID Overview Chapter 4: Basic Characteristics* explains the inner details of the language. You would not normally read this chapter unless you are an experienced programmer.

*RAPID Overview Chapter 5: Motion and I/O Principles* describes the various coordinate systems of the robot, its velocity and other motion characteristics during different types of execution.

*System DataTypes and Routines Chapters 1-3* 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.

*RAPID Overview Chapter 7: 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 *RAPID Overview Chapter 6: Programming off-line*.

To make things easier to locate and understand, *RAPID Overview chapter 8* contains an *Index*, *Glossary* and *System DataTypes and Routines Chapter 4* contains an *index*.

---

## 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;*

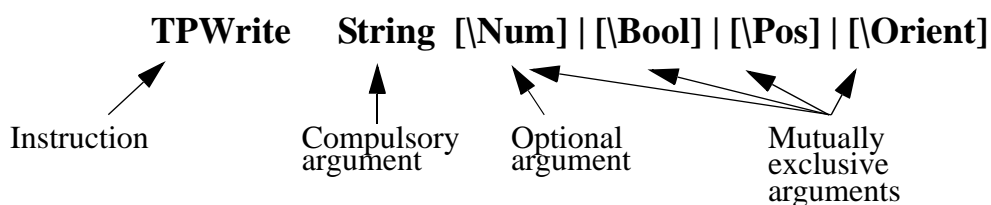
---

## 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.

## ***Introduction***



---

---

## 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.  $5 + \text{reg1} * 2$
- 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

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

---

### 2.3 Program control within the routine

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

---

**2.4 Stopping program execution**

<u>Instruction</u>	<u>Used to:</u>
<i>Stop</i>	Stop program execution
<i>EXIT</i>	Stop program execution when a program restart is not allowed
<i>Break</i>	Stop program execution temporarily for debugging purposes

---

**2.5 Stop current cycle**

<u>Instruction</u>	<u>Used to:</u>
<i>Exit cycle</i>	Stop the current cycle and move the program pointer to the first instruction in the main routine. When the execution mode <i>CONT</i> is selected, execution will continue with the next program cycle.

---



---

### 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.  $reg1 + 5 * reg3$ .

<u>Instruction</u>	<u>Used to:</u>
<code>:=</code>	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.

<u>Instruction</u>	<u>Used to:</u>
<code>WaitTime</code>	Wait a given amount of time or to wait until the robot stops moving
<code>WaitUntil</code>	Wait until a condition is satisfied
<code>WaitDI</code>	Wait until a digital input is set
<code>WaitDO</code>	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.

<u>Instruction</u>	<u>Used to:</u>
<code>comment</code>	Comment on the program

---

### 3.4 Loading program modules

Program modules can be loaded from mass memory or erased from the program memory. In this way large programs can be handled with only a small memory.

<u>Instruction</u>	<u>Used to:</u>
<i>Load</i>	Load a program module into the program memory
<i>UnLoad</i>	Unload a program module from the program memory
<i>Start Load</i>	Load a program module into the program memory during execution
<i>Wait Load</i>	Connect the module, if loaded with <i>StartLoad</i> , to the program task
<i>Save</i>	Save a program module

---

### 3.5 Various functions

<u>Function</u>	<u>Used to:</u>
<i>OpMode</i>	Read the current operating mode of the robot
<i>RunMode</i>	Read the current program execution mode of the robot
<i>Dim</i>	Obtain the dimensions of an array
<i>Present</i>	Find out whether an optional parameter was present when a routine call was made
<i>IsPers</i>	Check whether a parameter is a persistent
<i>IsVar</i>	Check whether a parameter is a variable

---

### 3.6 Basic data

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

---

### 3.7 Conversion function

Function*StrToByte**ByteToStr*Used to:

Convert a byte to a string data with a defined byte data format.

Convert a string with a defined byte data format to a byte data.





---

---

## 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.

<u>Instruction</u>	<u>Used to define:</u>
<i>VelSet</i>	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.

<u>Instruction</u>	<u>Used to define:</u>
<i>AccSet</i>	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.

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

---

### 4.5 Defining the payload

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

<u>Instruction</u>	<u>Used to define:</u>
<i>GripLoad</i>	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.

<u>Instruction</u>	<u>Used to define:</u>
<i>SingArea</i>	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.

<u>Instruction</u>	<u>Used to:</u>
<i>PDispOn</i>	Activate program displacement
<i>PDispSet</i>	Activate program displacement by specifying a value
<i>PDispOff</i>	Deactivate program displacement
<i>EOffsOn</i>	Activate an external axis offset
<i>EOffsSet</i>	Activate an external axis offset by specifying a value
<i>EOffsOff</i>	Deactivate an external axis offset
<u>Function</u>	<u>Used to:</u>
<i>DefDFrame</i>	Calculate a program displacement from three positions
<i>DefFrame</i>	Calculate a program displacement from six positions
<i>ORobT</i>	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.

<u>Instruction</u>	<u>Used to:</u>
<i>SoftAct</i>	Activate the soft servo for one or more axes
<i>SoftDeact</i>	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.

<u>Instruction</u>	<u>Used to:</u>
<i>TuneServo</i> <sup>1</sup>	Adjust the robot tuning values
<i>TuneReset</i>	Reset tuning to normal
<i>PathResol</i>	Adjust the geometric path resolution
<u>Data type</u>	<u>Used to:</u>
<i>tunetype</i>	Represent the tuning type as a symbolic constant

1. Only when the robot is equipped with the option “Advanced Motion”

---

## 4.10 World Zones

Up to 10 different volumes can be defined within the working area of the robot. These can be used for:

- Indicating that the robot's TCP is a definite part of the working area.
- Delimiting the working area for the robot and preventing a collision with the tool.
- Creating a working area common to two robots. The working area is then available only to one robot at a time.

<u>Instruction</u>	<u>Used to:</u>
<i>WZBoxDef<sup>1</sup></i>	Define a box-shaped global zone
<i>WZCylDef<sup>1</sup></i>	Define a cylindrical global zone
<i>WZSphDef</i>	Define a spherical global zone
<i>WZLimSup<sup>1</sup></i>	Activate limit supervision for a global zone
<i>WZDOSet<sup>1</sup></i>	Activate global zone to set digital outputs
<i>WZDisable<sup>1</sup></i>	Deactivate supervision of a temporary global zone
<i>WZEnable<sup>1</sup></i>	Activate supervision of a temporary global zone
<i>WZFree<sup>1</sup></i>	Erase supervision of a temporary global zone
<u>Data type</u>	<u>Used to:</u>
<i>wztemporary</i>	Identify a temporary global zone
<i>wzstationary</i>	Identify a stationary global zone
<i>shapedata</i>	Describe the geometry of a global zone

---

## 4.11 Data for motion settings

<u>Data type</u>	<u>Used to define:</u>
<i>motsetdata</i>	Motion settings except program displacement
<i>progdisp</i>	Program displacement

---

1. Only when the robot is equipped with the option "Advanced functions"

---

---

## 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 7):

- 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

<u>Instruction</u>	<u>Type of movement:</u>
<i>MoveC</i>	TCP moves along a circular path
<i>MoveJ</i>	Joint movement
<i>MoveL</i>	TCP moves along a linear path
<i>MoveAbsJ</i>	Absolute joint movement
<i>MoveCDO</i>	Moves the robot circularly and sets a digital output in the middle of the corner path.
<i>MoveJDO</i>	Moves the robot by joint movement and sets a digital output in the middle of the corner path.
<i>MoveLDO</i>	Moves the robot linearly and sets a digital output in the middle of the corner path.
<i>MoveCSync</i> <sup>1</sup>	Moves the robot circularly and executes a RAPID procedure
<i>MoveJSync</i> <sup>1</sup>	Moves the robot by joint movement and executes a RAPID procedure
<i>MoveLSync</i> <sup>1</sup>	Moves the robot linearly and executes a RAPID procedure

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

---

## 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.

<u>Instruction</u>	<u>Type of movement:</u>
<i>SearchC</i>	TCP along a circular path
<i>SearchL</i>	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.

<u>Instruction</u>	<u>Used to:</u>
<i>TriggIO</i> <sup>1</sup>	Define a trigg condition to set an output at a given position
<i>TriggInt</i> <sup>1</sup>	Define a trigg condition to execute a trap routine at a given position

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

<i>TriggEquip</i> <sup>1</sup>	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
<i>TriggC</i> <sup>1</sup>	Run the robot (TCP) circularly with an activated trigg condition
<i>TriggJ</i> <sup>1</sup>	Run the robot axis-by-axis with an activated trigg condition
<i>TriggL</i> <sup>1</sup>	Run the robot (TCP) linearly with an activated trigg condition
<u>Data type</u>	<u>Used to define:</u>
<i>triggdata</i> <sup>1</sup>	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.

<u>Instruction</u>	<u>Used to:</u>
<i>StopMove</i>	Stop the robot movements
<i>StartMove</i>	Restart the robot movements
<i>StorePath</i> <sup>1</sup>	Store the last path generated
<i>RestoPath</i> <sup>1</sup>	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.

<u>Instruction</u>	<u>Used to:</u>
<i>DeactUnit</i>	Deactivate an external mechanical unit
<i>ActUnit</i>	Activate an external mechanical unit

---

## 5.7 Independent axes

The robot axis 6 (and 4 on IRB 2400 /4400) 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.

<u>Function</u>	<u>Used to:</u>
<i>IndAMove</i> <sup>2</sup>	Change an axis to independent mode and move the axis to an absolute position
<i>IndCMove</i> <sup>2</sup>	Change an axis to independent mode and start the axis moving continuously
<i>IndDMove</i> <sup>2</sup>	Change an axis to independent mode and move the axis a delta distance
<i>IndRMove</i> <sup>2</sup>	Change an axis to independent mode and move the axis to a relative position (within the axis revolution)
<i>IndReset</i> <sup>2</sup>	Change an axis to dependent mode or/and reset the working area
<i>IndInpos</i> <sup>2</sup>	Check whether an independent axis is in position
<i>IndSpeed</i> <sup>2</sup>	Check whether an independent axis has reached programmed speed
<u>Instruction</u>	<u>Used to:</u>
<i>HollowWristReset</i> <sup>2</sup>	Reset the position of the wrist joints on hollow wrist manipulators, such as IRB 5402 and IRB 5403.

2. Only if the robot is equipped with the option "Advanced Motion".  
(The Instruction HollowWristReset can only be used on Robot IRB 5402 AND IRB 5403)

---

## 5.8 Position functions

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



---

## 5.9 Motion data

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

<u>Data type</u>	<u>Used to define:</u>
<i>robtarg</i>	The end position
<i>jointtarg</i>	The end position for a <i>MoveAbsJ</i> instruction
<i>speeddata</i>	The speed
<i>zonedata</i>	The accuracy of the position (stop point or fly-by point)
<i>tooldata</i>	The tool coordinate system and the load of the tool
<i>wobjdata</i>	The work object coordinate system

---

## 5.10 Basic data for movements

<u>Data type</u>	<u>Used to define:</u>
<i>pos</i>	A position (x, y, z)
<i>orient</i>	An orientation
<i>pose</i>	A coordinate system (position + orientation)
<i>confdata</i>	The configuration of the robot axes
<i>extjoint</i>	The position of the external axes
<i>robjoint</i>	The position of the robot axes
<i>o_robtarg</i>	Original robot position when <i>Limit ModPos</i> is used
<i>o_jointtarg</i>	Original robot position when <i>Limit ModPos</i> is used for <i>MoveAbsJ</i>
<i>loaddata</i>	A load
<i>mecunit</i>	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

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

<u>Function</u>	<u>Used to read:</u>
<i>DOutput</i>	The value of a digital output signal
<i>GOutput</i>	The value of a group of digital output signals
<i>AOutput</i>	The current value from an analog output signal

---

## 6.5 Testing input on output signals

<u>Instruction</u>	<u>Used to:</u>
WaitDI	Wait until a digital input is set or reset
WaitDO	Wait until a digital output is set on reset
<u>Function</u>	<u>Used to:</u>
TestDI	Test whether a digital input is set

---

## 6.6 Disabling and enabling I/O modules

I/O modules are automatically enabled at start-up, but they can be disabled during program execution and re-enabled later.

<u>Instruction</u>	<u>Used to:</u>
IODisable	Disable an I/O module
IOEnable	Enable an I/O module

---

## 6.7 Defining input and output signals

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

<u>Instruction</u>	<u>Used to:</u>
<i>AliasIO</i> <sup>1</sup>	Define a signal with an alias name

\* Only to be defined using system parameters.

---

1. Only if the robot is equipped with the option “Developer’s Functions”

---

---

## 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

<u>Instruction</u>	<u>Used to:</u>
<i>TPEraser</i>	Clear the teach pendant operator display
<i>TPWrite</i>	Write text on the teach pendant operator display
<i>ErrWrite</i>	Write text on the teach pendant display and simultaneously store that message in the program's error log.
<i>TPReadFK</i>	Label the function keys and to read which key is pressed
<i>TPReadNum</i>	Read a numeric value from the teach pendant
<i>TPShow</i>	Choose a window on the teach pendant from RAPID

---

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

<u>Instruction</u>	<u>Used to:</u>
<i>Open</i> <sup>1</sup>	Open a channel/file for reading or writing
<i>Write</i> <sup>1</sup>	Write text to the channel/file
<i>Close</i> <sup>1</sup>	Close the channel/file
<u>Function</u>	<u>Used to:</u>
<i>ReadNum</i> <sup>1</sup>	Read a numeric value
<i>ReadStr</i> <sup>1</sup>	Read a text string

---

### 7.4 Communicating using binary serial channels/files

<u>Instruction</u>	<u>Used to:</u>
<i>Open</i> <sup>1</sup>	Open a serial channel/file for binary transfer of data
<i>WriteBin</i> <sup>1</sup>	Write to a binary serial channel/file
<i>WriteStrBin</i> <sup>1</sup>	Write a string to a binary serial channel/file
<i>Rewind</i> <sup>1</sup>	Set the file position to the beginning of the file
<i>Close</i> <sup>1</sup>	Close the channel/file
<u>Function</u>	<u>Used to:</u>
<i>ReadBin</i> <sup>1</sup>	Read from a binary serial channel

---

### 7.5 Data for serial channels

<u>Data type</u>	<u>Used to define:</u>
<i>iodev</i>	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

<u>Instruction</u>	<u>Used to:</u>
CONNECT	Connect a variable (interrupt identity) to a trap routine

---

### 8.3 Ordering interrupts

<u>Instruction</u>	<u>Used to order:</u>
<i>ISignalDI</i>	An interrupt from a digital input signal
<i>SignalDO</i>	An interrupt from a digital output signal
<i>ITimer</i>	A timed interrupt
<i>TriggInt</i> <sup>1</sup>	A position-fixed interrupt (from the Motion pick list )

---

### 8.4 Cancelling interrupts

<u>Instruction</u>	<u>Used to:</u>
<i>IDelete</i>	Cancel (delete) an interrupt

---

### 8.5 Enabling/disabling interrupts

<u>Instruction</u>	<u>Used to:</u>
<i>ISleep</i>	Deactivate an individual interrupt
<i>IWatch</i>	Activate an individual interrupt
<i>IDisable</i>	Disable all interrupts
<i>IEnable</i>	Enable all interrupts

---

### 8.6 Data type of interrupts

<u>Data type</u>	<u>Used to define:</u>
<i>intnum</i>	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

<u>Instruction</u>	<u>Used to:</u>
<i>RAISE</i>	“Create” an error and call the error handler

**9.3 Restarting/returning from the error handler**

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

---

**9.4 Data for error handling**

<u>Data type</u>	<u>Used to define:</u>
<i>errnum</i>	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

<u>Instruction</u>	<u>Used to:</u>
<i>ClkReset</i>	Reset a clock used for timing
<i>ClkStart</i>	Start a clock used for timing
<i>ClkStop</i>	Stop a clock used for timing
<u>Function</u>	<u>Used to:</u>
<i>ClkRead</i>	Read a clock used for timing
<u>Data Type</u>	<u>Used for:</u>
<i>clock</i>	Timing – stores a time measurement in seconds

---

### 10.3 Reading current time and date

<u>Function</u>	<u>Used to:</u>
<i>CDate</i>	Read the Current Date as a string
<i>CTime</i>	Read the Current Time as a string
<i>GetTime</i>	Read the Current Time as a numeric value

**10.4 Retrieve time information from file**Function*FileTime**ModTime*Used to:

Retrieve the last time for modification of a file.

Retrieve the time of loading a specified module.

---

---

## 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

<u>Instruction</u>	<u>Used to:</u>
<i>Clear</i>	Clear the value
<i>Add</i>	Add or subtract a value
<i>Incr</i>	Increment by 1
<i>Decr</i>	Decrement by 1

---

### 11.3 More advanced calculations

<u>Instruction</u>	<u>Used to:</u>
$:=$	Perform calculations on any type of data

---

## 11.4 Arithmetic functions

<u>Function</u>	<u>Used to:</u>
<i>Abs</i>	Calculate the absolute value
<i>Round</i>	Round a numeric value
<i>Trunc</i>	Truncate a numeric value
<i>Sqrt</i>	Calculate the square root
<i>Exp</i>	Calculate the exponential value with the base “e”
<i>Pow</i>	Calculate the exponential value with an arbitrary base
<i>ACos</i>	Calculate the arc cosine value
<i>ASin</i>	Calculate the arc sine value
<i>ATan</i>	Calculate the arc tangent value in the range [-90,90]
<i>ATan2</i>	Calculate the arc tangent value in the range [-180,180]
<i>Cos</i>	Calculate the cosine value
<i>Sin</i>	Calculate the sine value
<i>Tan</i>	Calculate the tangent value
<i>EulerZYX</i>	Calculate Euler angles from an orientation
<i>OrientZYX</i>	Calculate the orientation from Euler angles
<i>PoseInv</i>	Invert a pose
<i>PoseMult</i>	Multiply a pose
<i>PoseVect</i>	Multiply a pose and a vector
<i>Vectmagn</i>	Calculate the magnitude of a <i>pos</i> vector.
<i>DotProd</i>	Calculate the dot (or scalar) product of two <i>pos</i> vectors.
<i>NOrient</i>	Normalise unnormalised orientation (quaternion)

---

---

## 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 is a package that can be extensively customised. It is the intention that the user adapts the routines to suit 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
- User defined open/close gun and supervision
- User defined pressure setting
- User defined preclose time calculation
- 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

- SpotL current data information
- Spot identity info: the current spotdata parameter name (string format)
- Spot identity transfer to the serial weld timer BOSCH PSS 5000
- User defined autonomous supervision, such as state-controlled weld current signal and water cooling start. Note: This feature requires the MultiTasking option
- Manual weld, gun open and gun close initiated by digital input
- Weld process start disregarding the in position event, is possible
- Optional user defined error recovery

---

## 12.2 Principles of SpotWare

SpotWare is based on a separate handling of motion, spot welding and, if MultiTasking is installed, continuous 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.

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.

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 and they may contain additional gun supervision.

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. The weld timer may be of the type, program schedule or start signal triggered.
- 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. (Note: MultiTasking option required).

---

### 12.3 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 modules 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

---

### 12.4 Spot welding instructions

<u>Instruction</u>	<u>Used to:</u>
<i>SpotL</i>	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

---

### 12.5 Spot welding data

<u>Data type</u>	<u>Used to define:</u>
<i>spotdata</i>	The spot weld process control
<i>gundata</i>	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

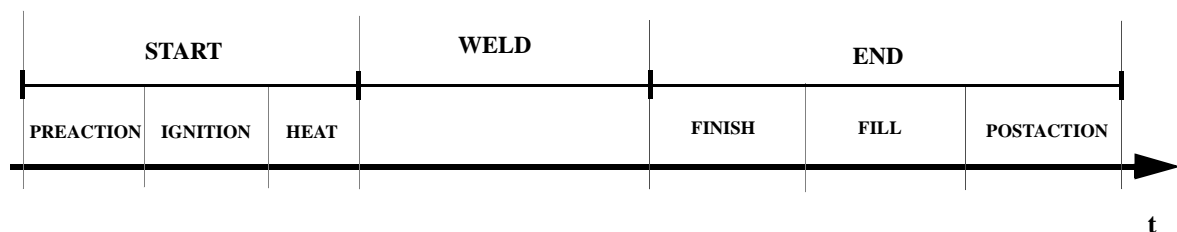
The same instructions are used both to control the robot's movements and the actual welding process. The arc welding instructions indicate which weld data and seam data are used in the weld in question.

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

<u>Instruction</u>	<u>Type of movement:</u>
<i>ArcC</i>	TCP along a circular path
<i>ArcL</i>	TCP along a linear path

**13.3 Arc welding data**Data type*welddata**seamdata**weavedata*Used to define:

The weld phase

The start and end phase of a weld

The weaving characteristics

---

## 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

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

---

14.4 Glue data

<u>Data type</u>	<u>Used to define:</u>
<i>ggundata</i>	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

*SCWrite*<sup>1</sup>

Used to:

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.

<u>Instruction</u>	<u>Used to:</u>
<i>TestSign</i>	Define and activate a test signal
<u>Instruction</u>	<u>Used to:</u>
<i>GetSysData</i>	Fetch data for and name of current active Tool or Work Object.
<u>Data type</u>	<u>Used to define:</u>
<i>testsignal</i>	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

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

---

### 17.2 Comparison and Searching

<u>Operator</u>	<u>Used to:</u>
<i>=</i>	Test if equal to
<i>&lt;&gt;</i>	Test if not equal to
<u>Function</u>	<u>Used to:</u>
<i>StrMemb</i>	Check if character belongs to a set
<i>StrFind</i>	Search for character in a string
<i>StrMatch</i>	Search for pattern in a string
<i>StrOrder</i>	Check if strings are in order

**17.3 Conversion**

<u>Function</u>	<u>Used to:</u>
<i>NumToStr</i>	Convert a numeric value to a string
<i>ValToStr</i>	Convert a value to a string
<i>StrToVal</i>	Convert a string to a value
<i>StrMap</i>	Map a string
<i>StrToByte</i>	Convert a string to a byte
<i>ByteToStr</i>	Convert a byte to string data

---

---

## 18 Multitasking

**Multitasking RAPID** is a way to execute programs in (pseudo) parallel with the normal execution. 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. (See Basic Characteristics Multitasking.)

---

### 18.1 Basics

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 (array dimension) 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....).

---

### 18.2 Resource access Protection

<u>Function</u>	<u>Used to</u>
<i>TestAndSet</i>	Retrieve exclusive right to specific RAPID code areas or system resources.



---

---

## 19 Syntax Summary

---

### 19.1 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

**CorrCon Descr**

**CorrDiscon Descr**

**CorrWrite Descr Data**

**CorrClear**

**DeactUnit** MecUnit

**Decr** Name

**EOffsSet** EAxOffs

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

**Exit**

**ExitCycle**

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

**GOTO** Label

**GripLoad** Load

**GetSysData** DestObject [ \ObjectName ]

**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

**IODisable** UnitName MaxTime

**IOEnable** UnitName MaxTime

**ISignalDI** [ \Single ] Signal TriggValue Interrupt

**ISignalDO** [ \Single ] Signal TriggValue Interrupt



**ISleep** Interrupt

**ITimer** [ \Single ] Time Interrupt

**IVarValue** VarNo Value, 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 ]

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

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

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

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

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

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

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

**MoveLSync** ToPoint Speed [ \T ] Zone Tool  
[ \WObj ] ProcName

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

**PathResol** Value

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

**PDispSet** DispFrame

**Procedure** { Argument }

**PulseDO** [ \PLength ] Signal

**RAISE** [ Error no ]

**Reset** Signal

**RETURN** [ Return value ]

**Rewind** IODevice

**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 ]

**TPShow** Window

**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  
**WriteStrBin** IODevice Str  
**WZBoxDef** [ \Inside ] | [ \Outside ] Shape LowPoint HighPoint 1  
**WZCylDef** [ \Inside ] | [ \Outside ] Shape CentrePoint Radius Height  
**WZDisable** WorldZone  
**WZDOSet** [ \Temp ] | [ \Stat ] WorldZone [ \Inside ] | [ \Before ] Shape  
 Signal SetValue  
**WZEnable** WorldZone  
**WZFree** WorldZone  
**WZLimSup** [ \Temp ] | [ \Stat ] WorldZone Shape  
**WZSphDef** [ \Inside ] | [ \Outside ] Shape CentrePoint Radius

---

## 19.2 Functions

**Abs** (Input)

**ACos** (Value)

**AOutput** (Signal)

**ArgName** (Parameter)

**ASin** (Value)

**ATan** (Value)

**ATan2** (Y X)

**ByteToStr** (ByteData [\Hex] | [\Okt] | [\Bin] | [\Char])

**ClkRead** (Clock)

**CorrRead**

**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)

**DotProd** (Vector1 Vector2)

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

**Exp** (Exponent)

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

**GOutput** (Signal)

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

**IndInpos** MecUnit Axis

**IndSpeed** MecUnit Axis [\InSpeed] | [\ZeroSpeed]  
**IsPers** (DatObj)  
**IsVar** (DatObj)  
**MirPos** (Point MirPlane [\WObj] [\MirY])  
**ModTime** ( Object )  
**NOrient** (Rotation)  
**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)

**StrToByte** (ConStr [\Hex] | [\Okt] | [\Bin] | [\Char])

**StrToVal** ( Str Val )

**Tan** (Angle)

**TestDI** (Signal)

**TestAndSet** Object

**Trunc** ( Val [\Dec] )

**ValToStr** ( Val )

**VectMagn** (Vector)

---

# 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.

ALIAS	AND	BACKWARD	CASE
CONNECT	CONST	DEFAULT	DIV
DO	ELSE	ELSEIF	ENDFOR
ENDFUNC	ENDIF	ENDMODULE	ENDPROC
ENDRECORD	ENDTEST	ENDTRAP	ENDWHILE
ERROR	EXIT	FALSE	FOR
FROM	FUNC	GOTO	IF
INOUT	LOCAL	MOD	MODULE
NOSTEPIN	NOT	NOVIEW	OR
PERS	PROC	RAISE	READONLY
RECORD	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. String length max. 80 characters.

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.

<u>Placeholder</u>	<u>Represents:</u>
<TDN>	data type definition
<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 LANGUAGE:ENGLISH %%%	(Program version M94 or M94A) (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 | <sup>1)</sup> | Ñ | Ò | Ó | Ô | Õ | Ö | Ø  
 | Ù | Ú | Û | Ü | <sup>2)</sup> | <sup>3)</sup> | 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 | <sup>1)</sup> | ñ | ò | ó | ô | õ | ö  
 | ø | ù | ú | û | ü | <sup>2)</sup> | <sup>3)</sup> | 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).

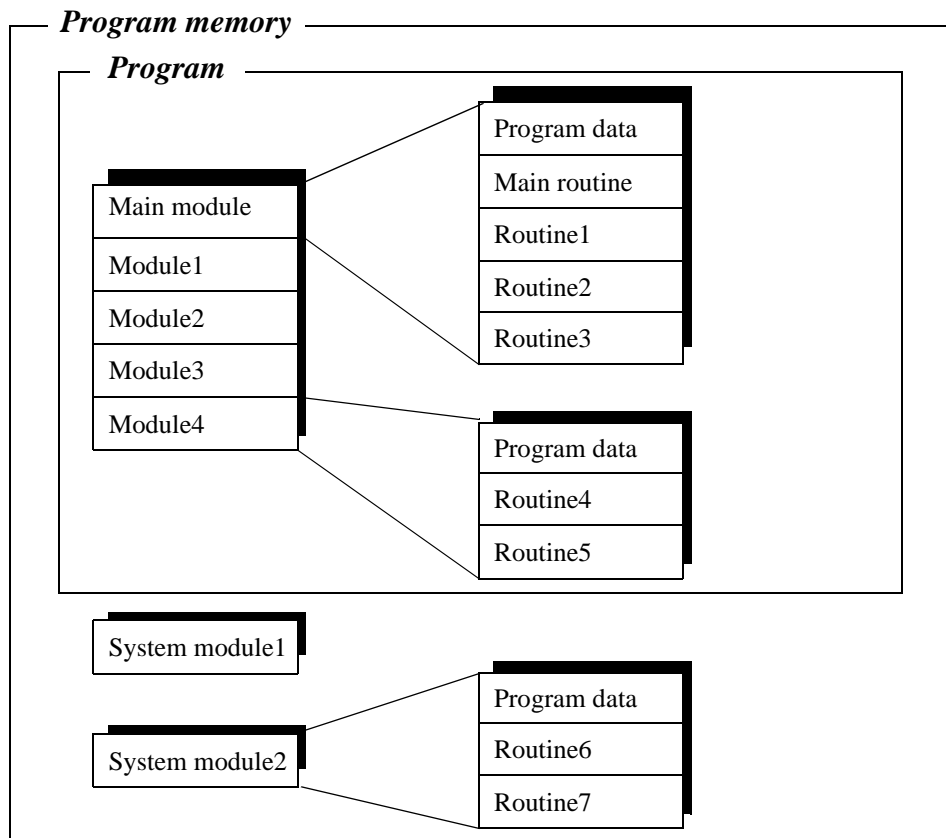


Figure 1 The program can be divided into modules.

### 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:

<u>Attribute</u>	<u>If specified, the module:</u>
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.	<pre> MODULE module_name (SYSMODULE, VIEWONLY) !data type definition !data declarations !routine declarations ENDMODULE </pre>

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> ]
    <type definition 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.)

<type definition list> ::= { <type definition> }  
<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.

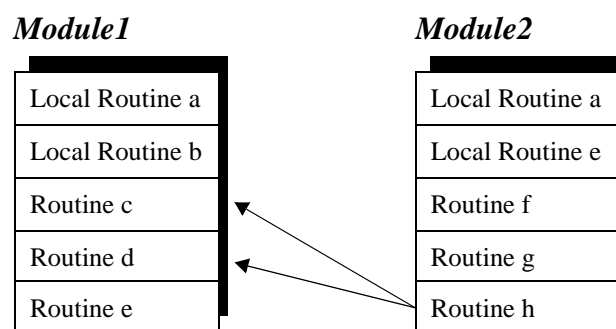


Figure 2 Example: The following routines can be called from Routine h:  
 Module1 - Routine c, d.  
 Module2 - All routines.

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.

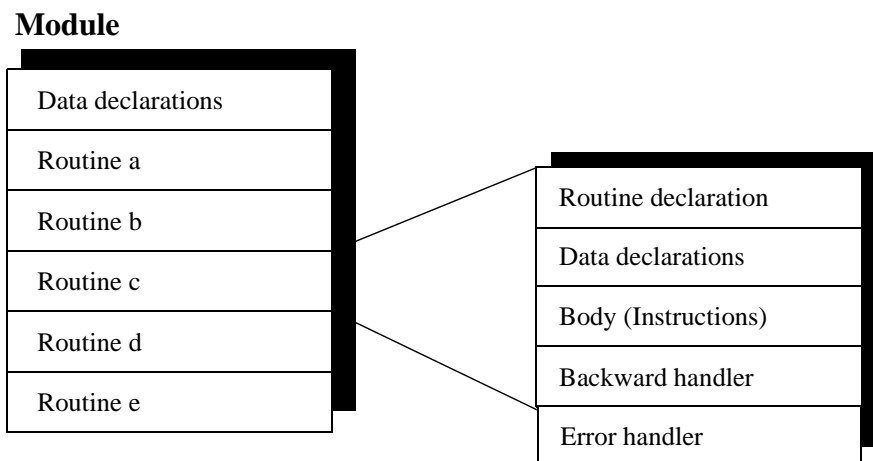


Figure 3 A routine can contain declarations, data, a body, a backward handler and an error handler.

***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 late 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. If a reference is made to an unknown procedure using late binding, the system variable ERRNO is set to ERR\_REFUNKPRC. If a reference is made to a procedure call error (syntax, not procedure) using late binding, the system variable ERRNO is set to ERR\_CALLPROC.

---

## 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>
    | ',' <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>  
    | ',' <optional procedure argument>  
    | <conditional 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*, *signalai*, *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;  
                      level:= high;

This is OK since dionum is an alias  
data type for num

---

### 4.3 Syntax

```
<type definition> ::=
[LOCAL] ( <record definition>
          | <alias definition> )
| <comment>
| <TDN>

<record definition> ::=
    RECORD <identifier>
        <record component list> ';'
    ENDRECORD

<record component list> ::=
    <record component definition> |
    <record component definition> <record component list>

<record component definition> ::=
    <data type> <record component name>

<alias definition> ::=
    ALIAS <data type> <identifier> ';'

<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.

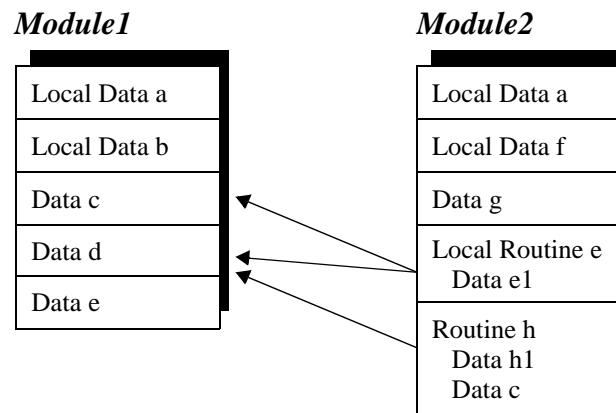


Figure 4 Example: The following data can be called from routine e:

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 always**

**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 repnt := [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;

It is possible to declare two persistents with the same name in different modules, if they are local within the module (PERS LOCAL), without any error being generated by the system (different data scope). But **note the limitation** that these two persistents always have the same current value (use the same storage in the memory).

---

## 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 see what is happening in various activities such as warm start, new program, program start etc.

Table 1

System event Affects	Power on (Warm start)	Open, Close or New program	Start program (Move PP to main)	Start program (Move PP to Routine)	Start program (Move PP to cursor)	Start program (Call Routine)	Start program (After cycle)	Start program (After stop)
Constant	Unchanged	Init	Init	Init	Unchanged	Unchanged	Unchanged	Unchanged
Variable	Unchanged	Init	Init	Init	Unchanged	Unchanged	Unchanged	Unchanged
Persistent	Unchanged	Init	Init	Init	Unchanged	Unchanged	Unchanged	Unchanged
Commanded interrupts	Re-ordered	Disappears	Disappears	Disappears	Unchanged	Unchanged	Unchanged	Unchanged
Start up routine SYS_RESET (with motion settings)	Not run	Run*	Run	Not run	Not run	Not run	Not run	Not run
Files	Closes	Closes	Closes	Closes	Unchanged	Unchanged	Unchanged	Unchanged
Path	Recreated at power on	Disappears	Disappears	Disappears	Disappears	Unchanged	Unchanged	Unchanged

\* Generates an error when there is a semantic error in the actual task program.

---

## 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:

<u>Instruction</u>	<u>Termination word</u>
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.  $a := \text{Abs}(3 * b)$ ;

### 7.1 Arithmetic expressions

An arithmetic expression is used to evaluate a numeric value.

Example:  $2 * \pi * \text{radius}$

Table 2 shows the different types of operations possible.

Table 2

Operator	Operation	Operand types	Result type
+	addition	num + num	num <sup>3)</sup>
+	unary plus; keep sign	+num or +pos	same <sup>1)3)</sup>
+	vector addition	pos + pos	pos
-	subtraction	num - num	num <sup>3)</sup>
-	unary minus; change sign	-num or -pos	same <sup>1)3)</sup>
-	vector subtraction	pos - pos	pos
*	multiplication	num * num	num <sup>3)</sup>
*	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 <sup>2)</sup>	integer division	num DIV num	num
MOD <sup>2)</sup>	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.  $14 \text{ DIV } 4 = 3$ ,  $14 \text{ MOD } 4 = 2$ .  
(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 <sup>1)</sup> = any <sup>1)</sup>	bool
>=	greater than or equal to	num >= num	bool
>	greater than	num > num	bool
<>	not equal to	any <sup>1)</sup> <> any <sup>1)</sup>	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

$\begin{array}{c c} a & b \end{array}$	True	False
True	True	False
False	False	False

a XOR b

$\begin{array}{c c} a & b \end{array}$	True	False
True	False	True
False	True	False

a OR b

$\begin{array}{c c} a & b \end{array}$	True	False
True	True	True
False	True	False

NOT b

$\begin{array}{c c} b & \end{array}$	
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:             $2 * \pi * \text{radius}$

### 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	<b>VAR</b> 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 sim-

ply 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

<u>Expression</u>	<u>Evaluation order</u>	<u>Comment</u>
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>
    | ',' <optional function argument>
    | <conditional 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 to 0";
        RETURN x;
    ENDIF
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 four 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.

When an error occurs in a routine that does not contain an error handler or when the end of the error handler is reached (**ENDFUNC**, **ENDPROC** or **ENDTRAP**), the *system error handler* is called. The system error handler just reports the error and stops the execution.

In a chain of routine calls, each routine may have its own error handler. If an error occurs in a routine with an error handler, and the error is explicitly propagated using the RAISE instruction, the same error is raised again at the point of the call of the routine - the error is *propagated*. When the top of the call chain (the entry routine of the task) is reached without any error handler being found or when the end of any error handler is reached within the call chain, the system error handler is called. The system error handler just reports the error and stops the execution. Since a trap routine can only be called by the system (as a response to an interrupt), any propagation of an error from a trap routine is made to 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 maximum number of defined interrupts at any one time is limited to **70 per program task**. The **total limitation** set by the I/O CPU is 100 interrupts.

---

### 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

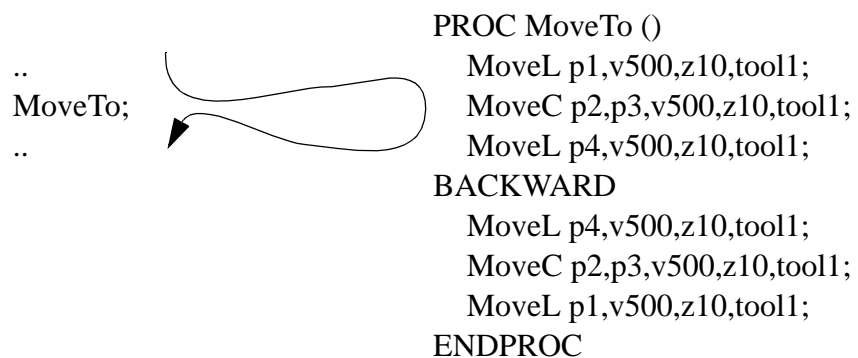
Procedures may contain a backward handler that defines the backward execution of a procedure call.

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.

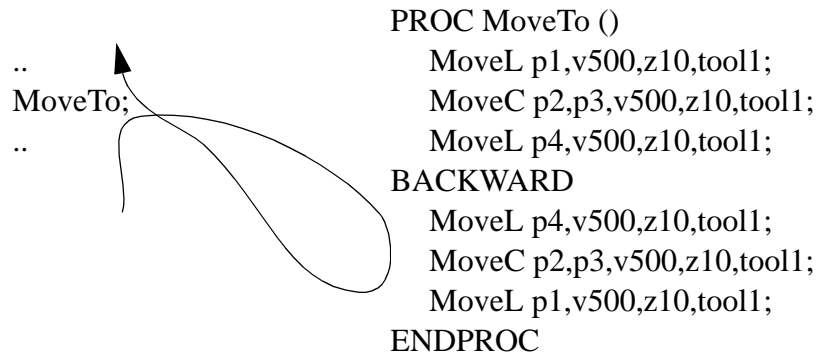
Example:

```
PROC MoveTo ()
  MoveL p1,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p4,v500,z10,tool1;
  BACKWARD
  MoveL p4,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p1,v500,z10,tool1;
ENDPROC
```

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



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



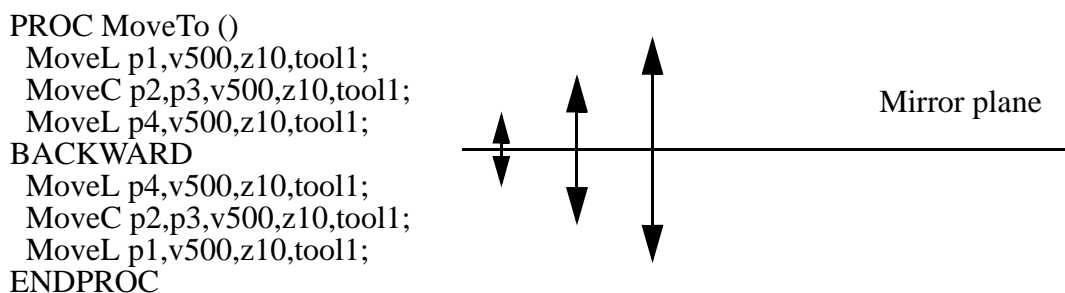
Instructions in the backward or error handler of a routine may not be executed backwards. Backward execution cannot be nested, i.e. two instructions in a call chain may not simultaneously be executed backwards.

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

---

## 10.2 Limitation of move instructions in the backward handler

The move instruction type and sequence in the backward handler must be a mirror of the move instruction type and sequence for forward execution in the same routine:



Note that the order of CirPoint  $p2$  and ToPoint  $p3$  in the MoveC should be the same.

By move instructions is meant all instructions that result in some movement of the robot or external axes such as MoveL, SearchC, TriggJ, ArcC, PaintL ...



**Any departures from this programming limitation in the backward handler can result in faulty backward movement. Linear movement can result in circular movement and vice versa, for some part of the backward path.**



---

---

## 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 (array dimension) 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 Pendant, 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 Pendant 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**

The instruction *SetDO* and *ISignalDO* are used.

**Example****TASK 0**

```
MODULE module1
PROC main()
```

```
SetDO do1,1;
```

```
    .
```

```
ENDPROC
ENDMODULE
```

### **TASK 1**

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

CONNECT isiint1 WITH isi_trap;
ISignalDO do1, 1, isiint1;

WHILE TRUE DO
    WaitTime 200;
ENDWHILE

IDelete isiint1;

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. A **SEMISTATIC** task will also in the restart sequence reload modules specified in the system parameters if the module file is newer than the loaded module.

It is also possible to set the task to type **NORMAL**, then it will behave in the same way as task 0 (the main task, controlling the robot movement). The teach pendant 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 robin 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 robin chain 1: tasks 0, 1, and 8 are busy

Round robin chain 2: tasks 0, 3, 4, 5 and 8 are busy  
tasks 1 and 2 are idle

Round robin chain 3: tasks 2, 4 and 5 are busy

tasks 0, 1, 8 and 9 are idle.

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

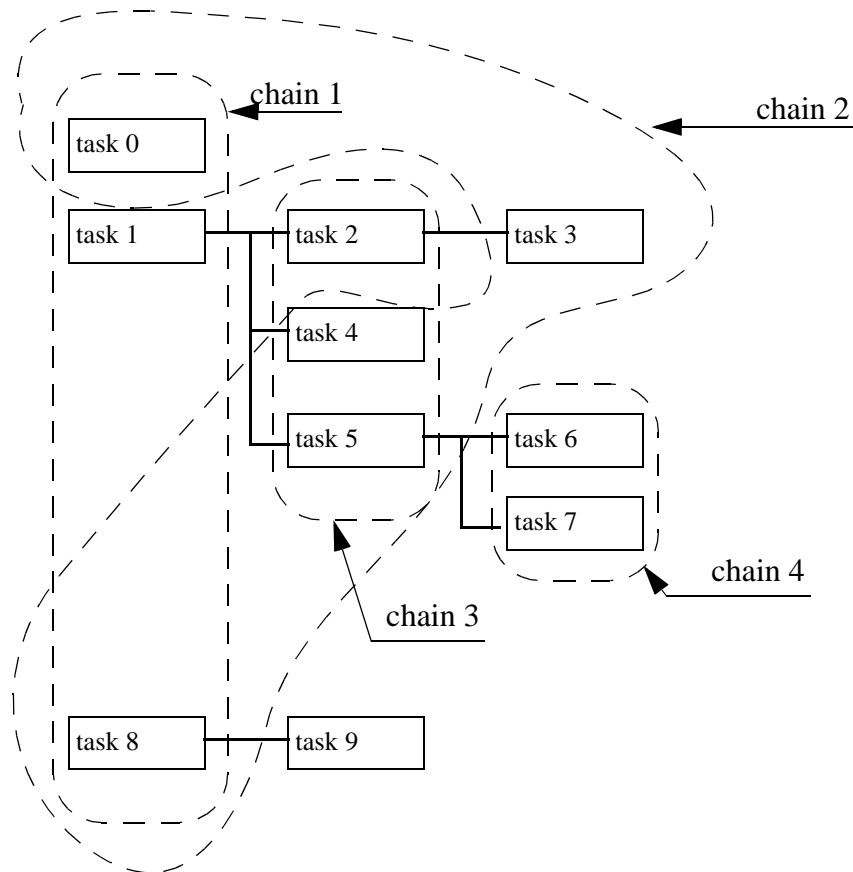


Figure 5 The tasks can have different priorities.

## 11.5 Trust Level

TrustLevel handel the system behavior when a **SEMISTATIC** or **STATIC** task is stopped for some reason or not executable.

**SysFail** - This is the default behaviour, all other **NORMAL** tasks (eg the MAIN task) will also stop, and the system is set to state **SYS\_FAIL**. All jogg and program start orders will be rejected. Only a new warm start reset the system. This should be used when the task has some security supervisions.

**SysHalt** - All **NORMAL** tasks will be stopped (normaly only the main task). The system is forced to “motors off”. When taking up the system to “motors on” it is possible to jogg the robot, but a new attempt to start the program will be rejected. A new warm start will reset the system.

**SysStop** - All **NORMAL** tasks will be stopped (eg the main task), but it is restartable. Jogging is also possible.

**NoSafety** - Only the actual task itself will stop.

See System parameters - Controller/Task

---

### 11.6 Task sizes

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

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*.

---

### 11.7 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 get any time to interact with the user. And if the supervision task is in foreground, it will never allow another task in background to execute.

---

### 11.8 Programming scheme

**The first time.**

1. Define the new task under system parameters (controller/task)  
Set the *type* to **NORMAL** and the *TrustLevel* to **NoSafety**.
2. Specify all modules that should be preloaded to this new task, also under system parameters (controller/task-modules).
3. Create the modules that should be in the task from the TeachPendant (in the MAIN task) or off-line.
4. Test and debug the modules in the MAIN task, until the functionality is satisfied. Note: this could only be done in **motors on** state.
5. Change the task *type* to **SEMISTATIC** (or **STATIC**).
6. Restart the system.

**Iteration phase.**

In many cases a iteration with point 3 and 5 is enough. It's only when the program has to be tested in the **MAIN** task and execution of the **RAPID** code in two task at the same time could confuse the user, all point should be used. Note: if a **STATIC** task is used it has to be forced to reload the new changed module and restarted from the beginning. If all point below is used it will take care of that for you.

1. Change the task type to **NORMAL** to inhibit the task. A **NORMAL** task will not start when the system restart and if it's not the **MAIN** task not it will be affected by the start button at the teach pendant.
2. Restart the system.
3. Load the moulde(s) to the **MAIN** task, test, change and save the module(s).  
Note: Don't save the task, save each module according to the system parameters.
4. Change back the task *type* to **SEMISTATIC** (or **STATIC**).
5. Restart the system.

**Finnish phase.**

1. Set the *TrustLevel* to desired level eg **SysFail**
2. Restart the system





---

# 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 8.

---

## 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).

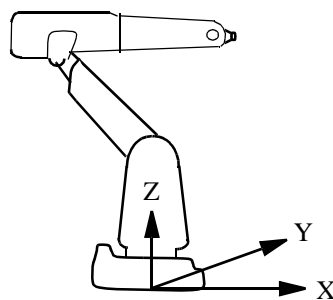


Figure 1 The base coordinate system.

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.

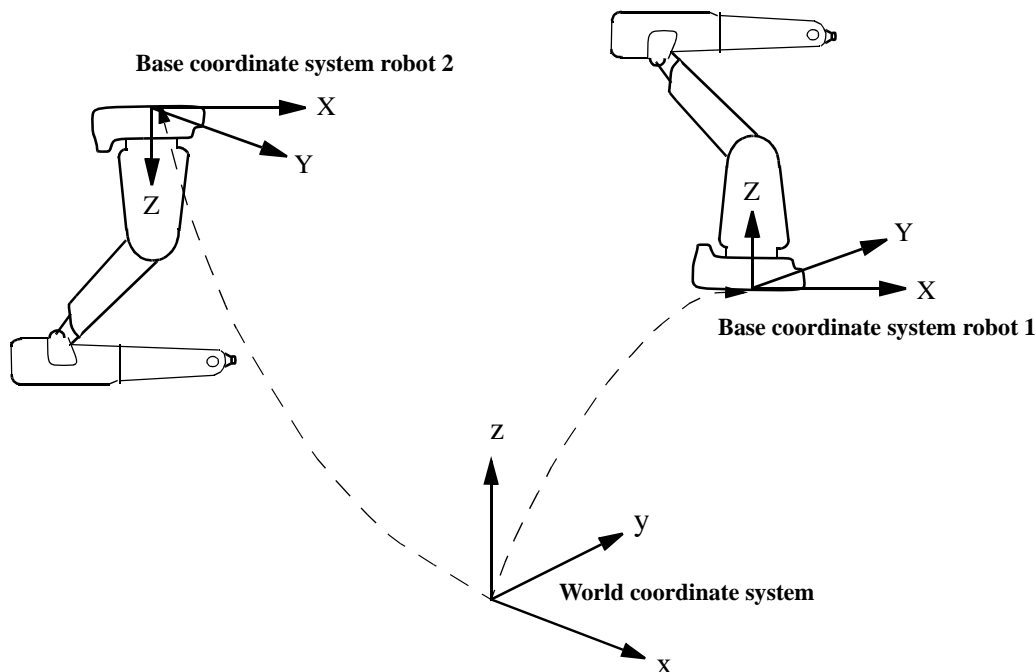


Figure 2 Two robots (one of which is suspended) with a common world coordinate system.

### 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).

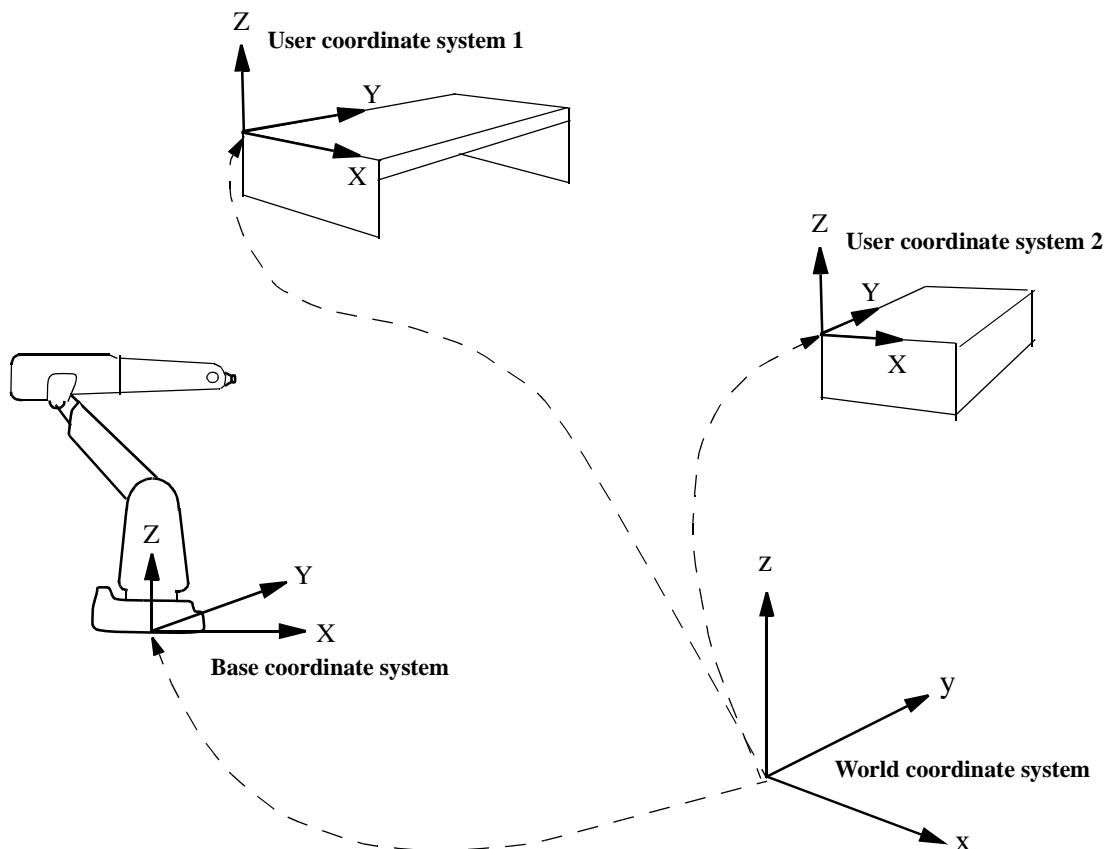


Figure 3 Two user coordinate systems describe the position of two different fixtures.

### 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).

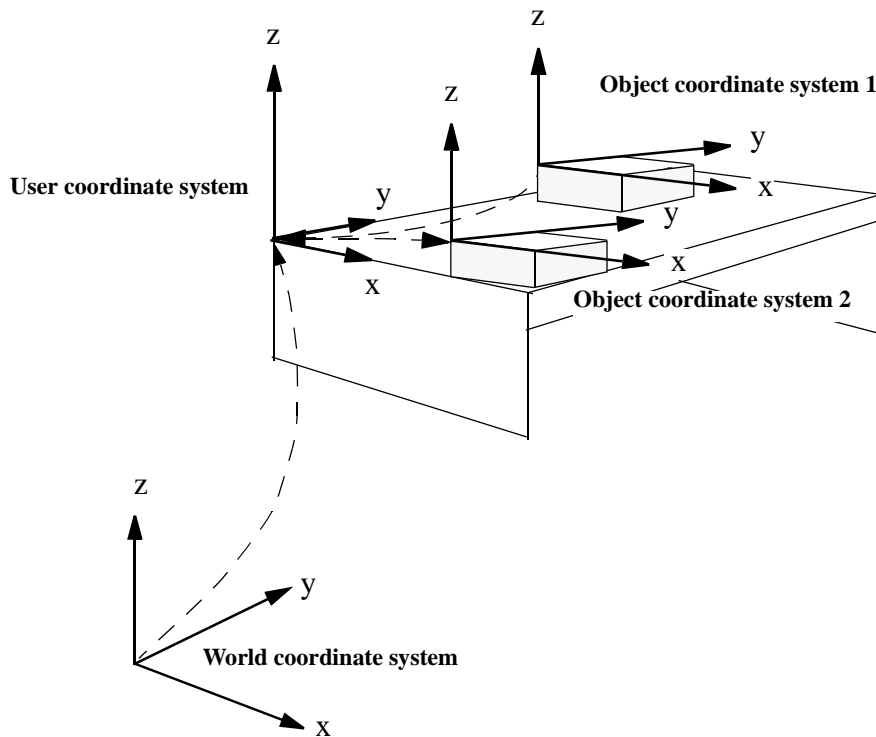


Figure 4 Two object coordinate systems describe the position of two different work objects located in the same fixture.

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).

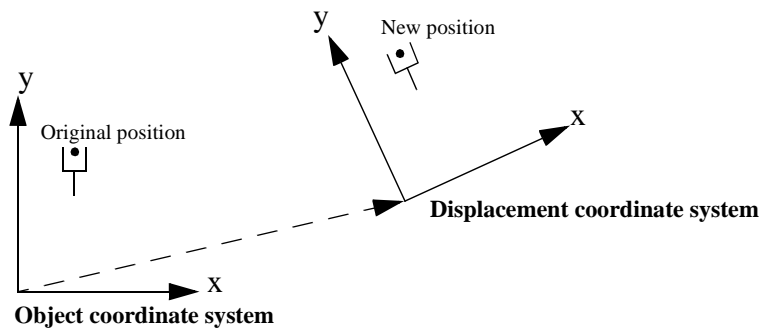


Figure 5 If program displacement is active, all positions are displaced.

### 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.

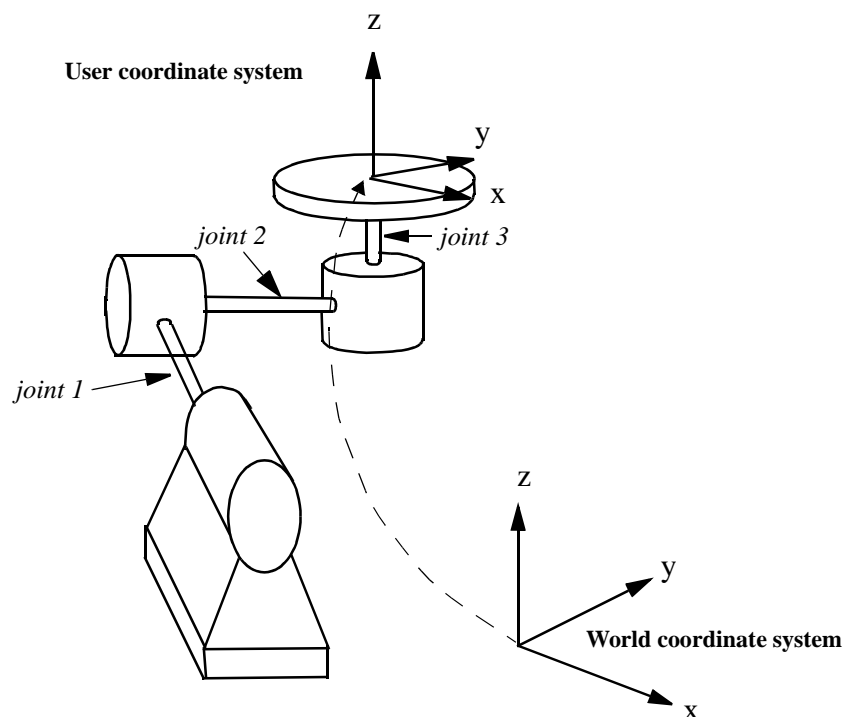


Figure 6 A user coordinate system, defined to follow the movements of a 3-axis external mechanical unit.

### *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.

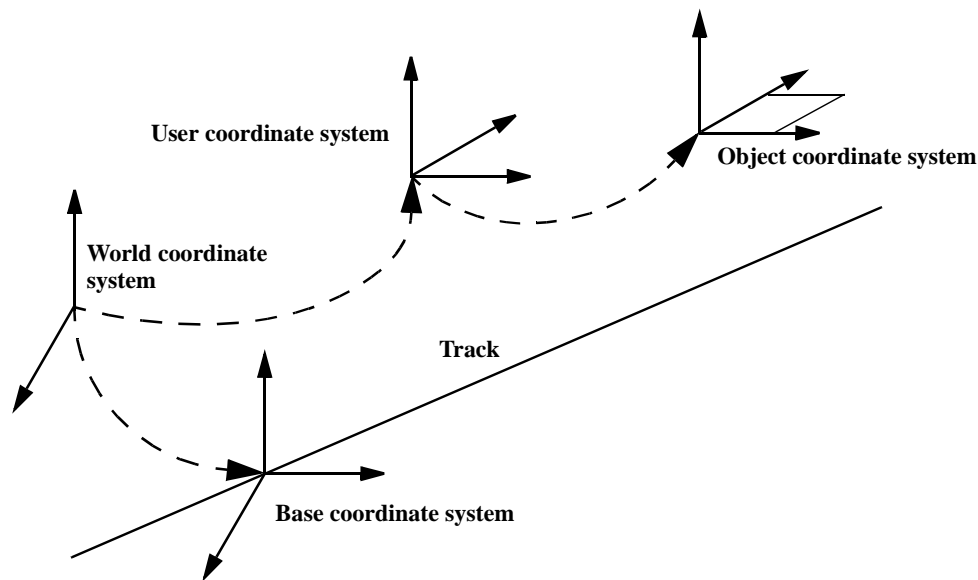


Figure 7 Coordinated interpolation with a track moving the base coordinate system of the robot.

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).

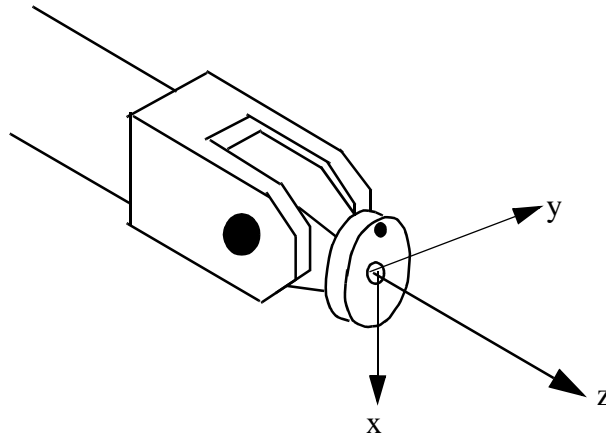


Figure 8 The wrist coordinate system.

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.

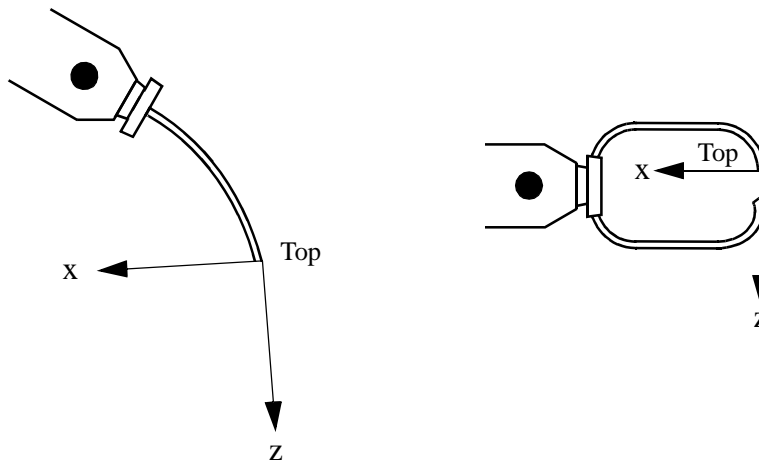


Figure 9 Tool coordinate system, as usually defined for an arc-welding gun (left) and a spot welding gun (right).

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

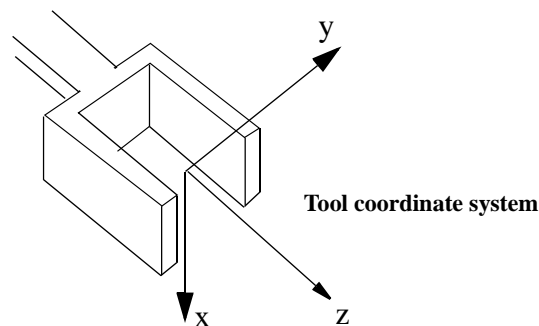


Figure 10 The tool coordinate system is defined relative to the wrist coordinate system, here for a gripper.

### 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.



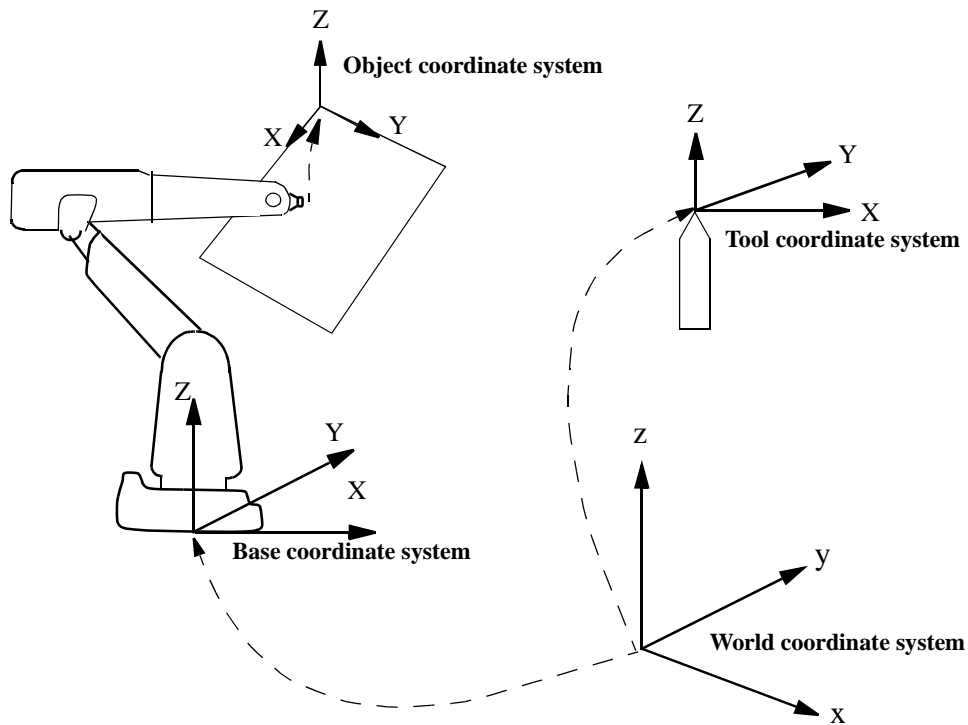


Figure 11 If a stationary TCP is used, the object coordinate system is usually based on the wrist coordinate system.

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.

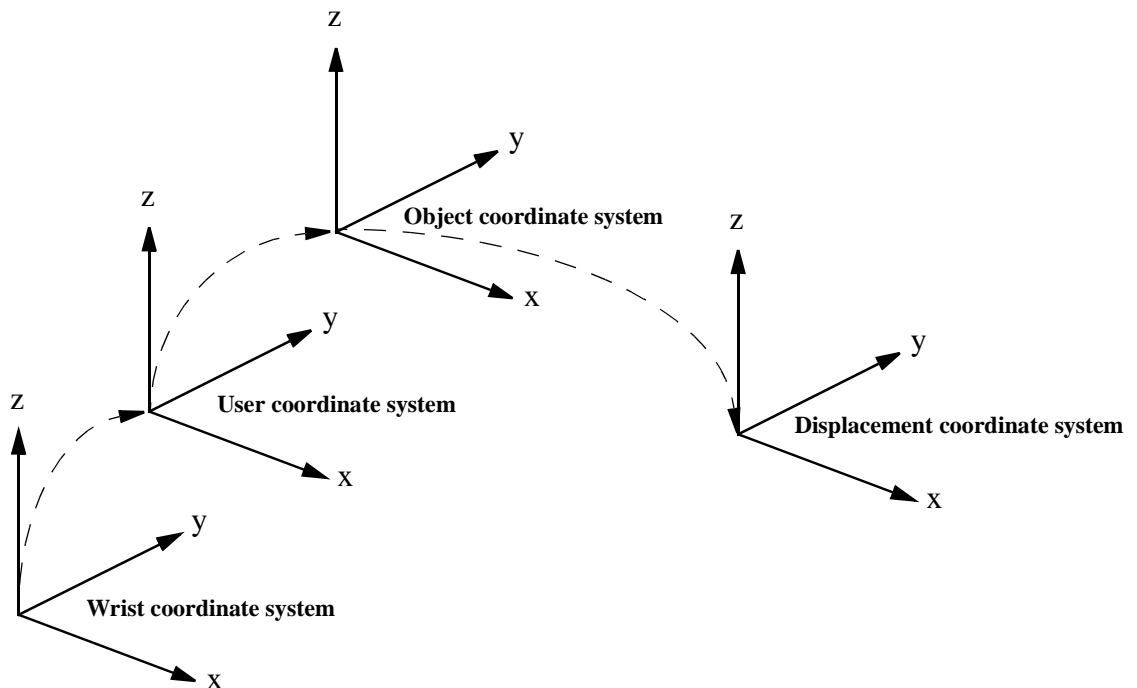


Figure 12 Program displacement can also be used together with stationary TCPs.

---

## 1.4 Related information

	<u>Described in:</u>
Definition of the world coordinate system	User's Guide - System Parameters
Definition of the user coordinate system	User's Guide - Calibration Data Types - <i>wobjdata</i>
Definition of the object coordinate system	User's Guide - Calibration Data Types - <i>wobjdata</i>
Definition of the tool coordinate system	User's Guide - Calibration Data Types - <i>tooldata</i>
Definition of a tool centre point	User's Guide - Calibration Data Types - <i>tooldata</i>
Definition of displacement frame	User's Guide - Calibration RAPID Summary - <i>Motion Settings</i>
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 pallet 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).

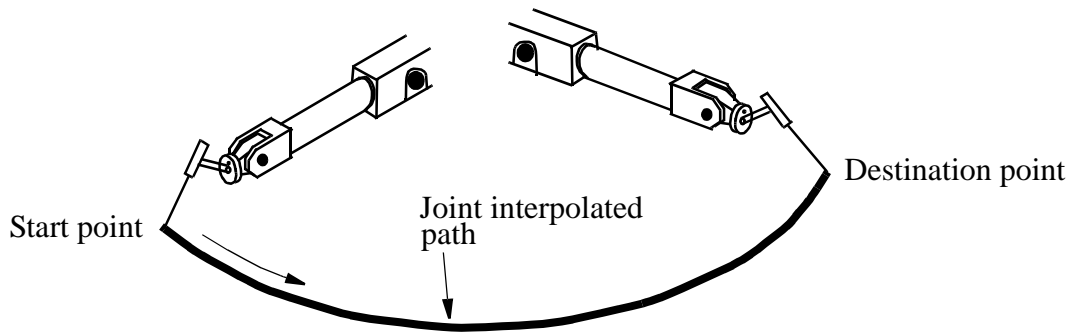


Figure 13 Joint interpolation is often the fastest way to move between two points as the robot axes follow the closest path between the start point and the destination point (from the perspective of the axis angles).

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).

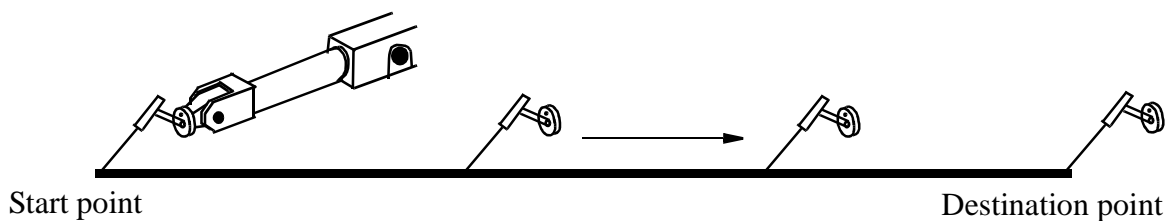


Figure 14 Linear interpolation without reorientation of the tool.

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 reorientated, 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.

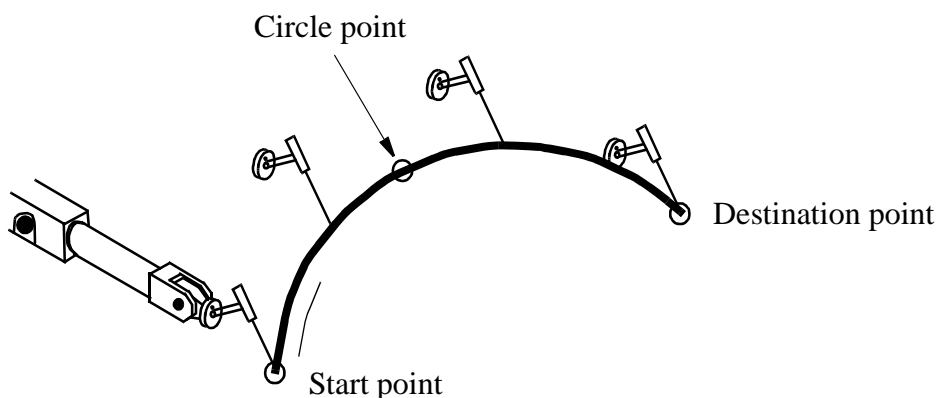


Figure 15 Circular interpolation with a short twist for part of a circle (circle segment) with a start point, circle point and destination point.

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

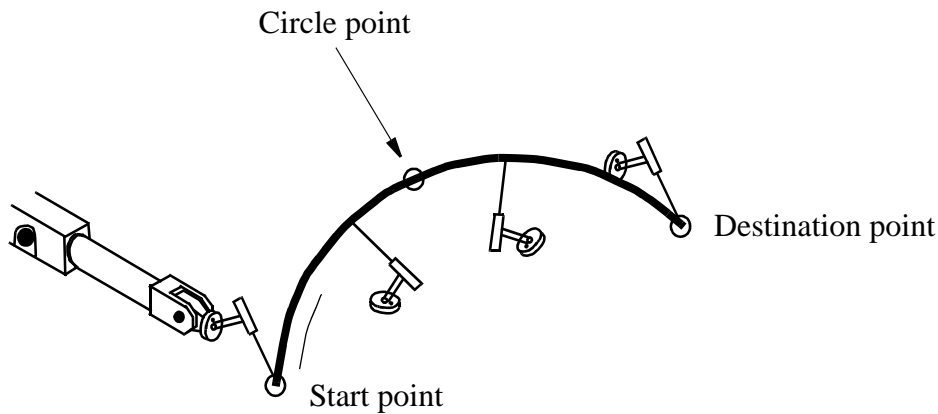


Figure 16 Circular interpolation with a long twist for orientation is achieved by defining the orientation in the circle point in the opposite direction compared to the start point.

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).

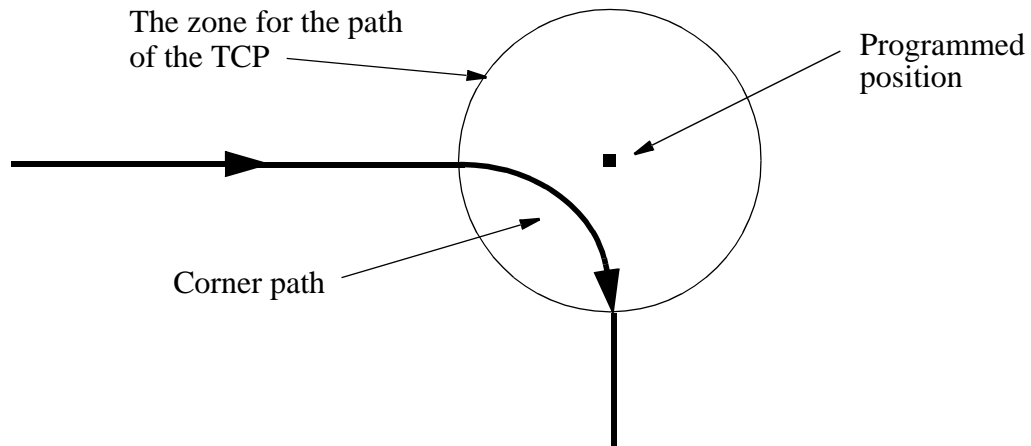


Figure 17 A fly-by point generates a corner path to pass the programmed position.

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.

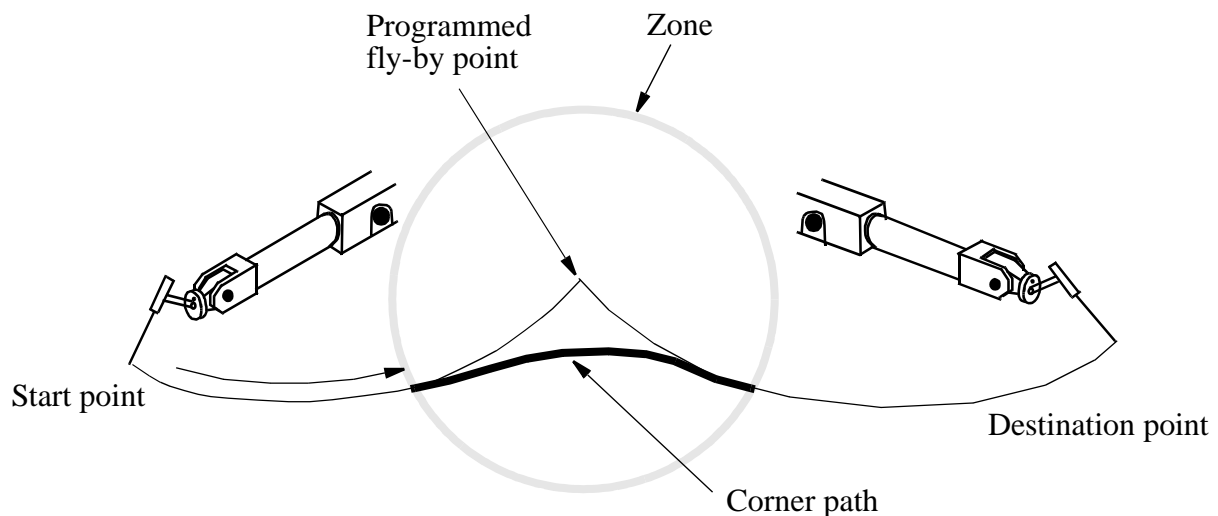


Figure 18 During joint interpolation, a corner path is generated in order to pass a fly-by point.

### 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).

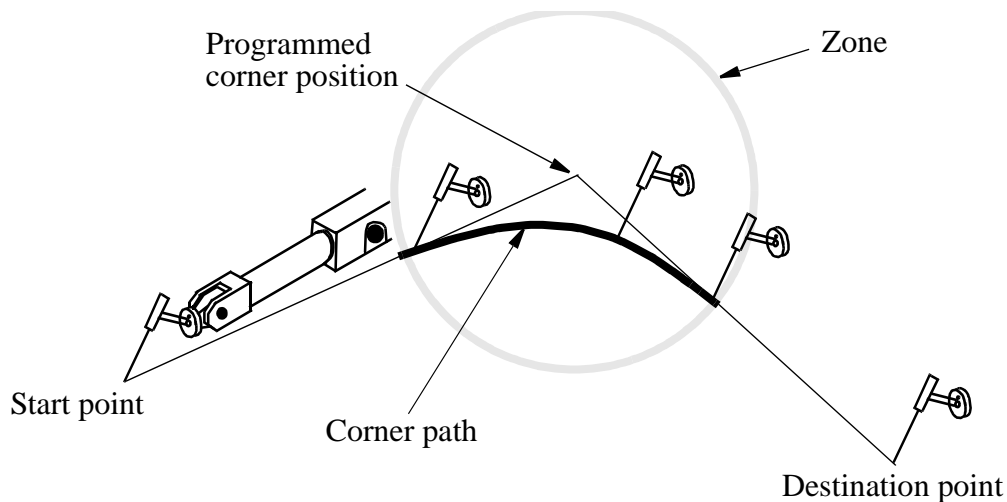


Figure 19 During linear interpolation, a corner path is generated in order to pass a fly-by point.

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 reorientated 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).

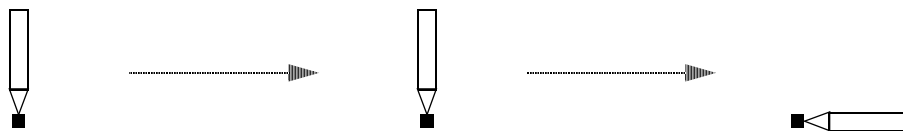


Figure 20a Three positions with different tool orientations are programmed as above.



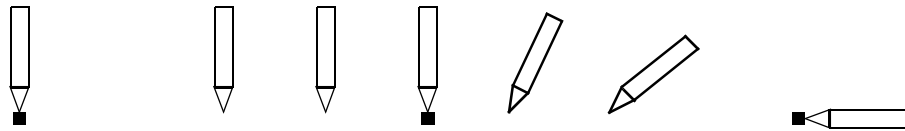


Figure 20b If all positions were stop points, program execution would look like this.

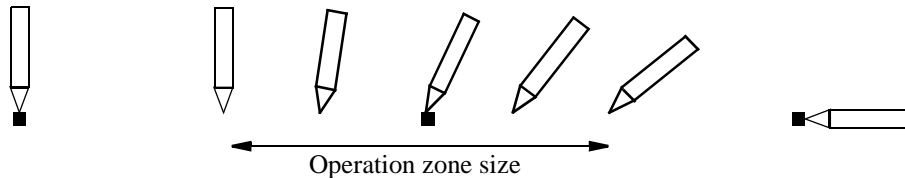


Figure 20c If the middle position was a fly-by point, program execution would look like this.

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 for 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).

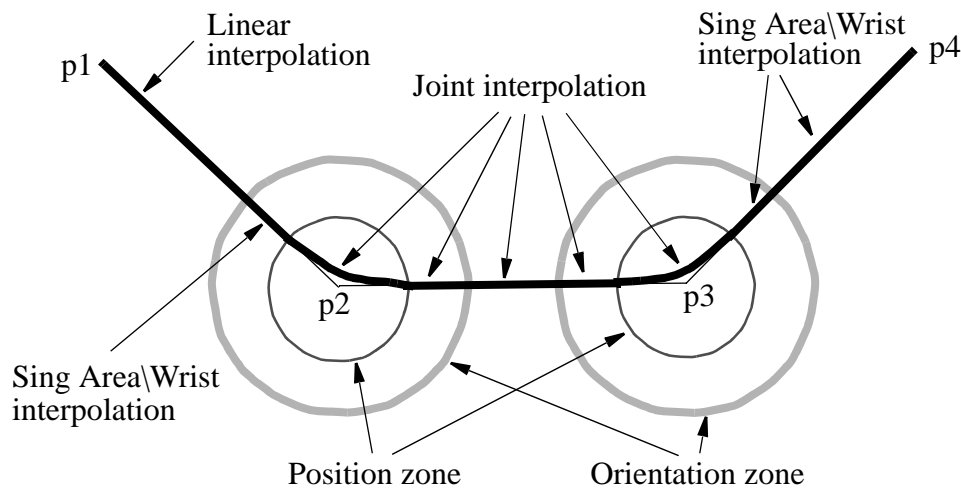


Figure 21 Interpolation when changing from one interpolation method to another. Linear interpolation has been programmed between p1 and p2; joint interpolation between p2 and p3; and Sing Area\Wrist interpolation between p3 and p4.

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.

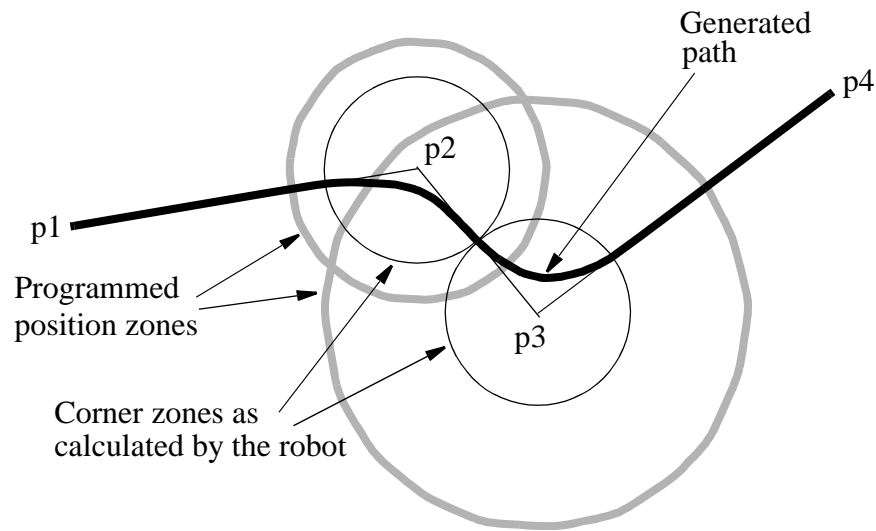


Figure 22 Interpolation with overlapping position zones. The zones around p2 and p3 are larger than half the distance from p2 to p3. Thus, the robot reduces the size of the zones to make them equal to half the distance from p2 to p3, thereby generating symmetrical corner paths within the zones.

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).

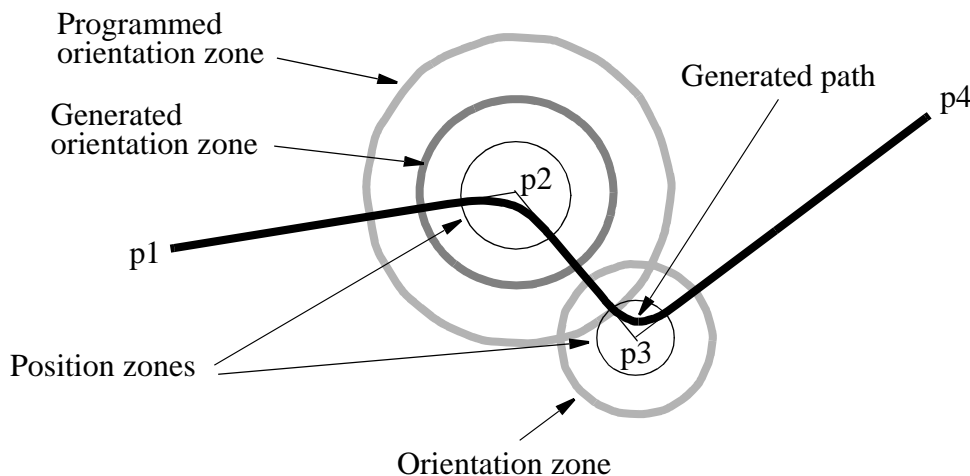


Figure 23 Interpolation with overlapping orientation zones. The orientation zone at p2 is larger than half the distance from p2 to p3 and is thus reduced to half the distance from p2 to p3. The position zones do not overlap and are consequently not reduced; the orientation zone at p3 is not reduced either.

### 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 immediately changes 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.

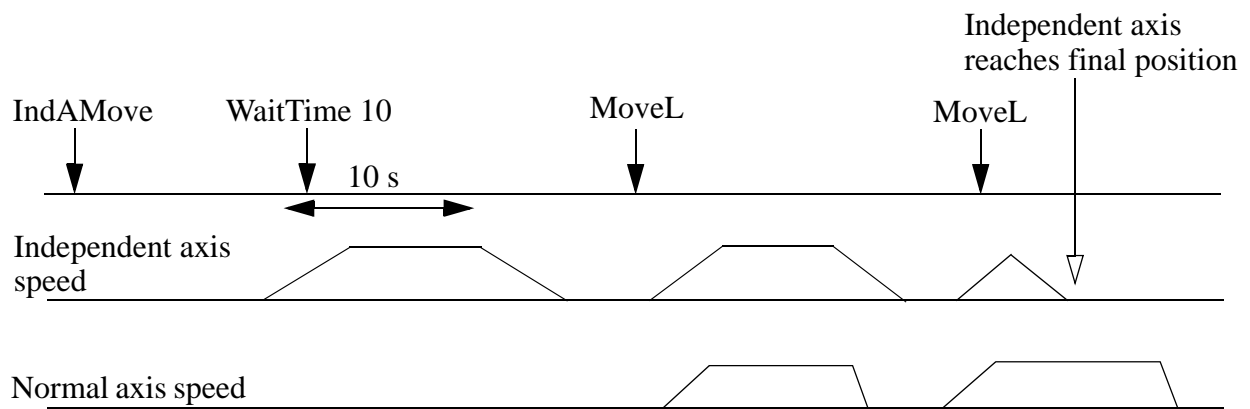


Figure 24 Stepwise execution of independent axes.

### 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.

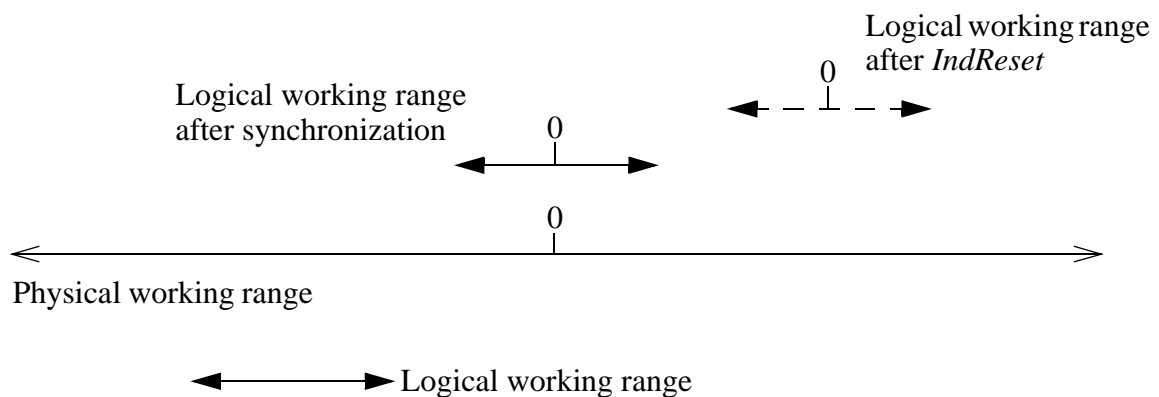


Figure 25 The logical working range can be moved, using the instruction *IndReset*.

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 an independent axis. Normally the *IndReset* instruction is used only for this axis. However, the *IndReset* instruction can also be used for axis 4 on IRB 2400 and 4400 models. If *IndReset* is used for robot axis 4, then axis 6 must not be in the independent mode.

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 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*.

---

**2.7 Related information**

	<u>Described in:</u>
Definition of speed	Data Types - <i>speeddata</i>
Definition of zones (corner paths)	Data Types - <i>zonedata</i>
Instruction for joint interpolation	Instructions - <i>MoveJ</i>
Instruction for linear interpolation	Instructions - <i>MoveL</i>
Instruction for circular interpolation	Instructions - <i>MoveC</i>
Instruction for modified interpolation	Instructions - <i>SingArea</i>
Singularity	Motion and I/O Principles- <i>Singularity</i>
Concurrent program execution	Motion and I/O Principles- <i>Synchronisation with logical instructions</i>
CPU Optimization	User's Guide - System parameters



### 3 Synchronisation 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).

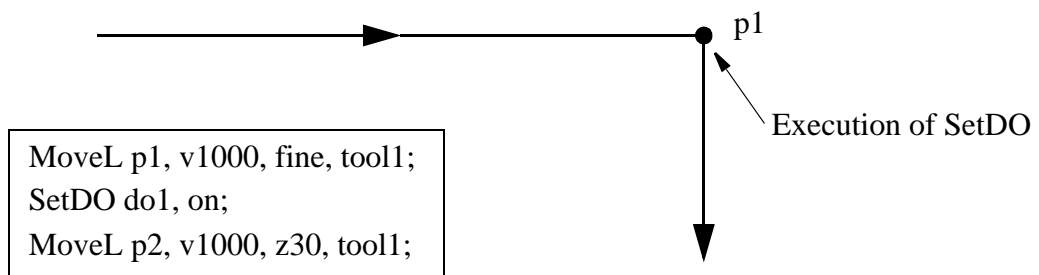


Figure 26 A logical instruction after a stop point is not executed until the destination position has been reached.

#### 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 and Figure 28. These instructions are then executed in order.

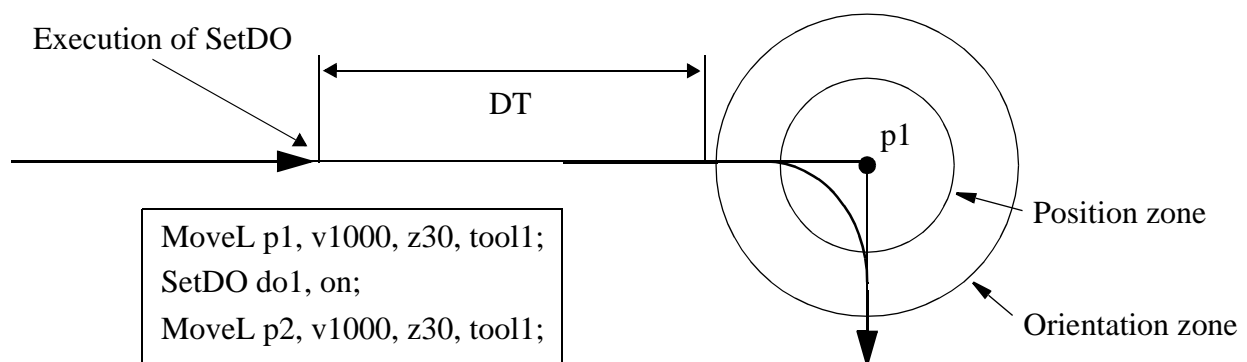


Figure 27 A logical instruction following a fly-by point is executed before reaching the largest zone.

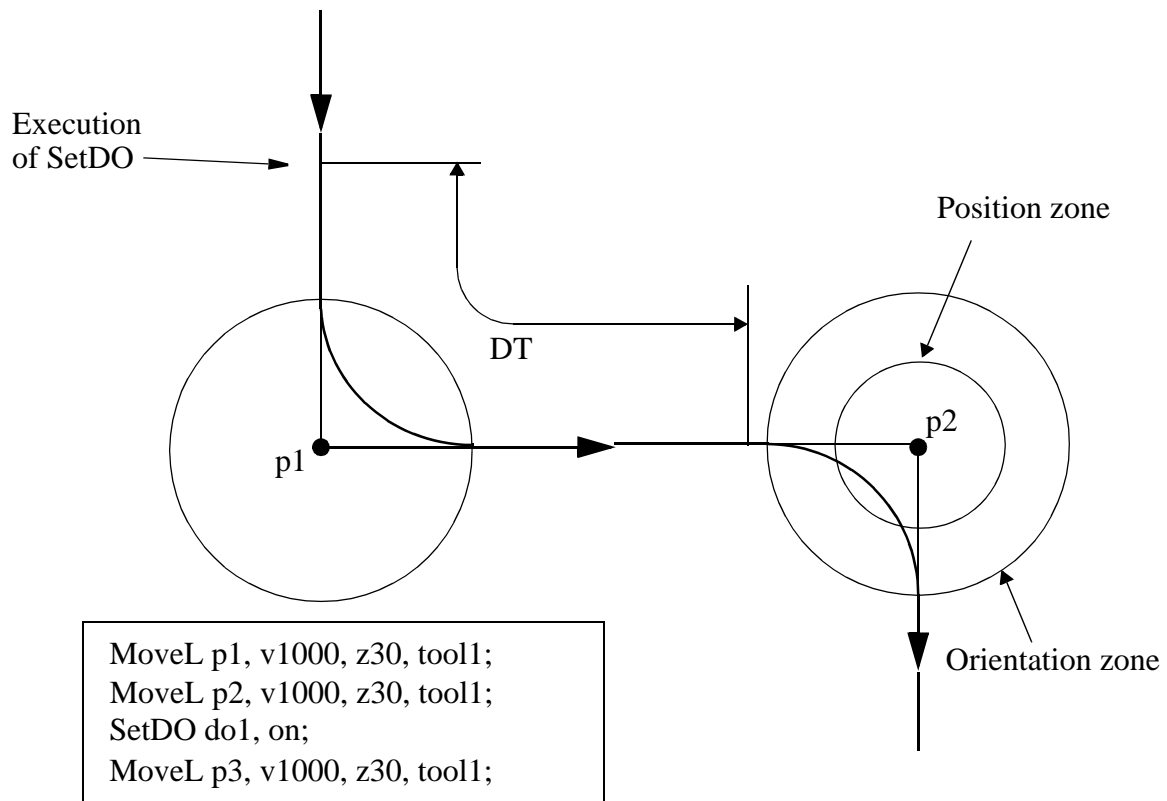


Figure 28 A logical instruction following a fly-by point is executed before reaching the largest zone.

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 29.
- 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 30.

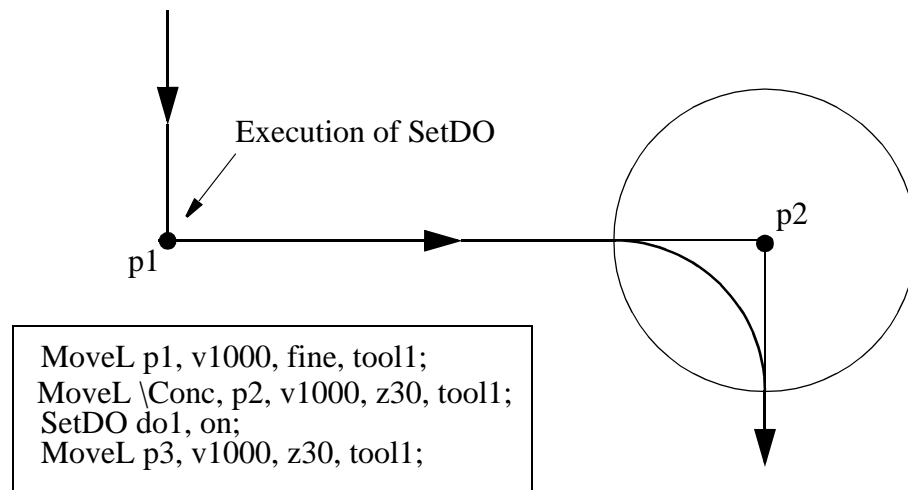


Figure 29 In the case of concurrent program execution after a stop point, a positioning instruction and subsequent logical instructions are started at the same time.

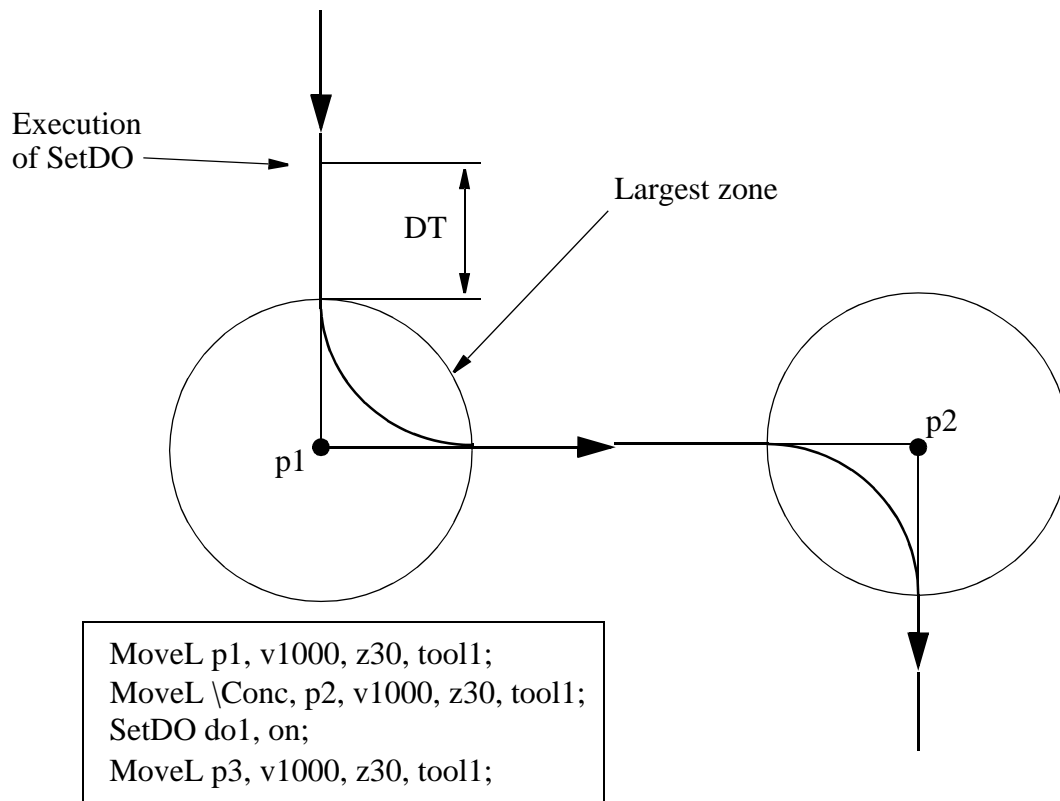
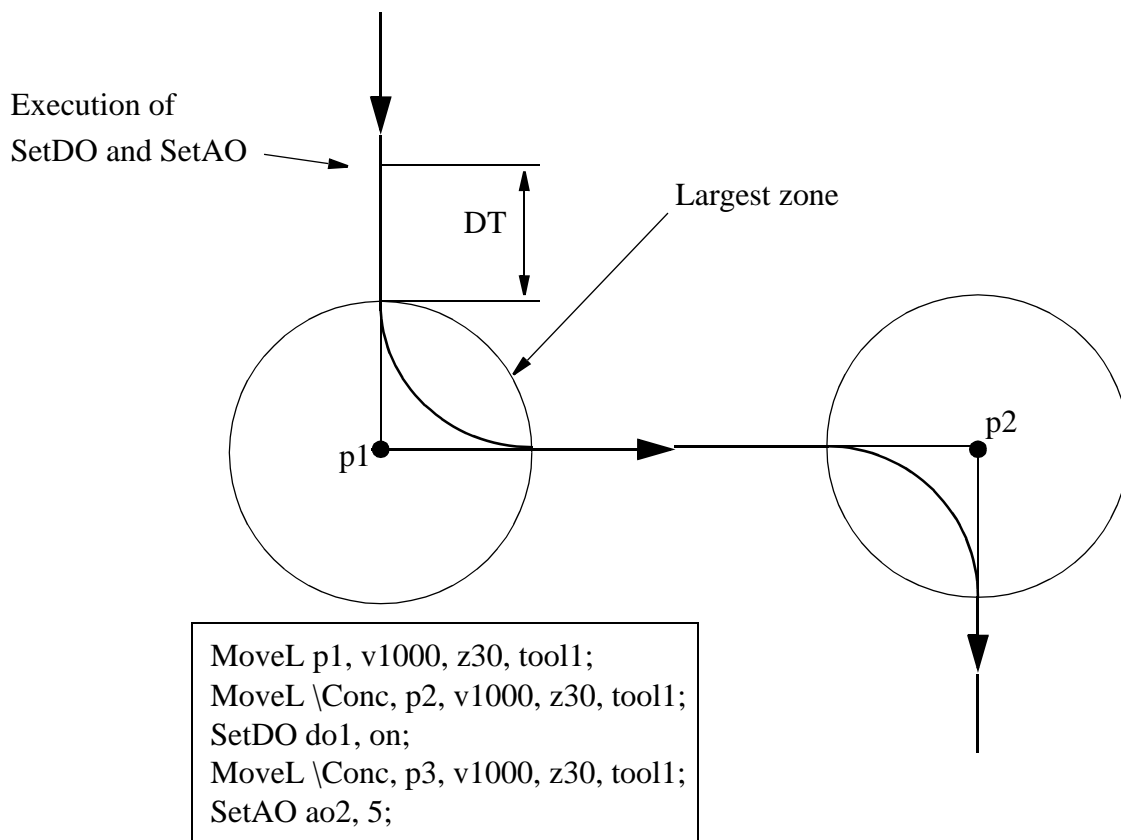


Figure 30 In the case of concurrent program execution after a fly-by point, the logical instructions start executing before the positioning instructions with the argument \Conc are started.

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:

- Logical instructions are executed directly, in the order they were programmed. This takes place at the same time as the movement (see Figure 31) which means that logical instructions are executed at an earlier stage on the path than they were programmed.



*Figure 31 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-synchronise positioning instructions and logical instructions:

- a positioning instruction to a stop point without the argument `\Conc`,
- the instruction `WaitTime` or `WaitUntil` with the argument `\Inpos`.

### 3.4 Path synchronisation

In order to synchronise process equipment (for applications such as gluing, painting and arc welding) with the robot movements, different types of path synchronisation 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 at 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 synchronised 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 defined by a programmed position and can be tuned as a path distance before the programmed position.

Typical repeat accuracy for a set of digital outputs on the path is +/- 2ms.

In the event of a power failure and restart in a Trigg instruction, all trigg events will be generated once again on the remaining movement path for the trigg instruction.

---

**3.5 Related information**

	<u>Described in:</u>
Positioning instructions	RAPID Summary - <i>Motion</i>
Definition of zone size	Data Types - <i>zonedata</i>

---

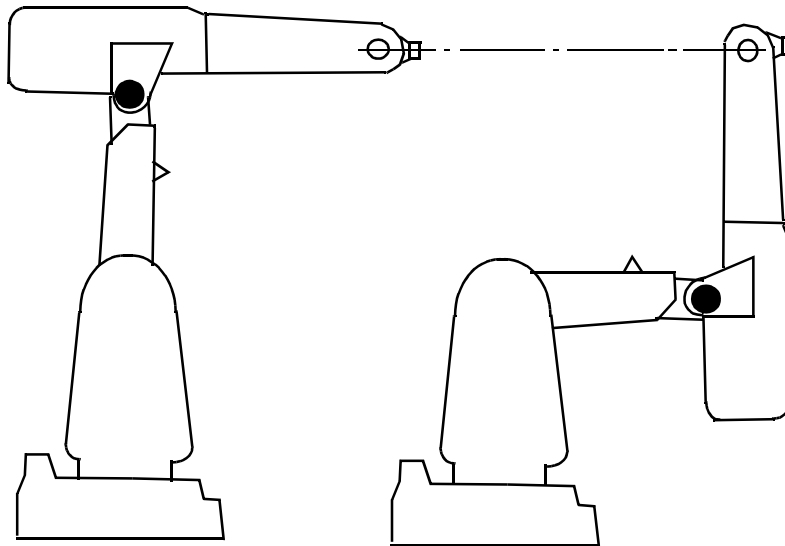
---

---

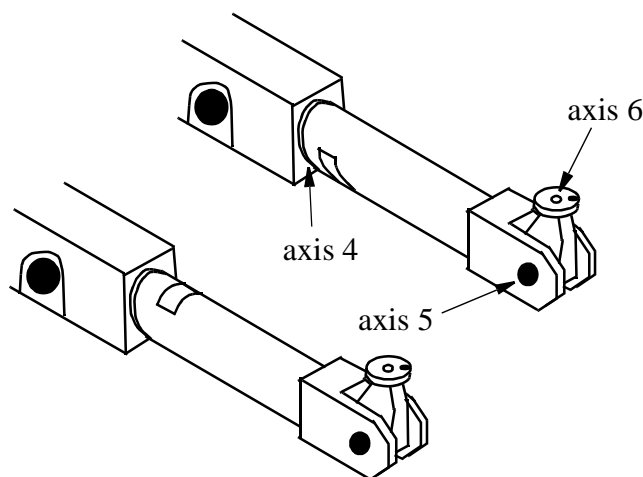
---

## 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.

For a rotational robot axis the robot configuration is specified by defining the appropriate quarter revolutions of the axis.

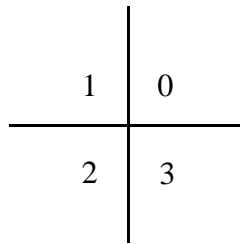


Figure 33 Quarter revolution for a positive joint angle:  $\text{int}(\text{jointangle} \times 2/\pi)$ .

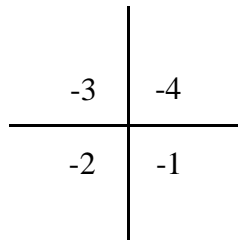


Figure 34 Quarter revolution for a negative joint angle:  $\text{int}(\text{jointangle} \times 2/\pi - 1)$ .

For a linear robot axis the value defines a meter interval for the robot axis. Value 0 means a position between 0 and 1 meters, 1 means a position between 1 and 2 meters. For negative values, -1 means a position between -1 and 0 meters, etc.

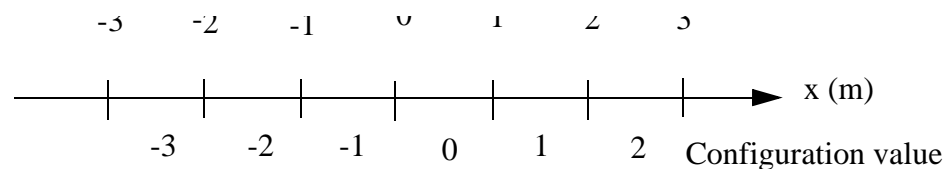


Figure 35 Configuration values for a linear axis

The configuration data set consists of four components cf1, cf4, cf6, cfx. cf1 specifies the value for axis 1, cf4, for axis 4, cf6 for axis 6. cfx is an additional component that is used for robot types with structures that need an extra component.

The configuration check involves comparing the configuration of the programmed position with that of the robot. The configuration of a specific axis is accepted if the attained axis value is within +/- 45 degrees from the specified quadrant for a rotational



joint or within +/- 0.5 m from the specified meter value of a linear axis.

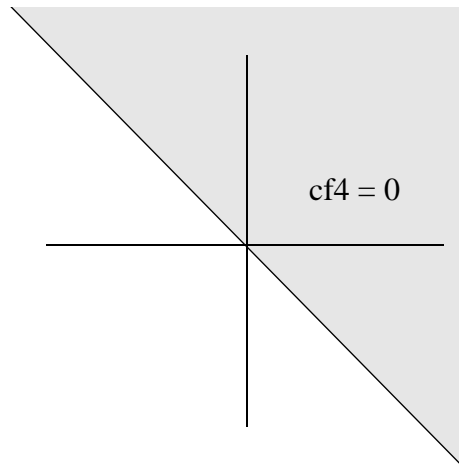


Figure 36 Example: Accepted axis value  $135 \text{ deg} < \text{joint } 4 < -45 \text{ deg}$  for rotational axis 4 when  $cf4 = 0$

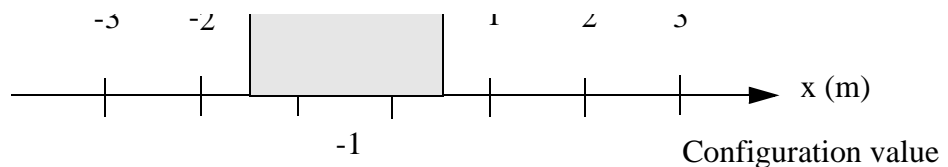


Figure 37 Example: Accepted axis value  $-1.5 \text{ m} < \text{joint } 1 < 0.5 \text{ m}$  for linear axis 1 when  $cf1 = -1$

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:

- The configuration of the programmed position will not be attained.
- The reorientation needed by any one of the wrist axes to get to the programmed position exceeds a limit (140-180 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 axes configurations will not be reached, 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 Related information**

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

## 5 Robot kinematic models

### 5.1 Robot kinematics

The position and orientation of a robot is determined from the kinematic model of its mechanical structure. The specific mechanical unit models must be defined for each installation. For standard ABB master and external robots, these models are predefined in the controller.

#### 5.1.1 Master robot

The kinematic model of the master robot models the position and orientation of the tool of the robot relative to its base as function of the robot joint angles.

The kinematic parameters specifying the arm-lengths, offsets and joint attitudes, are predefined in the configuration file for each robot type.

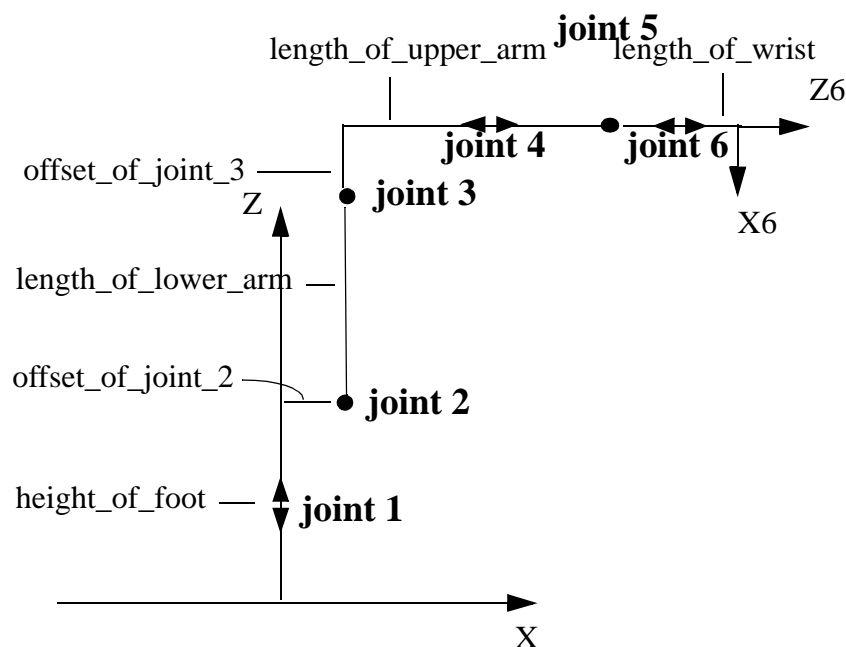


Figure 37 Kinematic structure of an IRB1400 robot

A calibration procedure supports the definition of the base frame of the master robot relative to the world frame.

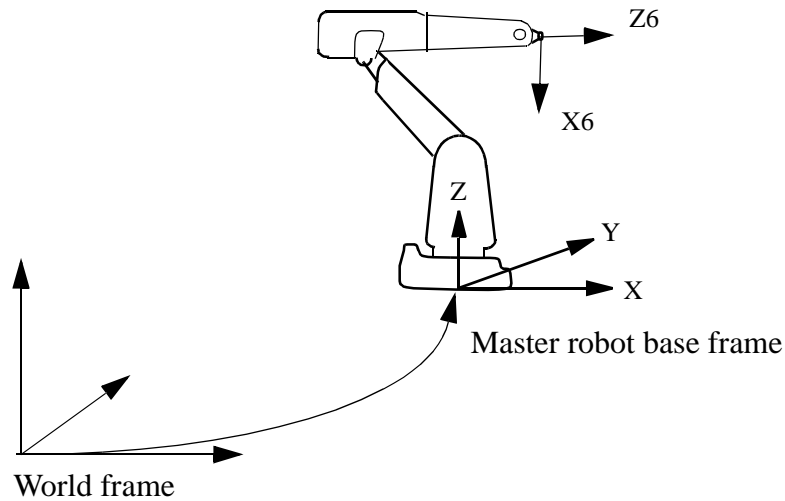


Figure 38 Base frame of master robot

### 5.1.2 External robot

Coordination with an external robot also requires a kinematic model for the external robot. A number of predefined classes of 2 and 3 dimensional mechanical structures are supported.

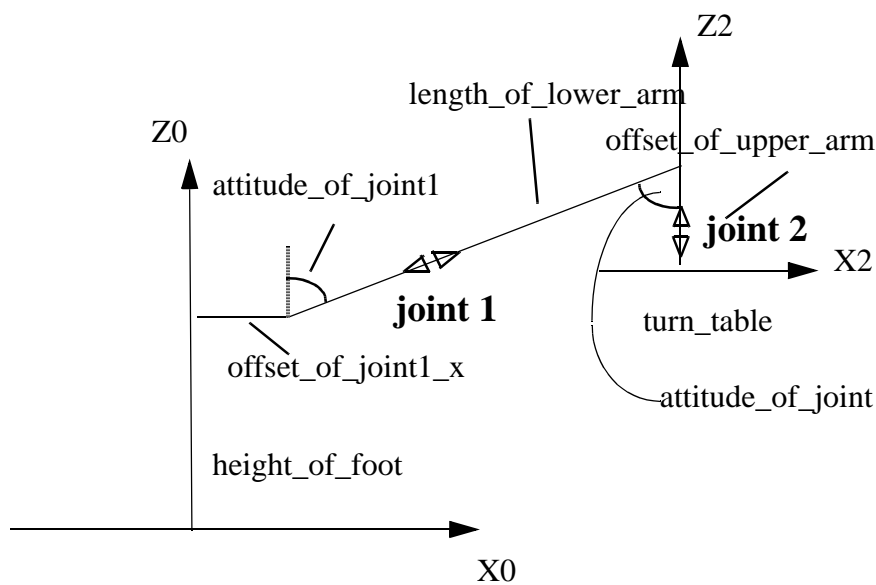


Figure 39 Kinematic structure of an ORBIT 160B robot using predefined model

Calibration procedures to define the base frame relative to the world frame are supplied for each class of structures.

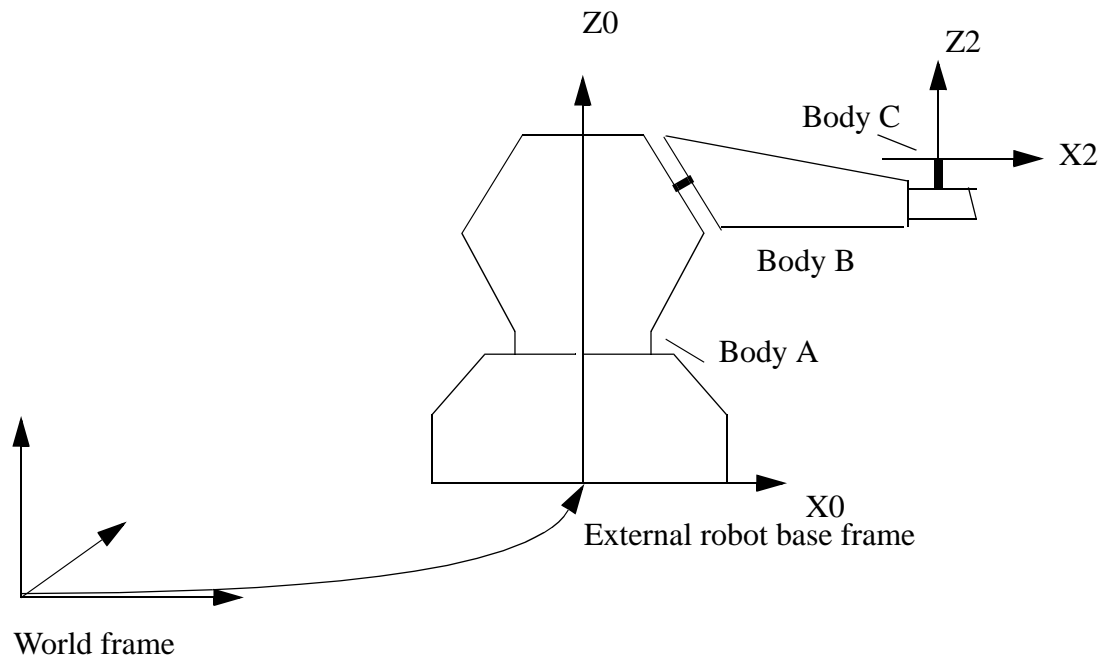


Figure 40 Base frame of an ORBIT\_160B robot

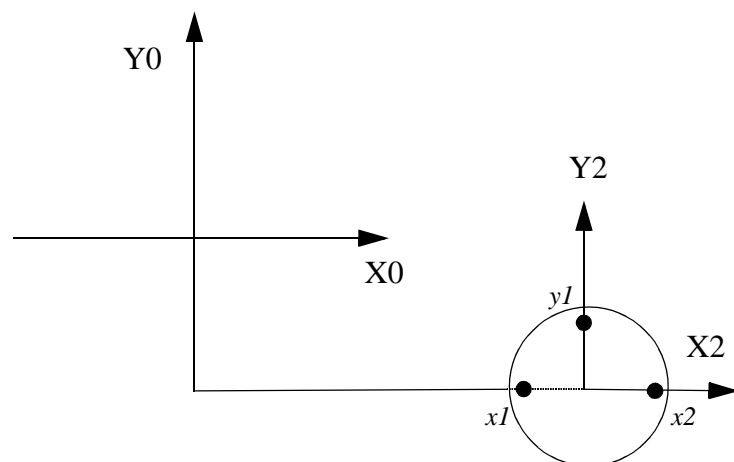


Figure 41 Reference points on turntable for base frame calibration of an ORBIT\_160B robot in the home position using predefined model

## 5.2 General kinematics

Mechanical structures not supported by the predefined structures may be modelled by using a general kinematic model. This is possible for external robots.

Modelling is based on the Denavit-Hartenberg convention according to Introduction to Robotics, Mechanics & Control, John J. Craig (Addison-Wesley 1986)

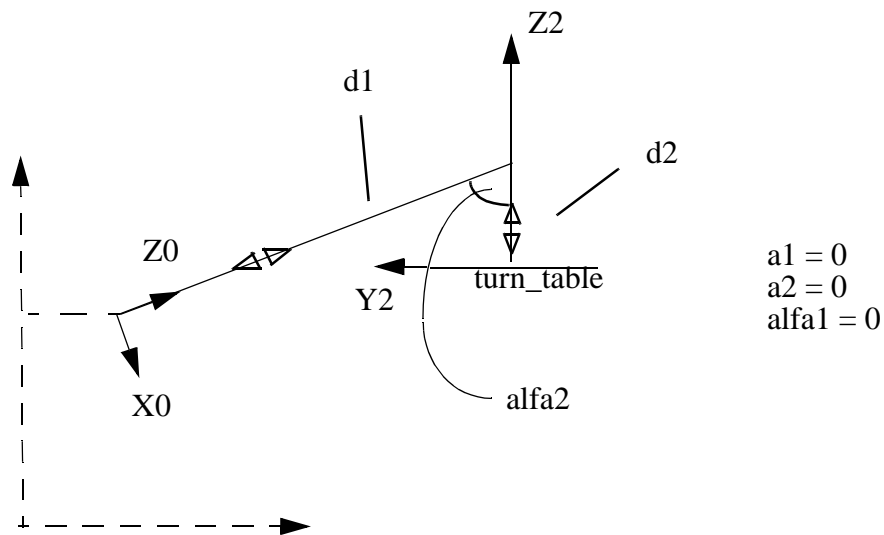


Figure 42 Kinematic structure of an ORBIT 160B robot using general kinematics model

A calibration procedure supports the definition of the base frame of the external robot relative to the world frame.

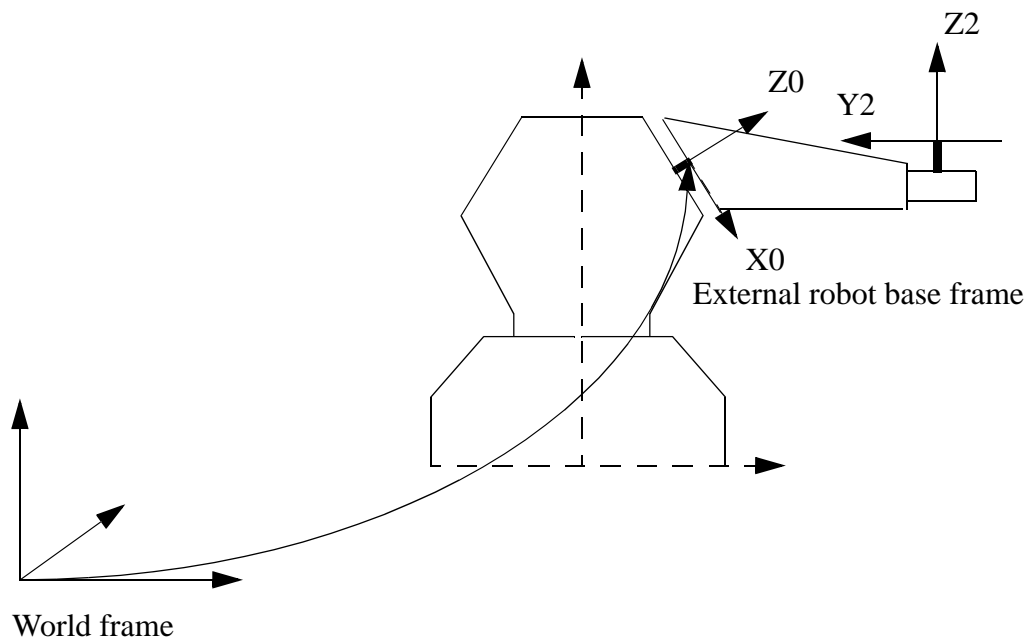


Figure 43 Base frame of an ORBIT\_160B robot using general kinematics model

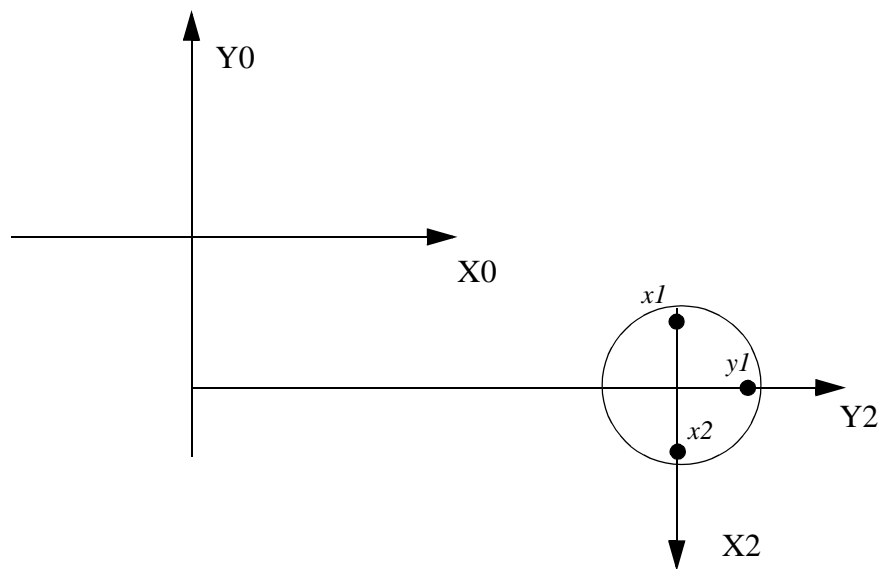


Figure 44 Reference points on turntable for base frame calibration of an ORBIT\_160B robot in the home position (joints = 0 degrees)

---

### 5.3 Related information

Definition of general kinematics of an external robot

#### Described in:

User's Guide - System Parameters





---

---

## 6 Motion Supervision/Collision Detection

Motion supervision is the name of a collection of functions for high sensitivity, model-based supervision of the robot's movements. Motion supervision includes functionality for the detection of collision, jamming, and incorrect load definition. This functionality is called Collision Detection.

---

### 6.1 Introduction

The collision detection will trigger if the load is incorrectly defined. If the load data is not known, the load identification functionality can be used to define it.

When the collision detection is triggered, the motor torques are reversed and the mechanical brakes applied in order to stop the robot. The robot then backs up a short distance along the path in order to remove any residual forces which may be present if a collision or jam occurred. After this, the robot stops again and remains in the motors on state. A typical collision is illustrated in the figure below.

The motion supervision is only active when at least one axis (including external axes) is in motion. When all axes are standing still, the function is deactivated. This is to avoid unnecessary triggering due to external process forces.

---

### 6.2 Tuning of Collision Detection levels

The collision detection uses a variable supervision level. At low speeds it is more sensitive than during high speeds. For this reason, no tuning of the function should be required by the user during normal operating conditions. However, it is possible to turn the function on and off and to tune the supervision levels. Separate tuning parameters are available for jogging and program execution. The different tuning parameters are described in more detail in the User's Guide under System Parameters: Manipulator.

There is a RAPID instruction called MotionSup which turns the function on and off and modifies the supervision level. This is useful in applications where external process forces act on the robot in certain parts of the cycle. The MotionSup instruction is described in more detail in the LoadId & CollDetect Manual.

---

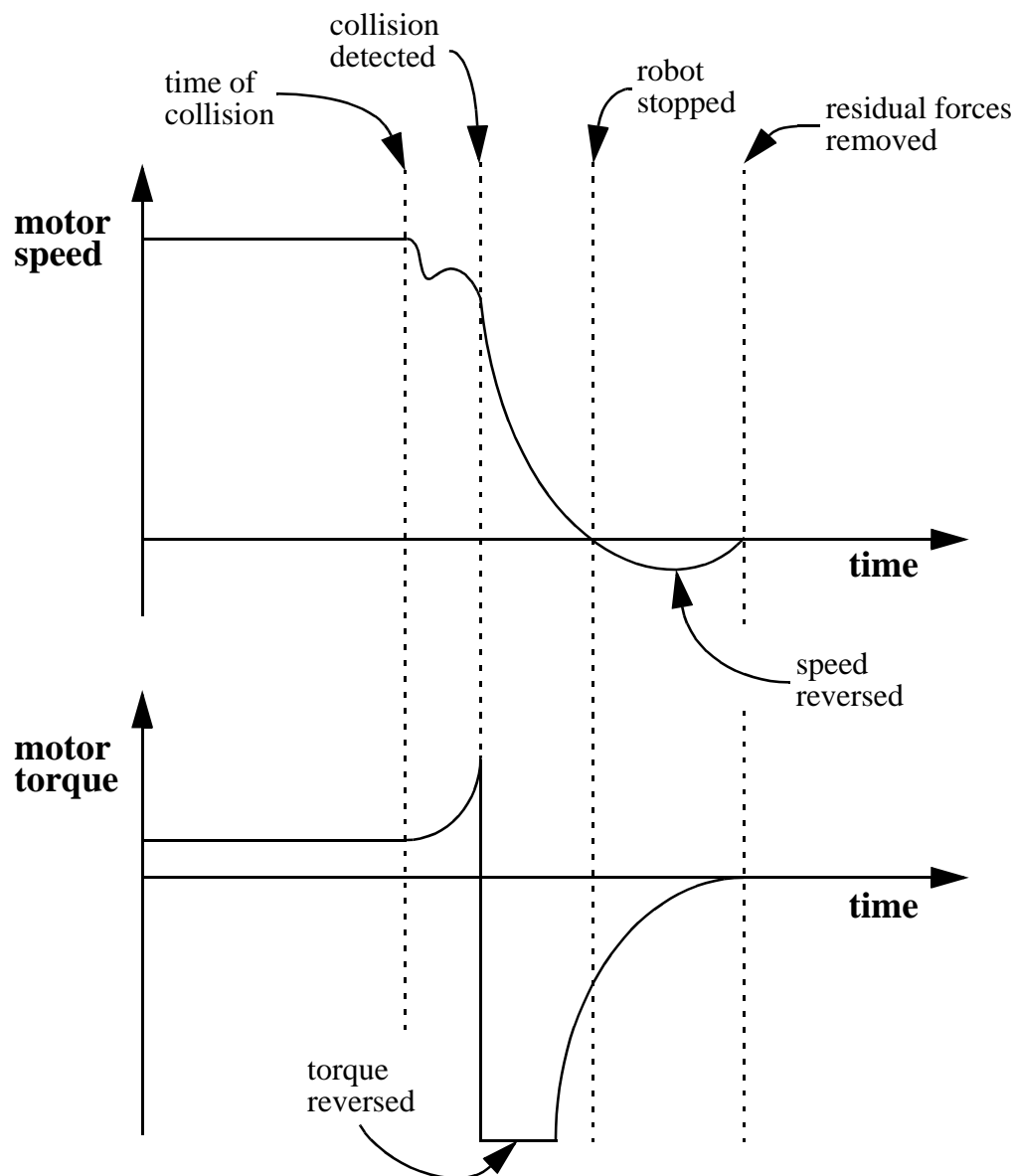
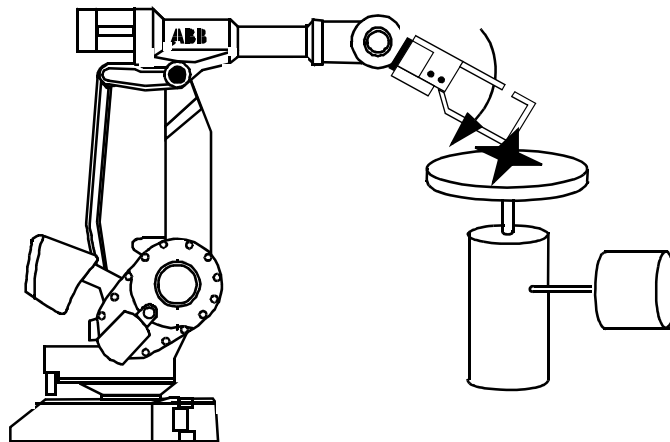
### 6.3 Motion supervision dialogue box

Select motion supervision under the special menu in the jogging window. This displays a dialogue box which allows the motion supervision to be turned on and off. **This will only affect the robot during jogging.** If the motion supervision is turned off in the dialogue box and a program is executed, the collision detection can still be active during the running of the program. If the program is then stopped and the robot jogged, the status flag in the dialogue window is set to on again. This is a safety measure to avoid turning the function off by accident.

*Figure: Typical collision*

*Phase 1 - The motor torque is reversed to stop the robot.*

*Phase 2 - The motor speed is reversed to remove residual forces on the tool and robot.*



---

## 6.4 Digital outputs

The digital output MotSupOn is high when the collision detection function is active and low when it is not active. Note that a change in the state of the function takes effect when a motion starts. Thus, if the collision detection is active and the robot is moving, MotSupOn is high. If the robot is stopped and the function turned off, MotSupOn is still high. When the robot starts to move, MotSupOn switches to low.

The digital output MotSupTrigg goes high when the collision detection triggers. It stays high until the error code is acknowledged, either from the teach pendant or through the digital input AckErrDialog.

The digital outputs are described in more detail in the User's Guide under System Parameters: IO Signals.

---

## 6.5 Limitations

The motion supervision is only available for the robot axes. It is not available for track motions, orbit stations, or any other external manipulators.

**In RobotWare 3.1, the motion supervision is only available for the IRB6400 robot family.**

The collision detection is deactivated when at least one axis is run in independent joint mode. This is also the case even when it is an external axis which is run as an independent joint.

The collision detection may trigger when the robot is used in soft servo mode. Therefore, it is advisable to turn the collision detection off when the robot is in soft servo mode.

If the RAPID instruction MotionSup is used to turn off the collision detection, this will only take effect once the robot starts to move. As a result, the digital output MotSupOn may temporarily be high at program start before the robot starts to move.

The distance the robot backs up after a collision is proportional to the speed of the motion before the collision. If repeated low speed collisions occur, the robot may not back up sufficiently to relieve the stress of the collision. As a result, it may not be possible to jog the robot without the supervision triggering. In this case use the jog menu to turn off the collision detection temporarily and jog the robot away from the obstacle.

In the event of a stiff collision during program execution, it may take a few seconds before the robot starts to back up.

---

## **6.6 Related information**

RAPID instruction MotionSup  
System parameters for tuning  
Motion supervision IO Signals  
Load Identification

Described in:

RAPID Summary - *Motion*  
System Parameters - *Manipulator*  
System Parameters - *IO Signals*  
Option Manual - *LoadId & CollDetect*

## 7 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 45).

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 46).

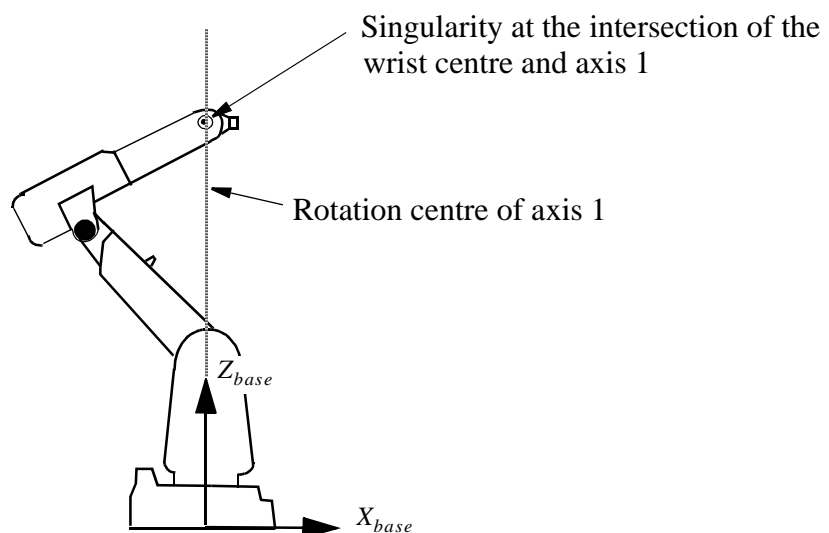


Figure 45 Arm singularity occurs where the wrist centre and axis 1 intersect.

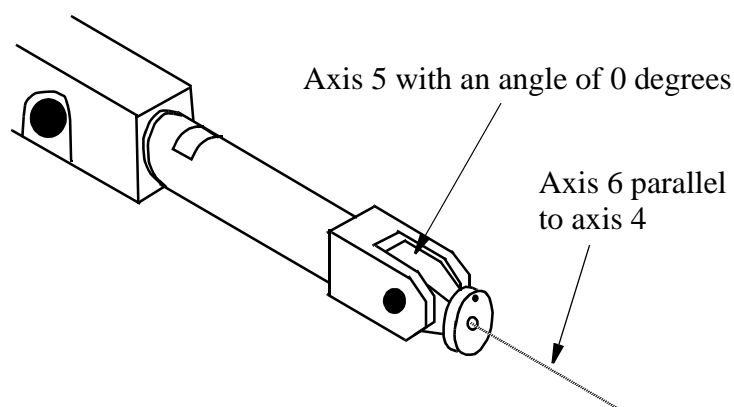


Figure 46 Wrist singularity occurs when axis 5 is 0 degrees.

---

## 7.1 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.

---

## 7.2 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.

---

## 7.3 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.

---

## 7.4 Related information

Controlling how the robot is to act on execution near singular points

Described in:

Instructions - *SingArea*

---

---

## 8 World Zones

---

### 8.1 Using global zones

When using this function, the robot stops or an output is automatically set if the robot is inside a special user-defined area. Here are some examples of applications:

- When two robots share a part of their respective work areas. The possibility of the two robots colliding can be safely eliminated by the supervision of these signals.
- When external equipment is located inside the robot's work area. A forbidden work area can be created to prevent the robot colliding with this equipment.
- Indication that the robot is at a position where it is permissible to start program execution from a PLC.

---

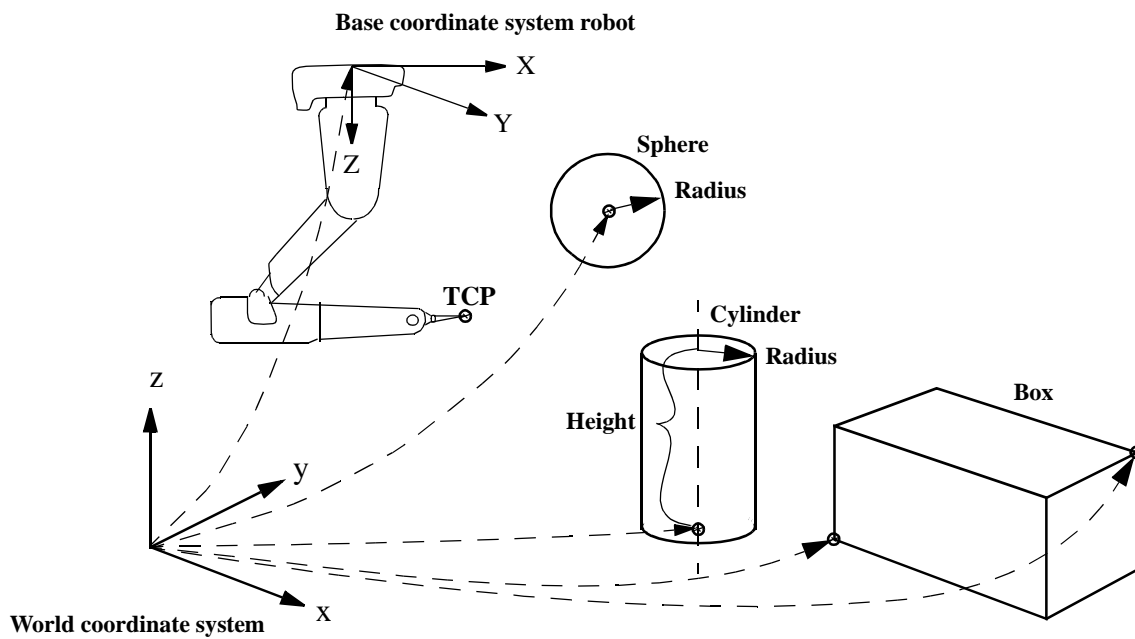
### 8.2 Using World Zones

To indicate that the tool centre point is in a specific part of the working area.  
To limit the working area of the robot in order to avoid collision with the tool.  
To make a common work area for two robots available to only one robot at a time.

---

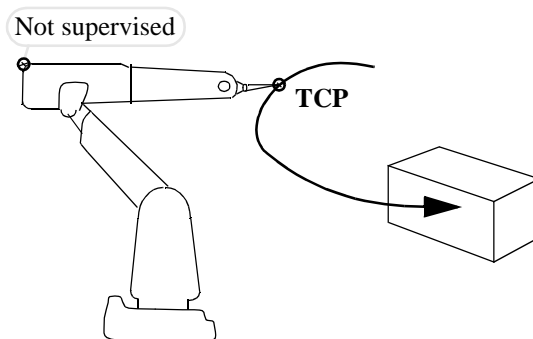
### 8.3 Definition of World Zones in the world coordinate system

All World Zones are to be defined in the world coordinate system.  
The sides of the Boxes are parallel to the coordinate axes and Cylinder axis is parallel to the Z axis of the world coordinate system.



A World Zone can be defined to be inside or outside the shape of the Box, Sphere or the Cylinder.

## 8.4 Supervision of the Robot TCP



The movement of the tool centre point is supervised and not any other points on the robot.

The TCP is always supervised irrespective of the mode of operation, for example, program execution and jogging.

### 8.4.1 Stationary TCPs

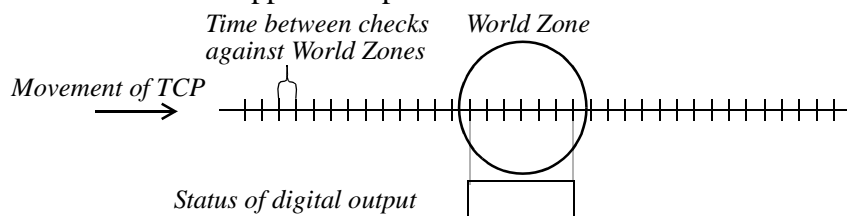
If the robot is holding a work object and working on a stationary tool, a stationary TCP is used. If that tool is active, the tool will not move and if it is inside a World Zone then it is always inside.



## 8.5 Actions

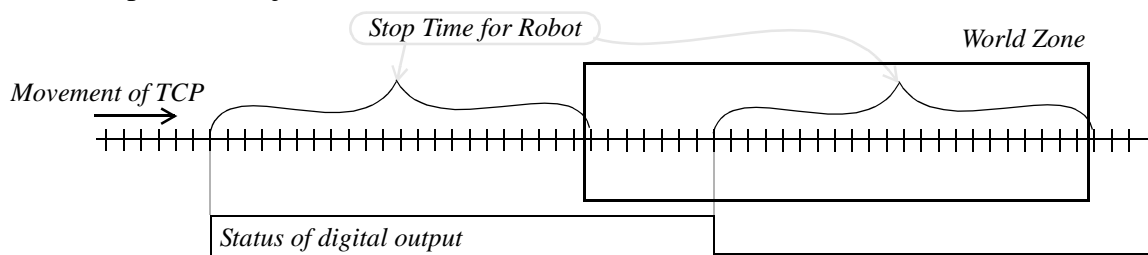
### 8.5.1 Set a digital output when the tcp is inside a World Zone.

This action sets a digital output when the tcp is inside a World Zone. It is useful to indicate that the robot has stopped in a specified area.



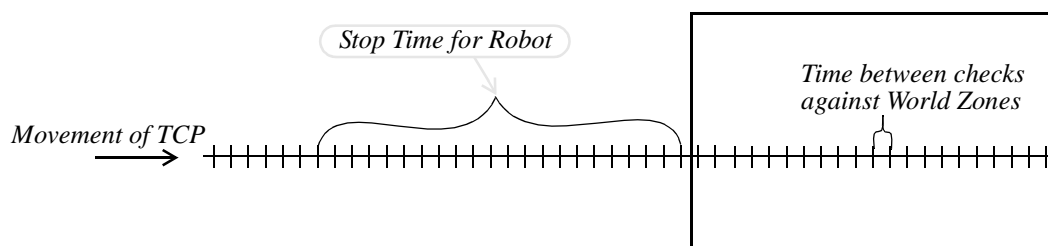
### 8.5.2 Set a digital output before the tcp reaches a World Zone.

This action sets a digital output before the tcp reaches a World Zone. It can be used to stop the robot just inside a World Zone



### 8.5.3 Stop the robot before the tcp reaches a World Zone.

A World Zone can be defined to be outside the work area. The robot will then stop with the Tool Centre Point just outside the World Zone, when heading towards the Zone



When the robot has been moved into a World Zone defined as an outside work area, for example, by releasing the brakes and manually pushing, then the only ways to get out of the Zone are by jogging or by manual pushing with the brakes released.

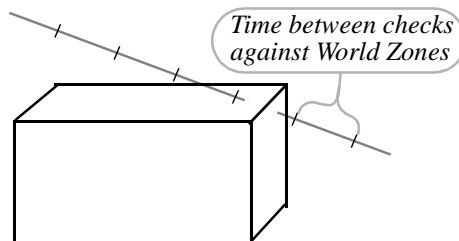
## 8.6 Minimum size of World Zones.

Supervision of the movement of the tool centre points is done at discrete points with a time interval between them that depends on the path resolution.

It is up to the user to make the zones large enough so the robot cannot move right through a zone without being checked inside the Zone.

Min. size of zone for used path_resolution and max. speed			
Speed Resol.	1000 mm/s	2000 mm/s	4000 mm/s
1	25 mm	50 mm	100 mm
2	50 mm	100 mm	200 mm
3	75 mm	150 mm	300 mm

If the same digital output is used for more than one World Zone, the distance between the Zones must exceed the minimum size, as shown in the table above, to avoid an incorrect status for the output.



It is possible that the robot can pass right through a corner of a zone without it being noticed, if the time that the robot is inside the zone is too short. Therefore, make the size of the zone larger than the dangerous area.

## 8.7 Maximum number of World Zones

A maximum of **ten** World Zones can be defined at the same time.

## 8.8 Power failure, restart, and run on

**Stationary World Zones** will be deleted at power off and must be reinserted at power on by an event routine connected to the event POWER ON.

**Temporary World Zones** will survive a power failure but will be erased when a new program is loaded or when a program is started from the main program.

The digital outputs for the World Zones will be updated first at **Motors on**.

---

## 8.9 Related information

RAPID Reference Manual

Motion and I/O Principles:

Data Types:

Coordinate Systems

wztemporary

wzstationary

shapedata

Instructions:

WZBoxDef

WZSphDef

WZCylDef

WZLimSup

WZDOSet

WZDisable

WZEnable

WZFree

WZTempFree



## 9 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 47).

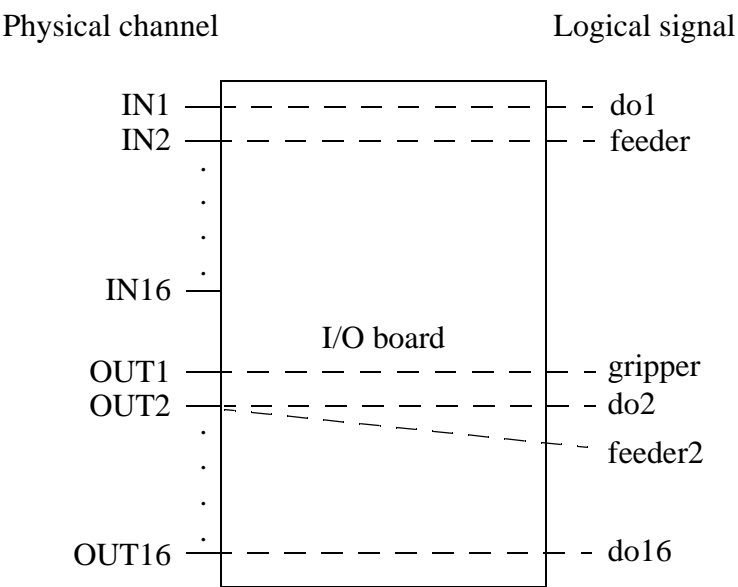


Figure 47 To be able to use an I/O board, its channels must be given logical names. In the above example, the physical output 2 is connected to two different logical names. IN16, on the other hand, has no logical name and thus cannot be used.

### 9.1 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 48).

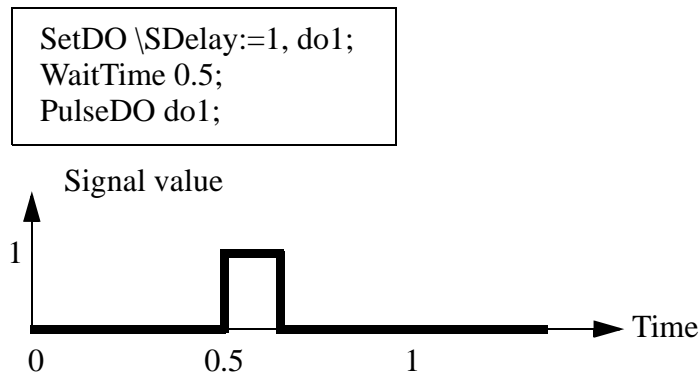


Figure 48 The instruction *SetDO* is not carried out at all because a new command is given before the time delay has elapsed.

---

## 9.2 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.

---

## 9.3 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.

---

## 9.4 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.

---

## 9.5 Related information

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





## 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 PGMCP1.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
```



---

# 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.


<u>Name</u>	<u>Data type</u>	<u>Declaration</u>
<b>reg1</b>	num	VAR num reg1:=0
<b>reg2</b>	.	.
<b>reg3</b>	.	.
<b>reg4</b>	.	.
<b>reg5</b>	num	VAR num reg5:=0
<b>wobj1</b>	wobjdata	PERS wobjdata wobj1:=wobj0
<b>clock1</b>	clock	VAR clock clock1
<b>high</b>	dionum	CONST dionum high:=1
<b>low</b>	dionum	CONST dionum low:=0
<b>edge</b>	dionum	CONST dionum edge:=2

*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*).

## INDEX

---

### A

- aggregate 4-71
- alias data type 4-71
- AND 4-82
- argument
  - conditional 4-84
- arithmetic expression 4-81
- array 4-75, 4-76
- assigning a value to data 3-5
- axis configuration 5-31

### B

- backward execution 4-1
- Backward Handler 6-3
- backward handler 4-65, 4-1, 4-3
- base coordinate system 5-1

### C

- calling a subroutine 3-3
- circular movement 5-13
- comment 3-5, 4-55
- communication 3-37
- communication instructions 3-19
- component of a record 4-71
- concurrent execution 5-26, 43
- conditional argument 4-84
- configuration check instructions 3-8
- CONST 4-76
- constant 4-74
- coordinate system 5-1, 5-35
- coordinated external axes 5-5
- corner path 5-14
- cross connections 5-54

### D

- data 4-74
  - used in expression 4-83
- data type 4-71
- declaration
  - constant 4-76
  - module 4-60
  - persistent 4-75
  - routine 4-65
  - variable 4-75
- displacement frame 5-4

- displacement instructions 3-9
- DIV 4-81

### E

- equal data type 4-71
- ERRNO 4-89
- error handler 4-89
- error number 4-89
- error recovery 4-89
- expression 4-81
- external axes
  - coordinated 5-5

### F

- file header 4-55
- file instructions 3-19
- function 4-63
- function call 4-84

### G

- global
  - data 4-74
  - routine 4-63
- GlueWare 3-35

### I

- I/O principles 5-53
- I/O synchronisation 5-25
- identifier 4-53
- input instructions 3-17
- interpolation 5-11
- interrupt 3-21, 4-91

### J

- joint movement 5-11

### L

- linear movement 5-12
- local
  - data 4-74
  - routine 4-63
- logical expression 4-82
- logical value 4-54

## **M**

- main routine 4-59
- mathematical instructions 3-27, 3-41
- MOD 4-81
- modified linear interpolation 5-14
- module 4-59
  - declaration 4-60
- motion instructions 3-12
- motion settings instructions 3-7
- multitasking 4-3
- Multitasking 3-43

## **N**

- non value data type 4-71
- NOT 4-82
- numeric value 4-54

## **O**

- object coordinate system 5-3
- offline programming 6-1
- operator
  - priority 4-85
- optional parameter 4-64
- OR 4-82
- output instructions 3-17

## **P**

- parameter 4-64
- path synchronization 5-29
- PERS 4-76
- persistent 4-74
- placeholder 4-55
- position
  - instruction 3-12
- position fix I/O 5-29
- procedure 4-63
- program 4-59
- program data 4-74
- program displacement 3-9
- program flow instructions 3-3
- program module 4-59
- programming 6-1

## **R**

- record 4-71
- reserved words 4-53

- robot configuration 5-31
- routine 4-63
  - declaration 4-65
- routine data 4-74

## **S**

- scope
  - data scope 4-74
  - routine scope 4-63
- searching instructions 3-12
- semi value data type 4-71
- singularity 5-45
- soft servo 3-9, 5-22
- spot welding 3-29
- stationary TCP 5-8
- stopping program execution 3-4
- string 4-54
- string expression 4-82
- switch 4-64
- syntax rules 2-2
- system module 4-60

## **T**

- TCP 5-1, 5-35
  - stationary 5-8
- time instructions 3-25
- tool centre point 5-1, 5-35
- tool coordinate system 5-7
- trap routine 4-63, 4-91
- typographic conventions 2-2

## **U**

- User - system module 5
- user coordinate system 5-3

## **V**

- VAR 4-75
- variable 4-74

## **W**

- wait instructions 3-5
- world coordinate system 5-2
- wrist coordinate system 5-7

## **X**

- XOR 4-82





---



---

## Glossary

<b>Argument</b>	The parts of an instruction that can be changed, i.e. everything except the name of the instruction.
<b>Automatic mode</b>	The applicable mode when the operating mode selector is set to  .
<b>Component</b>	One part of a record.
<b>Configuration</b>	The position of the robot axes at a particular location.
<b>Constant</b>	Data that can only be changed manually.
<b>Corner path</b>	The path generated when passing a fly-by point.
<b>Declaration</b>	The part of a routine or data that defines its properties.
<b>Dialog/Dialog box</b>	Any dialog boxes appearing on the display of the teach pendant must always be terminated (usually by pressing <b>OK</b> or <b>Cancel</b> ) before they can be exited.
<b>Error handler</b>	A separate part of a routine where an error can be taken care of. Normal execution can then be restarted automatically.
<b>Expression</b>	A sequence of data and associated operands; e.g. <i>reg1+5</i> or <i>reg1&gt;5</i> .
<b>Fly-by point</b>	A point which the robot only passes in the vicinity of – without stopping. The distance to that point depends on the size of the programmed zone.
<b>Function</b>	A routine that returns a value.
<b>Group of signals</b>	A number of digital signals that are grouped together and handled as one signal.
<b>Interrupt</b>	An event that temporarily interrupts program execution and executes a trap routine.
<b>I/O</b>	Electrical inputs and outputs.
<b>Main routine</b>	The routine that usually starts when the <b>Start</b> key is pressed.
<b>Manual mode</b>	The applicable mode when the operating mode switch is set to  .
<b>Mechanical unit</b>	A group of external axes.
<b>Module</b>	A group of routines and data, i.e. a part of the program.
<b>Motors On/Off</b>	The state of the robot, i.e. whether or not the power supply to the motors is switched on.
<b>Operator's panel</b>	The panel located on the front of the control system.
<b>Orientation</b>	The direction of an end effector, for example.
<b>Parameter</b>	The input data of a routine, sent with the routine call. It corresponds to the argument of an instruction.
<b>Persistent</b>	A variable, the value of which is persistent.
<b>Procedure</b>	A routine which, when called, can independently form an instruction.

## *Glossary*

<b>Program</b>	The set of instructions and data which define the task of the robot. Programs do not, however, contain system modules.
<b>Program data</b>	Data that can be accessed in a complete module or in the complete program.
<b>Program module</b>	A module included in the robot's program and which is transferred when copying the program to a diskette, for example.
<b>Record</b>	A compound data type.
<b>Routine</b>	A subprogram.
<b>Routine data</b>	Local data that can only be used in a routine.
<b>Start point</b>	The instruction that will be executed first when starting program execution.
<b>Stop point</b>	A point at which the robot stops before it continues on to the next point.
<b>System module</b>	A module that is always present in the program memory. When a new program is read, the system modules remain in the program memory.
<b>System parameters</b>	The settings which define the robot equipment and properties; configuration data in other words.
<b>Tool Centre Point (TCP)</b>	The point, generally at the tip of a tool, that moves along the programmed path at the programmed velocity.
<b>Trap routine</b>	The routine that defines what is to be done when a specific interrupt occurs.
<b>Variable</b>	Data that can be changed from within a program, but which loses its value (returns to its initial value) when a program is started from the beginning.
<b>Window</b>	The robot is programmed and operated by means of a number of different windows, such as the Program window and the Service window. A window can always be exited by choosing another window.
<b>Zone</b>	The spherical space that surrounds a fly-by point. As soon as the robot enters this zone, it starts to move to the next position.