

Technical reference manual

RAPID overview

Controller software IRC5
RobotWare 5.13



Technical reference manual

RAPID overview

RobotWare 5.13

3HAC16580-1

Revision J

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

Except as may be expressly stated anywhere in this manual, nothing herein shall be construed as any kind of guarantee or warranty by ABB for losses, damages to persons or property, fitness for a specific purpose or the like.

In no event shall ABB be liable for incidental or consequential damages arising from use of this manual and products described herein.

This manual and parts thereof must not be reproduced or copied without ABB'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 manual may be obtained from ABB at its then current charge.

© Copyright 2004-2010 ABB All rights reserved.

ABB AB
Robotics Products
SE-721 68 Västerås
Sweden

1 Introduction	11
1.1 Other manuals	11
1.2 How to read this manual	11
2 Basic RAPID programming	15
2.1 Program structure	15
2.1.1 Basic elements.....	17
2.1.2 Modules.....	23
2.1.3 System module User	27
2.1.4 Routines	29
2.2 Program data.....	37
2.2.1 Data types.....	37
2.2.2 Data declarations.....	39
2.3 Expressions	45
2.3.1 Arithmetic expressions.....	45
2.3.2 Logical expressions.....	46
2.3.3 String expressions	47
2.3.4 Using data in expressions.....	47
2.3.5 Using aggregates in expressions	48
2.3.6 Using function calls in expressions.....	48
2.3.7 Priority between operators	49
2.3.8 Example	50
2.3.9 Syntax.....	50
2.4 Instructions	53
2.4.1 Syntax.....	53
2.5 Controlling the program flow	55
2.5.1 Programming principles.....	55
2.5.2 Calling another routine.....	55
2.5.3 Program control within the routine	55
2.5.4 Stopping program execution	56
2.5.5 Stop current cycle.....	56
2.6 Various instructions	57
2.6.1 Assigning a value to data	57
2.6.2 Wait	57
2.6.3 Comments	57
2.6.4 Loading program modules	58
2.6.5 Various functions.....	58
2.6.6 Basic data	59
2.6.7 Conversion function.....	59

2.7	Motion settings	61
2.7.1	Programming principles.....	61
2.7.2	Maximum TCP speed	61
2.7.3	Defining velocity	62
2.7.4	Defining acceleration	62
2.7.5	Defining configuration management	62
2.7.6	Defining the payload.....	62
2.7.7	Defining the behavior near singular points	63
2.7.8	Displacing a program.....	63
2.7.9	Soft servo	63
2.7.10	Adjust the robot tuning values	64
2.7.11	World zones.....	64
2.7.12	Various for motion settings	65
2.8	Motion	67
2.8.1	Programming principles.....	67
2.8.2	Positioning instructions.....	68
2.8.3	Searching.....	68
2.8.4	Activating outputs or interrupts at specific positions	68
2.8.5	Control of analog output signal proportional to actual TCP	69
2.8.6	Motion control if an error/interrupt takes place	69
2.8.7	Get robot info in a MultiMove system.....	70
2.8.8	Controlling additional axes	70
2.8.9	Independent axes.....	71
2.8.10	Path correction	71
2.8.11	Path Recorder	72
2.8.12	Conveyor tracking.....	72
2.8.13	Servo Tracking for Indexing Conveyor	72
2.8.14	Sensor synchronization	73
2.8.15	Load identification and collision detection.....	73
2.8.16	Position functions.....	73
2.8.17	Check interrupted path after power failure	74
2.8.18	Status functions.....	74
2.8.19	Motion data	75
2.8.20	Basic data for movements.....	75
2.9	Input and output signals	77
2.9.1	Programming principles.....	77
2.9.2	Changing the value of a signal.....	77
2.9.3	Reading the value of an input signal.....	77

2.9.4 Reading the value of an output signal	78
2.9.5 Testing input or output signals	78
2.9.6 Disabling and enabling I/O modules	78
2.9.7 Defining input and output signals	79
2.9.8 Get status of I/O bus and unit	79
2.9.9 Start of I/O bus	79
2.10 Communication	81
2.10.1 Programming principles	81
2.10.2 Communicating using the FlexPendant, function group TP	81
2.10.3 Communicating using the FlexPendant, function group UI	82
2.10.4 Reading from or writing to a character-based serial channel/file	82
2.10.5 Communicating using binary serial channels/files/field buses	83
2.10.6 Communication using rawbytes	83
2.10.7 Data for serial channels/files/fieldbuses	84
2.10.8 Communicating using sockets	84
2.10.9 Communication using RAPID Message Queues	84
2.11 Interrupts	87
2.11.1 Programming principles	87
2.11.2 Connecting interrupts to trap routines	88
2.11.3 Ordering interrupts	88
2.11.4 Cancelling interrupts	88
2.11.5 Enabling/disabling interrupts	89
2.11.6 Interrupt data	89
2.11.7 Data type of interrupts	89
2.11.8 Safe Interrupt	91
2.11.9 Interrupt manipulation	91
2.11.10 Trap routines	91
2.12 Error recovery	93
2.12.1 Programming principles	93
2.12.2 Creating an error situation from within the program	93
2.12.3 Booking an error number	93
2.12.4 Restarting/returning from the error handler	94
2.12.5 User defined errors and warnings	94
2.12.6 IGenerate process error	94
2.12.7 Data for error handling	95
2.12.8 Configuration for error handling	95
2.12.9 Error handlers	97
2.12.10 System error handler	97

2.12.11 Errors raised by the program.....	98
2.12.12 The event log.....	98
2.12.13 UNDO.....	99
2.13 System & time.....	103
2.13.1 Programming principles.....	103
2.13.2 Using a clock to time an event.....	103
2.13.3 Reading current time and date.....	103
2.13.4 Retrieve time information from file.....	104
2.13.5 Get the size of free program memory.....	104
2.14 Mathematics.....	105
2.14.1 Programming principles.....	105
2.14.2 Simple calculations on numeric data.....	105
2.14.3 More advanced calculations.....	105
2.14.4 Arithmetic functions.....	105
2.14.5 String digit functions.....	106
2.14.6 Bit functions.....	106
2.15 External computer communication.....	107
2.15.1 Programming principles.....	107
2.15.2 Sending a program-controlled message from the robot to a computer.....	107
2.16 File operation functions.....	109
2.17 RAPID support instructions.....	111
2.17.1 Get system data.....	111
2.17.2 Get information about the system.....	111
2.17.3 Get information about memory.....	112
2.17.4 Read configuration data.....	112
2.17.5 Write configuration data.....	112
2.17.6 Restart the controller.....	112
2.17.7 Text tables instructions.....	112
2.17.8 Get object name.....	113
2.17.9 Get information about the tasks.....	113
2.17.10 Get current event type, execution handler or execution level.....	113
2.17.11 Search for symbols.....	114
2.18 Calibration & service instructions.....	115
2.18.1 Calibration of the tool.....	115
2.18.2 Various calibration methods.....	115
2.18.3 Directing a value to the robot's test signal.....	115
2.18.4 Recording of an execution.....	116
2.19 String functions.....	117

2.19.1 Basic operations	117
2.19.2 Comparison and searching	117
2.19.3 Conversion	118
2.20 Multitasking.....	119
2.20.1 Basics	120
2.20.2 General instructions and functions.....	120
2.20.3 MultiMove system with coordinated robots	121
2.20.4 Synchronizing the tasks	123
2.20.5 Synchronizing using polling	123
2.20.6 Synchronizing using an interrupt	124
2.20.7 Intertask communication.....	125
2.20.8 Type of task.....	126
2.20.9 Priorities	126
2.20.10 TrustLevel	127
2.20.11 Something to think about	128
2.21 Backward execution	129
2.21.1 Backward handlers	129
2.21.2 Limitation of move instructions in the backward handler	130
2.21.3 Behavior of the backward execution.....	131
2.22 Syntax summary	135
2.22.1 Instructions.....	135
2.22.2 Functions	148
3 Motion and I/O programming	155
3.1 Coordinate systems.....	155
3.1.1 The robot's tool center point (TCP)	155
3.1.2 Coordinate systems used to determine the position of the TCP	155
3.1.3 Coordinate systems used to determine the direction of the tool	160
3.1.4 Related information.....	164
3.2 Positioning during program execution	165
3.2.1 General.....	165
3.2.2 Interpolation of the position and orientation of the tool	165
3.2.3 Interpolation of corner paths	169
3.2.4 Independent axes.....	175
3.2.5 Soft Servo.....	178
3.2.6 Stop and restart.....	178
3.2.7 Related information.....	179
3.3 Synchronization with logical instructions	181
3.3.1 Sequential program execution at stop points	181

3.3.2 Sequential program execution at fly-by points	181
3.3.3 Concurrent program execution	182
3.3.4 Path synchronization	185
3.3.5 Related information	186
3.4 Robot configuration.....	187
3.4.1 Different types of robot configurations.....	187
3.4.2 Specifying robot configuration	189
3.4.3 Configuration check.....	189
3.4.4 Related information	191
3.5 Robot kinematic models.....	193
3.5.1 Robot kinematics	193
3.5.2 General kinematics.....	195
3.5.3 Related information	197
3.6 Motion supervision/collision detection	199
3.6.1 Introduction.....	199
3.6.2 Tuning of collision detection levels	199
3.6.3 Motion supervision dialog box	201
3.6.4 Digital outputs.....	201
3.6.5 Limitations	201
3.6.6 Related information	202
3.7 Singularities.....	203
3.7.1 Singularity points of IRB140	204
3.7.2 Program execution through singularities	204
3.7.3 Jogging through singularities	205
3.7.4 Related information	205
3.8 Optimized acceleration limitation	207
3.9 World Zones	209
3.9.1 Using global zones	209
3.9.2 Using World Zones	209
3.9.3 Definition of World Zones in the world coordinate system.....	209
3.9.4 Supervision of the robot TCP	210
3.9.5 Stationary TCPs	210
3.9.6 Actions	211
3.9.7 Minimum size of World Zones	212
3.9.8 Maximum number of World Zones.....	212
3.9.9 Power failure, restart, and run on.....	213
3.9.10 Related information	213
3.10 I/O principles.....	215

3.10.1 Signal characteristics.....	215
3.10.2 Signals connected to interrupt.....	216
3.10.3 System signals.....	216
3.10.4 Cross connections	217
3.10.5 Limitations	217
3.10.6 Related information.....	218
4 Glossary.....	219

1 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 *Operating manual - IRC5 with FlexPendant* until you are familiar with the system.

1.1 Other manuals

The *Operating manual - IRC5 with FlexPendant* 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.

The *Product Specification* contains an overview of the characteristics and performance of the robot.

1.2 How to read this manual

To answer the questions *Which instruction should I use?* or *What does this instruction mean?*, see *RAPID Overview Chapter 2: Basic RAPID programming*. 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. It also explains the inner details of the language.

RAPID Overview Chapter 3: Motion and I/O Programming describes the various coordinate systems of the robot, its velocity and other motion characteristics during different types of execution.

To make things easier to locate and understand, *RAPID Overview chapter 4* contains a *Glossary* and *Index*.

Typographic conventions

The commands located under any of the five menu keys at the top of the FlexPendant 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, for example ***Modpos***.

Words belonging to the actual programming language, such as instruction names, are written in italics, for example *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 FlexPendant in the following ways:

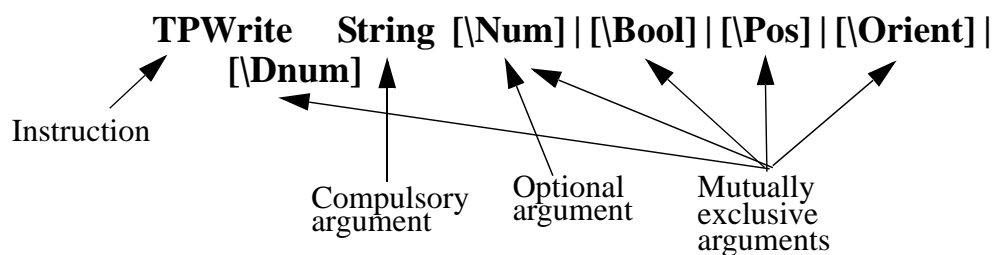
- Certain control words that are masked in the FlexPendant display are printed, for example words indicating the start and end of a routine.
- Data and routine declarations are printed in the formal form, for example *VAR num reg1;*

Syntax rules

Instructions and functions are described using both simplified syntax and formal syntax. If you use the FlexPendant 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, that is 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 curly brackets { }.

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*>]
 ['\Dnum':=' <expression (**IN**) of *dnum*>'];'

- The text within the square brackets [] may be omitted.
- Arguments that are mutually exclusive, that is 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 curly brackets { }.
- 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

How to read this manual

2 Basic RAPID programming

2.1 Program structure

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; for example *Reset do5*. These arguments can be specified in one of the following ways:

- as a numeric value, for example 5 or 4.6
- as a reference to data, for example *reg1*
- as an expression, for example $5 + \text{reg1} * 2$
- as a function call, for example *Abs(reg1)*
- as a string value, for example *"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; for example when an input is set, it is automatically executed if that particular interrupt occurs.

Information can also be stored in data, for example 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 *persistent*.

- 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
- Multitasking

The language is not case sensitive, for example upper case and lower case letters are considered the same.

2.1.1 Basic elements

2.1.1.1 Identifiers

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

for example *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 32 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.

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	UNDO
VAR	VIEWONLY	WHILE	WITH
XOR			

2.1.1.2 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.

2.1.1.3 Numeric values

A numeric value can be expressed as

- an integer, for example 3, -100, 3E2
- a decimal number, for example 3.5, -0.345, -245E-2

The value must be in the range specified by the ANSI IEEE 754 Standard for Floating-Point Arithmetic.

2.1.1.4 Logical values

A logical value can be expressed as TRUE or FALSE.

2.1.1.5 String values

A string value is a sequence of characters (ISO 8859-1 (Latin-1)) and control characters (non-ISO 8859-1 (Latin-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"

2.1.1.6 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 outside a module declaration;

```
for example      ! comment
                  IF reg1 > 5 THEN
                    ! comment
                    reg2 := 0;
                  ENDIF
```

2.1.1.7 Placeholders

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

Placeholder	Represents:
<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

2.1.1.8 File header

A program file starts with the following file header:

```
%%%
VERSION:1
LANGUAGE:ENGLISH          (or some other language:
                           GERMAN or FRENCH)
%%%
```

2.1.1.9 Syntax

Identifiers

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

Numeric values

```
<num literal> ::=  
    <integer> [ <exponent> ]  
    | <decimal integer> [ <exponent> ]  
    | <hex integer>  
    | <octal integer>  
    | <binary integer>  
    | <integer> '.' [ <integer> ] [ <exponent> ]  
    | [ <integer> ] '.' <integer> [ <exponent> ]  
  
<integer> ::= <digit> { <digit> }  
<hex integer> ::= '0' ('X' | 'x') <hex digit> { <hex digit> }  
<octal integer> ::= '0' ('O' | 'o') <octal digit> { <octal digit> }  
<binary integer> ::= '0' ('B' | 'b') <binary digit> { <binary digit> }  
  
<exponent> ::= ('E' | 'e') ['+' | '-' ] <integer>  
  
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d | e | f  
<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  
<binary digit> ::= 0 | 1
```

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 (Latin-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 | Å | Á | Â | Ã
    | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í
    | Î | Ï | 1) Ñ | Ò | Ó | Ô | Õ | Ö | Ø
    | Ù | Ú | Û | Ü | 2) | 3) | ß
<lower case letter> ::=
    a | b | c | d | e | f | g | h | i | j
    | k | l | m | n | o | p | q | r | s | t
    | u | v | w | x | y | z | ß | à | á | â
    | ã | ä | å | æ | ç | è | é | ê | ë | ì
    | í | î | ï | 1) ñ | ò | ó | ô | õ | ö
    | ø | ù | ú | û | ü | 2) | 3) | ÿ
```

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

2.1.2 Modules

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

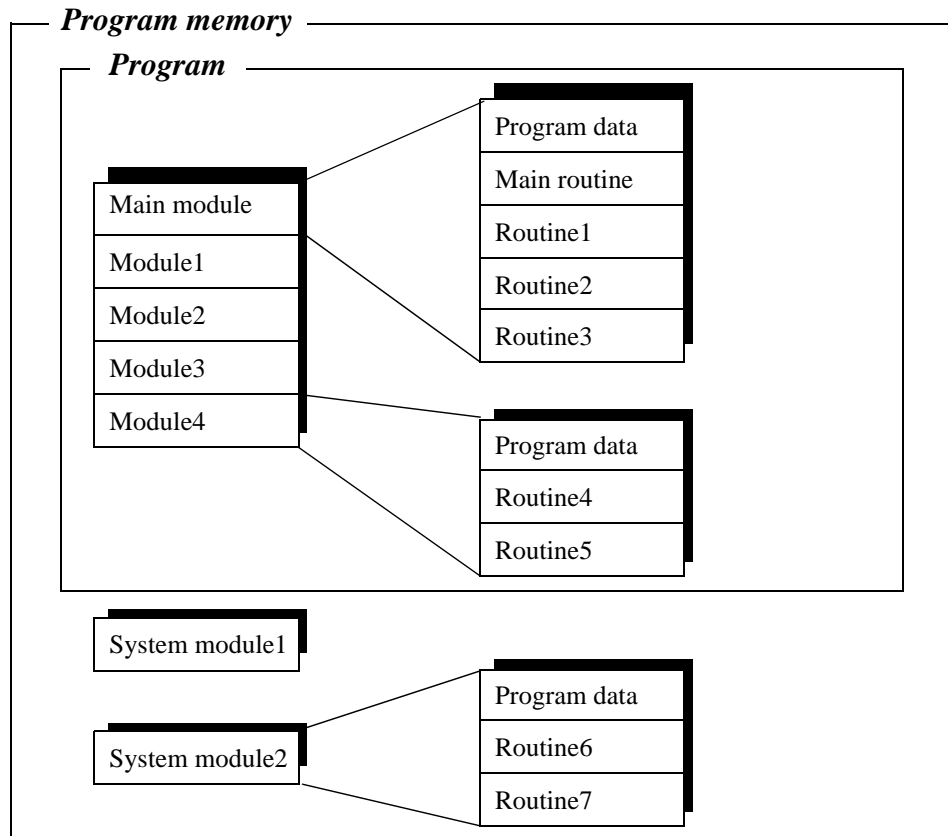


Figure 1 The program can be divided into modules.

2.1.2.1 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.

2.1.2.2 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.

2.1.2.3 Module declarations

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

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

for example `MODULE module_name (SYSMODULE, VIEWONLY)`
 `!data type definition`
 `!data declarations`
 `!routine declarations`
 `ENDMODULE`

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

2.1.2.4 Program file structure

As indicated above all program modules are contained in a program with a specific program name. When saving a program on the flash-disk or mass memory, then a new directory is created with the name of the program. In this directory all program modules will be saved with a file extension .mod together with a description file with the same name as the program and with the extension .pgf. The description file will include a list of all modules contained in the program.

2.1.2.5 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> }

```


2.1.3 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.

2.1.3.1 Contents

User comprises five numerical data (registers), one work object data, one clock and two symbolic values for digital signals.

Name	Data type	Declaration
reg1	num	VAR num reg1:=0
reg2	.	.
reg3	.	.
reg4	.	.
reg5	num	VAR num reg5:=0
clock1	clock	VAR clock clock1

User is a system module, which means that it is always present in the memory of the robot regardless of which program is loaded.

2.1.4 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.

2.1.4.1 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 in the task.
- 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.

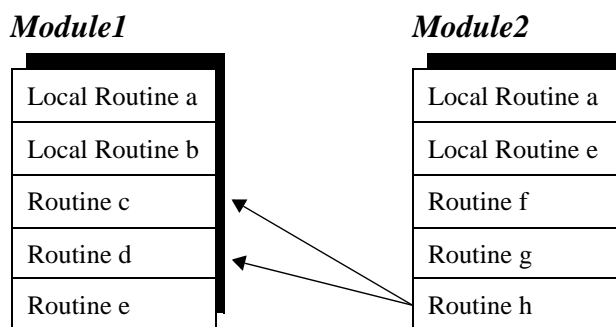


Figure 2 Example: The following routines can be called from Routine h:
 Module1 - Routine c, d.
 Module2 - All routines.

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

2.1.4.2 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, that is 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 (that is 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, that is 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{*,*})

2.1.4.3 Routine termination

The execution of a procedure is either explicitly terminated by a RETURN instruction or implicitly terminated when the end (ENDPROC, BACKWARD, ERROR or UNDO) 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, ERROR or UNDO) of that trap routine is reached. Execution continues from the point where the interrupt occurred.

2.1.4.4 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, that is it is not possible to declare a routine within a routine.

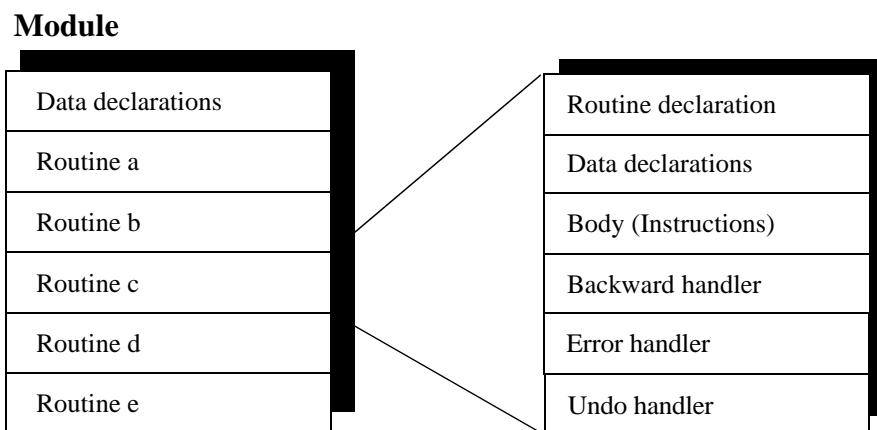


Figure 3 A routine can contain declarations, data, a body, a backward handler, an error handler and an undo handler.

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
```

2.1.4.5 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 2.3.6 *Using function calls in expressions* on page 48 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.

2.1.4.6 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> ]
    [ UNDO <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> ]
    [ UNDO <instruction list> ]
    ENDFUNC
<function name> ::= <identifier>

```

Trap routine declaration

```

<trap declaration> ::=
    TRAP <trap name>
    <data declaration list>
    <instruction list>
    [ ERROR <instruction list> ]
    [ UNDO <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> )

```

2.2 Program data

2.2.1 Data types

There are three 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, for example *num*.
- A *record* data type is a composite type with named, ordered components, for example *pos*. A component may be of an atomic or record type.

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

for example [300, 500, depth] *pos* record aggregate value.

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

for example *pos1.x* := 300; assignment of the x-component of *pos1*.

- An alias data type is by definition equal to another type. Alias types make it possible to classify data objects.

2.2.1.1 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:

- Initialization
- 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*, *signal di*, *signal gi*) are of the data type *semi value*. These data can be used in value-oriented operations, except initialization 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.

2.2.1.2 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
----------	--	--

2.2.1.3 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>
```

2.2.2 Data declarations

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 initialization value of the persistent declaration to be updated. (When a program is saved the initialization 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.

2.2.2.1 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.

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.

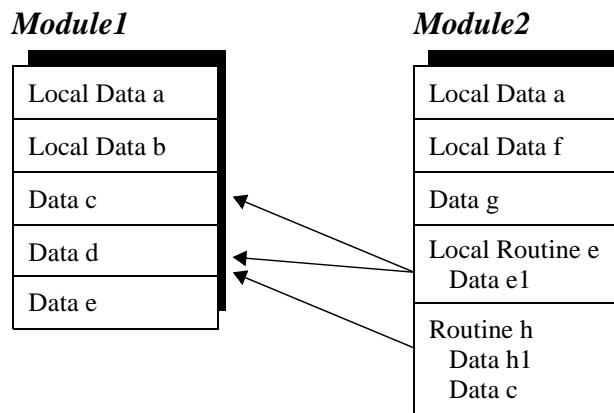


Figure 4 Example: The following data can be called from routine e:

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.

2.2.2.2 Variable declaration

A variable is introduced by a variable declaration and can be declared as system global, task global or local.

Example: VAR num globalvar := 123;
 TASK VAR num taskvar := 456;
 LOCAL VAR num localvar := 789;

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 initialized (given an initial value). The expression used to initialize a program variable must be constant. Note that the value of an uninitialized variable may be used, but it is undefined, that is 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 initialization value is set when:

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

2.2.2.3 Persistent declaration

Persistents can only be declared at module level, not inside a routine. Persistents can be declared as system global, task global or local.

Example: PERS num globalpers := 123;
 TASK PERS num taskpers := 456;
 LOCAL PERS num localpers := 789;

All system global persistents with the same name share current value. Task global and local persistents **do not** share current value with other persistents.

Local and task global persistents must be given an initialization value. For system global persistents the initial value may be omitted. The initialization value must be a single value (without data references or operands), or a single aggregate with members which, in turn, are single values or single aggregates.

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

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

Example: PERS pos pallet{ 14, 18 } := [...];

Note that if the current value of a persistent is changed, this causes the initialization value (if not omitted) of the persistent declaration to be updated. However, due to performance issues this update will not take place during program execution. The initial value will be updated when the module is saved (Backup, Save Module, Save Program). It will also be updated when editing program. The program data window on the FlexPendant will always show the current value of the persistent.

Example: PERS num reg1 := 0;
 ...
 reg1 := 5;
 After module save, the saved module looks like this:
 PERS num reg1 := 5;
 ...
 reg1 := 5;

2.2.2.4 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]];

2.2.2.5 Initiating data

The initialization value for a constant or variable can be a constant expression.
The initialization value for a persistent can only be a literal expression.

Example:

```
CONST num a := 2;  
CONST num b := 3;  
! Correct syntax  
CONST num ab := a + b;  
VAR num a_b := a + b;  
PERS num a__b := 5;  
! Faulty syntax  
PERS num a__b := a + b;
```

In the table below, you can see what is happening in various activities such as warm start, new program, program start etc.

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**/Unchanged	Unchanged	Unchanged	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.

**** Persistents without initial value is only initialized if not already declared**

2.2.2.6 Storage class

The *storage class* of a data object determines when the system allocates and de-allocates memory for the data object. The storage class of a data object is determined by the kind of data object and the context of its declaration and can be either *static* or *volatile*.

Constants, persistents, and module variables are static, that is they have the same storage during the lifetime of a task. This means that any value assigned to an persistent or a module variable, always remains unchanged until the next assignment.

Routine variables are volatile. The memory needed to store the value of a volatile variable is allocated first upon the call of the routine in which the declaration of the variable is contained. The memory is later de-allocated at the point of the return to the caller of the routine. This means that the value of a routine variable is always undefined before the call of the routine and is always lost (becomes undefined) at the end of the execution of the routine.

In a chain of recursive routine calls (a routine calling itself directly or indirectly) each instance of the routine receives its own memory location for the “same” routine variable - a number of *instances* of the same variable are created.

2.2.2.7 Syntax

Data declaration

```
<data declaration> ::=
    [LOCAL] ( <variable declaration>
              | <persistent declaration>
              | <constant declaration> )
    | TASK <persistent 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>]

Note! The literal expression may only be omitted for system global persistents.

Constant declaration

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

2.3 Expressions

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

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

2.3.1 Arithmetic expressions

An arithmetic expression is used to evaluate a numeric value.

Example: `2*pi*radius`

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

Figure 5 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, dnum or pos).

Figure 6 2. Integer operations, for example `14 DIV 4=3`, `14 MOD 4=2`. (Non-integer operands are illegal.)

Figure 7 3. Preserves integer (exact) representation as long as operands and result are kept within the integer subdomain of the numerical type.

2.3.2 Logical expressions

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

Example: $a > 5 \text{ AND } b = 3$

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

Figure 8 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

2.3.3 String expressions

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

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

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

2.3.4 Using data in expressions

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

Example: 2*pi*radius

2.3.4.1 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

2.3.4.2 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

2.3.5 Using aggregates in expressions

An aggregate is used for record or array values.

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

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

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

2.3.6 Using function calls in expressions

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

Example: Sin(angle)

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

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

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

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

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

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

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

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

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

2.3.7 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.

2.3.8 Example

Expression	Evaluation order	Comment
$a + b + c$	$(a + b) + c$	left to right rule
$a + b * c$	$a + (b * c)$	* higher than +
$a \text{ OR } b \text{ OR } c$	$(a \text{ OR } b) \text{ OR } c$	Left to right rule
$a \text{ AND } b \text{ OR } c \text{ AND } d$	$(a \text{ AND } b) \text{ OR } (c \text{ AND } d)$	AND higher than OR
$a < b \text{ AND } c < d$	$(a < b) \text{ AND } (c < d)$	< higher than AND

2.3.9 Syntax

2.3.9.1 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> ')'

```

2.3.9.2 Operators

```

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

```

2.3.9.3 Constant values

```

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

```

2.3.9.4 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>

```

2.3.9.5 Aggregates

```

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

```

2.3.9.6 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>

```

2.3.9.7 Special expressions

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

2.3.9.8 Parameters

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

2.4 Instructions

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

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

Instruction	Termination word
IF	ENDIF
FOR	ENDFOR
WHILE	ENDWHILE
TEST	ENDTEST

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

All instructions are collected into specific groups, which are described in the following sections. This grouping is the same as can be found in the pick lists used when adding new instructions to a program on the FlexPendant program editor.

2.4.1 Syntax

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


2.5 Controlling the program flow

The program is executed sequentially as a rule, that is 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.

2.5.1 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.

2.5.2 Calling another routine

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

2.5.3 Program control within the routine

Instruction	Used to:
<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 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)

2.5.4 Stopping program execution

Instruction	Used to:
<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
<i>SystemStopAction</i>	Stop program execution and robot movement

2.5.5 Stop current cycle

Instruction	Used to:
<i>ExitCycle</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.

2.6 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.

2.6.1 Assigning a value to data

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

Instruction	Used to:
<code>:=</code>	Assign a value to data

2.6.2 Wait

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

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

2.6.3 Comments

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

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

2.6.4 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.

Instruction	Used to:
<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>CancelLoad</i>	Cancel the loading of a module that is being or has been loaded with the instruction <i>StartLoad</i>
<i>CheckProgRef</i>	Check program references
<i>Save</i>	Save a program module
<i>EraseModule</i>	Erase a module from the program memory.
Data type	Used to:
<i>loadsession</i>	Program a load session

2.6.5 Various functions

Instruction	Used to:
<i>TryInt</i>	Test if data object is a valid integer
Function	Used to:
<i>OpMode</i>	Read the current operating mode of the robot
<i>RunMode</i>	Read the current program execution mode of the robot
<i>NonMotionMode</i>	Read the current Non-Motion execution mode of the program task
<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>Type</i>	Returns the data type name for a specified variable
<i>IsPers</i>	Check whether a parameter is a persistent
<i>IsVar</i>	Check whether a parameter is a variable

2.6.6 Basic data

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

2.6.7 Conversion function

Function	Used to:
<i>StrToByte</i>	Convert a byte to a string data with a defined byte data format.
<i>ByteToStr</i>	Convert a string with a defined byte data format to a byte data.

2.7 Motion settings

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

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

2.7.1 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_SHARED*)

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

2.7.2 Maximum TCP speed

Function	Used to:
<i>MaxRobSpeed</i>	Return the maximum TCP speed for the used robot type

2.7.3 Defining velocity

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

Instruction	Used to define:
<i>VelSet</i>	The maximum velocity and velocity override
<i>SpeedRefresh</i>	Update speed override for ongoing movement

2.7.4 Defining acceleration

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

Instruction	Used to:
<i>AccSet</i>	Define the maximum acceleration.
<i>WorldAccLim</i>	Limiting the acceleration/deceleration of the tool (and griplload) in the world coordinate system.
<i>PathAccLim</i>	Set or reset limitations on TCP acceleration and/or TCP deceleration along the movement path.

2.7.5 Defining configuration management

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

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

2.7.6 Defining the payload

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

Instruction	Used to define:
<i>GripLoad</i>	The payload of the gripper

2.7.7 Defining the behavior near singular points

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

Instruction	Used to define:
<i>SingArea</i>	The interpolation method through singular points

2.7.8 Displacing a program

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

Instruction	Used to:
<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 additional axis offset
<i>EOffsSet</i>	Activate an additional axis offset by specifying a value
<i>EOffsOff</i>	Deactivate an additional axis offset

Function	Used to:
<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
<i>DefAccFrame</i>	Define a frame from original positions and displaced positions

2.7.9 Soft servo

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

Instruction	Used to:
<i>SoftAct</i>	Activate the soft servo for one or more axes
<i>SoftDeact</i>	Deactivate the soft servo
<i>DitherAct^a</i>	Enables dither functionality for soft servo
<i>DitherDeact^a</i>	Disables dither functionality for soft servo

a. Only for IRB 7600

2.7.10 Adjust the robot tuning values

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

Instruction	Used to:
<i>TuneServo</i>	Adjust the robot tuning values
<i>TuneReset</i>	Reset tuning to normal
<i>PathResol</i>	Adjust the geometric path resolution
<i>CirPathMode</i>	Choose the way the tool reorients during circular interpolation.
Data type	Used to:
<i>tunetype</i>	Represent the tuning type as a symbolic constant

2.7.11 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.

Instruction	Used to:
<i>WZBoxDef^a</i>	Define a box-shaped global zone
<i>WZCylDef^a</i>	Define a cylindrical global zone
<i>WZSphDef^a</i>	Define a spherical global zone
<i>WZHomeJointDef^a</i>	Define a global zone in joints coordinates
<i>WZLimJointDef^a</i>	Define a global zone in joints coordinates for limitation of working area.
<i>WZLimSup^a</i>	Activate limit supervision for a global zone
<i>WZDOSet^a</i>	Activate global zone to set digital outputs
<i>WZDisable^a</i>	Deactivate supervision of a temporary global zone
<i>WZEnable^a</i>	Activate supervision of a temporary global zone
<i>WZFree^a</i>	Erase supervision of a temporary global zone

a. Only when the robot is equipped with the option *World Zones*.

Data type	Used to:
<i>wztemporary^a</i>	Identify a temporary global zone
<i>wzstationary^a</i>	Identify a stationary global zone
<i>shapedata^a</i>	Describe the geometry of a global zone

2.7.12 Various for motion settings

Instruction	Used to:
<i>WaitRob</i>	Wait until the robot and additional axis have reached stop point or have zero speed.

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

2.8 Motion

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

2.8.1 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 additional axes)
- Speed data (desired speed)
- Zone data (position accuracy)
- Tool data (for example the position of the TCP)
- Work object data (for example 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 61):

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

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

2.8.2 Positioning instructions

Instruction	Type of movement:
<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>MoveExtJ</i>	Moves a linear or rotational additional axis without TCP.
<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>	Moves the robot circularly and executes a RAPID procedure.
<i>MoveJSync</i>	Moves the robot by joint movement and executes a RAPID procedure.
<i>MoveLSync</i>	Moves the robot linearly and executes a RAPID procedure.

2.8.3 Searching

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

Instruction	Type of movement:
<i>SearchC</i>	TCP along a circular path.
<i>SearchL</i>	TCP along a linear path.
<i>SearchExtJ</i>	Joint movement of mechanical unit without TCP.

2.8.4 Activating outputs or interrupts at specific positions

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

Instruction	Used to:
<i>TriggIO</i>	Define a trigg condition to set an output at a given position
<i>TriggInt</i>	Define a trigg condition to execute a trap routine at a given position
<i>TriggCheckIO</i>	Define an IO check at a given position

Instruction	Used to:
<i>TriggEquip</i>	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>TriggRampAO</i>	Define a trigg condition to ramp up or down analog output signal at a given position with the possibility to include time compensation for the lag in the external equipment.
<i>TriggC</i>	Run the robot (TCP) circularly with an activated trigg condition.
<i>TriggJ</i>	Run the robot axis-by-axis with an activated trigg condition.
<i>TriggL</i>	Run the robot (TCP) linearly with an activated trigg condition.
<i>TriggLIOs</i>	Run the robot (TCP) linearly with an activated I/O trigg condition.
<i>StepBwdPath</i>	Move backwards on its path in a RESTART event routine.
<i>TriggStopProc</i>	Create an internal supervision process in the system for zero setting of specified process signals and the generation of restart data in a specified persistent variable at every program stop (STOP) or emergency stop (QSTOP) in the system.

Data type	Used to define:
<i>triggdata</i>	Trigg conditions
<i>aiotrigg</i>	Analog I/O trigger condition
<i>restartdata</i>	Data for <i>TriggStopProc</i>
<i>triggios</i>	Trigg conditions for <i>TriggLIOs</i>
<i>triggstrgo</i>	Trigg conditions for <i>TriggLIOs</i>
<i>triggiosdnum</i>	Trigg conditions for <i>TriggLIOs</i>

2.8.5 Control of analog output signal proportional to actual TCP

<i>TriggSpeed</i>	Define conditions and actions for control of an analog output signal with output value proportional to the actual TCP speed.
-------------------	--

2.8.6 Motion control if an error/interrupt takes place

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

Instruction	Used to:
<i>StopMove</i>	Stop the robot movements
<i>StartMove</i>	Restart the robot movements
<i>StartMoveRetry</i>	Restart the robot movements and make a retry in one indivisible sequence

Instruction	Used to:
<i>StopMoveReset</i>	Reset the stop move status, but don't start the robot movements
<i>StorePath</i>	Store the last path generated
<i>RestoPath</i>	Regenerate a path stored earlier
<i>ClearPath</i>	Clear the whole motion path on the current motion path level.
<i>PathLevel</i>	Get the current path level.
<i>SyncMoveSuspend^a</i>	Suspend synchronized coordinated movements on StorePath level.
<i>SyncMoveResume^a</i>	Resume synchronized coordinated movements on StorePath level.

Function	Used to:
<i>IsStopMoveAct</i>	Get status of the stop move flags.

2.8.7 Get robot info in a MultiMove system

Used to retrieve name or reference to the robot in current program task.

Function	Used to:
<i>RobName</i>	Get the controlled robot name in current program task, if any.

Data	Used to:
<i>ROB_ID</i>	Get data containing a reference to the controlled robot in current program task, if any.

2.8.8 Controlling additional axes

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

Instruction	Used to:
<i>DeactUnit</i>	Deactivate an external mechanical unit
<i>ActUnit</i>	Activate an external mechanical unit
<i>MechUnitLoad</i>	Defines a payload for a mechanical unit

Function	Used to:
<i>GetNextMechUnit</i>	Retrieving name of mechanical units in the robot system
<i>IsMechUnitActive</i>	Check whether a mechanical unit is activated or not

2.8.9 Independent axes

The robot axis 6 (and 4 on IRB 2400 /4400) or an additional axis can be moved independently of other movements. The working area of an axis can also be reset, which will reduce the cycle times.

Instruction	Used to:
<i>IndAMove^a</i>	Change an axis to independent mode and move the axis to an absolute position
<i>IndCMove^a</i>	Change an axis to independent mode and start the axis moving continuously.
<i>IndDMove^a</i>	Change an axis to independent mode and move the axis a delta distance
<i>IndRMove^a</i>	Change an axis to independent mode and move the axis to a relative position (within the axis revolution)
<i>IndReset^a</i>	Change an axis to dependent mode or/and reset the working area
<i>HollowWristReset^b</i>	Reset the position of the wrist joints on hollow wrist manipulators, such as IRB 5402 and IRB 5403.

a. Only if the robot is equipped with the option *Independent movement*.

b. Only if the robot is equipped with the option *Independent movement*. The instruction *HollowWristReset* can only be used on IRB 5402 and IRB 5403.

Function	Used to:
<i>IndInpos^a</i>	Check whether an independent axis is in position
<i>IndSpeed^a</i>	Check whether an independent axis has reached programmed speed

2.8.10 Path correction

Instruction	Used to:
<i>CorrCon^a</i>	Connect to a correction generator
<i>CorrWrite^a</i>	Write offsets in the path coordinate system to a correction generator
<i>CorrDiscon^a</i>	Disconnect from a previously connected correction generator
<i>CorrClear^a</i>	Remove all connected correction generators

a. Only if the robot is equipped with the option *Path offset* or *RobotWare-Arc sensor*.

Function	Used to:
<i>CorrRead^a</i>	Read the total corrections delivered by all connected correction generators

Data type	Used to:
<i>corrdescr^a</i>	Add geometric offsets in the path coordinate system

2.8.11 Path Recorder

Instruction	Used to:
<i>PathRecStart^a</i>	Start recording the robot's path
<i>PathRecStop^a</i>	Stop recording the robot's path
<i>PathRecMoveBwd^a</i>	Move the robot backwards along a recorded path
<i>PathRecMoveFwd^a</i>	Move the robot back to the position where PathRecMoveBwd was executed

a. Only if the robot is equipped with the option *Path recovery*.

Function	Used to:
<i>PathRecValidBwd^a</i>	Check if the path recorder is active and if a recorded backward path is available
<i>PathRecValidFwd^a</i>	Check if the path recorder can be used to move forward
Data type	Used to:
<i>pathrecid</i>	Identify a breakpoint for the path recorder

2.8.12 Conveyor tracking

Instruction	Used to:
<i>WaitWObj^a</i>	Wait for work object on conveyor
<i>DropWObj^a</i>	Drop work object on conveyor

a. Only if the robot is equipped with the option *Conveyor tracking*.

2.8.13 Servo Tracking for Indexing Conveyor

Instruction	Used to:
<i>IndCnvAddObject^a</i>	Used to manually add an object to the object queue.
<i>IndCnvEnable^a</i>	Start to listen to the digital input and execute an indexing movement when triggered.
<i>IndCnvDisable^a</i>	The system stop listen to the digital input.
<i>IndCnvInit^a</i>	Set up the indexed conveyor functionality
<i>IndCnvReset^a</i>	To be able to jog or execute a move instruction for the indexing conveyor the system needs to be set to Normal Mode, and that is done with this instruction, or when moving PP to main.

a. Only if the robot is equipped with the option *Conveyor tracking*.

Data type	Used to:
<i>indcnvdata</i>	Used to setup the behavior of the indexing conveyor functionality.

2.8.14 Sensor synchronization

Sensor Synchronization, is the function whereby the robot speed follows a sensor which can be mounted on a moving conveyor or a press motor axis .

Instruction	Used to:
<i>WaitSensor^a</i>	Connect to an object in the start window on a sensor mechanical unit.
<i>SyncToSensor^a</i>	Start or stop synchronization of robot movement to sensor movement.
<i>DropSensor^a</i>	Disconnect from the current object.

a. Only if the robot is equipped with the option *Sensor synchronization*.

2.8.15 Load identification and collision detection

Instruction	Used to:
<i>MotionSup^a</i>	Deactivates/activates motion supervision
<i>ParIdPosValid</i>	Valid robot position for parameter identification
<i>ParIdRobValid</i>	Valid robot type for parameter identification
<i>LoadId</i>	Load identification of tool or payload
<i>ManLoadId</i>	Load identification of external manipulator

a. Only if the robot is equipped with the option *Collision detection*.

Data type	Used to:
<i>loadidnum</i>	Represent an integer with a symbolic constant
<i>paridnum</i>	Represent an integer with a symbolic constant
<i>paridvalidnum</i>	Represent an integer with a symbolic constant

2.8.16 Position functions

Function	Used to:
<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>CalcRobT</i>	Calculates <i>robtarg</i> from <i>jointtarget</i>

Function	Used to:
<i>CPos</i>	Read the current position (only x, y, z of the robot)
<i>CRobT</i>	Read the current position (the complete <i>robtarget</i>)
<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
<i>CalcJointT</i>	Calculates joint angles from <i>robtarget</i>
<i>Distance</i>	The distance between two positions

2.8.17 Check interrupted path after power failure

Function	Used to:
<i>PFRestart</i>	Check if the path has been interrupted at power failure.

2.8.18 Status functions

Function	Used to:
<i>CSpeedOverride</i>	Read the speed override set by the operator from the Program or Production Window.

2.8.19 Motion data

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

Data type	Used to define:
<i>robtargt</i>	The end position
<i>jointtargt</i>	The end position for a <i>MoveAbsJ</i> or <i>MoveExtJ</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
<i>stoppointdata</i>	The termination of the position
<i>identno</i>	A number used to control synchronizing of two or more coordinated synchronized movement with each other

2.8.20 Basic data for movements

Data type	Used to define:
<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 additional axes
<i>robjoint</i>	The position of the robot axes
<i>loaddata</i>	A load
<i>mecunit</i>	An external mechanical unit

2.9 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.

2.9.1 Programming principles

The signal names are defined in the system parameters. These names are always available in the program for reading or setting I/O operations.

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

2.9.2 Changing the value of a signal

Instruction	Used to:
<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; for example <i>high/low</i>)
<i>SetGO</i>	Change the value of a group of digital output signals

2.9.3 Reading the value of an input signal

The value of an input signal can be read directly in the program, for example:

```
! Digital input
  IF di1 = 1 THEN ...
! Digital group input
  IF gi1 = 5 THEN ...
! Analog input
  IF ai1 > 5.2 THEN ...
```

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

ERR_NORUNUNIT No contact with the unit

2.9.4 Reading the value of an output signal

Function	Used to:
<i>AOutput</i>	Read the current value of an analog output signal
<i>DOutput</i>	Read the current value of a digital output signal
<i>GOutput</i>	Read the current value of a group of digital output signals
<i>GOutputDnum</i>	Read the current value of a group of digital output signals. Can handle digital group signals with up to 32 bits. Return the read value in a dnum data type.
<i>GInputDnum</i>	Read the current value of a group of digital input signals. Can handle digital group signals with up to 32 bits. Return the read value in a dnum data type.

2.9.5 Testing input or output signals

Instruction	Used to:
<i>WaitDI</i>	Wait until a digital input is set or reset
<i>WaitDO</i>	Wait until a digital output is set on reset
<i>WaitGI</i>	Wait until a group of digital input signals is set to a value
<i>WaitGO</i>	Wait until a group of digital output signals is set to a value
<i>WaitAI</i>	Wait until an analog input is less or greater than a value
<i>WaitAO</i>	Wait until an analog output is less or greater than a value

Function	Used to:
<i>TestDI</i>	Test whether a digital input is set
<i>ValidIO</i>	Valid I/O signal to access

2.9.6 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.

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

2.9.7 Defining input and output signals

Data type	Used to define:
<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>signaldi</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 *
Instruction	Used to:
<i>AliasIO</i>	Define a signal with an alias name

2.9.8 Get status of I/O bus and unit

Data type	Used to define:
<i>iounit_state</i>	The status of the I/O unit
<i>bustate</i>	The status of the I/O bus
Function	Used to:
<i>IOUnitState</i>	Returns current status of the I/O unit.
Instruction	Used to:
<i>IOBusState</i>	Get current status of the I/O bus.

2.9.9 Start of I/O bus

Instruction	Used to:
<i>IOBusStart</i>	Start an I/O bus.

2.10 Communication

There are four possible ways to communicate via serial channels:

- Messages can be output to the FlexPendant 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 in 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.

2.10.1 Programming principles

The decision whether to use character-based or binary information depends 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 FlexPendant 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.

2.10.2 Communicating using the FlexPendant, function group TP

Instruction	Used to:
<i>TPERase</i>	Clear the FlexPendant operator display
<i>TPWrite</i>	Write text on the FlexPendant operator display
<i>ErrWrite</i>	Write text on the FlexPendant 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>TPReadDnum</i>	Read a numeric value from the FlexPendant
<i>TPReadNum</i>	Read a numeric value from the FlexPendant
<i>TPShow</i>	Choose a window on the FlexPendant from RAPID

Data type	Used to:
<i>tpnum</i>	Represent FlexPendant window with a symbolic constant

2.10.3 Communicating using the FlexPendant, function group UI

Instruction	Used to:
<i>UIMsgBox</i>	Write message to FlexPendant Read pressed button from FlexPendant Type basic
<i>UIShow</i>	Open an application on the FlexPendant from RAPID

Function	Used to:
<i>UIMessageBox</i>	Write message to FlexPendant Read pressed button from FlexPendant Type advanced
<i>UIDnumEntry</i>	Read a numeric value from the FlexPendant
<i>UIDnumTune</i>	Tune a numeric value from the FlexPendant
<i>UINumEntry</i>	Read a numeric value from the FlexPendant
<i>UINumTune</i>	Tune a numeric value from the FlexPendant
<i>UIAlphaEntry</i>	Read text from the FlexPendant
<i>UIListView</i>	Select item in a list from the FlexPendant
<i>UIClientExist</i>	Is the FlexPendant connected to the system

Data type	Used to:
<i>icondata</i>	Represent icon with a symbolic constant
<i>buttondata</i>	Represent button with a symbolic constant
<i>listitem</i>	Define menu list items
<i>btnres</i>	Represent selected button with a symbolic constant
<i>uishownum</i>	Instance Id for UIShow

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

Instruction	Used to:
<i>Open</i>	Open a channel/file for reading or writing
<i>Write</i>	Write text to the channel/file
<i>Close</i>	Close the channel/file

Function	Used to:
<i>ReadNum</i>	Read a numeric value
<i>ReadStr</i>	Read a text string

2.10.5 Communicating using binary serial channels/files/field buses

Instruction	Used to:
<i>Open</i>	Open a serial channel/file for binary transfer of data
<i>WriteBin</i>	Write to a binary serial channel/file
<i>WriteAnyBin</i>	Write to any binary serial channel/file
<i>WriteStrBin</i>	Write a string to a binary serial channel/file
<i>Rewind</i>	Set the file position to the beginning of the file
<i>Close</i>	Close the channel/file
<i>ClearIOBuff</i>	Clear input buffer of a serial channel
<i>ReadAnyBin</i>	Read from any binary serial channel
<i>WriteRawBytes</i>	Write data of type rawbytes to a binary serial channel/file/field bus
<i>ReadRawBytes</i>	Read data of type rawbytes from a binary serial channel/file/field bus

Function	Used to:
<i>ReadBin</i>	Read from a binary serial channel
<i>ReadStrBin</i>	Read a string from a binary serial channel/file

2.10.6 Communication using rawbytes

The instructions and functions below are used to support the communication instructions *WriteRawBytes* and *ReadRawBytes*.

Instruction	Used to:
<i>ClearRawBytes</i>	Set a rawbytes variable to zero
<i>CopyRawBytes</i>	Copy from one rawbytes variable to another
<i>PackRawBytes</i>	Pack the contents of a variable into a "container" of type rawbytes
<i>UnPackRawBytes</i>	Unpack the contents of a "container" of type rawbytes to a variable
<i>PackDNHeader</i>	Pack the header of a DeviceNet message into a "container" of rawbytes

Function	Used to:
<i>RawBytesLen</i>	Get the current length of valid bytes in a rawbyte variable

2.10.7 Data for serial channels/files/fieldbuses

Data type	Used to define:
<i>iodev</i>	A reference to a serial channel/file, which can then be used for reading and writing
<i>rawbytes</i>	A general data “container”, used for communication with I/O devices

2.10.8 Communicating using sockets

Instruction	Used to:
<i>SocketCreate</i>	Create a new socket
<i>SocketConnect</i>	Connect to remote computer (only client applications)
<i>SocketSend</i>	Send data to remote computer
<i>SocketReceive</i>	Receive data from remote computer
<i>SocketClose</i>	Close the socket
<i>SocketBind</i>	Bind a socket to a port (only server applications)
<i>SocketListen</i>	Listen for connections (only server applications)
<i>SocketAccept</i>	Accept connections (only server applications)

Function	Used to:
<i>SocketGetStatus</i>	Get current socket state

Data type	Used to define:
<i>socketdev</i>	Socket device
<i>socketstatus</i>	Socket status

2.10.9 Communication using RAPID Message Queues

Data type	Used to define:
<i>rmqheader^a</i>	The rmqheader is a part of the data type rmqmessage and is used to describe the message
<i>rmqmessage^a</i>	A general data container, used when communicate with RAPID Message Queue functionality
<i>rmqslot^a</i>	Identity number of a RAPID task or Robot Application Builder client

Instruction	Used to:
<i>IRMQMessage^a</i>	Order and enable interrupts for a specific data type
<i>RMQFindSlot^a</i>	Find the identity number of the queue configured for a RAPID task or Robot Application Builder client
<i>RMQGetMessage^a</i>	Get the first message from the queue of this task
<i>RMQGetMsgData^a</i>	Extract the data from a message
<i>RMQGetMsgHeader^a</i>	Extract header information from a message
<i>RMQSendMessage^a</i>	Send data to the queue of the queue configured for a RAPID task or Robot Application Builder client
<i>RMQSendWait^a</i>	Send a message and wait for the answer
<i>RMQEmptyQueue</i>	Empty the RMQ connected to the task executing instruction.
<i>RMQReadWait</i>	Wait until a message has arrived, or timeout occurs.

Function	Used to:
<i>RMQGetSlotName^a</i>	Get the name of a RAPID Message Queue client from a given identity number, that is from a given rmqslot

a. Only if the robot is equipped with at least one of the options *FlexPendant Interface*, *PC Interface*, or *Multitasking*

2.11 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, for example *WaitUntil*,
- any time during the execution of a movement instruction, for example *MoveL*.

This normally results in a delay of 2-30 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 in queue at stop will be thrown away and any interrupts generated under stop will not be dealt, except for safe interrupts, see Safe Interrupt on page 91.

The maximum number of defined interrupts at any one time is limited to 100 **per program task**.

2.11.1 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, that is 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.

2.11.2 Connecting interrupts to trap routines

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

2.11.3 Ordering interrupts

Instruction	Used to order:
<i>ISignalDI</i>	An interrupt from a digital input signal
<i>ISignalDO</i>	An interrupt from a digital output signal
<i>ISignalGI</i>	An interrupt from a group of digital input signals
<i>ISignalGO</i>	An interrupt from a group of digital output signals
<i>ISignalAI</i>	An interrupt from an analog input signal
<i>ISignalAO</i>	An interrupt from an analog output signal
<i>ITimer</i>	A timed interrupt
<i>TriggInt</i>	A position-fixed interrupt (from the Motion pick list)
<i>IPers</i>	An interrupt when changing a persistent.
<i>IError</i>	Order and enable an interrupt when an error occurs
<i>IRMQMessage^a</i>	An interrupt when a specified data type is received by a RAPID Message Queue.

a. Only if the robot is equipped with the option *FlexPendant Interface*, *PC Interface*, or *Multi-tasking*.

2.11.4 Cancelling interrupts

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

2.11.5 Enabling/disabling interrupts

Instruction	Used to:
<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

2.11.6 Interrupt data

Instruction	Is Used:
<i>GetTrapData</i>	in a trap routine to obtain all information about the interrupt that caused the trap routine to be executed.
<i>ReadErrData</i>	in a trap routine, to obtain numeric information (domain, type and number) about an error, a state change, or a warning, that caused the trap routine to be executed.

2.11.7 Data type of interrupts

Data type	Used to:
<i>intnum</i>	Define the identity of an interrupt.
<i>trapdata</i>	Contain the interrupt data that caused the current TRAP routine to be executed.
<i>errtype</i>	Specify an error type (gravity)
<i>errdomain</i>	Order and enable an interrupt when an error occur.
<i>errdomain</i>	Specify an error domain.

2.11.9 Interrupt manipulation

Some instructions, for example *ITimer* and *ISignalDI*, can be used together with Safe Interrupt. Safe Interrupts are interrupts that will be queued if they arrive during stop or stepwise execution. The queued interrupts will be dealt with as soon as continuous execution is started, they will be handled in *FIFO* order. Interrupts in queue at stop will also be dealt with. The instruction *ISleep* can not be used together with safe interrupts.

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

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

An activated interrupt may in turn be deactivated (and vice versa). During the deactivation time, any generated interrupts of the specified type are thrown away without any trap execution.

Example:	ISleep sig1 int;	deactivate
	·	
	IWatch sig1 int;	activate

An enabled interrupt may in turn be disabled (and vice versa). During the disable time, any generated interrupts of the specified type are queued and raised first when the interrupts are enabled again.

Example:	IDisable sig1int;	disable
	·	
	IEnable sig1int;	enable

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;

2.11.10 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, that is causes immediate termination of program execution.

```
Example:  VAR intnum empty;
          VAR intnum full;

          PROC main()

              ! Connect trap routines
              CONNECT empty WITH etrap;
              CONNECT full WITH ftrap;

              ! Define feeder interrupts
              ISignalDI di1, high, empty;
              ISignalDI di3, high, full;
              .
              .
              ! Delete interrupts
              IDelete empty;
              IDelete full;
          ENDPROC

          ! Responds to "feeder empty" interrupt
          TRAP etrap
              open_valve;
              RETURN;
          ENDTRAP

          ! Responds to "feeder full" interrupt
          TRAP ftrap
              close_valve;
              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.

2.12 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 system, such as division by zero, or of a type that is raised by the program, such as a program raising an error when an incorrect value is read by a bar code reader.

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 system. 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.

2.12.1 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.

In an 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 (RETURN, RETRY, TRYNEXT, and RAISE).

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

2.12.2 Creating an error situation from within the program

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

2.12.3 Booking an error number

Instruction	Used to:
<i>BookErrNo</i>	Book a new RAPID system error number.

2.12.4 Restarting/returning from the error handler

Instruction	Used to:
<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
<i>RaiseToUser</i>	From a NOSTEPIN routine, the error is raised to the error handler at user level.
<i>StartMoveRetry</i>	An instruction that replaces the two instructions <i>StartMove</i> and <i>RETRY</i> . It both resumes movements and re-execute the instruction that caused the error.
<i>SkipWarn</i>	Skip the latest requested warning message.
<i>ResetRetryCount</i>	Reset the number of counted retries.

Function	Used to:
<i>RemainingRetries</i>	Remaining retries left to do.

2.12.5 User defined errors and warnings

2.12.6 IGenerate process error

Instruction	Used to:
<i>ErrLog</i>	Display an error message on the teach pendant and write it in the robot message log.
<i>ErrRaise</i>	Create an error in the program and then call the error handler of the routine.

Instruction	Used to:
<i>ProcerrRecovery</i>	Generate process error during robot movement.

2.12.7 Data for error handling

Data type	Used to define:
<i>errnum</i>	The reason for the error
<i>errstr</i>	Text in an error message

2.12.8 Configuration for error handling

System parameter	Used to define:
<i>No Of Retry</i>	The number of times a failing instruction will be retried if the error handler use <i>RETRY</i> . <i>No Of Retry</i> belongs to the type <i>System Misc</i> in the topic <i>Controller</i> .

2.12.9 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. To be able to make more than four retries, you have to configure the system parameter *No Of Retry*, see *Technical reference manual - System parameters*.
- 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.

2.12.10 System error handler

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.

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

2.12.11 Errors raised by the program

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.

2.12.12 The event log

Errors that are handled by an error handler still result in a warning in the event log. By looking in the event log it is possible to track what errors that have occurred.

If you want an error to be handled without writing a warning in the event log, use the instruction *SkipWarn* in the error handler. This can be useful when using the error handler to test something (for example if a file exists) without leaving any trails if the test fails.

2.12.13 UNDO

Rapid routines may contain an UNDO-handler. The handler is executed automatically if the PP is moved out of the routine. This is supposed to be used for cleaning up remaining side-effects after partially executed routines, for example canceling modal instructions (such as opening a file). Most parts of the Rapid language can be used in an UNDO-handler, but there are some limitations, for example motion instructions.

2.12.13.1 Definitions/terminology

In order to avoid ambiguousness in the following text, next follows a list of explanations to terms related to UNDO.

UNDO: The execution of clean-up code prior to a program reset.

UNDO-handler: An optional part of a Rapid procedure or function containing Rapid code that is executed on an UNDO.

UNDO-routine: A procedure or a function with an UNDO-handler.

Call-chain: All procedures or functions currently associated to each other through not-yet finished routine invocations. Assumed to start in the Main routine if nothing else is specified.

UNDO context: When the current routine is part of a call-chain starting in an UNDO-handler.

2.12.13.2 When to use UNDO

A Rapid routine can be aborted at any point by moving the program pointer out of the routine. In some cases, when the program is executing certain sensitive routines, it is unsuitable to abort. Using UNDO it is possible to protect such sensitive routines against an unexpected program reset. With UNDO it is possible to have certain code executed automatically if the routine is aborted. This code should typically perform clean-up actions, for instance closing a file.

2.12.13.3 UNDO behavior in detail

When UNDO is activated, all UNDO-handlers in the current call-chain are executed. These handlers are optional parts of a Rapid procedure or function, containing Rapid code. The currently active UNDO-handlers are those who belong to procedures or functions that has been invoked but not yet terminated, that is the routines in the current call-chain.

UNDO is activated when the PP is unexpectedly moved out of an UNDO-routine, for instance if the user moves PP to Main. UNDO is also started if an EXIT instruction is executed, causing the program to be reset, or if the program is reset for some other

reason, for instance when changing some configuration or if the program or module is deleted. However, UNDO is not started if the program reaches the end of the routine or a return-statement and returns as usual from the routine.

If there is more than one UNDO-routine in the call-chain, the UNDO-handlers of the routines will be processed in the same order the routines would have returned, bottom-up. The UNDO-handler closest to the end of the call-chain will execute first and the one closest to Main will execute last.

2.12.13.4 Limitations

An UNDO-handler can access any variable or symbol reachable from the normal routine body, including locally declared variables. Rapid-code that are to be executed in UNDO-context has however limitations.

An UNDO-handler must not contain STOP, BREAK, RAISE or RETURN. If an attempt is made to use any of these instructions in UNDO context, the instruction will be ignored and an ELOG warning is generated.

Motion-instructions, that is MoveL, are not allowed in UNDO context either.

The execution is always continuous in UNDO, it is not possible to step. When UNDO starts, the execution mode is set to continuous automatically. After the UNDO session is finished, the old execution mode is restored.

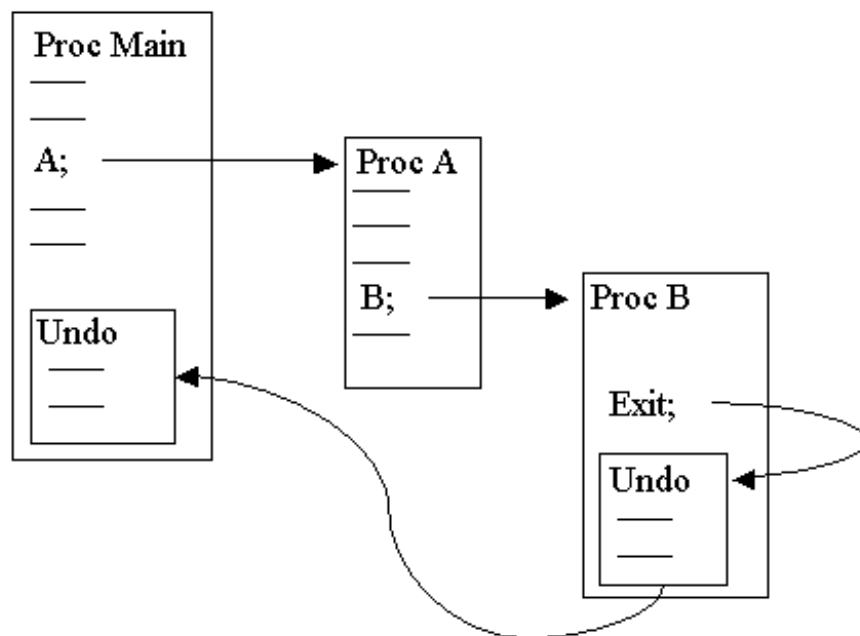
If the program is stopped while executing an UNDO-handler, the rest of the handler will not be executed. If there are additional UNDO-handlers in the call-chain that have not yet been executed, they will be ignored as well. This will result in an ELOG warning. This also includes stopping due to a runtime error.

The PP is not visible in an UNDO-handler. When UNDO executes, the PP remains at its old location, but is updated when the UNDO-handler(s) are finished.

An EXIT instruction aborts UNDO in similar way as a Run-time error or a Stop. The rest of the UNDO-handlers are ignored and the PP is moved to Main.

2.12.13.5 Example

<p>The Program:</p> <pre> PROC B TPWrite "In Routine B"; Exit; UNDO TPWrite "In UNDO of routine B"; ENDPROC PROC A TPWrite "In Routine A"; B; ENDPROC PROC main TPWrite "In main"; A; UNDO TPWrite "In UNDO of main"; ENDPROC </pre>	<p>The output:</p> <pre> In main In Routine A In Routine B In UNDO of routine B In UNDO of main </pre>
--	--



UNDO

2.13 System & time

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

2.13.1 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 FlexPendant 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.

2.13.2 Using a clock to time an event

Instruction	Used to:
<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
Function	Used to:
<i>ClkRead</i>	Read a clock used for timing
Data Type	Used for:
<i>clock</i>	Timing – stores a time measurement in seconds

2.13.3 Reading current time and date

Function	Used to:
<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

2.13.4 Retrieve time information from file

Function	Used to:
<i>FileTime</i>	Retrieve the last time for modification of a file.
<i>ModTime</i>	Retrieve file modify time for the loaded module.
<i>ModExist</i>	Check if program module exist.

2.13.5 Get the size of free program memory

Function	Used to:
<i>ProgMemFree</i>	Retrieve the size of free program memory.

2.14 Mathematics

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

2.14.1 Programming principles

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

2.14.2 Simple calculations on numeric data

Instruction	Used to:
<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

2.14.3 More advanced calculations

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

2.14.4 Arithmetic functions

Function	Used to:
<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

Function	Used to:
<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>	Normalize unnormalized orientation (quaternion)

2.14.5 String digit functions

Function	Used to:
<i>StrDigCmp</i>	Numeric compare of two strings with only digits
<i>StrDigCalc</i>	Arithmetic operations on two strings with only digits

Data type	Used to:
<i>stringdig</i>	String with only digits

2.14.6 Bit functions

Instruction	Used to:
<i>BitClear</i>	Clear a specified bit in a defined <i>byte</i> data.
<i>BitSet</i>	Set a specified bit to 1 in a defined <i>byte</i> data.

Function	Used to:
<i>BitCheck</i>	Check if a specified bit in a defined byte data is set to 1.
<i>BitAnd</i>	Execute a logical bitwise <i>AND</i> operation on data types <i>byte</i> .
<i>BitNeg</i>	Execute a logical bitwise <i>NEGATION</i> operation on data types <i>byte</i> .
<i>BitOr</i>	Execute a logical bitwise <i>OR</i> operation on data types <i>byte</i> .
<i>BitXOr</i>	Execute a logical bitwise <i>XOR</i> operation on data types <i>byte</i> .
<i>BitLSh</i>	Execute a logical bitwise <i>LEFT SHIFT</i> operation on data types <i>byte</i> .
<i>BitRSh</i>	Execute a logical bitwise <i>RIGHT SHIFT</i> operation on data types <i>byte</i> .

Data type	Used to:
<i>byte</i>	Used together with instructions and functions that handle bit manipulation.

2.15 External computer communication

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

2.15.1 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.

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

Instruction	Used to:
<i>SCWrite</i> ^a	Send a message to the superordinate computer

- a. Only if the robot is equipped with the option *PC interface/backup*.

2.16 File operation functions

Instruction	Used to:
<i>MakeDir</i>	Create a new directory.
<i>RemoveDir</i>	Remove a directory.
<i>OpenDir</i>	Open a directory for further investigation.
<i>CloseDir</i>	Close a directory in balance with <i>OpenDir</i> .
<i>RemoveFile</i>	Remove a file.
<i>RenameFile</i>	Rename a file.
<i>CopyFile</i>	Copy a file.

Function	Used to:
<i>ISFile</i>	Check the type of a file.
<i>FSSize</i>	Retrieve the size of a file system.
<i>FileSize</i>	Retrieve the size of a specified file.
<i>ReadDir</i>	Read next entry in a directory.

Data Type	Used to:
<i>dir</i>	Traverse directory structures.

2.17 RAPID support instructions

Various functions for supporting of the RAPID language:

- Get system data
- Read configuration data
- Write configuration data
- Restart the controller
- Test system data
- Get object name
- Get task name
- Search for symbols
- Get current event type, execution handler or execution level

2.17.1 Get system data

Instruction to fetch the value and (optional) the symbol name for the current system data of specified type.

Instruction	Used to:
<i>GetSysData</i>	Fetch data and name of current active Tool or Work Object.
<i>ResetPPMoved</i>	Reset state for the program pointer moved in manual mode.
<i>SetSysData</i>	Activate a specified system data name for a specified data type.

Function	Used to:
<i>IsSysID</i>	Test the system identity.
<i>IsStopStateEvent</i>	Get information about the movement of the Program Pointer.
<i>PPMovedInManMode</i>	Test whether the program pointer is moved in manual mode.
<i>RobOS</i>	Check if the execution is performed on Robot Controller RC or Virtual Controller VC.

2.17.2 Get information about the system

Function to get information about Serial Number, SoftWare Version, Robot Type, LAN ip address or Controller Language.

Function	Used to:
<i>GetSysInfo</i>	Get information about the system.

2.17.3 Get information about memory

Function	Used to:
<i>ProgMemFree</i>	Get the size of free program memory

2.17.4 Read configuration data

Instruction to read one attribute of a named system parameter.

Instruction	Used to:
<i>ReadCfgData</i>	Read one attribute of a named system parameter.

2.17.5 Write configuration data

Instruction to write one attribute of a named system parameter.

Instruction	Used to:
<i>WriteCfgData</i>	Write one attribute of a named system parameter.

2.17.6 Restart the controller

Instruction	Used to:
<i>WarmStart</i>	Restart the controller for example when you have changed system parameters from RAPID.

2.17.7 Text tables instructions

Instructions to administrate text tables in the system.

Instruction	Used to:
<i>TextTabInstall</i>	Install a text table in the system.

Function	Used to:
<i>TextTabGet</i>	Get the text table number of a user defined text table.
<i>TextGet</i>	Get a text string from the system text tables.

Function	Used to:
<i>TextTabFreeToUse</i>	Test whether the text table name (text resource string) is free to use or not.

2.17.8 Get object name

Instruction to get the name of an original data object for a current argument or current data.

Function	Used to:
<i>ArgName</i>	Return the original data object name.

2.17.9 Get information about the tasks

Function	Used to:
<i>GetTaskName</i>	Get the identity of the current program task, with its name and number.
<i>MotionPlannerNo</i>	Get the number of the current motion planner.

2.17.10 Get current event type, execution handler or execution level

Function	Used to:
<i>EventType</i>	Get current event routine type.
<i>ExecHandler</i>	Get type of execution handler.
<i>ExecLevel</i>	Get execution level.

Data Type	Used to:
<i>event_type</i>	Event routine type.
<i>handler_type</i>	Type of execution handler.
<i>exec_level</i>	Execution level.

2.17.11 Search for symbols

Instructions to search for data objects in the system.

Instruction	Used to:
<i>SetAllDataVal</i>	Set a new value to all data objects of a certain type that match a given grammar.
<i>SetDataSearch</i>	Together with <i>GetNextSym</i> data objects can be retrieved from the system.
<i>GetDataVal</i>	Get a value from a data object that is specified with a string variable.
<i>SetDataVal</i>	Set a value for a data object that is specified with a string variable.

Function	Used to:
<i>GetNextSym</i>	Together with <i>SetDataSearch</i> data objects can be retrieved from the system.

Data type	Used to:
<i>datapos</i>	Holds information of where a certain object is defined in the system.

2.18 Calibration & service instructions

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

2.18.1 Calibration of the tool

Instructions	Used to:
<i>MToolRotCalib</i>	Calibrate the rotation of a moving tool.
<i>MToolTCPCalib</i>	Calibrate Tool Center Point - TCP for a moving tool.
<i>SToolRotCalib</i>	Calibrate the TCP and rotation of a stationary tool.
<i>SToolTCPCalib</i>	Calibrate Tool Center Point - TCP for a stationary tool

2.18.2 Various calibration methods

Functions	Used to:
<i>CalcRotAxisFrame</i>	Calculate the user coordinate system of a rotational axis type.
<i>CalcRotAxFrameZ</i>	Calculate the user coordinate system of a rotational axis type when the master robot and the additional axis are located in different RAPID tasks.
<i>DefAccFrame</i>	Define a frame from original positions and displaced positions.

2.18.3 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.

Instructions	Used to:
<i>TestSignDefine</i>	Define a test signal
<i>TestSignReset</i>	Reset all test signals definitions

Function	Used to:
<i>TestSignRead</i>	Read test signal value

Data type	Used to:
<i>testsignal</i>	For programming instruction <i>TestSignDefine</i>

2.18.4 Recording of an execution

The recorded data is stored in a file for later analysis, and is intended for debugging RAPID programs, specifically for multi-tasking systems.

Instructions	Used to:
<i>SpyStart</i>	Start the recording of instruction and time data during execution.
<i>SpyStop</i>	Stop the recording of time data during execution.

2.19 String functions

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

2.19.1 Basic operations

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

2.19.2 Comparison and searching

Operator	Used to:
<code>=</code>	Test if equal to
<code><></code>	Test if not equal to
Function	Used to:
<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

2.19.3 Conversion

Function	Used to:
<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
<i>DecToHex</i>	Convert a number specified in a readable string in the base 10 to the base 16
<i>HexToDec</i>	Convert a number specified in a readable string in the base 16 to the base 10

2.20 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. 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 additional program. Each task can be of type **NORMAL**, **STATIC**, or **SEMISTATIC**.

Up to 20 different tasks can be run in pseudo parallel. Each task consists of a set of modules that are local in each task.

Variables, constants, and persistents are local in each task, but global persistents are not. A persistent is global by default, if not declared as LOCAL or TASK. A global persistent with the same name and type is reachable in all tasks that it is declared in. If two global 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 (for example 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
- When running a Wait instruction in manual mode, a simulation box will come up after 3 seconds. This will only occur in a **NORMAL** task.
- Move instructions can only be executed in the motion task (the task bind to program instance 0, see Technical reference manual - 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 FlexPendant cannot access other tasks than a **NORMAL** task. So, the development of RAPID programs for other **SEMISTATIC** or **STATIC** tasks can only be done if the code is loaded into a **NORMAL** task, or off-line.

For all settings, see *Technical reference manual - System parameters*.

2.20.1 Basics

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

Up to 20 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 (for example Start/Stop/Restart....).

2.20.2 General instructions and functions

Instruction	Used to
<i>WaitSyncTask^a</i>	Synchronize several program tasks at a special point in each program

a. If the robot is equipped with the option *MultiTasking*.

Function	Used to
<i>TestAndSet</i>	Retrieve exclusive right to specific RAPID code areas or system resources (type user poll)
<i>WaitTestAndSet</i>	Retrieve exclusive right to specific RAPID code areas or system resources (type interrupt control)
<i>TaskRunMec</i>	Check if the program task controls any mechanical unit.
<i>TaskRunRob</i>	Check if the program task controls any TCP-robot
<i>GetMecUnitName</i>	Get the name of the mechanical unit

Data types	Used to
<i>taskid</i>	Identify available program tasks in the system.
<i>syncident^a</i>	Specify the name of a synchronization point
<i>tasks¹</i>	Specify several RAPID program tasks

2.20.3 MultiMove system with coordinated robots

Instruction	Used to
<i>SyncMoveOn</i> ^a	Start a sequence of synchronized movements
<i>SyncMoveOff</i> ^a	To end synchronized movements
<i>SyncMoveUndo</i>	Reset synchronized movements

a. If the robot is equipped with the option *MultiMove Coordinated*.

Function	Used to
<i>IsSyncMoveOn</i>	Tell if the current task is in synchronized mode
<i>TasksInSync</i>	Returns the number of synchronized tasks

Data types	Used to
<i>syncident</i> ^a	To specify the name of a synchronization point
<i>tasks</i> ^a	Specify several RAPID program tasks
<i>identno</i>	Identity for move instructions

a. If the robot is equipped with the option *MultiTasking*.

2.20.4 Synchronizing 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 synchronization mechanism is necessary. But there are other applications which need to know what the main task is doing, for example.

2.20.5 Synchronizing 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 1

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

```
    startsync:= TRUE;
    .
```

```
ENDPROC
ENDMODULE
```

TASK 2

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

```
    WaitUntil startsync;
    .
```

```
ENDPROC
ENDMODULE
```

2.20.6 Synchronizing using an interrupt

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

Example

TASK 1

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

```
SetDO do1,1;  
.
```

```
ENDPROC  
ENDMODULE
```

TASK 2

```
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
```

2.20.7 Intertask communication

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

A global 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.

Example

TASK 1

```
MODULE module1
PERS bool startsync:=FALSE;
PERS string stringtosend:="";
PROC main()
```

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

```
    startsync:= TRUE
```

```
ENDPROC
ENDMODULE
```

TASK 2

```
MODULE module2
PERS bool startsync:=FALSE;
PERS string stringtosend:="";
PROC main()
```

```
    WaitUntil startsync;
```

```
    !read string
```

```
    IF stringtosend = "this is a test" THEN
```

```
ENDPROC
ENDMODULE
```

2.20.8 Type of task

Each task can be of type **NORMAL**, **STATIC** or **SEMISTATIC**.

STATIC and **SEMISTATIC** tasks are started in the system startup 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). If the type is set to **SEMISTATIC**, it will be started from the beginning each time the power is turned on, and modules specified in the system parameters will be reloaded if the module file is newer than the loaded module.

Tasks of type **NORMAL** will not be started at startup. They are started in the normal way, for example, from the FlexPendant.

2.20.9 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 9)

Round robin chain 1: tasks 1, 2, and 9 are busy

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

Round robin chain 3: tasks 3, 5 and 6 are busy
tasks 1, 2, 9 and 10 are idle.

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

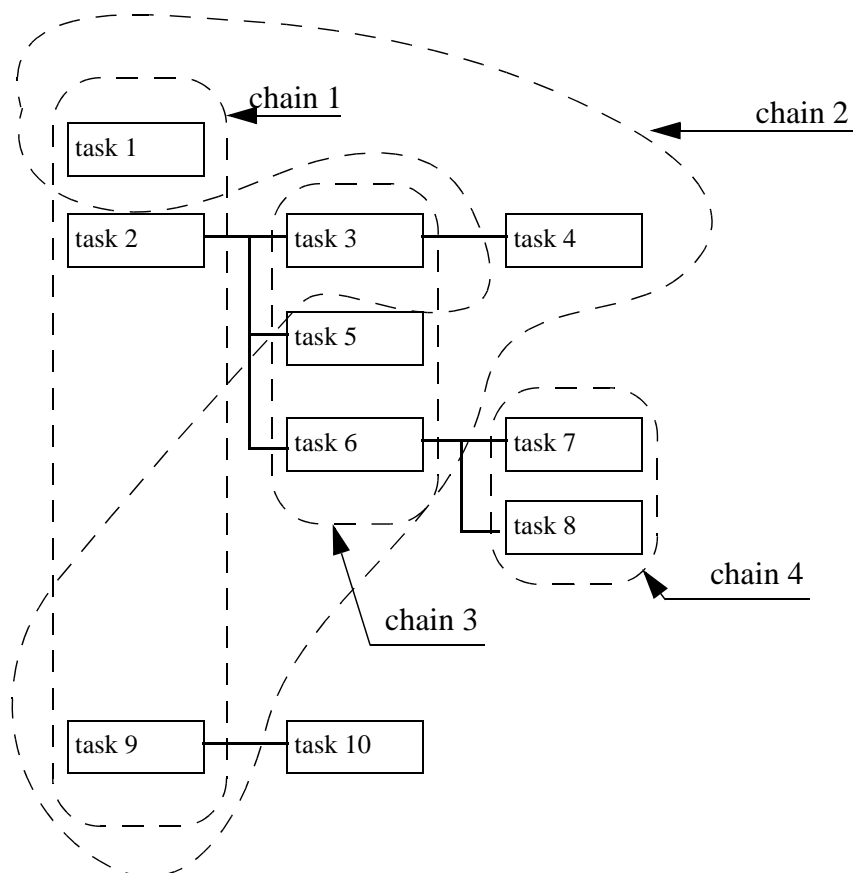


Figure 9 The tasks can have different priorities.

2.20.10 TrustLevel

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

SysFail - This is the default behavior, all other **NORMAL** tasks will also stop, and the system is set to state **SYS_FAIL**. All jog 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. The system is forced to “motors off”. When taking up the system to “motors on” it is possible to jog 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, but can be restarted. Jogging is also possible.

NoSafety - Only the actual task itself will stop.

See *Technical reference manual - System parameters - Controller/Task*

2.20.11 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 FlexPendant 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.

2.21 Backward execution

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

- It is not possible to step backwards out of a IF, FOR, WHILE and TEST statement.
- It is not possible to step backwards out of a routine when reaching the beginning of the routine.
- Motion settings instructions, and some other instructions affecting the motion, cannot be executed backwards. If attempting to execute such an instruction a warning will be written in the event log.

2.21.1 Backward handlers

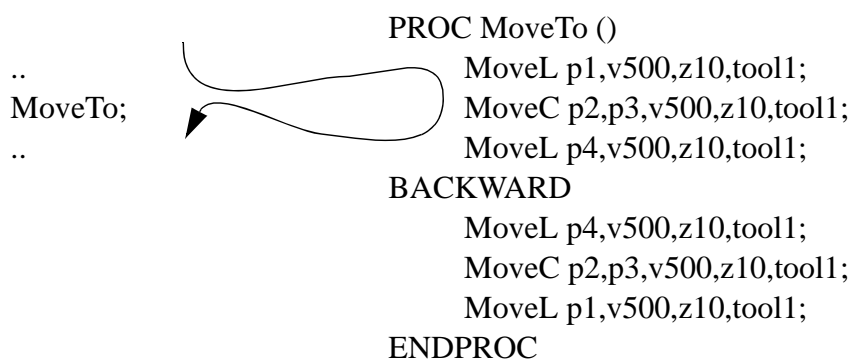
Procedures may contain a backward handler that defines the backward execution of a procedure call. If calling a routine inside the backward handler, the routine is executed forward.

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.

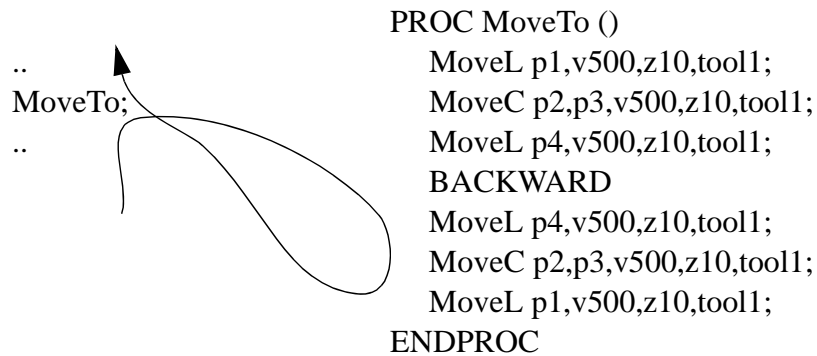
Example:

```
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, that is two instructions in a call chain may not simultaneously be executed backwards.

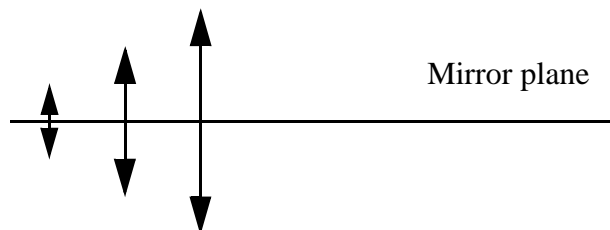
A procedure with no backward handler cannot be executed backwards. A procedure with an empty backward handler is executed as “no operation”.

2.21.2 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:

```

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
  
```



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 additional 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.

2.21.3 Behavior of the backward execution

2.21.3.1 MoveC and nostepin routines

When stepping forward through a MoveC instruction, the robot stops at the circular point (the instruction is executed in two steps). However, when stepping backwards through a MoveC instruction, the robot does not stop at the circular point (the instruction is executed in one step).

It is not allowed to change from forward to backward execution when the robot is executing a MoveC instruction.

It is not allowed to change from forward to backward execution, or vice versa, in a nostepin routine.

2.21.3.2 Target, movement type and speed

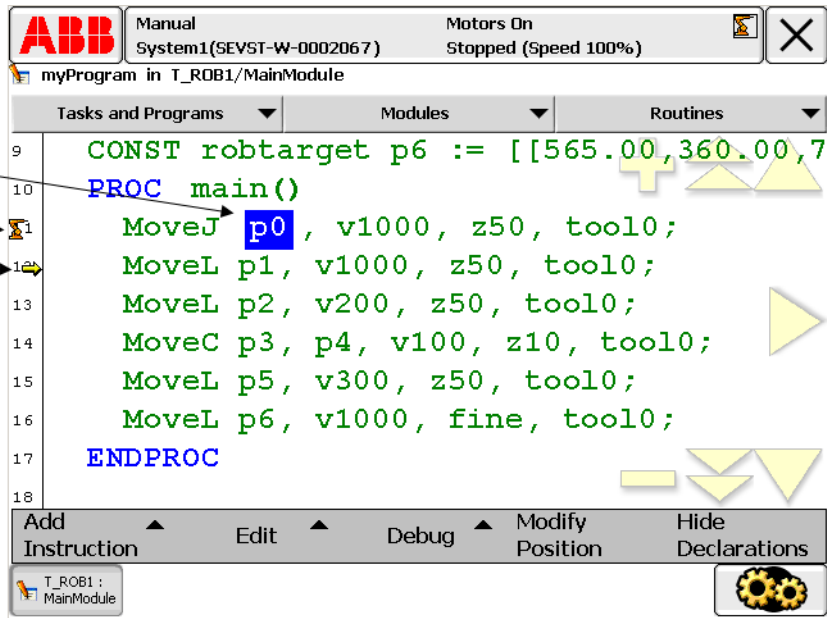
When stepping forward through the program code, a program pointer indicates the next instruction to execute and a motion pointer indicates the move instruction that the robot is performing.

When stepping backward through the program code, the program pointer indicates the instruction above the motion pointer. When the program pointer indicates one move instruction and the motion pointer indicates another, the next backward movement will move to the target indicated by the program pointer, using the movement type and speed indicated by the motion pointer.

An exception, in terms of backward execution speed, is the instruction MoveExtJ. This instruction uses the speed related to the robtarget for both forward and backward execution.

2.21.3.3 Example

This example illustrates the behavior when stepping backwards through move instructions. The program pointer and motion pointer helps you keep track of where the RAPID execution is and where the robot is.



The screenshot shows the ABB Robot Studio interface. At the top, there's a status bar with 'ABB' logo, 'Manual System1(SEVST-W-0002067)', and 'Motors On Stopped (Speed 100%)'. Below this is a menu bar with 'Tasks and Programs', 'Modules', and 'Routines'. The main area displays a RAPID program with line numbers 9 to 18. The program code is as follows:

```

9  CONST robtarget p6 := [[565.00,360.00,7
10 PROC main()
11   MoveJ p0, v1000, z50, tool0;
12   MoveL p1, v1000, z50, tool0;
13   MoveL p2, v200, z50, tool0;
14   MoveC p3, p4, v100, z10, tool0;
15   MoveL p5, v300, z50, tool0;
16   MoveL p6, v1000, fine, tool0;
17 ENDPROC
18

```

On the left side, there are three pointers labeled A, B, and C. Pointer A is at line 11, pointing to the 'MoveJ' instruction. Pointer B is at line 12, pointing to the 'MoveL p1' instruction. Pointer C is at line 10, pointing to the 'PROC main()' instruction. The 'robtarget p6' is highlighted in green. At the bottom, there's a toolbar with buttons: 'Add Instruction', 'Edit', 'Debug', 'Modify Position', and 'Hide Declarations'. Below the toolbar, there's a small window showing 'T_ROB1 : MainModule' and a gear icon.

A	Program pointer
B	Motion pointer
C	Highlighting of the robtarget that the robot is moving towards, or already have reached.

- 1 The program is stepped forward until the robot is in p5. The motion pointer will indicate p5 and the program pointer will indicate the next move instruction (MoveL p6).
- 2 The first press of the BWD button will not move the robot, but the program pointer will move to the previous instruction (MoveC p3, p4). This indicates that this is the instruction that will be executed the next time BWD is pressed.
- 3 The second press of the BWD button will move the robot to p4 linearly with the speed v300. The target for this movement (p4) is taken from the MoveC instruction. The type of movement (linear) and the speed are taken from the instruction below (MoveL p5). The motion pointer will indicate p4 and the program pointer will move up to MoveL p2.
- 4 The third press of the BWD button will move the robot circularly, via p3, to p2 with the speed v100. The target p2 is taken from the instruction MoveL p2. The type of movement (circular), the circular point (p3) and the speed are taken from the MoveC instruction. The motion pointer will indicate p2 and the program pointer will move up to MoveL p1.
- 5 The fourth press of the BWD button will move the robot linearly to p1 with the speed v200. The motion pointer will indicate p1 and the program pointer will move

up to MoveJ p0.

- 6 The first press of the FWD button will not move the robot, but the program pointer will move to the next instruction (MoveL p2).
- 7 The second press of the FWD button will move the robot to p2 with the speed v200.

2.22 Syntax summary

2.22.1 Instructions

Data := Value

AccSet Acc Ramp

ActUnit MecUnit

Add Name | Dname AddValue | AddDvalue

AliasIO FromSignal ToSignal

ArcRefresh

ArcKill

BitClear BitData BitPos

BitSet BitData BitPos

BookErrNo ErrorName

Break

CallByVar Name Number

CancelLoad LoadNo

CheckProgRef

CirPathMode [\PathFrame] | [\ObjectFrame] | [\CirPointOri] |
[\Wrist45] | [\Wrist46] | [\Wrist56]

Clear Name | Dname

ClearIOBuff IODevice

ClearPath

ClearRawBytes RawData [\FromIndex]

ClkReset Clock

ClkStart Clock

ClkStop Clock

Close IODevice

CloseDir Dev

! Comment

ConfJ [\On] | [\Off]

ConfL [\On] | [\Off]

CONNECT Interrupt **WITH** Trap routine

CopyFile OldPath NewPath

CopyRawBytes FromRawData FromIndex ToRawData ToIndex
[\NoOfBytes]

CorrClear

CorrCon Descr

CorrDiscon Descr

CorrWrite

CorrWrite Descr Data

CorrClear

DeactUnit MecUnit

Decr Name | Dname

DitherAct [\MechUnit] Axis [\Level]

DitherDeact

DropSensor Mecunt

DropWObj WObj

EOffsOff

EOffsOn [\ExeP] ProgPoint

EOffsSet EAxOffs

EraseModule ModuleName

ErrLog ErrorID [\W][\I] Argument1 Argument2 Argument3
Argument4 Argument5

ErrRaise ErrorName ErrorID Argument1 Argument2
Argument3 Argument4 Argument5

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

Exit

ExitCycle

FOR Loopcounter **FROM** Startvalue **TO** Endvalue
[**STEP** Stepvalue] **DO ... ENDFOR**

GetDataVal Object [\Block] | [\TaskRef] | [\TaskName] Value

GetSysData DestObject [\ObjectName]

GetTrapData TrapEvent

GOTO Label

GripLoad Load

IDelete Interrupt

IDisable

IEnable

IError ErrorDomain [\ErrorId] ErrorType Interrupt

IF Condition ...

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

Incr Name | Dname

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

IndCMove MecUnit Axis Speed [\Ramp]

IndCnvAddObject MecUnit

IndCnvInit MecUnit Signal Data

IndCnvReset MecUnit

IndCnvEnable MecUnit

IndCnvDisable MecUnit

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

IOBusStart BusName

IOBusState BusName State [\Phys] | [\Logic]

IODisable UnitName MaxTime

IOEnable UnitName MaxTime

IPers Name Interrupt

IRMQMessage InterruptDataType Interrupt

ISignalAI [\Single] | [\SingleSafe] Signal Condition HighValue
LowValue DeltaValue [\DPos] | [\DNeg] Interrupt

ISignalAO [\Single] | [\SingleSafe] Signal Condition HighValue
LowValue DeltaValue [\DPos] | [\DNeg] Interrupt

ISignalDI [\Single] | [\SingleSafe] Signal TriggValue Interrupt

ISignalDO [\Single] | [\SingleSafe] Signal TriggValue Interrupt

ISignalGI [\Single] | [\SingleSafe] Signal Interrupt

ISignalGO [\Single] | [\SingleSafe] Signal Interrupt

ISleep Interrupt

ITimer [\Single] | [\SingleSafe] Time Interrupt

IVarValue VarNo Value Interrupt

IWatch Interrupt ParIdType LoadIdType Tool [\PayLoad]
[\WObj] [\ConfAngle] [\SlowTest] [\Accuracy]

Load [\Dynamic] FilePath [\File] [\CheckRef]

LoadId ParIdType LoadIdType Tool [\PayLoad] [\WObj]
[\ConfAngle] [\SlowTest] [\Accuracy]

MakeDir Path

ManLoadIdProc [\ParIdType] [\MechUnit] [\MechUnitName]
[\AxisNumber] [\PayLoad] [\ConfigAngle] [\DeactAll]
[\AlreadyActive] [\DefinedFlag]

MechUnitLoad MechUnit AxisNo Load

MotionSup [\On] | [\Off] [\TuneValue]

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

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

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

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

MoveExtJ [\Conc] ToJointPos [\ID] Speed [\T] Zone [\Inpos]

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

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

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

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

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

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

MToolRotCalib RefTip ZPos [\XPos] Tool

MToolTCPCalib Pos1 Pos2 Pos3 Pos4 Tool MaxErr MeanErr

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

OpenDir Dev Path

PackDNHeader Service Path RawData

PackRawBytes Value RawData [\Network] StartIndex
[\Hex1] | [\IntX] | [\Float4] | [\ASCII]

PathAccLim AccLim [\AccMax] DecelLim [\DecelMax]

PathRecMoveBwd [\ID] [\ToolOffs] [\Speed]

PathRecMoveFwd [\ID] [\ToolOffs] [\Speed]

PathRecStart ID

PathRecStop [\Clear]

PathResol Value

PDispOff

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

PDispSet DispFrame

Procedure { Argument }

ProcerrRecovery [\SyncOrgMoveInst] | [\SyncLastMoveInst]
[\ProcSignal]

PulseDO [\PLength] Signal

RAISE [Error no]

RaiseToUser [\Continue] | [\BreakOff] [\ErrorNumber]

ReadAnyBin IODevice Data [\Time]

ReadCfgData InstancePath Attribute CfgData [\ListNo]

ReadErrData TrapEvent ErrorDomain ErrorId ErrorType [\Str1]
[\Str2] [\Str3] [\Str4] [\Str5]

ReadRawBytes IODevice RawData NoOfBytes[\Time]

RemoveDir Path

RemoveFile Path

RenameFile OldPath NewPath

Reset Signal

ResetPPMoved

ResetRetryCount

RestoPath

RETURN [Return value]

Rewind IODevice

RMQEmptyQueue

RMQFindSlot Slot Name

RMQGetMessage Message

RMQGetMsgData Message Data

RMQGetMsgHeader Message [\Header] [\SenderId] [\UserDef]

RMQReadWait Message [\TimeOut]

RMQSendMessage Slot SendData [\UserDef]

RMQSendWait Slot SendData [\UserDef] Message
ReceiveDataType [\TimeOut]

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

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

SearchExtJ [\Stop] | [\PStop] | [\SStop] [\Sup] Signal
SearchJointPos ToJointPos [\ID] Speed | [\T]

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

Set Signal

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

SetAO Signal Value

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

SetDataVal Object [\Block] | [\TaskRef] | [\TaskName] Value

SetDO [\SDelay] Signal Value

SetGO [\SDelay] Signal Value | Dvalue

SetSysData SourceObject [\ObjectName]

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

SkipWarn

SocketAccept Socket ClientSocket [\ClientAddress] [\Time]

SocketBind Socket LocalAddress LocalPort

SocketClose Socket

SocketConnect Socket Address Port [\Time]

SocketCreate Socket

SocketListen Socket

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

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

SoftAct Axis Softness [\Ramp]

SoftDeact [\Ramp]

SpeedRefresh Override

SpotJ

SpotL ToPoint Speed Spot [\InPos] [\NoConc] [\Retract] Gun
Tool [\WObj]

SpotML

SpyStart File

SpyStop

StartLoad [\Dynamic] FilePath [\File] LoadNo

StartMove [\AllMotionTasks]

StartMoveRetry

StepBwdPath StepLength StepTime

SToolRotCalib RefTip ZPos XPos Tool

SToolTCPCalib Pos1 Pos2 Pos3 Pos4 Tool MaxErr MeanErr

Stop [\NoRegain] | [\AllMoveTasks]

StopMove [\Quick] [\AllMotionTasks]

StopMoveReset [\AllMotionTasks]

StorePath [\KeepSync]

SyncMoveOff SyncID [\TimeOut]

SyncMoveOn SyncID TaskList [\TimeOut]

SyncMoveSuspend

SyncMoveResume

SyncMoveUndo

SyncToSensor Mecunt [On/Off]

SystemStopAction [\Stop] [\StopBlock] [\Halt]

TasksInSync TaskList

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

ENDTEST

TestSignDefine Channel SignalId MechUnit Axis SampleTime

TestSignReset

TextTabInstall File

TPErase

TPReadDnum Answer String [\MaxTime] [\DIBreak]
[\DOBreak] [\BreakFlag]

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

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

TPShow Window

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

TriggC CirPoint ToPoint [\ID] Speed [\T] Trigg_1 [\T2] [\T3]
[\T4] [\T5] [\T6] [\T7] [\T8] Zone Tool [\WObj]

TriggCheckIO TriggData Distance [\Start] | [\Time] Signal
Relation CheckValue | CheckDvalue [\StopMove] Interrupt

TriggEquip TriggData Distance [\Start] Equiplag [\DOp] | [\GOp] |
[\AOp] [\ProcID] SetValue | SetDvalue [\Inhib]

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

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

TriggJ ToPoint [\ID] Speed [\T] Trigg_1 [\T2] [\T3] [\T4]
[\T5] [\T6] [\T7] [\T8] Zone Tool [\WObj]

TriggL ToPoint [\ID] Speed [\T] Trigg_1 [\T2] [\T3] [\T4]
[\T5] [\T6] [\T7] [\T8] Zone Tool [\WObj]

TriggLIOs ToPoint [\ID] Speed [\T] [\TriggData1] [\TriggData2]
[\TriggData3] Zone Tool [\WObj]

TriggRampAO TriggData Distance [\Start] EquipLag AOutput **SetValue**
RampLength [\Time]

TriggSpeed TriggData Distance [\Start] ScaleLag AO
ScaleValue [\DipLag] [\ErrDO] [\Inhib]

TriggStopProc RestartRef [\DO1] [\GO1] [\GO2] [\GO3] [\GO4]
ShadowDO

TryInt DataObj | DataObj2

TuneReset

TuneServo MecUnit Axis TuneValue [\Type]

UIMsgBox [\Header] MsgLine1 [\MsgLine2] [\MsgLine3]
[\MsgLine4] [\MsgLine5] [\Wrap] [\Buttons] [\Icon] [\Image]
[\Result] [\MaxTime] [\DIBreak] [\DOBreak] [\BreakFlag]

UIMShow AssemblyName TypeName [\InitCmd] [\InstanceId]
[\Status] [\NoCloseBtn]

UnLoad [\ErrIfChanged] | [\Save] FilePath [\File]

UnpackRawBytes RawData [\Network] StartIndex Value
[\Hex1] | [\IntX] | [\Float4] | [\ASCII]

WaitAI Signal [\LT] | [\GT] Value [\MaxTime] [\ValueAtTimeout]

WaitAO Signal [\LT] | [\GT] Value [\MaxTime] [\ValueAtTimeout]

WaitDI Signal Value [\MaxTime] [\TimeFlag]

WaitDO Signal Value [\MaxTime] [\TimeFlag]

WaitGI Signal [\NOTEQ] | [\LT] | [\GT] Value | Dvalue [\MaxTime]
[\ValueAtTimeout] | [\DvalueAtTimeout]

WaitGO Signal [\NOTEQ] | [\LT] | [\GT] Value | Dvalue [\MaxTime]
[\ValueAtTimeout] | [\DvalueAtTimeout]

WaitLoad [\UnloadPath] [\UnloadFile] LoadNo [\CheckRef]

WaitRob [\InPos] | [\ZeroSpeed]

WaitSensor Mecunt[\RelDist] [\PredTime] [\MaxTime]
[\TimeFlag]

WaitSyncTask SyncID TaskList [\TimeOut]

WaitTime [\InPos] Time

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

WaitWObj WObj [\RelDist]

WarmStart

VelSet Override Max

WHILE Condition **DO** ...
ENDWHILE

WorldAccLim [\On] | [\Off]

Write IODeviceString [\Num] | [\Bool] | [\Pos] | [\Orient] | [\Dnum]
[\NoNewLine]

WriteAnyBin IODevice Data

WriteBin IODevice Buffer NChar

WriteCfgData InstancePath Attribute CfgData [\ListNo]

WriteRawBytes IODevice RawData [\NoOfBytes]

WriteStrBin IODevice Str

WZBoxDef [\Inside] | [\Outside] Shape LowPoint HighPoint

WZCylDef [\Inside] | [\Outside] Shape CentrePoint Radius Height

WZDisable WorldZone

WZDOSet [\Temp] | [\Stat] WorldZone [\Inside] | [\Before] Shape
Signal SetValue

WZEnable WorldZone

WZFree WorldZone

WZHomeJointDef [\Inside] | [\Outside] Shape MiddleJointVal
DeltaJointVal

WZLimJointDef [\Inside] | [\Outside] Shape LowJointVal
HighJointVal

WZLimSup [\Temp] | [\Stat] WorldZone Shape

WZSphDef [\Inside] | [\Outside] Shape CentrePoint Radius

2.22.2 Functions

Abs (Input)

ACos (Value)

AOutput (Signal)

ArgName (Parameter)

ASin (Value)

ATan (Value)

ATan2 (Y X)

BitAnd (BitData1 BitData2)

BitCheck (BitData BitPos)

BitLSh (BitData ShiftSteps)

BitNeg (BitData1)

BitOr (BitData1 BitData2)

BitRSh (BitData1 ShiftSteps)

BitXOr (BitData1 BitData2)

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

CalcJointT (Rob_target Tool [\WObj])

CalcRobT (Joint_target Tool [\WObj])

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

CalcRotAxFrameZ (TargetList TargetsInList PositiveZPoint
MaxErr MeanErr)

CDate

CJointT

ClkRead (Clock)

CorrRead

Cos (Angle)

CPos ([\Tool] [\WObj])

CRobT ([\Tool] [\WObj])

CSpeedOverride ([\CTask])

CTime

CTool

CWObj

DecToHex (Str)

DefAccFrame (TargetListOne TargetListTwo TargetsInList
MaxErr MeanErr)

DefDFrame (OldP1 OldP2 OldP3 NewP1 NewP2 NewP3)

DefFrame (NewP1 NewP2 NewP3 [\Origin])

Dim (ArrPar DimNo)

Distance (Point1 Point2)

DnumToNum (Value [\Integer])

DOutput (Signal)

DotProd (Vector1 Vector2)

EventType

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

ExecHandler

ExecLevel

Exp (Exponent)

FileSize (Path)

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

FSSize (Name [\Total] | [\Free] [\Kbyte] [\Mbyte])

GetMecUnitName (MechUnit)

GetNextMechUnit (ListNumber UnitName) [\MecRef]
[\TCPro] [\NoOfAxes] [\MecTaskNo] [\MotPlanNo] [\Active]

GetNextSym (Object Block [\Recursive])

GetSysInfo ([\SerialNo] | [\SWVersion] | [\RobotType] | [\CtrlId] |
[\LanIp] | [\CtrlLang])

GetTaskName ([\TaskNo])

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

GOutput (Signal)

GOutputDnum (Signal)

GInputDnum (Signal)

HexToDec (Str)

IndInpos (MechUnit Axis)

IndSpeed (MechUnit Axis [\InSpeed] | [\ZeroSpeed])

IOUnitState (UnitName [\Phys] | [\Logic])

IsFile (Path[\Directory] [\Fifo] [\RegFile] [\BlockSpec]
[\CharSpec])

IsMechUnitActive (MechUnit)

IsPers (DatObj)

IsStopMoveAct ([\FromMoveTask] | [\FromNonMoveTask])

IsStopStateEvent ([\PPMoved] | [\PPToMain])

IsSyncMoveOn

IsSysId (SystemId)

IsVar (DatObj)
MaxRobSpeed
MirPos (Point MirPlane [\WObj] [\MirY])
ModExist (ModuleName)
ModTime (Object [\StrDig])
MotionPlannerNo
NonMotionMode ([\Main])
NOrient (Rotation)
NumToDnum (Value)
NumToStr (Val | Dval Dec [\Exp])
Offs (Point XOffset YOffset ZOffset)
OpMode
OrientZYX (ZAngle YAngle XAngle)
ORobT (OrgPoint [\InPDisp] | [\InEOffs])
ParIdPosValid (ParIdType Pos AxValid [\ConfAngle])
ParIdRobValid (ParIdType)
PathLevel ()
PathRecValidBwd ([\ID])
PathRecValidFwd ([\ID])
PFRestart ([\Base] | [\Irpt])
PoseInv (Pose)
PoseMult (Pose1 Pose2)
PoseVect (Pose Pos)
Pow (Base Exponent)

PPMovedInManMode

Present (OptPar)

ProgMemFree

RawBytesLen (RawData)

ReadBin (IODevice [\Time])

ReadDir (Dev FileName)

ReadMotor ([\MecUnit] Axis)

ReadNum (IODevice [\Time])

ReadStr (IODevice [\Time])

ReadStrBin (IODevice NoOfChars [\Time])

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

RemainingRetries

RMQGetSlotName (Slot)

RobOS

Round (Val [\Dec])

RunMode ([\Main])

Sin (Angle)

SocketGetStatus (Socket)

Sqrt (Value)

StrDigCmp (StrDig1 Relation StrDig2)

StrDigCalc (StrDig1 Operation StrDig2)

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)

TaskRunMec

TaskRunRob

TestAndSet (Object)

TestDI (Signal)

TestSignRead (Channel)

TextGet (Table Index)

TextTabFreeToUse (TableName)

TextTabGet (TableName)

Trunc (Val [\Dec])

Type (Data [\BaseName])

UIAlphaEntry ([\Header] [\Message] | [\MsgArray] [\Wrap] [\Icon]
[\InitString] [\MaxTime] [\DIBreak] [\DOBreak] [\BreakFlag])

UIClientExist

UIDnumEntry ([\Header] [\Message] | [\MsgArray] [\Wrap] [\Icon]
[\InitValue] [\MinValue] [\MaxValue] [\AsInteger] [\MaxTime]
[\DIBreak] [\DOBreak] [\BreakFlag])

UIDnumTune ([\Header] [\Message] | [\MsgArray] [\Wrap] [\Icon]
InitValue Increment [\MinValue] [\MaxValue] [\MaxTime]
[\DIBreak] [\DOBreak] [\BreakFlag])

UICollection ([\Result] [\Header] ListItems [\Buttons] | [\BtnArray]
[\Icon] [\DefaultIndex] [\MaxTime] [\DIBreak] [\DOBreak]
[\BreakFlag]

UIMessageBox ([\Header] [\Message] | [\MsgArray] [\Wrap]
[\Buttons] | [\BtnArray] [\DefaultBtn] [\Icon] [\Image] [\MaxTime]
[\DIBreak] [\DOBreak] [\BreakFlag])

UINumEntry ([\Header] [\Message] | [\MsgArray] [\Wrap] [\Icon]
[\InitValue] [\MinValue] [\MaxValue] [\AsInteger] [\MaxTime]
[\DIBreak] [\DOBreak] [\BreakFlag])

UINumTune ([\Header] [\Message] | [\MsgArray] [\Wrap] [\Icon]
InitValue Increment [\MinValue] [\MaxValue] [\MaxTime]
[\DIBreak] [\DOBreak] [\BreakFlag])

VaidIO (Signal)

ValToStr (Val)

VectMagn (Vector)

3 Motion and I/O programming

3.1 Coordinate systems

3.1.1 The robot's tool center point (TCP)

The position of the robot and its movements are always related to the tool center point. This point is normally defined as being somewhere on the tool, for example in the muzzle of a glue gun, at the center 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 3.1.3.3 *Stationary TCPs* on page 162.

3.1.2 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.

3.1.2.1 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 10).

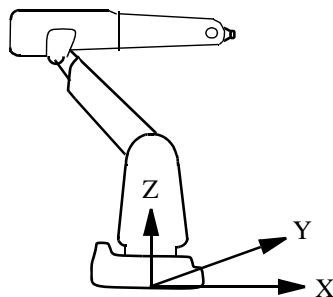


Figure 10 The base coordinate system.

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.

3.1.2.2 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 11.

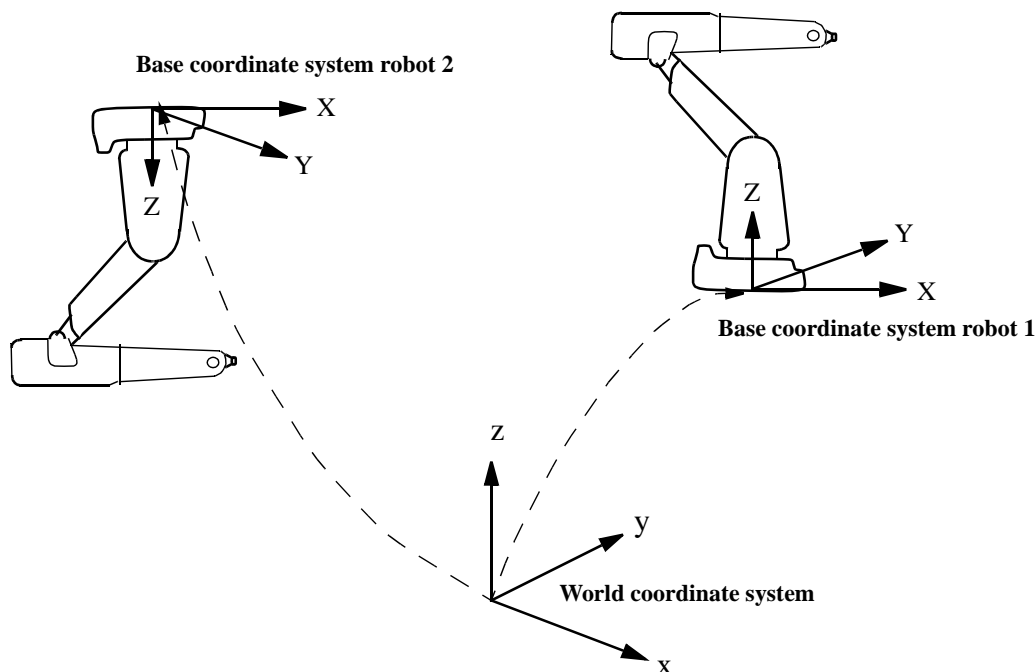


Figure 11 Two robots (one of which is suspended) with a common world coordinate system.

3.1.2.3 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 8).

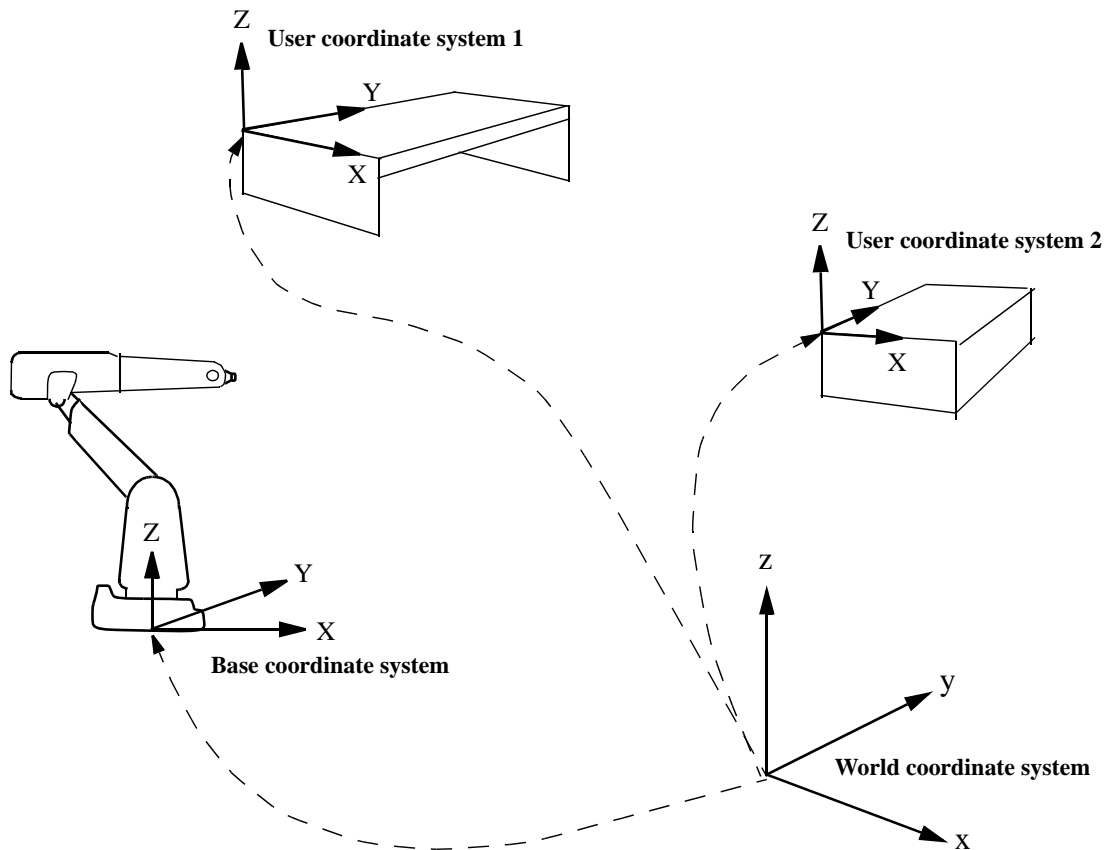


Figure 12 Two user coordinate systems describe the position of two different fixtures.

3.1.2.4 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 13).

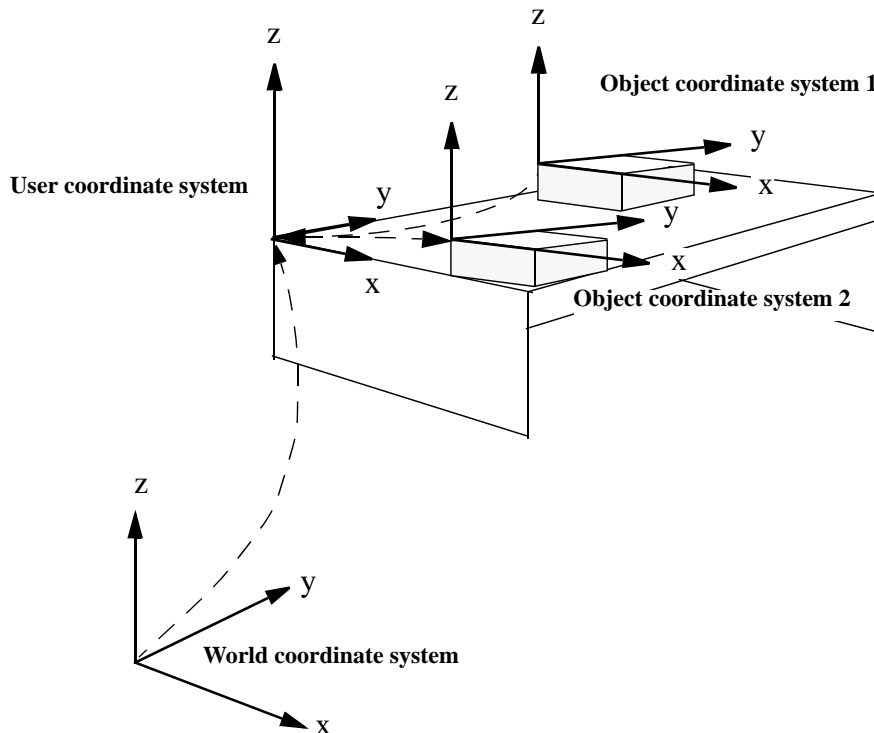


Figure 13 Two object coordinate systems describe the position of two different work objects located in the same fixture.

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 additional 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.

3.1.2.5 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 14).

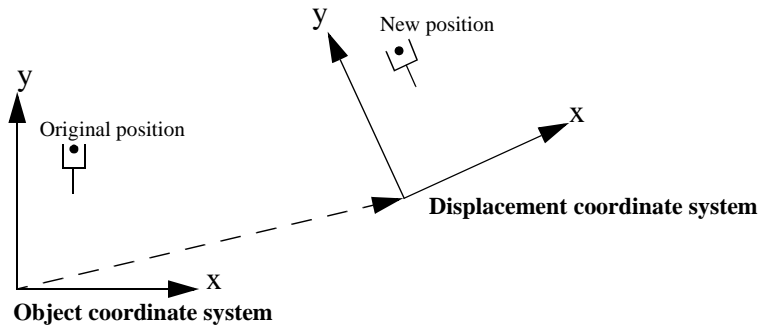


Figure 14 If program displacement is active, all positions are displaced.

3.1.2.6 Coordinated additional 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 15) and there is no need to consider the fact that the object is moved by the external unit.

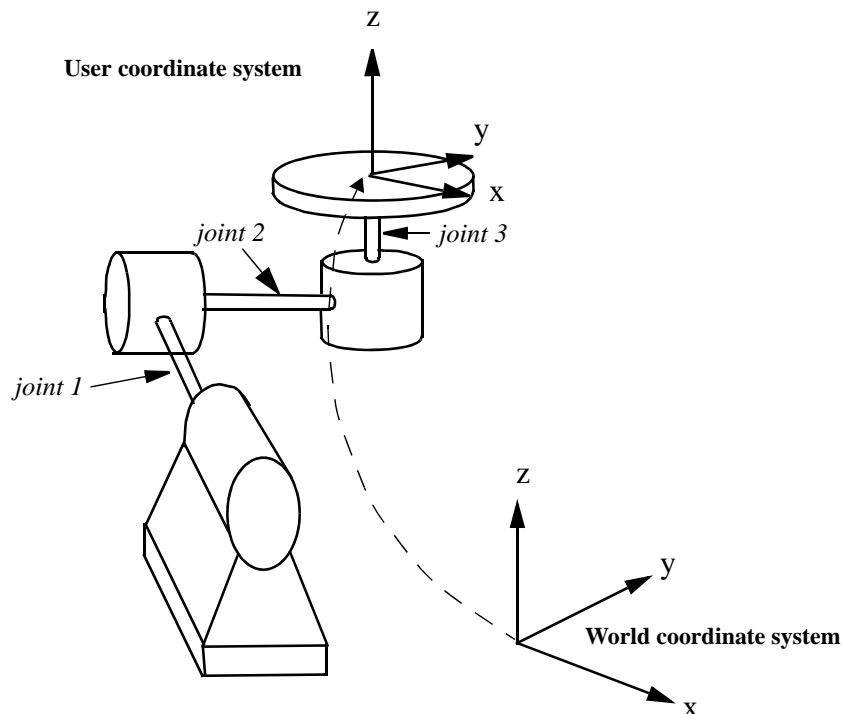


Figure 15 A user coordinate system, defined to follow the movements of a 3-axis external mechanical unit.

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 16) 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.

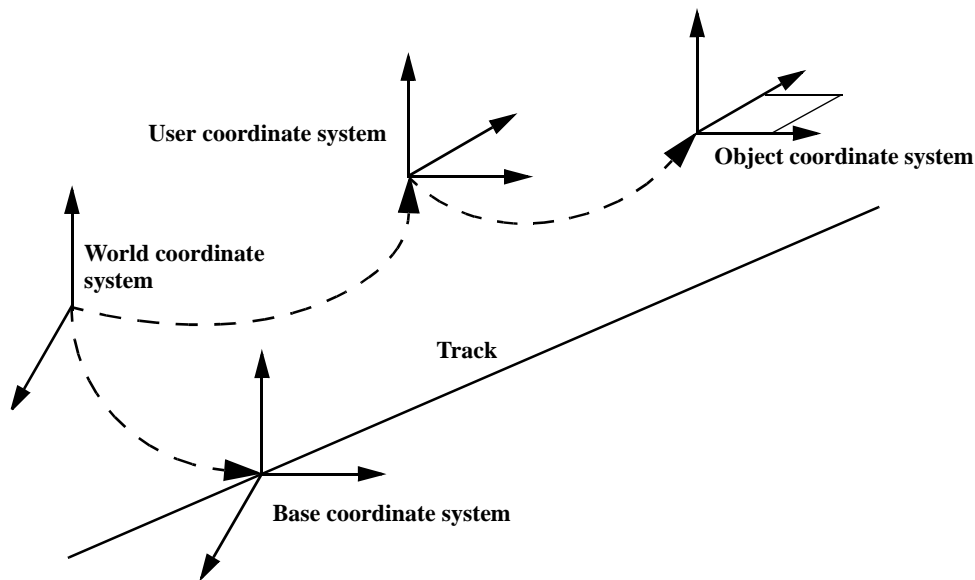


Figure 16 Coordinated interpolation with a track moving the base coordinate system of the robot.

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 additional axes and the translation/rotation of the user and the base coordinate systems.

These relations are defined in the system parameters.

3.1.3 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.

3.1.3.1 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 17).

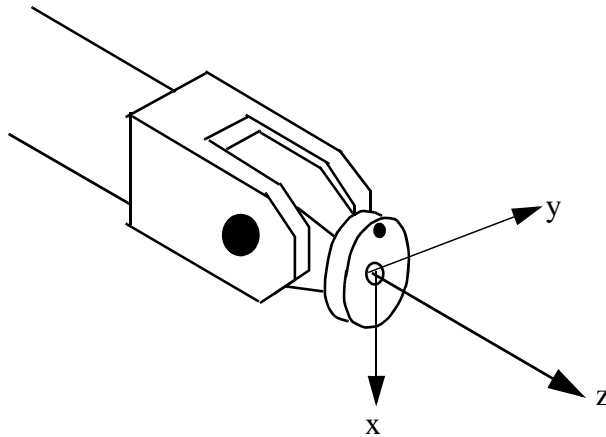


Figure 17 The wrist coordinate system.

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 center 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.

3.1.3.2 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, for example the muzzle on a glue gun. The tool coordinate axes are defined as those natural for the tool in question.

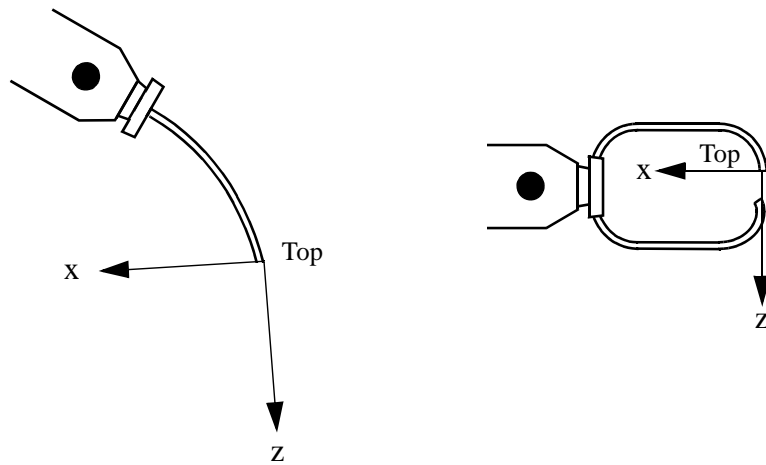


Figure 18 Tool coordinate system, as usually defined for an arc-welding gun (left) and a spot welding gun (right).

The tool coordinate system is defined based on the wrist coordinate system (see Figure 19).

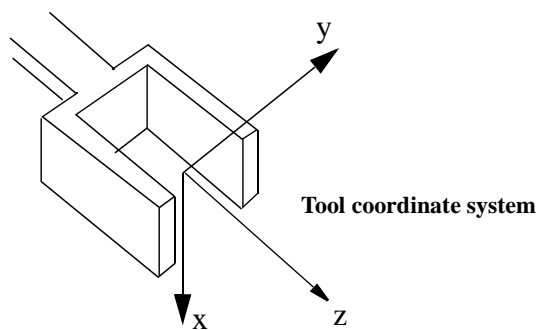


Figure 19 The tool coordinate system is defined relative to the wrist coordinate system, here for a gripper.

3.1.3.3 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 20.

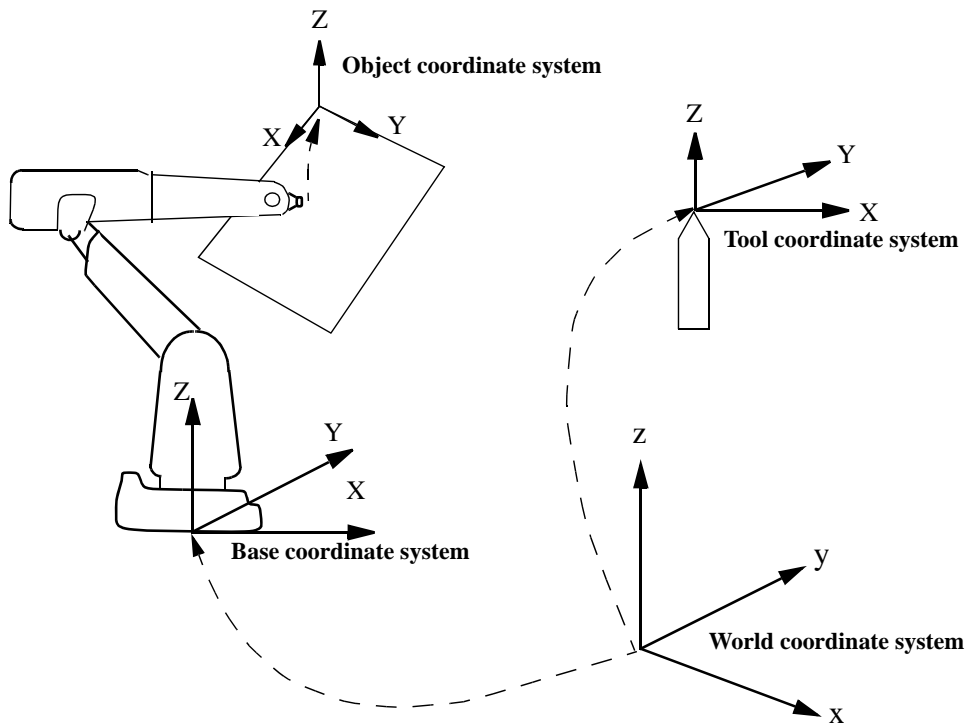


Figure 20 If a stationary TCP is used, the object coordinate system is usually based on the wrist coordinate system.

In the example in Figure 20, 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 21.

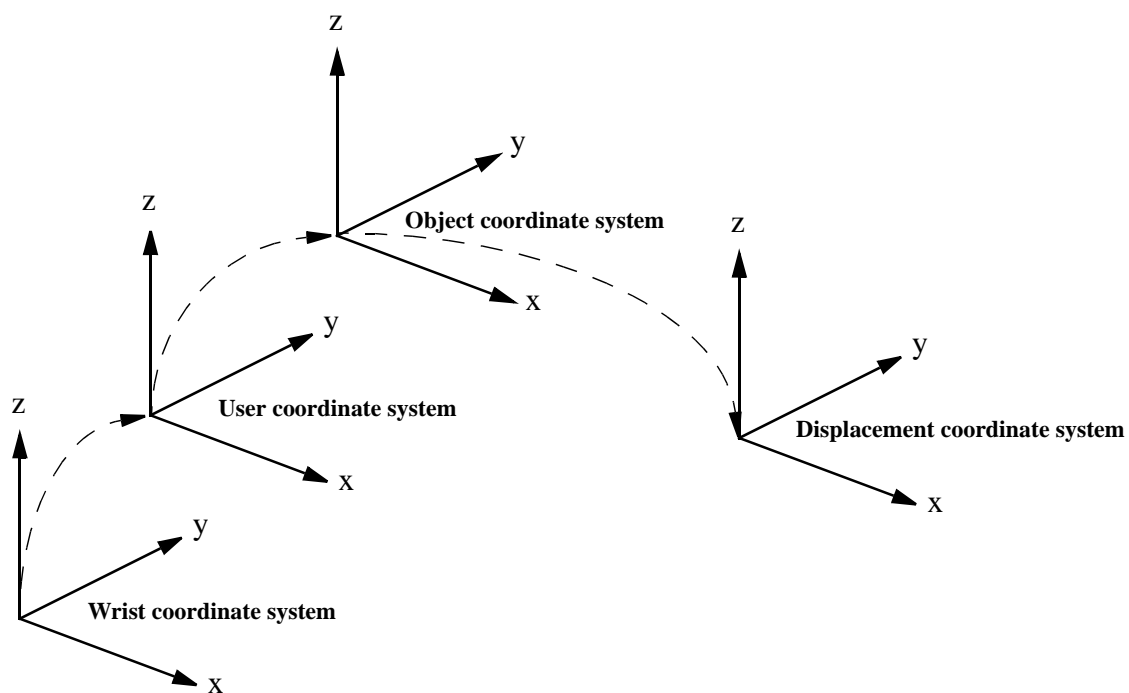


Figure 21 Program displacement can also be used together with stationary TCPs.

3.1.4 Related information

	Described in:
Definition of the world coordinate system	<i>Technical reference manual - System parameters</i>
Definition of the user coordinate system	<i>Operating manual - IRC5 with FlexPendant</i>
Definition of the object coordinate system	<i>Operating manual - IRC5 with FlexPendant</i>
Definition of the tool coordinate system	<i>Operating manual - IRC5 with FlexPendant</i>
Definition of a tool center point	<i>Operating manual - IRC5 with FlexPendant</i>
Definition of displacement frame	<i>Operating manual - IRC5 with FlexPendant</i>
Jogging in different coordinate systems	<i>Operating manual - IRC5 with FlexPendant</i>

3.2 Positioning during program execution

3.2.1 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 center point, the orientation of the tool, the configuration of the robot and the position of the additional axes).
- The interpolation method used to reach the destination point, for example joint interpolation, linear interpolation or circle interpolation.
- The velocity of the robot and additional axes.
- The zone data (defines how the robot and the additional 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 additional 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 additional 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.

3.2.2 Interpolation of the position and orientation of the tool

3.2.2.1 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 22).

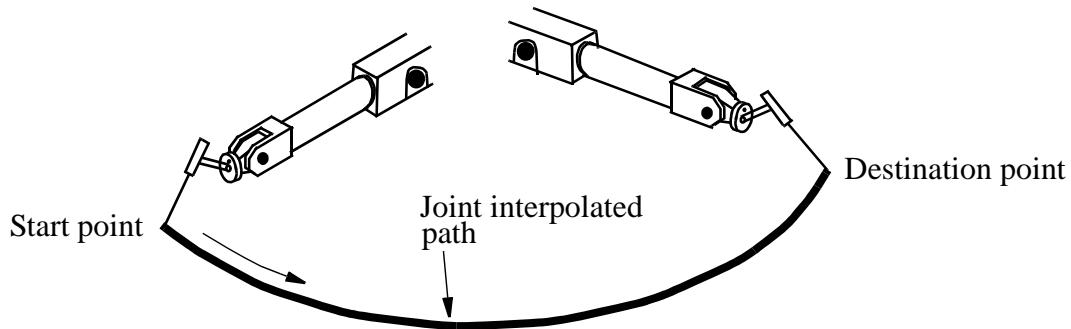


Figure 22 Joint interpolation is often the fastest way to move between two points as the robot axes follow the closest path between the start point and the destination point (from the perspective of the axis angles).

The velocity of the tool center 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, that is 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 optimized to the max performance of the robot.

3.2.2.2 Linear interpolation

During linear interpolation, the TCP travels along a straight line between the start and destination points (see Figure 23).

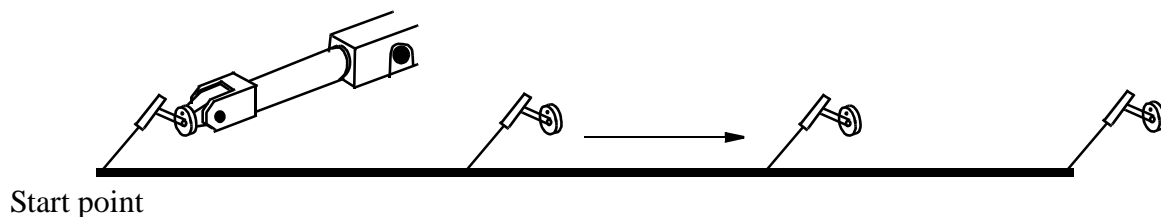


Figure 23 Linear interpolation without reorientation of the tool.

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 (for example 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 center 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 optimized automatically.

3.2.2.3 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 24).

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.

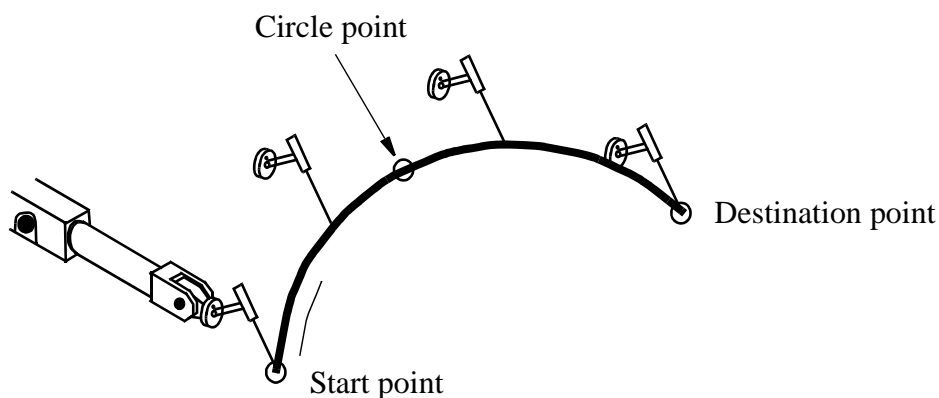


Figure 24 Circular interpolation with a short twist for part of a circle (circle segment) with a start point, circle point and destination point.

However, if the orientation at the support point is programmed closer to the orientation rotated 180° , the alternative twist is selected (see Figure 25).

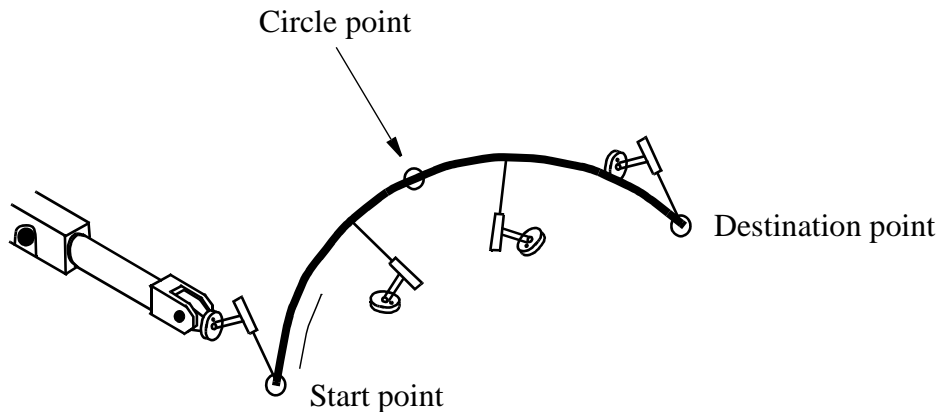


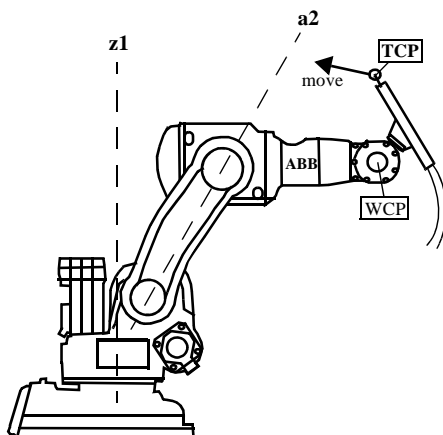
Figure 25 Circular interpolation with a long twist for orientation is achieved by defining the orientation in the circle point in the opposite direction compared to the start point.

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 optimized automatically.

3.2.2.4 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. The resulting orientation in the programmed point may also differ from the programmed orientation due to two singularities (see below).



The first singularity is when TCP is straight ahead from axis 2 (a2 in the figure above). The TCP cannot pass to the other side of axis 2, instead will axis 2 and 3 fold a bit more to keep the TCP on the same side and the end orientation of the move will then be turned away from the programmed orientation with the same size.

The second singularity is when TCP will pass near the z-axis of axis 1 (z1 in the figure above). The axis 1 will in this case turn around with **full speed** and the tool reorientation will follow in the same way. The direction of the turn is dependent of what side the TCP will go. **We recommend to change to joint interpolation (MoveJ) near the z-axis.** Note that it's the TCP that make the singularity not the WCP as when SingArea\Off is used.

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).

3.2.3 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 additional 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 26).

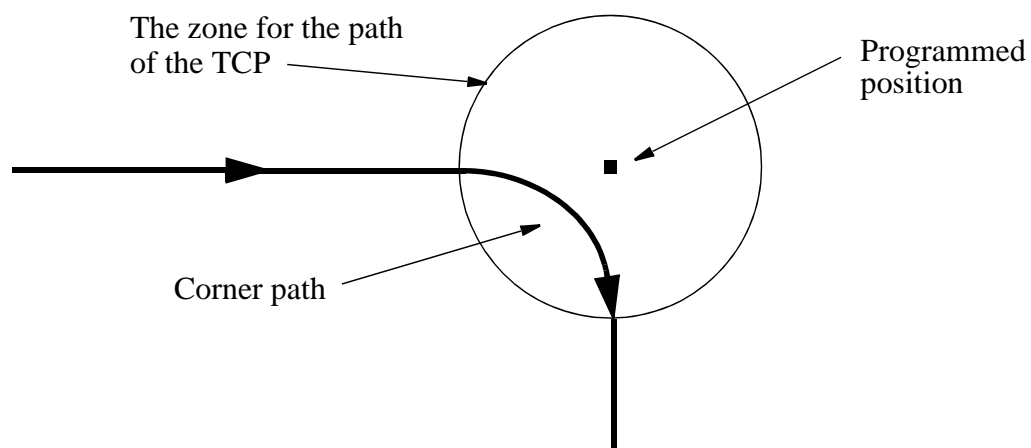


Figure 26 A fly-by point generates a corner path to pass the programmed position.

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

3.2.3.1 Joint interpolation in corner paths

The size of the corner paths (zones) for the TCP movement is expressed in mm (see Figure 27). 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.

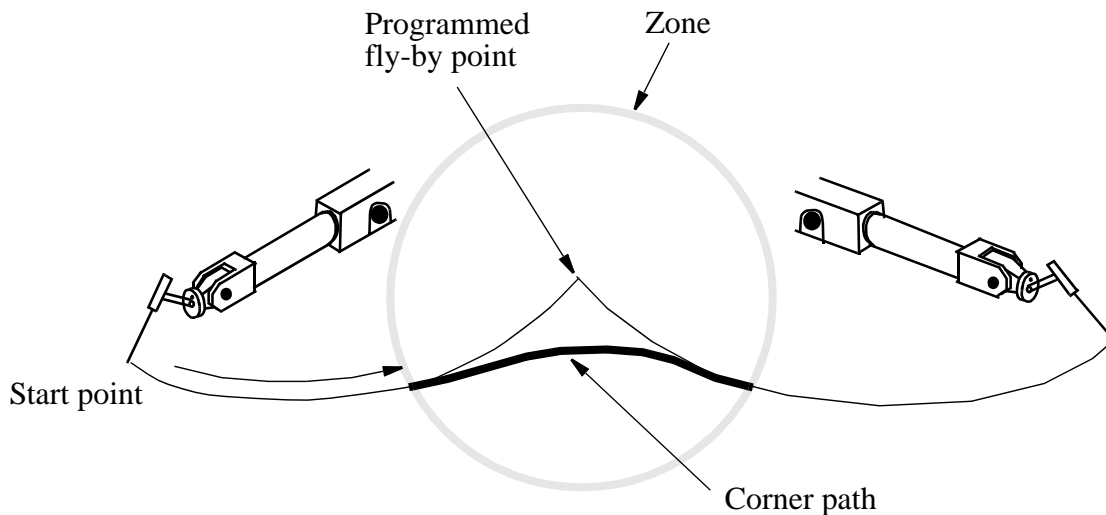


Figure 27 During joint interpolation, a corner path is generated in order to pass a fly-by point.

3.2.3.2 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 28).

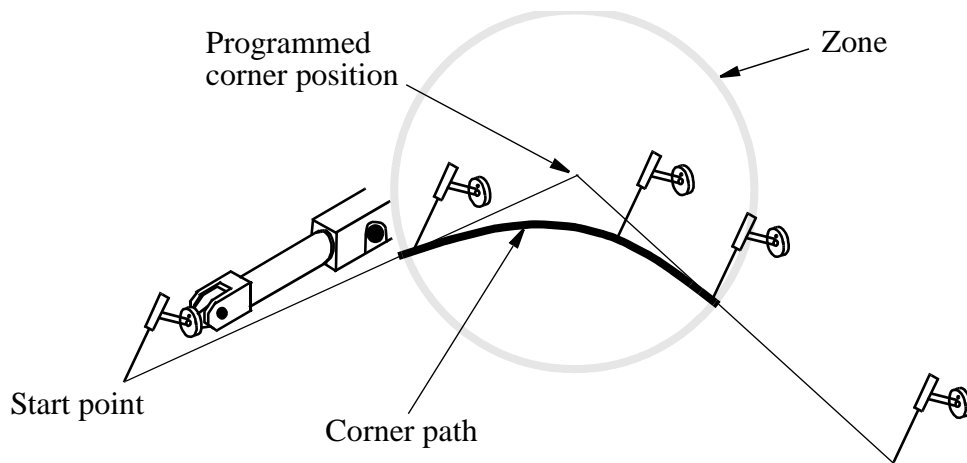


Figure 28 During linear interpolation, a corner path is generated in order to pass a fly-by point.

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.

3.2.3.3 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 29a-c).

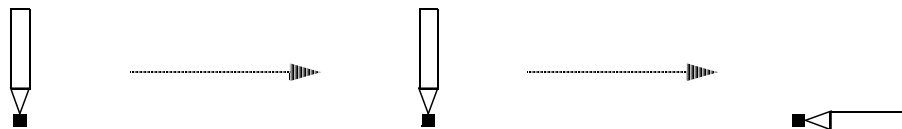


Figure 29a Three positions with different tool orientations are programmed as above.

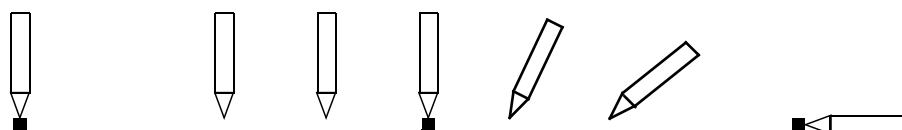


Figure 29b If all positions were stop points, program execution would look like this.

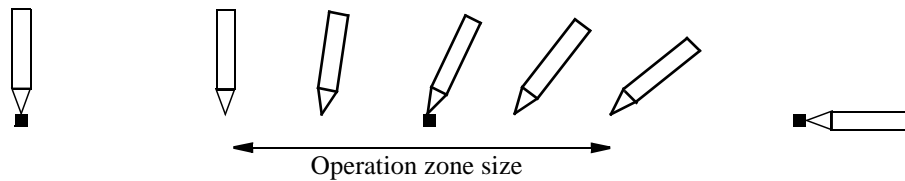


Figure 29c If the middle position was a fly-by point, program execution would look like this.

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.

3.2.3.4 Interpolation of additional axes in corner paths

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

3.2.3.5 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 30).

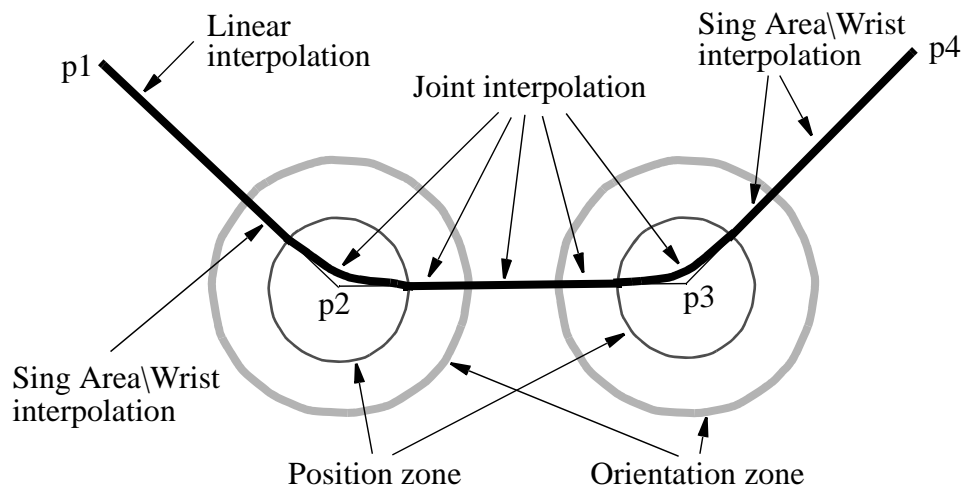


Figure 30 Interpolation when changing from one interpolation method to another. Linear interpolation has been programmed between p1 and p2; joint interpolation between p2 and p3; and Sing Area\Wrist interpolation between p3 and p4.

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.

3.2.3.6 Interpolation when changing coordinate system

When there is a change of coordinate system in a corner path, for example 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.

3.2.3.7 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 31). The same zone radius is always used for inputs to or outputs from a programmed position, in order to obtain symmetrical corner paths.

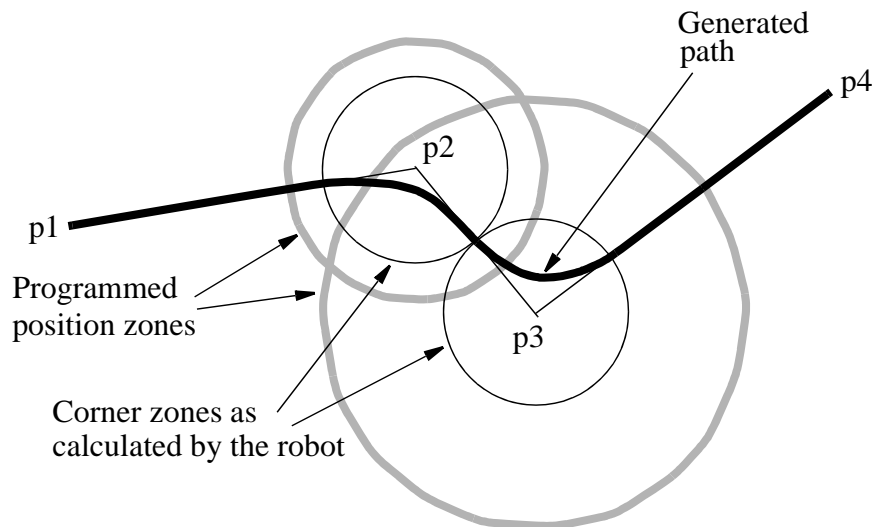


Figure 31 Interpolation with overlapping position zones. The zones around p2 and p3 are larger than half the distance from p2 to p3. Thus, the robot reduces the size of the zones to make them equal to half the distance from p2 to p3, thereby generating symmetrical corner paths within the zones.

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 32).

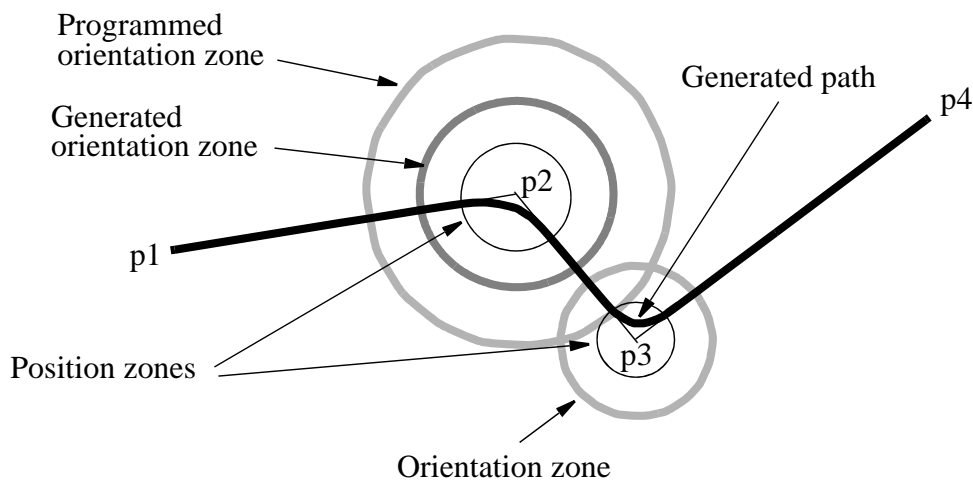


Figure 32 Interpolation with overlapping orientation zones. The orientation zone at p2 is larger than half the distance from p2 to p3 and is thus reduced to half the distance from p2 to p3. The position zones do not overlap and are consequently not reduced; the orientation zone at p3 is not reduced either.

3.2.3.8 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.

3.2.4 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.

3.2.4.1 Program execution

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

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

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

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

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

3.2.4.2 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 33.

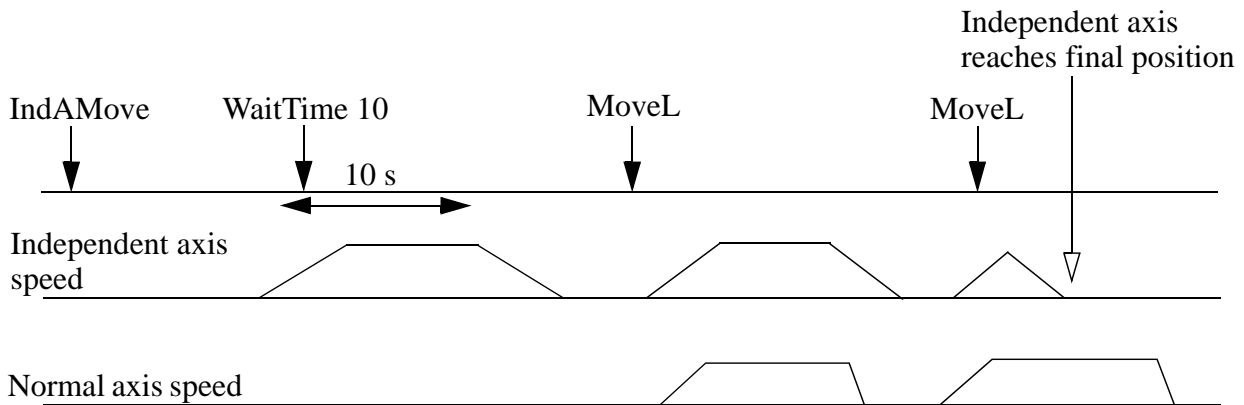


Figure 33 Stepwise execution of independent axes.

3.2.4.3 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.

3.2.4.4 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 34.

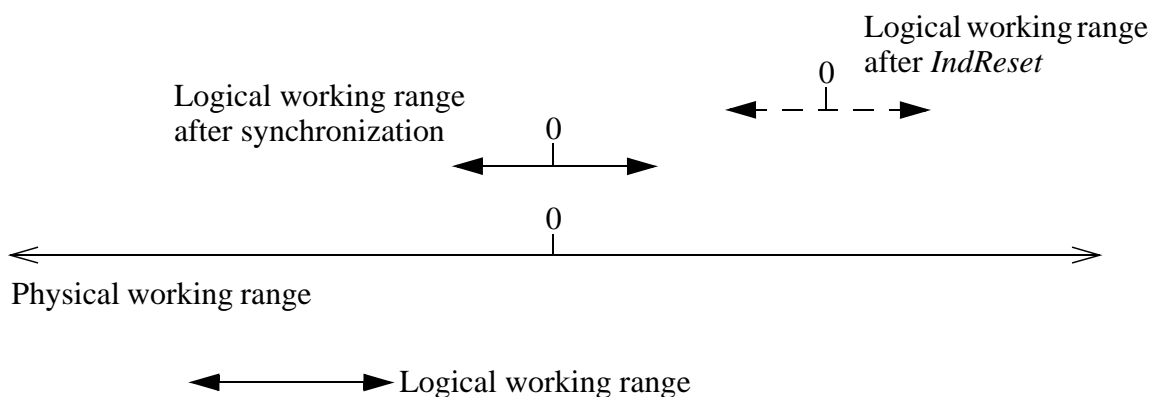


Figure 34 The logical working range can be moved, using the instruction *IndReset*.

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.

3.2.4.5 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.

3.2.4.6 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 FlexPendant for axis 6 is compensated with the positions of axis 4 and 5 via the gear coupling.

3.2.5 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.

3.2.6 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*.

3.2.7 Related information

	Described in:
Definition of speed	<i>Technical reference manual - RAPID Instructions, functions and data types - speeddata</i>
Definition of zones (corner paths)	<i>Technical reference manual - RAPID Instructions, functions and data types - zonedata</i>
Instruction for joint interpolation	<i>Technical reference manual - RAPID Instructions, functions and data types - MoveJ</i>
Instruction for linear interpolation	<i>Technical reference manual - RAPID Instructions, functions and data types - MoveL</i>
Instruction for circular interpolation	<i>Technical reference manual - RAPID Instructions, functions and data types - MoveC</i>
Instruction for modified interpolation	<i>Technical reference manual - RAPID Instructions, functions and data types - SingArea</i>
Singularity	<i>Singularities on page 203</i>
Concurrent program execution	<i>Synchronization with logical instructions on page 181</i>
CPU Optimization	<i>Technical reference manual - System parameters</i>

3.3 Synchronization with logical instructions

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

A logical instruction is any instruction that does not generate a robot movement or an additional axis movement, for example an I/O instruction.

3.3.1 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 additional axes have come to a standstill, that is when the programmed position has been attained (see Figure 35).

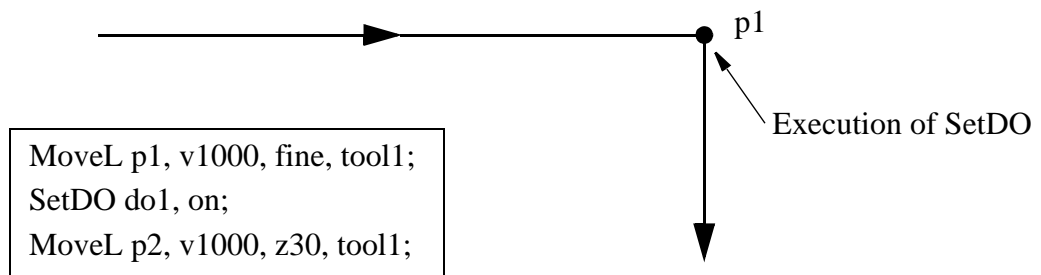


Figure 35 A logical instruction after a stop point is not executed until the destination position has been reached.

3.3.2 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 additional axes). See Figure 36 and Figure 37. These instructions are then executed in order.

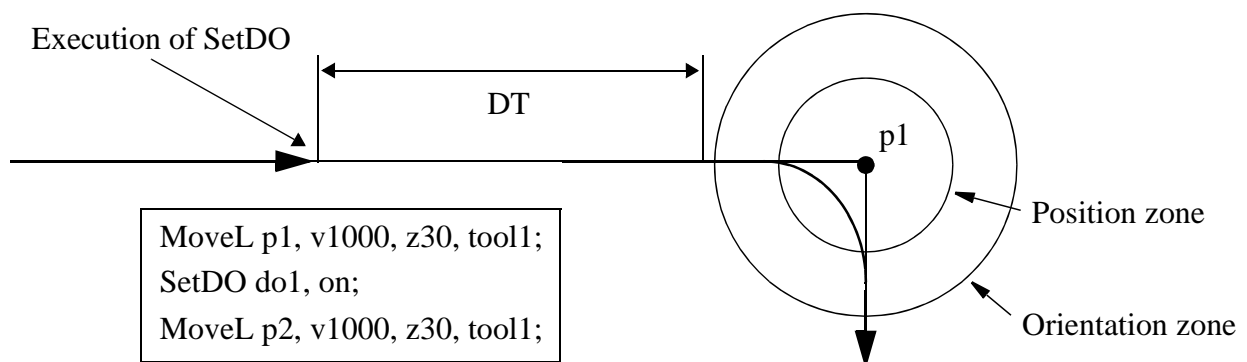


Figure 36 A logical instruction following a fly-by point is executed before reaching the largest zone.

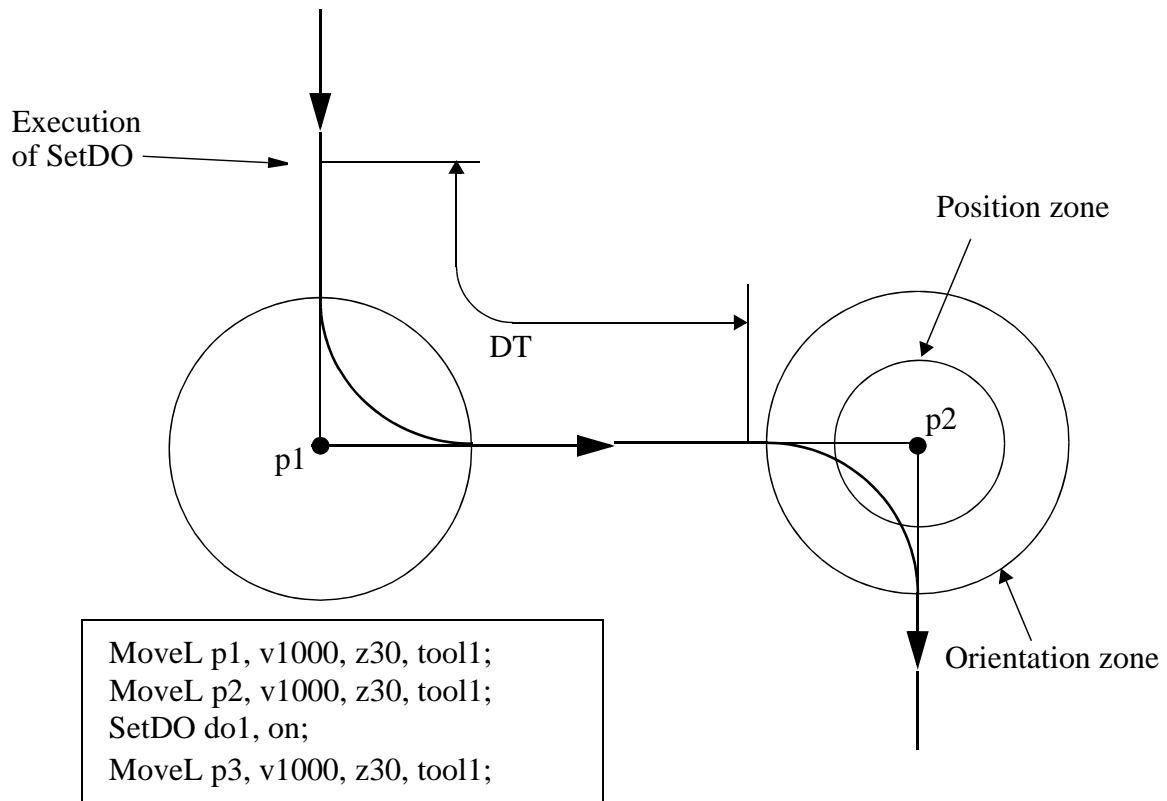


Figure 37 A logical instruction following a fly-by point is executed before reaching the largest zone.

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: approximately 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.

3.3.3 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 (for example 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 38.
- 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 additional axes). See Figure 39.

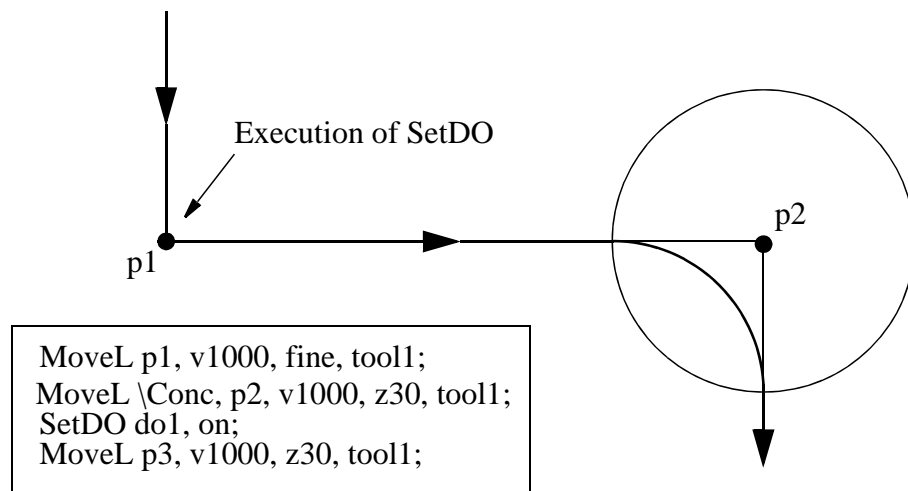


Figure 38 In the case of concurrent program execution after a stop point, a positioning instruction and subsequent logical instructions are started at the same time.

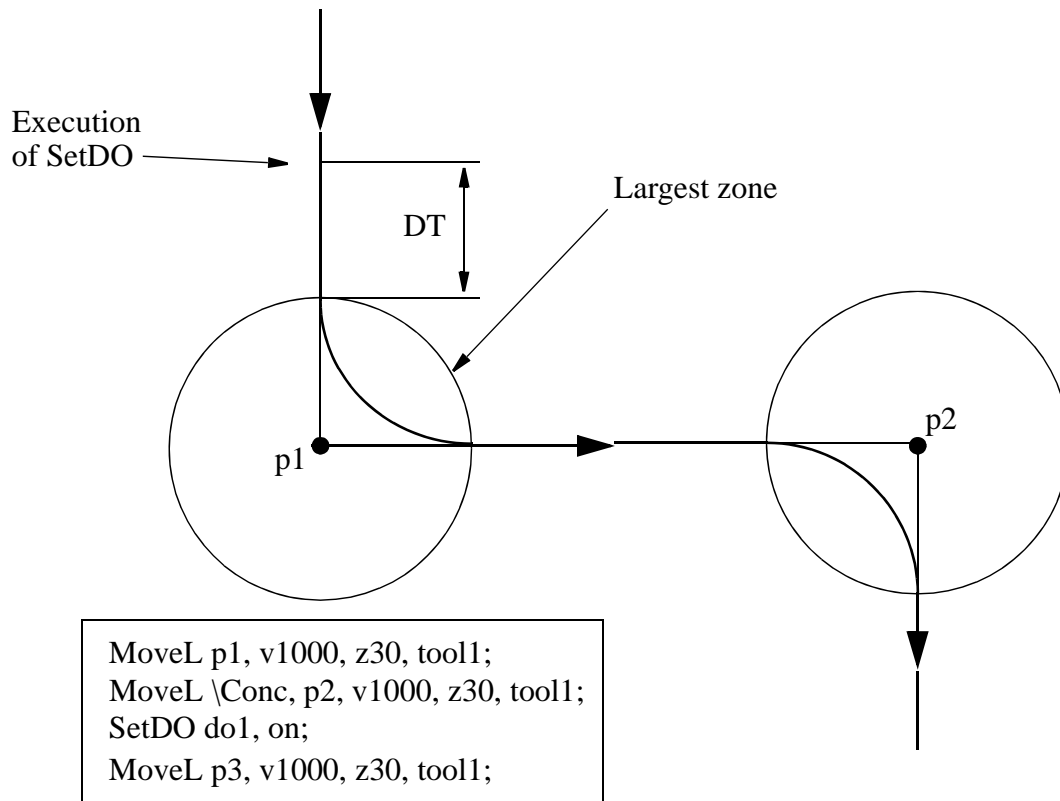


Figure 39 In the case of concurrent program execution after a fly-by point, the logical instructions start executing before the positioning instructions with the argument \Conc are started.

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 40) which means that logical instructions are executed at an earlier stage on the path than they were programmed.

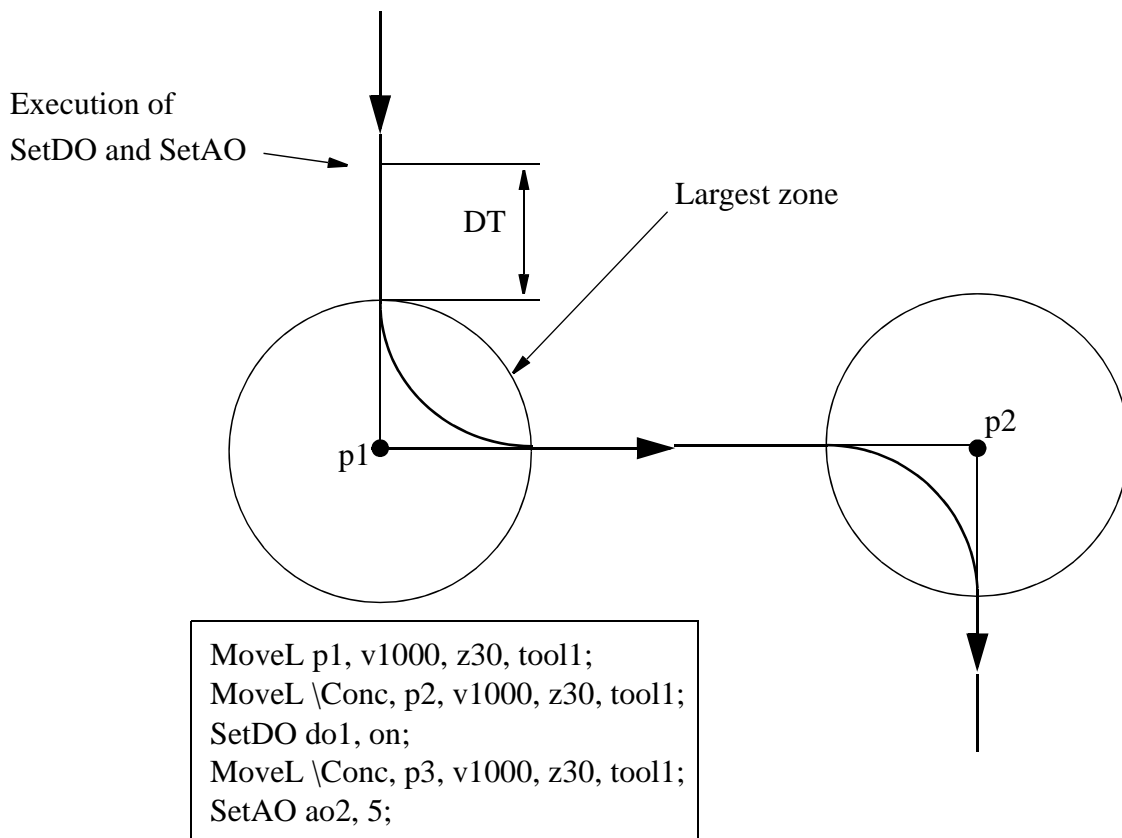


Figure 40 If several positioning instructions with the argument `\Conc` are programmed in sequence, all connected logical instructions are executed at the same time as the first position is executed.

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

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

3.3.4 Path synchronization

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

With a so-called positions event, a trig signal will be generated when the robot passes a predefined position on the path. With a time event, a signal will be generated in a predefined time before the robot stops 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 synchronized signals can be achieved both before (look ahead time) and after (delay time) the time that the robot passes the predefined position. The position is 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.

3.3.5 Related information

	Described in:
Positioning instructions	<i>Motion on page 67</i>
Definition of zone size	<i>Technical reference manual - RAPID Instructions, functions and data types - <code>zonedata</code></i>

3.4 Robot configuration

3.4.1 Different types of robot configurations

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 below when using different axis 1 directions (see Figure 41).

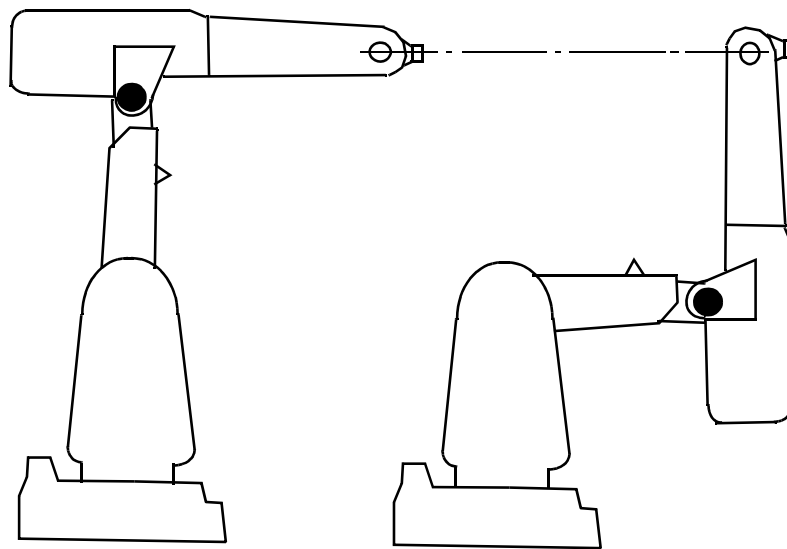


Figure 41 Two different arm configurations used to attain the same position and orientation. The configuration on the right side is attained by rotating the arm backward. Axis 1 is rotated 180 degrees.

Some robots may also get to that position from above and below while using the same axis 1 direction. This is possible for robot types with extended axis 3 working range (see Figure 42).

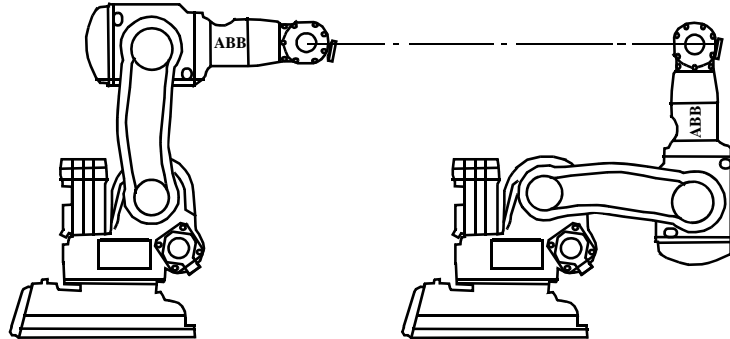


Figure 42 IRB140 with two different arm configurations used to attain same position and orientation. The Axis 1 angle is the same for both configurations. The configuration on the right side is attained by rotating the lower arm forward and the upper arm backward.

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 43).

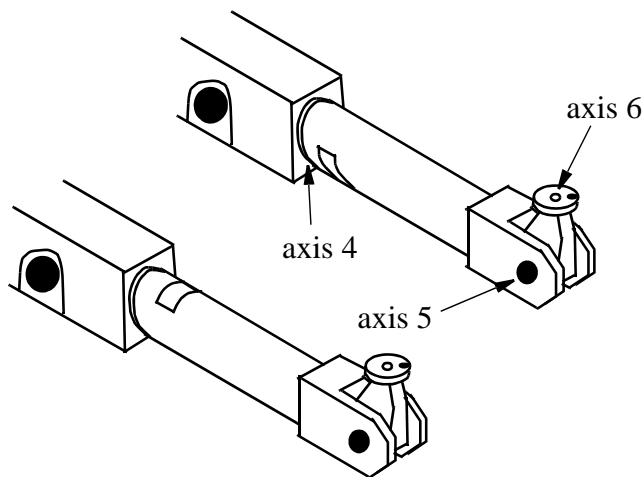


Figure 43 Two different wrist configurations used to attain the same position and orientation. In the configuration in which the front part of the upper arm points upwards (lower), axis 4 has been rotated 180 degrees, axis 5 through 180 degrees and axis 6 through 180 degrees in order to attain the configuration in which the front part of the upper arm points downwards (upper).

3.4.2 Specifying robot configuration

When programming a robot position, also a robot configuration is specified with *confdata* *cf1*, *cf4*, *cf6*, *cfx*.

The way of specifying robot configuration differs for different kinds of robot types (see *Technical reference manual - RAPID Instructions, functions and data types - confdata*, for a complete description). However, for most robot types this includes defining the appropriate quarter revolutions of axes 1, 4 and 6. For example, if axis 1 is between 0 and 90 degrees, then *cf1*=0, see figure below.

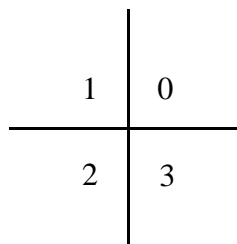


Figure 44 Quarter revolution for a positive joint angle: $\text{int}\left(\text{joint} - \frac{\text{angle}}{\pi/2}\right)$.

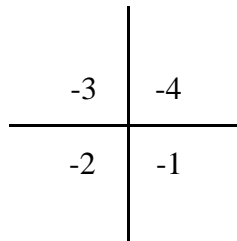


Figure 45 Quarter revolution for a negative joint angle: $\text{int}\left(\text{joint} - \frac{\text{angle}}{\pi/2} - 1\right)$.

3.4.3 Configuration check

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 by using *ConfL\On* or *ConfJ\On* and, if the correct configuration is not attained, program execution will stop. If the configuration is not checked, the robot may unexpectedly start to move its arms and wrists which, in turn, may cause it to collide with peripheral equipment.

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

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

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

When the execution of a programmed position is stopped because of a configuration error, it may often be caused by some 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 FlexPendant.

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

3.4.3.1 Detailed information of ConfL and ConfJ

MoveJ with ConfJ:

\Off:

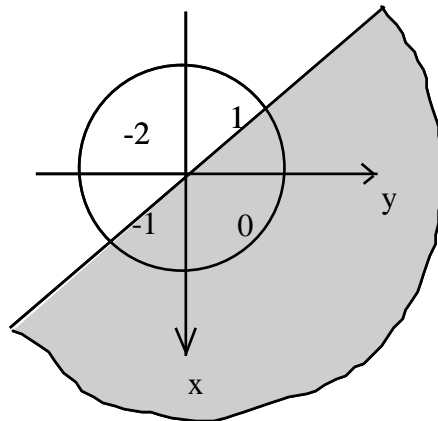
- The robot is moved to the programmed position, with a configuration (angle) for axes 1, 4 and 6, which is the closest to the configuration (angle) of the start position. This means that the configuration in the confdata is not used.

\On:

- The robot is moved to the programmed position, with a configuration equal or close to what is programmed in the confdata.
- If a program displacement is active, the arm configuration may vary within a 180 degrees area, see picture next page.
- If the calculated position is outside the 180 degree area, the robot will stop with an error message.

Arm configuration control

When executing a MoveJ with ConfJ\On, this area will be permitted if quadrant 0 is programmed (for example cf1=0):



MoveL with ConfL:

\Off:

- In this case the robot, for the end position, will choose the configuration, which is closest to the configuration of the start position. So, the robot will move along a straight line to the closest configuration, independent of what is programmed in confdata.

\On:

- In this case there will be a supervision like following: First the end position is calculated in joints, using the programmed confdata to determine the solution. Then the joint values for axes 1, 4 and 6 are compared to the corresponding for the start position. If the difference is less than 180 degree and if axis 5 has not changed the sign, then the movement will be permitted. In other cases the robot will stop with error message in the start position. If this test is OK, then the movement is performed and when in the end position the system will again check the configuration with the programmed, and if wrong stop the robot.

3.4.4 Related information

	Described in:
Definition of robot configuration	<i>Technical reference manual - RAPID Instructions, functions and data types - confdata</i>
Activating/deactivating the configuration check	<i>Motion settings on page 61</i>

3.5 Robot kinematic models

3.5.1 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.

3.5.1.1 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.

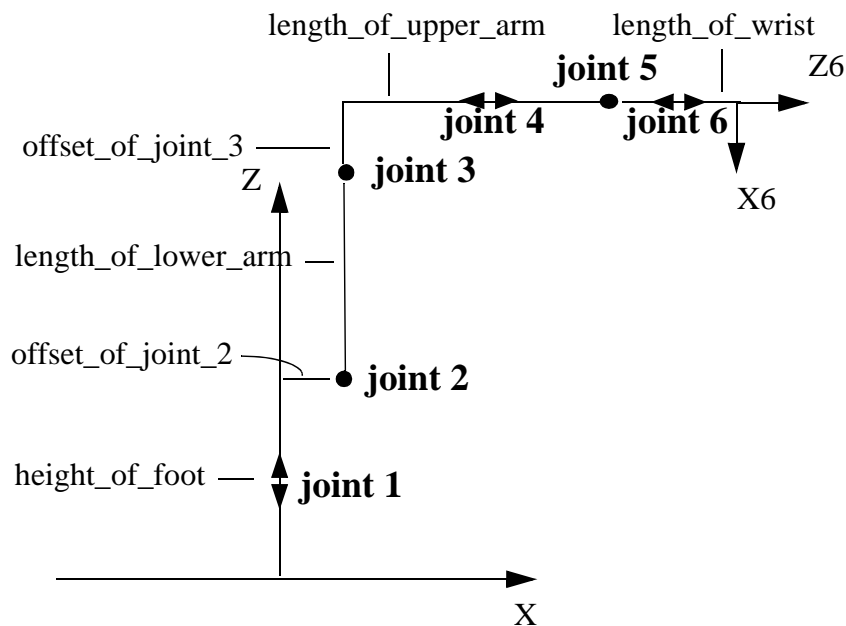


Figure 46 Kinematic structure of an IRB1400 robot

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

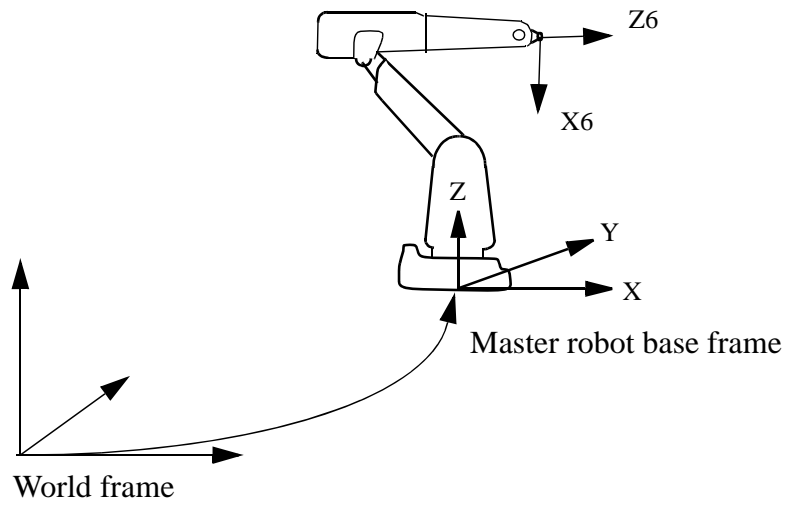


Figure 47 Base frame of master robot

3.5.1.2 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.

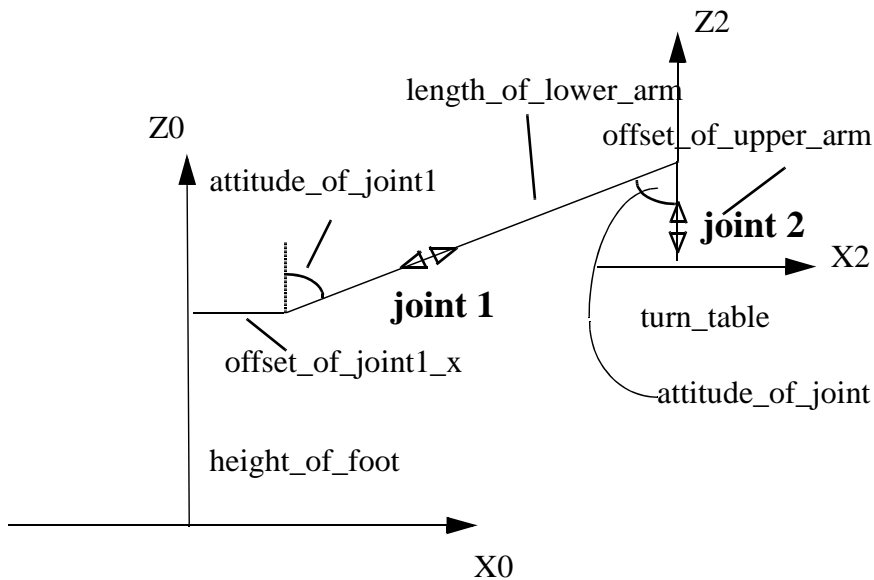


Figure 48 Kinematic structure of an ORBIT 160B robot using predefined model

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

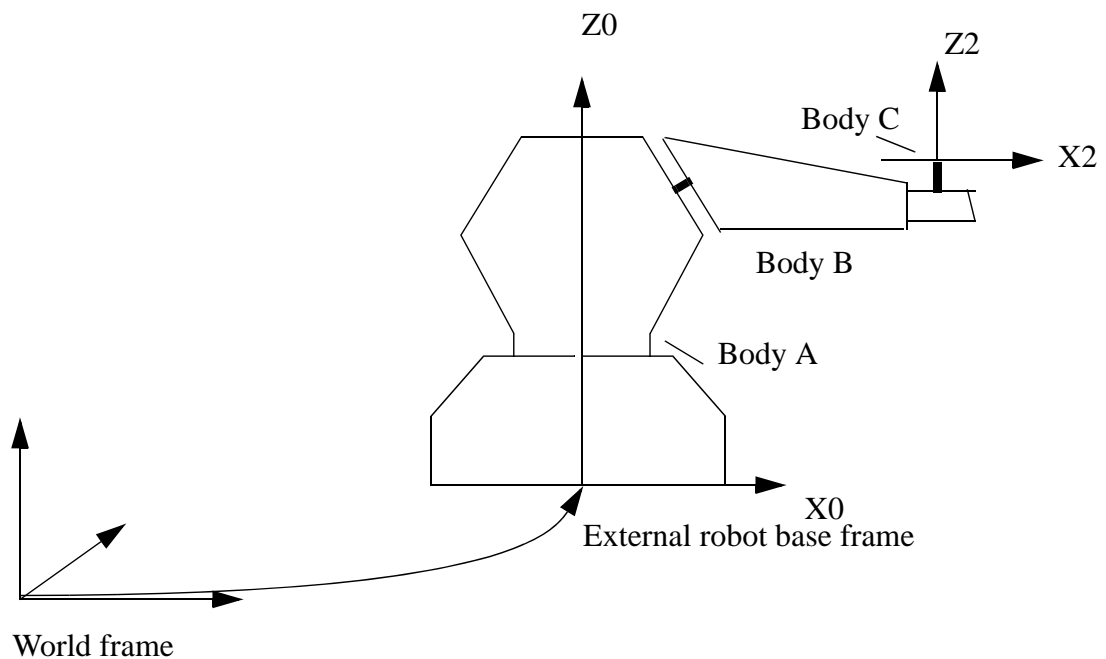


Figure 49 Base frame of an ORBIT_160B robot

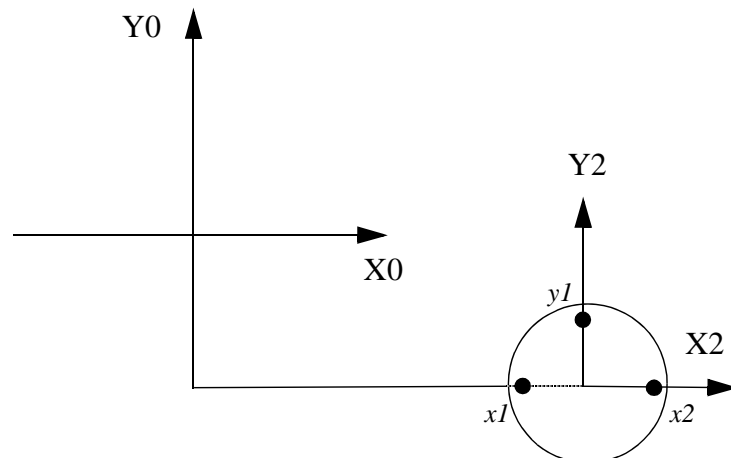


Figure 50 Reference points on turntable for base frame calibration of an ORBIT_160B robot in the home position using predefined model

3.5.2 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)

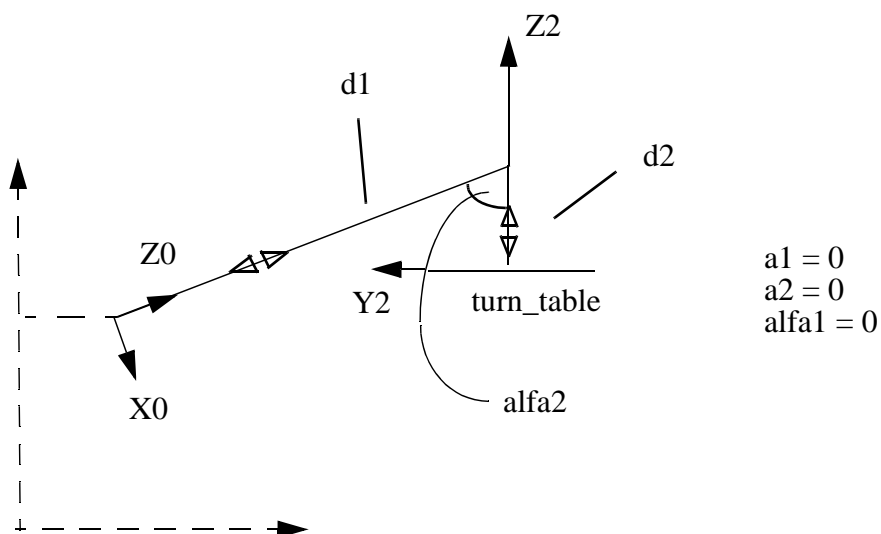


Figure 51 Kinematic structure of an ORBIT 160B robot using general kinematics model

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

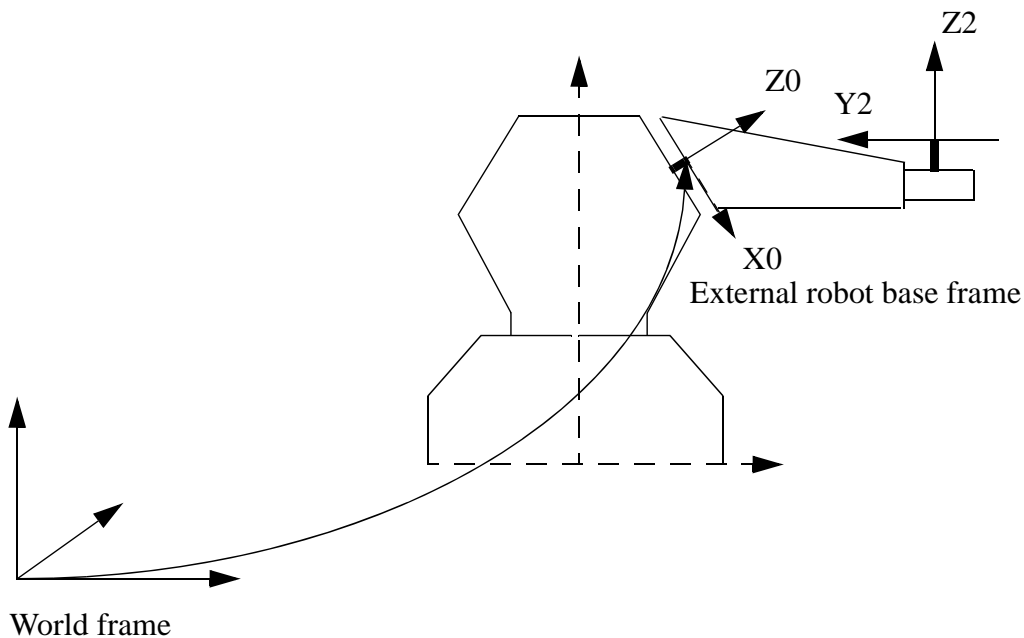


Figure 52 Base frame of an ORBIT_160B robot using general kinematics model

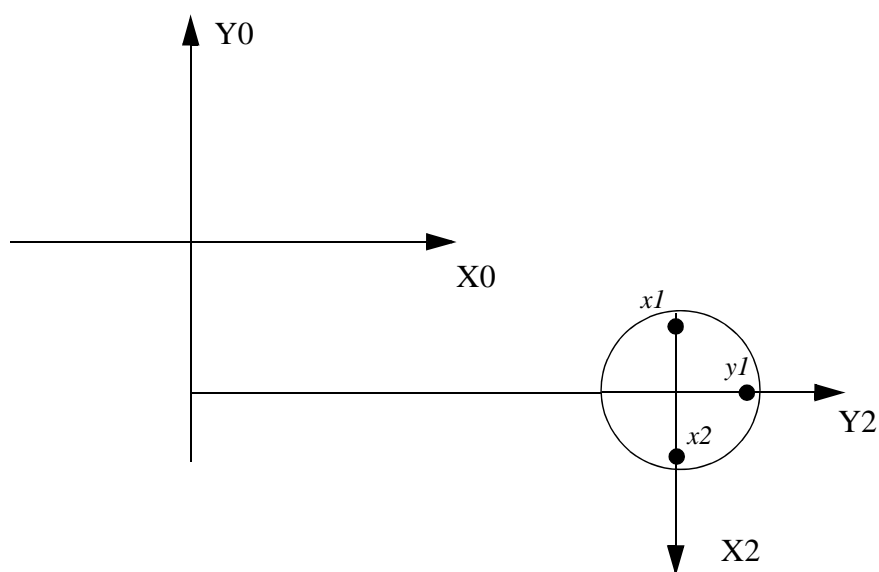


Figure 53 Reference points on turntable for base frame calibration of an ORBIT_160B robot in the home position (joints = 0 degrees)

3.5.3 Related information

	Described in:
Definition of general kinematics of an external robot	Technical reference manual -System parameters

3.6 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.

3.6.1 Introduction

The collision detection may trigger if the data for the loads mounted on the robot are not correct. This includes load data for tools, payloads and arm loads. If the tool data or payload data are not known, the load identification functionality can be used to define it. Arm load data cannot be identified.

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 additional 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.

3.6.2 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 *Technical reference manual - System parameters*.

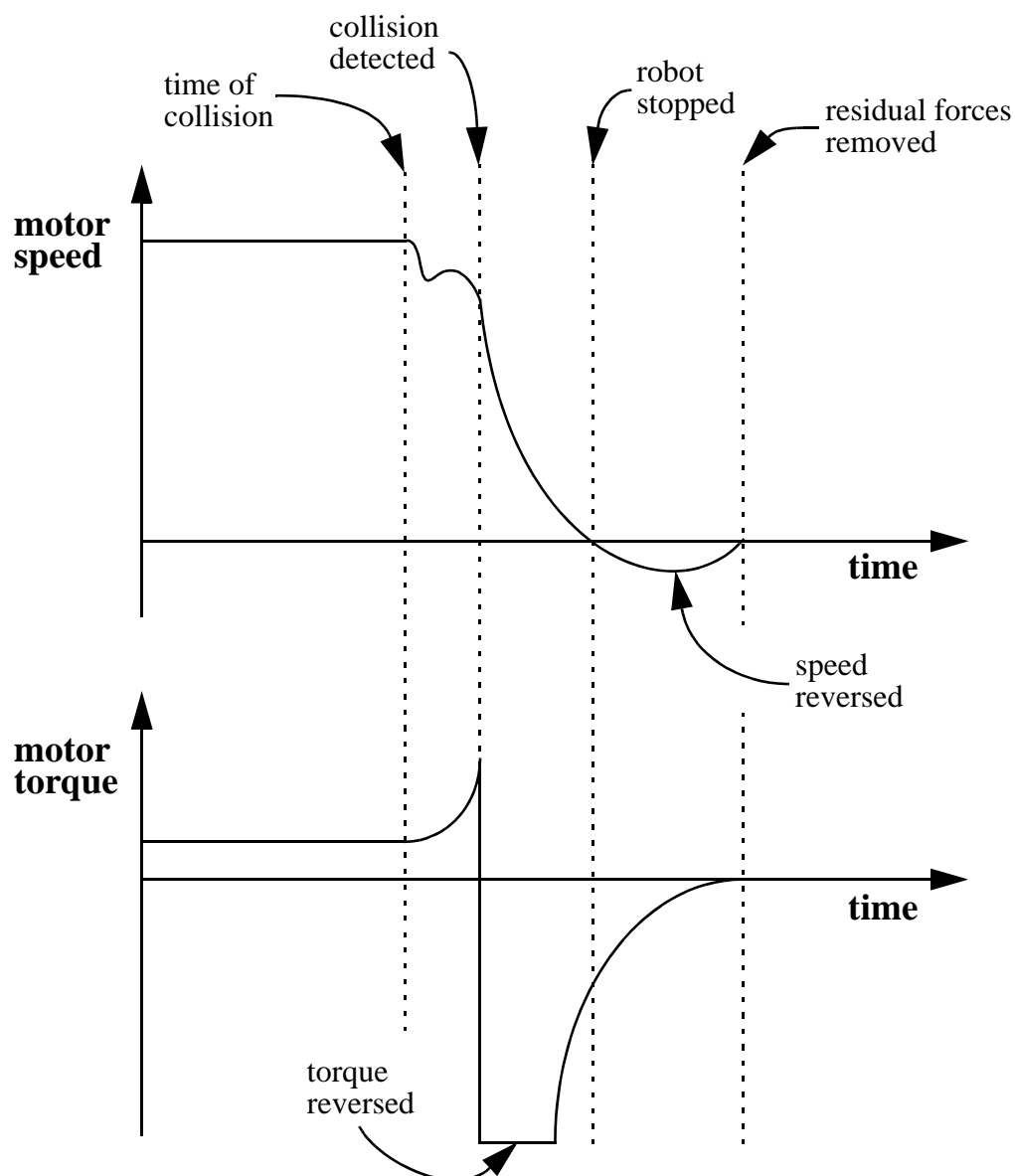
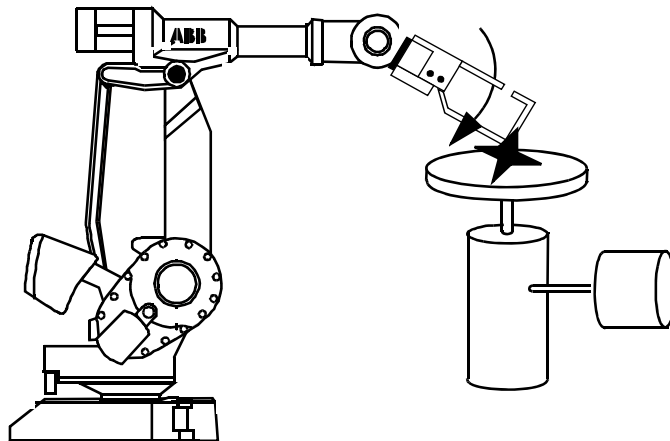
There is a RAPID instruction called MotionSup which turns the function on and off and modifies the supervision level. This is useful in applications where external process forces act on the robot in certain parts of the cycle. The MotionSup instruction is described in more detail in the *Technical reference manual - RAPID Instructions, Functions and data types*.

The tune values are set in percent of the basic tuning where 100% corresponds to the basic values. Increasing the percentage gives a less sensitive system and decreasing gives the opposite effect. It is important to note that if tune values are set in the system parameters and in the RAPID instruction both values are taken into consideration. Example: If the tune value in the system parameters is set to 150% and the tune value is set to 200% in the RAPID instruction the resulting tune level will be 300%.

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.



There is a maximum level to which the total collision detection tune level can be changed. This value is set by default to 300% but it can be modified via the system parameter **motion_sup_max_level** which is only available if the system is installed in Service mode.

3.6.3 Motion supervision dialog box

On the **ABB** menu tap **Control Panel** and then **Supervision**.

Tap the **Task** list and select a task. If you have more than one task, you need to set the desired values for each task separately.

Tap **OFF/ON** to remove or activate path supervision. Tap **-/+** to adjust sensitivity. The sensitivity can be set between 0 and 300. Unless you have the option *Collision Detection* installed, path supervision only affects the robot in auto and manually full speed mode.

Tap **OFF/ON** to remove or activate jog supervision. Tap **-/+** to adjust sensitivity. The sensitivity can be set between 0 and 300. This setting has no effect, unless you have the option *Collision Detection* installed.

For more information on *Collision Detection*, see *Application manual - Motion coordination and supervision*.

3.6.4 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 FlexPendant or through the digital input AckErrDialog.

The digital outputs are described in more detail in the *Operating manual - IRC5 with FlexPendant* and *Technical reference manual - System parameters*.

3.6.5 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.

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 additional 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.

If the robot is mounted on a track the collision detection should be set to off when the track is moving. If it is not turned off the collision detection may trigger when the track moves, even if there is no collision.

3.6.6 Related information

	Described in:
RAPID instruction MotionSup	<i>Motion</i> on page 67
System parameters for tuning	<i>Technical reference manual - System Parameters</i>
Motion supervision IO Signals	<i>Technical reference manual - System Parameters</i>
Load Identification	<i>Motion and I/O Principles</i>

3.7 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 center (the intersection of axes 4, 5, and 6) ends up directly above axis 1 (see Figure 54).

Wrist singularities are configurations where axis 4 and axis 6 are on the same line, that is axis 5 has an angle equal to 0 (see Figure 55).

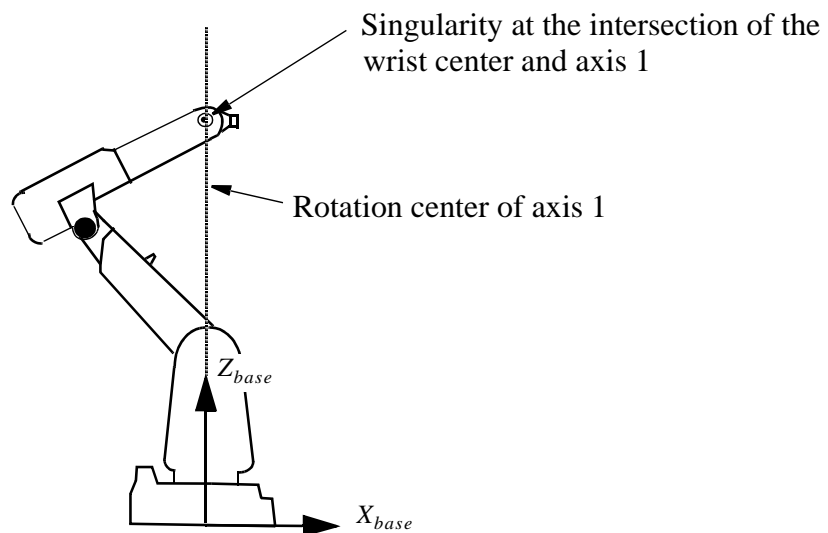


Figure 54 Arm singularity occurs where the wrist center and axis 1 intersect.

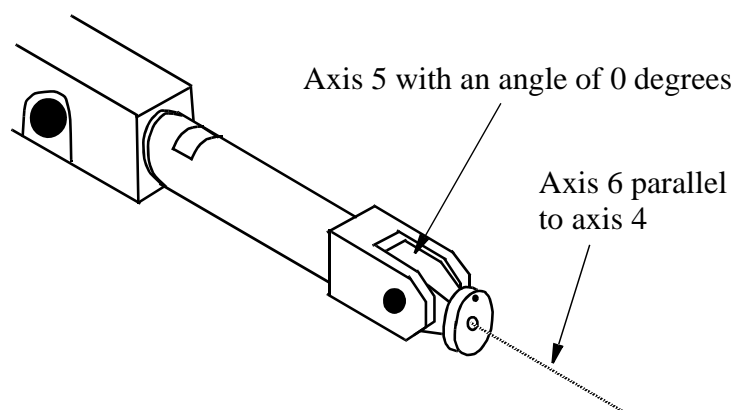


Figure 55 Wrist singularity occurs when axis 5 is at 0 degrees.

3.7.1 Singularity points of IRB140

The robot has the wrist singularity and the arm singularity. There is also a third kind of singularity. This singularity occurs at robot positions where the wrist center and the rotation centers of axes 2 and 3 are all in a straight line (see figure below).

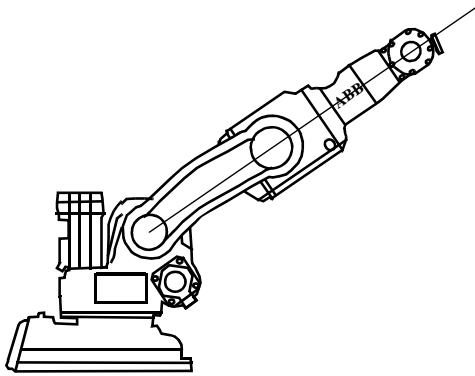


Figure 56 The additional singular point for IRB140.

3.7.2 Program execution through singularities

During joint interpolation, problems do not occur when the robot passes 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, it should be noted that the robot must not be in its singularity when only external joints are moved. This may cause robot joints to make unnecessary movements.

3.7.3 Jogging through singularities

During joint interpolation, problems do not occur when the robot passes singular points.

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

3.7.4 Related information

	Described in:
Controlling how the robot is to act on execution near singular points	<i>Technical reference manual - RAPID Instructions, functions and data types - SingArea</i>

3.8 Optimized acceleration limitation

The acceleration and speed of the robot is continuously controlled so that the defined limits are not exceeded.

The limits are defined by the user program (for example programmed speed or AccSet) or defined by the system itself (for example maximum torque in gearbox or motor, maximum torque or force in robot structure).

As long as the load data (mass, center of gravity, and inertia) is within the limits on the load diagram and correctly entered into the tool data, then no user defined acceleration limits are needed and the service life of the robot is automatically ensured.

If the load data lies outside the limits on the load diagram, then special restrictions may be necessary, that is AccSet or lower speed, as specified on request from ABB.

TCP acceleration and speed are controlled by the path planner with the help of a complete dynamic model of the robot arms, including the user defined loads.

The TCP acceleration and speed depends on the position, speed, and acceleration of all the axes at any instant in time and thus the actual acceleration varies continuously. In this way the optimal cycle time is obtained, that is one or more of the limits is at its maximum value at every instant. This means that the robot motors and structure are utilized to their maximum capability at all times.

3.9 World Zones

3.9.1 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:

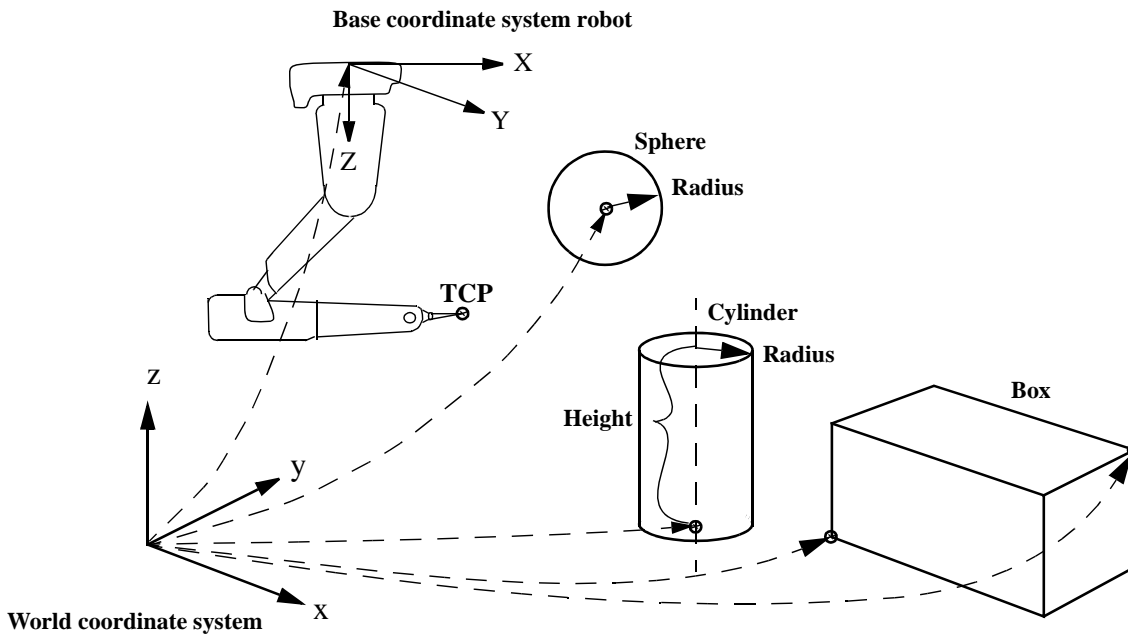
- When two robots share a part of their respective work areas. The possibility of the two robots colliding can be safely eliminated by the supervision of these signals.
- When external equipment is located inside the robot's work area. A forbidden work area can be created to prevent the robot colliding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

3.9.2 Using World Zones

To indicate that the tool center 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.

3.9.3 Definition of World Zones in the world coordinate system

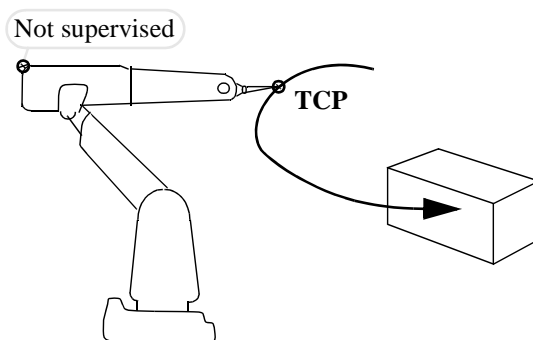
World Zones are to be defined in the world coordinate system.
The sides of the Boxes are parallel to the coordinate axes and Cylinder axis is parallel to the z-axis of the world coordinate system.



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

World Zone can also be defined in joints. The zone is to be defined between (inside) or not between (outside) two joint values for any robot or additional axes.

3.9.4 Supervision of the robot TCP



The movement of the tool center 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.

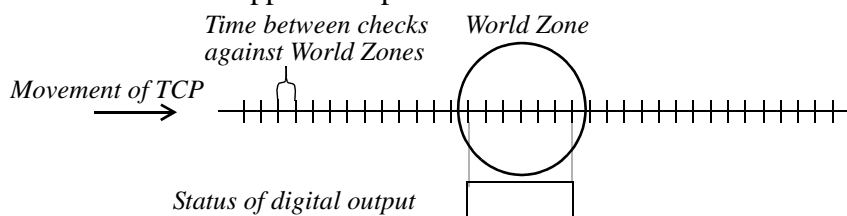
3.9.5 Stationary TCPs

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.

3.9.6 Actions

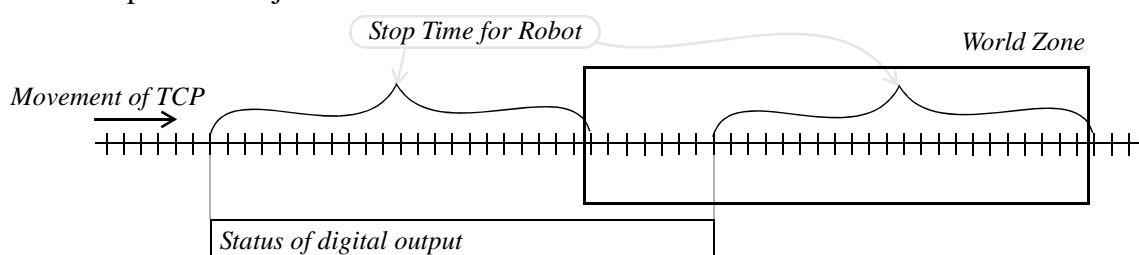
3.9.6.1 Set a digital output when the tcp is inside a World Zone

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.



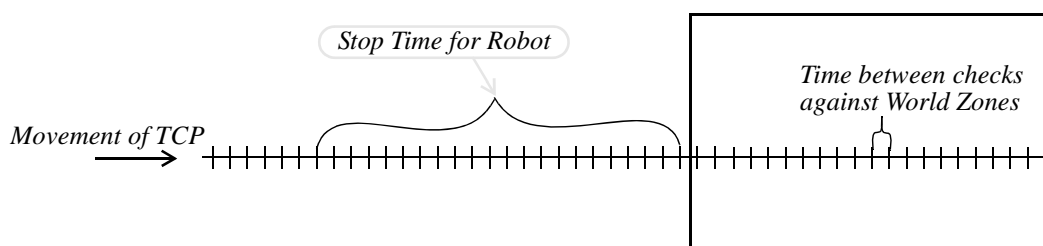
3.9.6.2 Set a digital output before the tcp reaches a World Zone

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



3.9.6.3 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 Center 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.

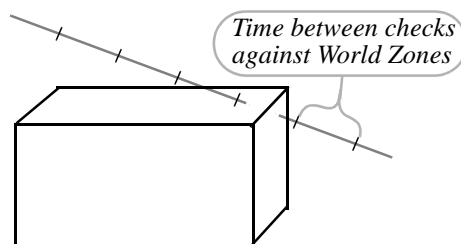
3.9.7 Minimum size of World Zones

Supervision of the movement of the tool center 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.

If World Zones are used in combination soft servo, the zone size must be additional increased to compensate for the lag from soft servo. The soft servo lag is the distance between the TCP of the robot and supervision of world zone at interpolation time. The soft servo lag will be increased with higher softness defined with the instruction SoftAct.

3.9.8 Maximum number of World Zones

A maximum of **20** World Zones can be defined at the same time.

3.9.9 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**. That is, when the controller is restarted the World Zone status will be set to outside during start. At first MOTORS ON after warm start the World Zone status will be updated correctly.

If the robot is moved during MOTORS OFF the World Zone status will not be updated until next MOTORS ON order.

A hard emergency stop (not SoftAS, SoftGS, or SoftES) can result in an incorrect World Zone status since the robot can move in or out of a Zone during the stopping movement without the World Zone signals being updated. The World Zone signals will be correctly updated after a MOTORS ON order.

3.9.10 Related information

In *Technical reference manual - RAPID Instructions, Functions and Data types*

Motion and I/O Principles:	Coordinate Systems
Data Types:	wztemporary
	wzstationary
	shapedata
Instructions:	WZBoxDef
	WZSphDef
	WZCylDef
	WZHomeJointDef
	WZLimJointDef
	WZLimSup
	WZDOSet
	WZDisable
	WZEnable
	WZFree

3.10 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 57).

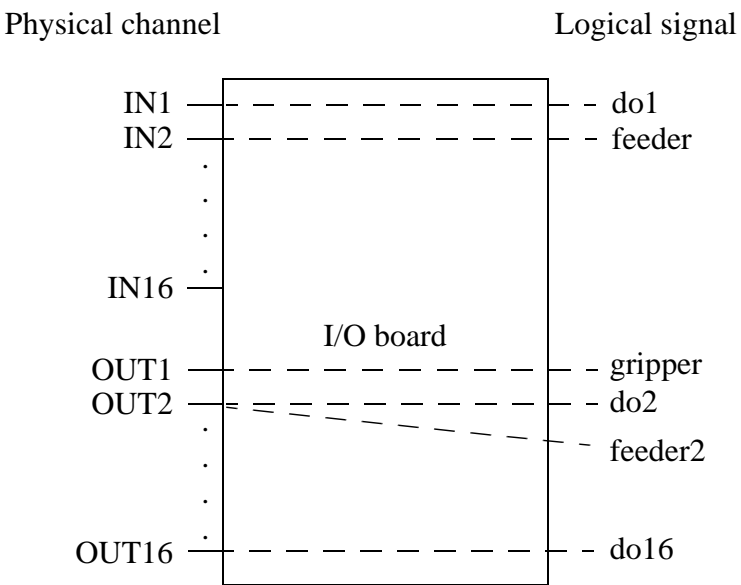


Figure 57 To be able to use an I/O board, its channels must be given logical names. In the above example, the physical output 2 is connected to two different logical names. IN16, on the other hand, has no logical name and thus cannot be used.

3.10.1 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 58).

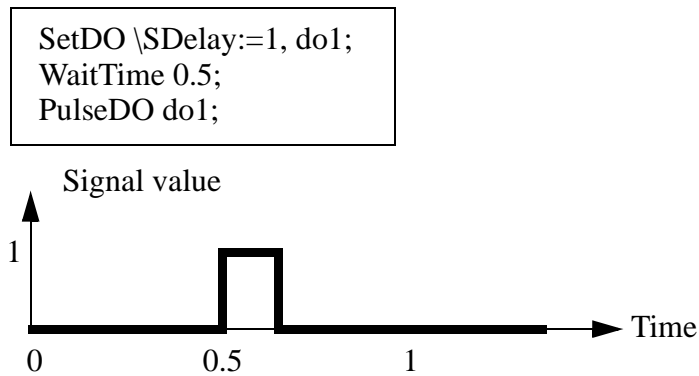


Figure 58 The instruction *SetDO* is not carried out at all because a new command is given before the time delay has elapsed.

3.10.2 Signals connected to interrupt

RAPID interrupt functions can be connected to digital signal changes. The function can be called on a raising or falling edge of the signal. However, if the digital signal changes very quickly, the interrupt can be missed.

Ex:

If a function is connected to a signal called do1 and you make a program like:

```
SetDO do1,1;
```

```
SetDO do1,0;
```

The signal will first go to High and then Low in a few milliseconds. In this case you may lose the interrupt. To be sure that you will get the interrupt, make sure that the output is set before resetting it.

Ex:

```
SetDO do1,1;
```

```
WaitDO do1,1;
```

```
SetDO do1,0;
```

In this way you will never lose any interrupt.

3.10.3 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.

3.10.4 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", for example 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.



3.10.5 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.

3.10.6 Related information

	Described in:
Definition of I/O boards and signals	<i>Technical reference manual - System parameters</i>
Instructions for handling I/O	<i>Input and output signals on page 77</i>
Manual manipulation of I/O	<i>Operating manual - IRC5 with FlexPendant</i>

4 Glossary

Term	Description
Argument	The parts of an instruction that can be changed, that is everything except the name of the instruction.
Automatic mode	The applicable mode when the operating mode selector is set to  .
Component	One part of a record.
Configuration	The position of the robot axes at a particular location.
Constant	Data that can only be changed manually.
Corner path	The path generated when passing a fly-by point.
Declaration	The part of a routine or data that defines its properties.
Dialog/Dialog box	Dialog boxes on the FlexPendant display must always be terminated (usually by tapping OK or Cancel) before they can be closed.
Error handler	A separate part of a routine where an error can be taken care of. Normal execution can then be restarted automatically.
Expression	A sequence of data and associated operands; for example <i>reg1+5</i> or <i>reg1>5</i> .
Fly-by point	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.
Function	A routine that returns a value.
Group signal	A number of digital signals that are grouped together and handled as one signal.
Interrupt	An event that temporarily interrupts program execution and executes a trap routine.
I/O	Electrical inputs and outputs.
Main routine	The routine that usually starts when the Start button is pressed.
Manual mode	The applicable mode when the operating mode switch is set to  .
Mechanical unit	A group of additional axes.
Module	A group of routines and data, that is a part of the program.
Motors On/Off	The state of the robot, that is whether or not the power supply to the motors is switched on.
Operator's panel	The panel located on the front of the controller.
Orientation	The direction of an end effector, for example.
Parameter	The input data of a routine, sent with the routine call. It corresponds to the argument of an instruction.
Persistent	A variable, the value of which is persistent.
Procedure	A routine which, when called, can independently form an instruction.
Program	The set of instructions and data which define the task of the robot. Programs do not, however, contain system modules.

Term	Description
Program data	Data that can be accessed in a complete module or in the complete program.
Program module	A module included in the robot's program and which is transferred when copying the program to a diskette, for example.
Record	A compound data type.
Routine	A subprogram.
Routine data	Local data that can only be used in a routine.
Start point	The instruction that will be executed first when starting program execution.
Stop point	A point at which the robot stops before it continues on to the next point.
System module	A module that is always present in the program memory. When a new program is read, the system modules remain in the program memory.
System parameters	The settings which define the robot equipment and properties; configuration data in other words.
Tool Center Point (TCP)	The point, generally at the tip of a tool, that moves along the programmed path at the programmed velocity.
Trap routine	The routine that defines what is to be done when a specific interrupt occurs.
Variable	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.
Window	The robot is programmed and operated using windows (or views) on the FlexPendant, for example the <i>Program Editor</i> window and the <i>Calibration</i> window. A window can be exited by switching to another window or by tapping the close button in the upper right corner.
Zone	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.

A

- additional axes
 - coordinated 159
- aggregate 37
- alias data type 38
- AND 46
- argument
 - conditional 49
- arithmetic expression 45
- array 40, 41
- assigning a value to data 57
- axis configuration 187

B

- backward execution 129
- backward handler 31, 123, 129
- base coordinate system 155

C

- circular movement 167
- comment 19, 57
- communication 107
- communication instructions 81
- component of a record 37
- concurrent execution 182, 201
- conditional argument 49
- CONST 41
- constant 39
- coordinate system 155, 193
- coordinated additional axes 159
- corner path 169
- cross connections 217

D

- data 39
 - used in expression 47
- data type 37
- declaration
 - constant 41
 - module 24
 - persistent 41
 - routine 31
 - variable 40
- displacement frame 158
- DIV 45

E

- equal data type 38
- ERRNO 97
- error handler 97
- error number 93
- Error Recovery 99
- expression 45

F

- file header 19
- file instructions 81
- function 29
- function call 48

G

- global
 - data 39
 - routine 29

I

- I/O principles 215
- I/O synchronization 181
- identifier 17
- input instructions 77
- interpolation 165
- interrupt 87

J

- joint movement 165

L

- linear movement 166
- local
 - data 39
 - routine 29
- logical expression 46
- logical value 18

M

- main routine 23
- mathematical instructions 105, 117
- MOD 45
- modified linear interpolation 168
- module 23
 - declaration 24

motion instructions 68
motion settings instructions 61
Multitasking 119
multitasking 123

N

non value data type 37
NOT 46
numeric value 18

O

object coordinate system 157
operator
 priority 49
optional parameter 30
OR 46
output instructions 77

P

parameter 30
path synchronization 185
PERS 41
persistent 39
placeholder 19
position
 instruction 68
position fix I/O 185
procedure 29
program 23
program data 39
program flow instructions 55
program module 23

R

record 37
reserved words 17
robot configuration 187
routine 29
 declaration 31
routine data 39

S

scope
 data scope 39
 routine scope 29
searching instructions 68

semi value data type 37
singularity 203, 207
soft servo 178
stationary TCP 162
stopping program execution 56
string 18
string expression 47
switch 30
syntax rules 12
system module 24

T

TCP 155, 193
 stationary 162
time instructions 103
tool center point 155, 193
tool coordinate system 161
trap routine 29, 87
typographic conventions 12

U

User - system module 27
user coordinate system 157

V

VAR 40
variable 39

W

wait instructions 57
world coordinate system 156
wrist coordinate system 161

X

XOR 46



ABB AB
Robotics Products
S-721 68 VÄSTERÅS
SWEDEN
Telephone: +46 (0) 21 344000
Telefax: +46 (0) 21 132592

3HAC16580-1, Revision J, en