ROBOTICS

Technical reference manual

RAPID overview

Trace back information:
Workspace 26C version a10
Checked in 2026-09-11
Skribenta version 5.6.019

Technical reference manual

RAPID overview

RobotWare 6.16.04

Document ID: 3HAC050947-001

Revision: Y

© Copyright 2004-2026 ABB. All rights reserved.
Specifications subject to change without notice.

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, damage 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.

Keep for future reference.

Additional copies of this manual may be obtained from ABB.

Original instructions.

© Copyright 2004-2026 ABB. All rights reserved.
Specifications subject to change without notice.

Table of contents

Overview of this manual 7
How to read this manual 10

1 Basic RAPID programming 13

1.1 Program structure 13
1.1.1 Introduction 13
1.1.2 Basic elements 15
1.1.3 Modules 19
1.1.4 System module *User* 22
1.1.5 Routines 23
1.2 Program data 29
1.2.1 Data types 29
1.2.2 Data declarations 31
1.3 Expressions 37
1.3.1 Types of expressions 37
1.3.2 Using data in expressions 40
1.3.3 Using aggregates in expressions 41
1.3.4 Using function calls in expressions 42
1.3.5 Priority between operators 44
1.3.6 Syntax 45
1.4 Instructions 47
1.5 Controlling the program flow 48
1.6 Various instructions 50
1.7 Motion settings 52
1.8 Motion 56
1.9 Input and output signals 65
1.10 Communication 69
1.11 Interrupts 73
1.12 Error recovery 77
1.13 UNDO 81
1.14 System & time 84
1.15 Mathematics 85
1.16 File operation functions 89
1.17 RAPID support instructions 90
1.18 Calibration & service 93
1.19 String functions 94
1.20 Multitasking 96
1.21 Backward execution 102

2 Motion and I/O programming 107

2.1 Coordinate systems 107
2.1.1 The tool center point of the robot (TCP) 107
2.1.2 Coordinate systems used to determine the position of the TCP 108
2.1.3 Coordinate systems used to determine the direction of the tool 114
2.2 Positioning during program execution 117
2.2.1 Introduction 117
2.2.2 Interpolation of the position and orientation of the tool 119
2.2.3 Interpolation of corner paths 123
2.2.4 Independent axes 128
2.2.5 Soft Servo 131
2.2.6 Stop and restart 132
2.3 Synchronization with logical instructions 133
2.4 Robot configuration 137
2.5 Robot kinematic models 141
2.6 Motion supervision/collision detection 145
2.7 Singularities 149

© Copyright 2004-2026 ABB. All rights reserved.

2.8 Optimized acceleration limitation 152
2.9 World Zones 153
2.10 I/O principles 158

3 Best practice for programming robots 161

4 Glossary 167

Index 169

© Copyright 2004-2026 ABB. All rights reserved.

Overview of this manual

About this manual

This is a reference manual containing a detailed explanation of the programming language as well as all instructions, functions, and data types. This manual is particularly useful when programming offline. Inexperienced users should start with *Operating manual - IRC5 with FlexPendant*.

This manual describes RobotWare 6.

Usage

This manual should be read during programming.

Who should read this manual?

This manual is intended for someone with some previous experience in programming, for example, a robot programmer.

Organization of chapters

The manual is organized in the following chapters:

Chapter	Contents
Basic RAPID programming	Answers questions like "Which instruction should I use?" or "What does this instruction mean?". 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 offline. It also explains the inner details of the language.
Motion and I/O programming	This chapter describes the coordinate systems of the robot, its velocity and other motion characteristics during execution.
Glossary	A glossary to make things easier to understand.

References

Reference	Document ID
Operating manual - IRC5 with FlexPendant	*3HAC050941-001*
Technical reference manual - RAPID Instructions, Functions and Data types	*3HAC050917-001*
Technical reference manual - RAPID kernel	*3HAC050946-001*
Technical reference manual - System parameters	*3HAC050948-001*
Application manual - Controller software IRC5	*3HAC050798-001*

Revisions

Revision	Description
-	Released with RobotWare 6.0.
A	Released with RobotWare 6.01. • Added instruction `TriggIOs`, see *Activating outputs or interrupts at specific positions on page 57*.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Continued

Revision	Description
B	Released with RobotWare 6.02. • Added trigonometric functions for data type `dnum`, see *Arithmetic functions on page 85*. • Added `TriggDataCopy`, `TriggDataReset`, and `TriggDataValid`, see *Activating outputs or interrupts at specific positions on page 57*. • Added instruction `SaveCfgData`, see *Save configuration data on page 91*.
C	Released with RobotWare 6.03. • The signal data types are now of the data type semi value, see *Non-value data types on page 29* and *Input and output signals on page 65*.
D	Released with RobotWare 6.04. • Updated the data declaration sections *Variable declaration on page 32* and *Persistent declaration on page 33*. • Minor corrections.
E	Released with RobotWare 6.05. • Removed the instructions `DitherAct` and `DitherDeact`. • Added *Matrix functions on page 87*. • Minor corrections.
F	Released with RobotWare 6.06. • Updated *RAPID support instructions on page 90*. • Updated *Behavior of the backward execution on page 103*. • Updated *Singularities on page 149*. • Minor corrections.
G	Released with RobotWare 6.07. • Added `CrossProd`, `Max` and `Min` to *Arithmetic functions on page 85*.
H	Released with RobotWare 6.08. • Minor corrections. • Updated section *Interpolation of corner paths on page 123*.
J	Released with RobotWare 6.09. • Made changes to text and examples for communication, focusing less on serial ports and more on I/O devices in general.
K	Released with RobotWare 6.11. • Added `StrFormat` to *String functions on page 94*.
L	Released with RobotWare 6.12. • Added information about independent collision stop without brake.
M	Released with RobotWare 6.13. • Added support for division of pos, see *Arithmetic expressions on page 37*. • Added information that both *PP to Main* and *PP to routine* will remove temporary world zones.
N	Released with RobotWare 6.13.02. • Clarified limitation for backward execution and execution errors.
P	Released with RobotWare 6.14. • Added support for scalar vector division of pos, see *Arithmetic expressions on page 37*.
Q	Released with RobotWare 6.15.03. • Added the function `GetNextOption`. • Added the instruction `TriggAbsJ`.
R	Released with RobotWare 6.15.04. • Added new matrix instructions.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Revision	Description
S	Released with RobotWare 6.15.06. • Added new `Fit` instructions.
T	Released with RobotWare 6.15.07. • Updated general information about backward execution.
U	Released with RobotWare 6.16. • Added the function `ReadDnum`. • Minor corrections.
V	Released with RobotWare 6.16.02. • Corrected the descriptions for `CorrCon` and `CorrWrite`. • Company name updated to reflect current legal entities.
W	Released with RobotWare 6.16.02. • Added section *Best practice for programming robots on page 161*.
X	Released with RobotWare 6.16.03. • Company name updated to reflect current legal entities.

© Copyright 2004-2026 ABB. All rights reserved.

How to read this manual

Typographic conventions

Examples of programs are always displayed in the same way as they are output to a file 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*;.

In descriptions in this manual, all names of instructions, functions, and data types are written in monospace font, for example: `TPWrite`. Names of variables, system parameters, and options are written in italic font. Comments in example code are not translated (even if the manual is translated).

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.

Example of simplified syntax

This is an example of simplified syntax with the instruction `TPWrite`.

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

- Mandatory arguments are not enclosed in brackets.
- 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 { }.

The above example uses the following arguments:

- `String` is a compulsory argument.
- `Num`, `Bool`, `Pos`, `Orient`, and `Dnum` are optional arguments.
- `Num`, `Bool`, `Pos`, `Orient`, and `Dnum` are mutually exclusive.

Example of formal syntax

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

- The text within the square brackets [ ] may be omitted.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

- 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 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.

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

Example

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


© Copyright 2004-2026 ABB. All rights reserved.

This page is intentionally left blank

1 Basic RAPID programming

1.1 Program structure

1.1.1 Introduction

Instructions

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+reg1*2`
- as a function call, for example `Abs(reg1)`
- as a string value, for example `"Producing part A"`

Routines

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.

Data

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 *persistents*.

- A constant represents a static value and can only be assigned a new value manually.
- A variable can also be assigned a new value during program execution.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

- 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

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.

© Copyright 2004-2026 ABB. All rights reserved.

1.1.2 Basic elements

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			

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.1.2 Basic elements
Continued

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.

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.

Logical values

A logical value can be expressed as TRUE or FALSE.

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 can be maximum 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"
```

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 the remainder of the line (starting from the exclamation mark) and cannot occur outside a module declaration.

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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	Description
<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

File header

A program file can start with the following file header (it is not required):

```
%%%
  VERSION:1
    LANGUAGE:ENGLISH
%%%
```

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>
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.1.2 Basic elements
Continued

```
<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

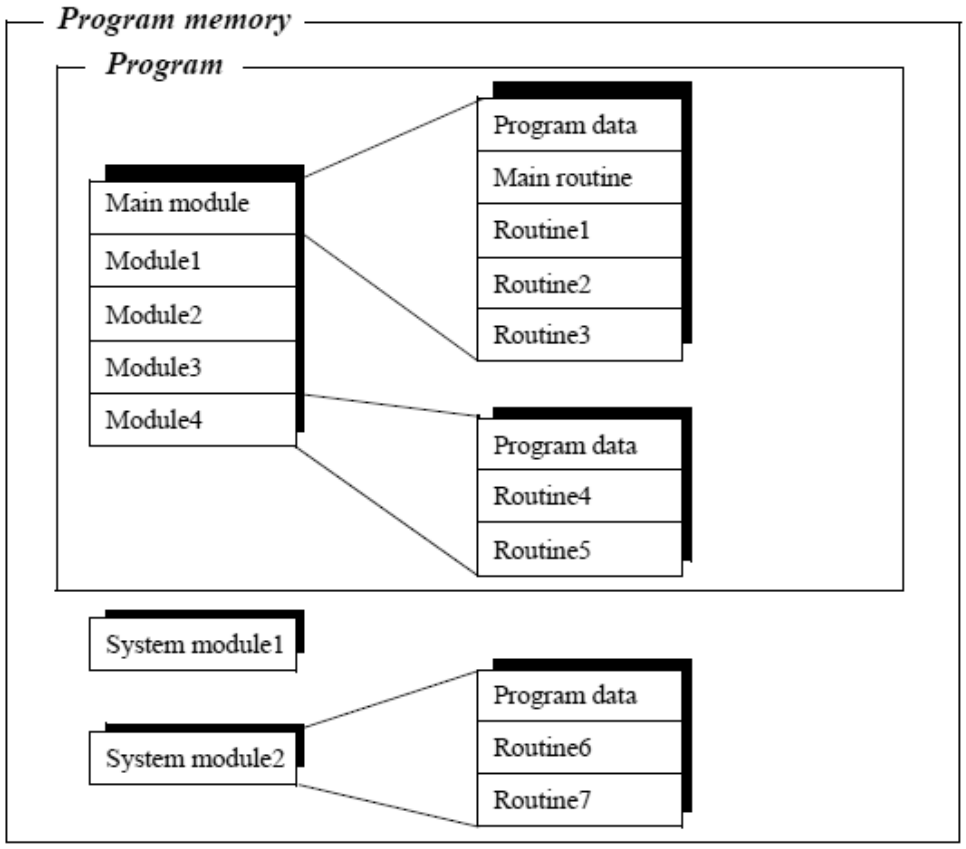
```
<character> ::= -- ISO 8859-1 (Latin-1)--
<newline> ::= -- newline control character --
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d |
     e | f
<letter> ::= <upper case letter> | <lower case letter>
<upper case letter> ::=
  A | B | C | D | E | F | G | H | I | J
  | K | L | M | N | O | P | Q | R | S | T
  | U | V | W | X | Y | Z | À | Á | Â | Ã
  | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í
  | Î | Ï | Ð | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
  | Ù | Ú | Û | Ü | Ý | Þ | 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 | ð | ñ | ò | ó | ô | õ | ö | ø
| ù | ú | û | ü | ý | þ | ÿ
```


© Copyright 2004-2026 ABB. All rights reserved.

1.1.3 Modules

Introduction

The program is divided into *program modules* and *system modules*.



xx1100000550

Program modules

A program module can consist of different data and routines. Each module, or the whole program, can be copied to 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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.1.3 Modules
Continued

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.

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
SYSMODULE	The module is a system module, otherwise a program module
NOSTEPIN	The module cannot be entered during stepwise execution
VIEWONLY	The module cannot be modified
READONLY	The module cannot be modified, but the attribute can be removed
NOVIEW	The module 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 FlexPendant. NOVIEW can only be defined offline 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.

Program file structure

As indicated above all program modules are contained in a program with a specific program name. When saving a program, 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.

Syntax

Module declaration

```
<module declaration> ::=
  MODULE <module name> [ <module attribute list> ]
  <type definition list>
  <data declaration list>
  <routine declaration list>
  ENDMODULE
<module name> ::= <identifier>
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
<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> }
```


© Copyright 2004-2026 ABB. All rights reserved.

1.1.4 System module *User*

Introduction

In order to simplify 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.

Contents

User contains 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	num	VAR num reg2:=0
reg3	num	VAR num reg3:=0
reg4	num	VAR num reg4:=0
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.

© Copyright 2004-2026 ABB. All rights reserved.

1.1.5 Routines

Introduction

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.

Routine scope

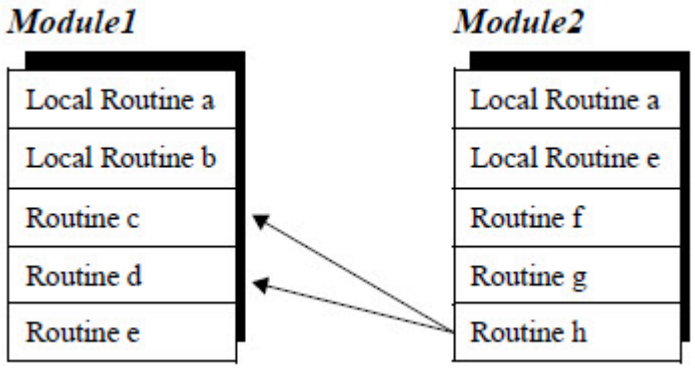
The scope of a routine denotes the area in which the routine is visible. The optional local directive of a routine declaration classifies a routine as local (within the module), otherwise it is global.

Example:

```
LOCAL PROC local_routine (...
PROC global_routine (...
```

The following scope rules apply to routines:

- 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.



xx1100000551

In the example above, 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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.1.5 Routines
Continued

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 (pipe charachter, |) 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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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{*,*})
```

Routine termination

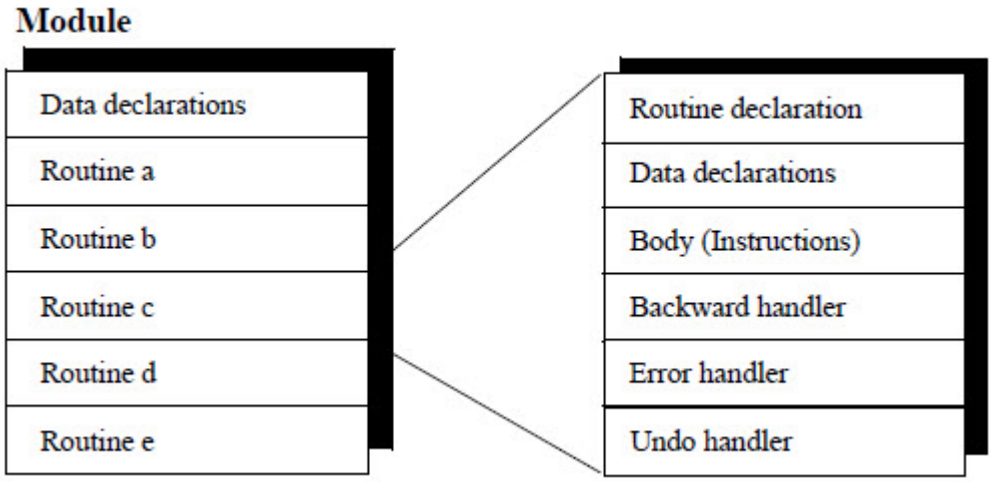
The execution of a procedure is either explicitly terminated by a RETURN instruction or implicitly terminated when the end (ENDPROC, BACKWARD, ERROR, or UNDO) of the procedure is reached.

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.

Routine declarations

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



xx1100000553

Procedure declaration

For 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.

For example, return the length of a vector.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.1.5 Routines
Continued

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

Trap declaration

For example, respond to feeder empty interrupt:

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

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 *Using function calls in expressions on page 42*.

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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> ]
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.1.5 Routines
Continued

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


© Copyright 2004-2026 ABB. All rights reserved.

1.2 Program data

1.2.1 Data types

Introduction

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.

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 signal data types (*signalai*, *signaldi*, *signalgi*,*signalao*, *signaldo*, *signalgo*) 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.

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 num level;
VAR dionum high:=1;
level:= high;
```

This is OK since `dionum` is an *alias* data type for `num`.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.2.1 Data types
Continued

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>
```


© Copyright 2004-2026 ABB. All rights reserved.

1.2.2 Data declarations

Introduction

There are three kinds of data:

- 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.

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;
```

Program data

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.

Routine data

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.

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.2.2 Data declarations
Continued

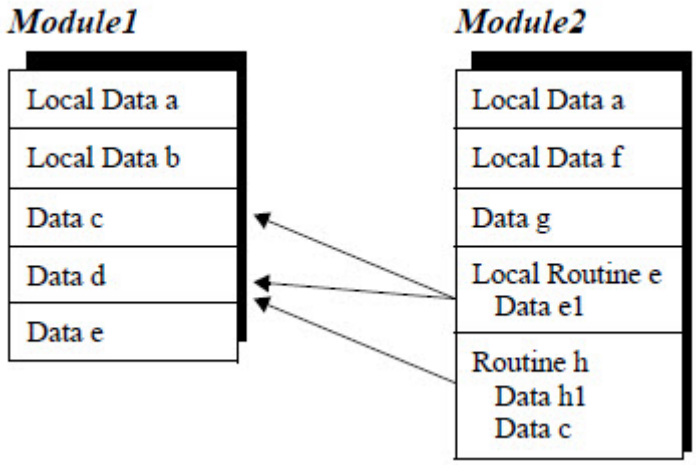
Example

In this 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.



xx1100000554

Variable declaration

A variable is introduced by a variable declaration and can be declared as global (no prescript needed) or local.

Example:

`MainModule` is loaded into task `T_ROB1`.

```
MODULE MainModule
  ! The scope of this variable is within T_ROB1, i.e. it can be
  ! accessed from any module in T_ROB1.
  VAR num global_var := 123;

  ! The scope of this variable is within this module.
  LOCAL VAR num local_var := 789;

  PROC main()
    ! The scope of this variable is within this procedure.
    VAR num local_var2 := 321;
    ...

  ENDPROC
ENDMODULE
```

A variable declared in a module that is installed shared can be declared with the prescript `TASK`, see *Technical reference manual - System parameters*, topic *Controller*, type *Automatic Loading of Modules*. Such a variable will be accessible

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

from all tasks but will have a unique value for each task. For example changing the variables value from one task will have no affect for other tasks.

Example:

SharedModule is installed shared in the system.

```
MODULE SharedModule(SYSMODULE)
  ! This variable is accessible from all tasks, but has a
  ! unique value for each task.
  TASK VAR num global_var := 123;
  ...
ENDMODULE
```

Using the TASK prescript in a module that is not installed shared will have no effect.

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 if it is a num. A string is set to empty string, and a boolean is set to FALSE.

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/module is loaded.
- the program pointer is reset, for example program pointer to main.

Persistent declaration

Persistent data can be declared only at module level, not inside a routine. Persistent variables can be declared as system-global (no prescript needed), task-global, or local.

Example:

The following module is loaded into both T_ROB1 and T_ROB2.

```
MODULE MainModule
  ! The scope of this persistent is within the task it's been
  ! loaded to. But, it will share the current value with any
  ! other task declaring the same persistent. I.e. changing the
  ! value in T_ROB1 will automatically change the value in T_ROB2.
  PERS num globalpers := 123;

  ! The scope of this persistent is within the task this
  ! module has been loaded to.
  TASK PERS num taskpers := 456;

  ! The scope of this persistent is within this module.
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.2.2 Data declarations
Continued

```
  LOCAL PERS num localpers := 789;
  ...
ENDMODULE
```

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 FlexPendant will always show the current value of the persistent.

Example:

```
PERS num reg1 := 0;
...
reg1 := 5;
```

After module save, if the code was executed, the saved module looks like this:

```
PERS num reg1 := 5;
...
reg1 := 5;
```

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.

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

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
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
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 restart, new program, program start etc.

System event affects	Power on (Restart)	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[i] / Unchanged	Unchanged	Unchanged	Unchanged	Unchanged	Unchanged	Unchanged
Commanded interrupts	Reordered	Disappears	Disappears	Disappears	Unchanged	Unchanged	Unchanged	Unchanged
Startup routine SYS_RESET (with motion settings)	Not run	Run[ii]	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

i Persistents without initial value is only initialized if not already declared.
ii Generates an error when there is a semantic error in the actual task program.

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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.

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>
```


© Copyright 2004-2026 ABB. All rights reserved.

1.3 Expressions

1.3.1 Types of expressions

Description

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);`

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 [i]
+	addition	dnum + num	dnum *i*
+	unary plus; keep sign	+num or +dnum or +pos	same [ii] , *i*
+	vector addition	pos + pos	pos
-	subtraction	num - num	num *i*
-	subtraction	dnum - dnum	dnum *i*
-	unary minus; change sign	-num or -pos	same *ii*, *i*
-	unary minus; change sign	-num or -dnum or -pos	same *ii*, *i*
-	vector subtraction	pos - pos	pos
*	multiplication	num * num	num *i*
*	multiplication	dnum * dnum	dnum *i*
*	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
/	scalar vector division	pos / num	pos
DIV [iii]	integer division	num DIV num	num
DIV *iii*	integer division	dnum DIV dnum	dnum
MOD *iii*	integer modulo; remainder	num MOD num	num

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.3.1 Types of expressions
Continued

Operator	Operation	Operand types	Result type
MOD [iii]	integer modulo; remainder	dnum MOD dnum	dnum

i Preserves integer (exact) representation as long as operands and result are kept within the integer sub-domain of the numerical type.
ii 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`).
iii Integer operations, for example 14 DIV 4=3, 14 MOD 4=2. (non-integer operands are illegal.)

Logical expressions

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

Example:

```
a>5 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 [i] = any	`bool`
>=	greater than or equal to	`num >= num`	`bool`
>=	greater than or equal to	`dnum >= dnum`	`bool`
>	greater than	`num > num`	`bool`
>	greater than or equal to	`dnum > dnum`	`bool`
<>	not equal to	any <> any	`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`

i Only value data types. Operands must have equal types.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

a AND b

b \ a	True	False
True	True	False
False	False	False

a XOR b

b \ a	True	False
True	False	True
False	True	False

a OR b

b \ a	True	False
True	True	True
False	True	False

NOT b

b	
True	False
False	True

xx1100000555

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

© Copyright 2004-2026 ABB. All rights reserved.

1.3.2 Using data in expressions

Introduction

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

Example:

```
2*pi*radius
```

Arrays

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

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

Example:

```
VAR num row{3};
VAR num column{3};
VAR num value;

! get one element from the array
value := column{3};

! get all elements in the array
row := column;
```

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;
..
! get the Y component only
yvalue := home.y;

! get the whole position
pos1 := home;
```


© Copyright 2004-2026 ABB. All rights reserved.

1.3.3 Using aggregates in expressions

Introduction

An aggregate is used for record or array values.

Example:

```
! pos record aggregate
pos := [x, y, 2*x];

! pos array aggregate
posarr := [[0, 0, 100], [0,0,z]];
```

Prerequisites

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 (aggregate type pos - determined by p1):

```
VAR pos pl;
p1 :=[1, -100, 12];
```

Example of what is **not** allowed (not allowed since the data type of neither of the aggregates can be determined by the context):

```
VAR pos pl;
IF [1, -100, 12] = [a,b,b,] THEN
```


© Copyright 2004-2026 ABB. All rights reserved.

1.3.4 Using function calls in expressions

Introduction

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

Example:

```
Sin(angle)
```

Arguments

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	Description

`Polar(3.937, 0.785398)`
`Polar(Dist:=3.937,`
`  Angle:=0.785398)`
	Two required arguments, without or with the parameter name.

`Cosine(45)` `Cosine(0.785398\Rad)`	One required argument, without or with one switch.
`Dist(p2)` `Dist(\distance:=pos1, p2)`	One required argument, without or with one optional argument.

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*.

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
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Parameters

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.

© Copyright 2004-2026 ABB. All rights reserved.

1.3.5 Priority between operators

Priority rules

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:

Priority	Operators
Highest	`* / DIV MOD`
	`+ -`
	`< > <> <= >= =`
	`AND`
Lowest	`XOR OR NOT`

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

Example expression	Evaluation order	Comment
`a + b + c`	`(a + b) + c`	Left to right rule
`a + b * c`	`a + (b * c)`	* higher than +
`a OR b OR c`	`(a OR b) OR c`	Left to right rule
`a AND b OR c AND d`	`(a AND b) OR (c AND d)`	AND higher than OR
`a < b AND c < d`	`(a < b) AND (c < d)`	< higher than AND

© Copyright 2004-2026 ABB. All rights reserved.

1.3.6 Syntax

Expressions

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

Operators

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

Constant values

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

Data

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

Aggregates

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.3.6 Syntax
Continued

Function calls

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

Special expressions

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

Parameters

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


© Copyright 2004-2026 ABB. All rights reserved.

1.4 Instructions

Description

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
```

Pick lists

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.

Syntax

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


© Copyright 2004-2026 ABB. All rights reserved.

1.5 Controlling the program flow

Introduction

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.

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.

Calling another routine

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

Program control within the routine

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

Stopping program execution

Instruction	Used to
`Stop`	Stop program execution
`EXIT`	Stop program execution when a program restart is not allowed

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to
`Break`	Stop program execution temporarily for debugging purposes
`SystemStopAction`	Stop program execution and robot movement

Stop current cycle

Instruction	Used to
`ExitCycle`	Stop the current cycle and move the program pointer to the first instruction in the main routine. When the execution mode `CONT` is selected, execution will continue with the next program cycle.

© Copyright 2004-2026 ABB. All rights reserved.

1.6 Various instructions

Introduction

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.

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
`:=`	Assign a value to data

Wait

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

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

Comments

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

Instruction	Used to
`!`	Comment on the program. A line starting with `!` (exclamation mark) is a comment and is discarded by the program execution.

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
`Load`	Load a program module into the program memory
`UnLoad`	Unload a program module from the program memory
`StartLoad`	Load a program module into the program memory during execution
`WaitLoad`	Connect the module, if loaded with `StartLoad`, to the program task

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to
CancelLoad	Cancel the loading of a module that is being or has been loaded with the instruction StartLoad
CheckProgRef	Check program references
Save	Save a program module
EraseModule	Erase a module from the program memory

Data type	Used to
loadsession	Program a load session

Various functions

Instruction	Used to
TryInt	Test if data object is a valid integer

Function	Used to
OpMode	Read the current operating mode of the robot
RunMode	Read the current program execution mode of the robot
NonMotionMode	Read the current Non-Motion execution mode of the program task
Dim	Obtain the dimensions of an array
Present	Find out whether an optional parameter was present when a routine call was made
Type	Return the data type name for a specified variable
IsPers	Check whether a parameter is a persistent
IsVar	Check whether a parameter is a variable

Basic data

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

Conversion function

Function	Used to
ByteToStr	Convert a byte to a string data with a defined byte data format.
StrToByte	Convert a string with a defined byte data format to a byte data.

© Copyright 2004-2026 ABB. All rights reserved.

1.7 Motion settings

Introduction

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
- Activation and deactivation of event buffer
- Suppress corner path warnings

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`)

- when using the restart mode **Reset system**,
- when a new program is loaded,
- when the program is started from the beginning.

Maximum TCP speed Function

Function	Used to
`MaxRobSpeed`	Return the maximum TCP speed for the used robot type

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.

A limitation of the speed can also be set, and it is later on limited when a system input signal is set.

Instruction	Used to define
`VelSet`	The maximum velocity and velocity override
`SpeedRefresh`	Update speed override for ongoing movement

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to define
SpeedLimAxis	Set speed limitation for an axis. It is later on applied by a system input signal.
SpeedLimCheckPoint	Set speed limitation for check points. It is later on applied by a system input signal.

Defining acceleration

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

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

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
ConfJ	Configuration control on/off during joint motion
ConfL	Configuration check on/off during linear motion

Defining the payload

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

Instruction	Used to define
GripLoad	Define the payload of the gripper

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
SingArea	Define the interpolation method through singular points

Activation and deactivation of event buffer

To achieve the best robot performance and good application behavior when combining an application using finepoints and a continous application where signals needs to be set in advance due to slow process equipment, the event buffer can be activated and deactivated.

Instruction	Used to define
ActEventBuffer	Activate the configured event buffer

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to define
DeactEventBuffer	Deactivate the use of the event buffer

Suppress corner path warnings

To avoid warning messages on the FlexPendant when a corner path failure occurs, these warnings can be activated and deactivated.

Instruction	Used to define
CornerPathWarning	Suppress or enable corner path warnings

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
PDispOn	Activate program displacement
PDispSet	Activate program displacement by specifying a value
PDispOff	Deactivate program displacement
EOffsOn	Activate an additional axis offset
EOffsSet	Activate an additional axis offset by specifying a value
EOffsOff	Deactivate an additional axis offset

Function	Used to
DefDFrame	Calculate a program displacement from three positions
DefFrame	Calculate a program displacement from six positions
ORobT	Remove program displacement from a position
DefAccFrame	Define a frame from original positions and displaced positions

Soft servo

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

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

Adjust the robot tuning values

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

Instruction	Used to
TuneServo	Adjust the robot tuning values
TuneReset	Reset tuning to normal
PathResol	Adjust the geometric path resolution

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to
CirPathMode	Choose the way the tool reorientates during circular interpolation.

Data type	Used to
tunetype	Represent the tuning type as a symbolic constant

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.

The instructions in the table below are only available when the robot is equipped with the option *World Zones*.

Instruction	Used to
WZBoxDef	Define a box-shaped global zone
WZCylDef	Define a cylindrical global zone
WZSphDef	Define a spherical global zone
WZHomeJointDef	Define a global zone in joints coordinates
WZLimJointDef	Define a global zone in joints coordinates for limitation of working area.
WZLimSup	Activate limit supervision for a global zone
WZDOSet	Activate global zone to set digital outputs
WZDisable	Deactivate supervision of a temporary global zone
WZEnable	Activate supervision of a temporary global zone
WZFree	Erase supervision of a temporary global zone
wztemporary	Identify a temporary global zone
wzstationary	Identify a stationary global zone
shapedata	Describe the geometry of a global zone

Various for motion settings

Instruction	Used to
WaitRob	Wait until the robot and additional axis have reached stop point or have zero speed.

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

© Copyright 2004-2026 ABB. All rights reserved.

1.8 Motion

Principle for robot movement

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.

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 52*):

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

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.

Positioning instructions

Instruction	Type of movement
`MoveC`	TCP moves along a circular path.
`MoveJ`	Joint movement.
`MoveL`	TCP moves along a linear path.
`MoveAbsJ`	Absolute joint movement.
`MoveExtJ`	Moves a linear or rotational additional axis without TCP.
`MoveCAO`	Moves the robot circularly and sets analog output in the corner
`MoveCDO`	Moves the robot circularly and sets a digital output in the middle of the corner path.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Type of movement
MoveCGO	Moves the robot circularly and set a group output signal in the corner
MoveJAO	Moves the robot by joint movement and sets analog output in the corner
MoveJDO	Moves the robot by joint movement and sets a digital output in the middle of the corner path.
MoveJGO	Moves the robot by joint movement and set a group output signal in the corner
MoveLAO	Moves the robot linearly and sets analog output in the corner
MoveLDO	Moves the robot linearly and sets a digital output in the middle of the corner path.
MoveLGO	Moves the robot linearly and sets group output signal in the corner
MoveCSync	Moves the robot circularly and executes a RAPID procedure.
MoveJSync	Moves the robot by joint movement and executes a RAPID procedure.
MoveLSync	Moves the robot linearly and executes a RAPID procedure.

Searching

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

Instruction	Type of movement
SearchC	TCP along a circular path.
SearchL	TCP along a linear path.
SearchExtJ	Joint movement of mechanical unit without TCP.

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
TriggAbsJ	Absolute joint robot movements with events
TriggC	Run the robot (TCP) circularly with an activated trigg condition.
TriggCheckIO	Define an I/O check at a given position
TriggEquip	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.
TriggDataCopy	Copy the content in a triggdata variable
TriggDataReset	Reset the content in a triggdata variable
TriggRampAO	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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to
TriggJ	Run the robot axis-by-axis with an activated trigg condition.
TriggJIOs	Run the robot (TCP) axis-by-axis with an activated I/O trigg condition.
TriggInt	Define a trigg condition to execute a trap routine at a given position
TriggIO	Define a trigg condition to set an output at a given position
TriggL	Run the robot (TCP) linearly with an activated trigg condition.
TriggLIOs	Run the robot (TCP) linearly with an activated I/O trigg condition.
StepBwdPath	Move backwards on its path in a RESTART event routine.
TriggStopProc	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.

Functions	Used to
TriggDataValid	Check if the content in a triggdata variable is valid

Data types	Used to
triggdata	Trigg conditions
aiotrigg	Analog I/O trigger condition
restartdata	Data for TriggStopProc
triggios	Trigg conditions for TriggJIOs and TriggLIOs
triggstrgo	Trigg conditions for TriggJIOs and TriggLIOs
triggiosdnum	Trigg conditions for TriggJIOs and TriggLIOs

Control of analog output signal proportional to actual TCP

Instruction	Used to
TriggSpeed	Define conditions and actions for control of an analog output signal with output value proportional to the actual TCP speed.

Motion control if an error/interrupt takes place

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

Instruction	Used to
StopMove	Define conditions and actions for control of an analog output signal with output value proportional to the actual TCP speed.
StartMove	Restart the robot movements
StartMoveRetry	Restart the robot movements and make a retry in one indivisible sequence
StopMoveReset	Reset the stop move status, but don't start the robot movements
StorePath	Store the last path generated

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to
RestoPath	Regenerate a path stored earlier
ClearPath	Clear the whole motion path on the current motion path level.
PathLevel	Get the current path level.
SyncMoveSuspend[i]	Suspend synchronized coordinated movements on StorePath level.
SyncMoveResume[i]	Resume synchronized coordinated movements on StorePath level

[i] If the robot is equipped with the option *MultiMove Coordinated*.

Function	Used to
IsStopMoveAct	Get status of the stop move flags.

Get robot info in a MultiMove system

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

Function	Used to
RobName	Get the controlled robot name in current program task, if any.

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

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
DeactUnit	Deactivate an external mechanical unit
ActUnit	Activate an external mechanical unit
MechUnitLoad	Define a payload for a mechanical unit
GetTorqueMargin	Read the least torque margin
ResetTorqueMargin	Reset least torque margin

Function	Used to
GetMotorTorque	Read the current torque of the robot and external axes motors, and can be used to detect if a servo gripper holds a load or not.
GetNextMechUnit	Retrieve the name and data for mechanical units
IsMechUnitActive	Check whether a mechanical unit is activated or not

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Independent axes

The robot axis 6 (and 4 on IRB 1600, 2600 and 4600 except ID versions) 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.

The instructions in the table below are only available when the robot is equipped with the option *Independent Axis*.

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

[i] Can only be used on IRB 5402 and IRB 5403.

The functions in the table below are only available when the robot is equipped with the option *Independent Axis*.

Function	Used to
IndInpos	Check whether an independent axis is in position.
IndSpeed	Check whether an independent axis has reached programmed speed.

Path correction

The instructions, functions, and data types in the tables below are only available when the robot is equipped with the options *Path offset* or *RobotWare-Arc sensor*.

Instruction	Used to
CorrCon	Connects to a correction generator
CorrWrite	Writes to a correction generator
CorrDiscon	Disconnect from a previously connected correction generator
CorrClear	Remove all connected correction generators

Function	Used to
CorrRead	Read the total corrections delivered by all connected correction generators

Data type	Used to
corrdescr	Add geometric offsets in the path coordinate system

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Path Recorder

The instructions, functions, and data types in the tables below are only available when the robot is equipped with the option *Path Recovery*.

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

Function	Used to
PathRecValidBwd	Check if the path recorder is active and if a recorded backward path is available
PathRecValidFwd	Check if the path recorder can be used to move forward

Data type	Used to
pathrecid	Identify a breakpoint for the path recorder

Conveyor tracking

The instructions in the table below are only available when the robot is equipped with the option *Conveyor tracking*.

Instruction	Used to
WaitWObj	Wait for work object on conveyor
DropWObj	Drop work object on conveyor

Servo Tracking for Indexing Conveyor

The instructions in the table below are only available when the robot is equipped with the option *Conveyor tracking*.

Instruction	Used to
IndCnvAddObject	Used to manually add an object to the object queue.
IndCnvEnable	Start to listen to the digital input and execute an indexing movement when triggered.
IndCnvDisable	The system stop listen to the digital input.
IndCnvInit	Set up the indexed conveyor functionality
IndCnvReset	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.
indcnvdata	Used to setup the behavior of the indexing conveyor functionality.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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.

The instructions in the table below are only available when the robot is equipped with the option *Sensor Synchronization*.

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

Load identification and collision detection

Instruction	Used to
`MotionSup` [i]	Deactivate/activate motion supervision
`ParIdPosValid`	Valid robot position for parameter identification
`ParIdRobValid`	Valid robot type for parameter identification
`LoadId`	Load identification of tool or payload
`ManLoadId`	Load identification of external manipulator

[i] Only if the robot is equipped with the option *Collision Detection*.

Data type	Used to
`loadidnum`	Represent an integer with a symbolic constant
`paridnum`	Represent an integer with a symbolic constant
`paridvalidnum`	Represent an integer with a symbolic constant

Position functions

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Function	Used to
Distance	The distance between two positions

Check interrupted path after power failure

Function	Used to
PFRestart	Check if the path has been interrupted at power failure.

Status functions

Function	Used to
CSpeedOverride	Read the speed override set by the operator from the **Program Editor** or **Production Window**.

Motion data

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

Data type	Used to define
robtarget	The end position
jointtarget	The end position for a MoveAbsJ/L, MoveExtJ, or TriggAbsJ instruction
speeddata	The speed
zonedata	The accuracy of the position (stop point or fly-by point)
tooldata	The tool coordinate system and the load of the tool
wobjdata	The work object coordinate system
stoppointdata	The termination of the position
identno	A number used to control synchronizing of two or more coordinated synchronized movement with each other

Basic data for movements

Data type	Used to define
pos	A position (x, y, z)
orient	An orientation
pose	A coordinate system (position + orientation)
confdata	The configuration of the robot axes
extjoint	The position of the additional axes
robjoint	The position of the robot axes
loaddata	A load
mecunit	An external mechanical unit

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Related information

Options	Described in
Collision Detection *Sensor Synchronization* *Independent Axis* *Path Corrections* *Path Offset* *Path Recovery*	*Application manual - Controller software IRC5*
Conveyor tracking	*Application manual - Conveyor tracking*
MultiMove	*Application manual - MultiMove*
RobotWare-Arc	*Application manual - Arc and Arc Sensor*

© Copyright 2004-2026 ABB. All rights reserved.

1.9 Input and output signals

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.

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.

Changing the value of a signal

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

Reading the value of input signals

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

```
! Digital input
IF di1 = 1 THEN ...
! Digital group input (smaller than 23 bits)
IF gi1 = 5 THEN ...
! Analog input
IF ai1 > 5.2 THEN ...
```

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

ERR_NO_ALIASIO_DEF if the signal variable is a variable declared in RAPID. It has not been connected to an I/O signal defined in the I/O configuration with instruction AliasIO.

ERR_NORUNUNIT if there is no contact with the I/O unit.

ERR_SIG_NOT_VALID if the I/O signal cannot be accessed (only valid for ICI field bus).

Reading the value of output signals

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

```
! Digital output
IF do1 = 1 THEN ...
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
! Digital group output (smaller than 23 bits)
IF go1 = 5 THEN ...
! Analog output
IF ao1 > 5.2 THEN ...
```

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

ERR_NO_ALIASIO_DEF if the signal variable is a variable declared in RAPID. It has not been connected to an I/O signal defined in the I/O configuration with instruction AliasIO.

ERR_NORUNUNIT if there is no contact with the I/O unit.

ERR_SIG_NOT_VALID if the I/O signal cannot be accessed (only valid for ICI fieldbus).

Reading the value of large group signals

Group signals that are between 23 bits up to 32 bits must read by a function and converted to dnum.

```
! Digital group input (larger than 23 bits)
IF GInputDnum(gi1) = 4294967295 THEN ...
! Digital group output (larger than 23 bits)
IF GOutputDnum(go1) = 4294967295 THEN ...
```

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

ERR_NO_ALIASIO_DEF if the signal variable is a variable declared in RAPID. It has not been connected to an I/O signal defined in the I/O configuration with instruction AliasIO.

ERR_NORUNUNIT if there is no contact with the I/O unit.

ERR_SIG_NOT_VALID if the I/O signal cannot be accessed (only valid for ICI fieldbus).

Testing input or output signals

Instruction	Used to define
WaitDI	Wait until a digital input is set or reset
WaitDO	Wait until a digital output is set on reset
WaitGI	Wait until a group of digital input signals is set to a value
WaitGO	Wait until a group of digital output signals is set to a value
WaitAI	Wait until a analog input is less or greather then a value
WaitAO	Wait until a analog output is less or greather then a value

Function	Used to define
TestDI	Test whether a digital input is set
ValidIO	Valid I/O signal to access
GetSignalOrigin	Get information about the origin of an I/O signal

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Data type	Used to define
signalorigin	Define the I/O signal origin

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 define
IODisable	Disable an I/O module
IOEnable	Enable an I/O module

Defining input and output signals

Instruction	Used to define
AliasIO	Define a signal with an alias name

Data type	Used to define
dionum	The symbolic value of a digital signal
signalai	The name of an analog input signal
signalao	The name of an analog output signal
signaldi	The name of a digital input signal
signaldo	The name of a digital output signal
signalgi	The name of a group of digital input signals
signalgo	The name of a group of digital output signals
signalorigin	Define the I/O signal origin

Get status of I/O network and device

Instruction	Used to define
IOBusState	Get current status of the I/O network

Function	Used to define
IOUnitState	Returns current status of the I/O device

Data type	Used to define
iounit_state	The status of the I/O device
bustate	The status of the I/O network

Start of I/O network

Instruction	Used to define
IOBusStart	Start an I/O bus.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Read group signal information

Instruction	Used to define
`GetGroupSignalInfo`	Read information about a digital group signal.

© Copyright 2004-2026 ABB. All rights reserved.

1.10 Communication

Communicating via I/O devices and files

There are four possible ways to communicate via I/O devices and files:

- 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.

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 I/O device or file used must first be opened. On doing this, the device/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.

Communicating using the FlexPendant, function group TP

Instruction	Used to
`TPErase`	Clear the FlexPendant operator display
`TPWrite`	Write text on the FlexPendant operator display
`ErrWrite`	Write text on the FlexPendant display and simultaneously store that message in the program error log.
`TPReadFK`	Label the function keys and to read which key is pressed
`TPReadDnum`	Read a numeric value from the FlexPendant
`TPReadNum`	Read a numeric value from the FlexPendant
`TPShow`	Choose a window on the FlexPendant from RAPID
`tpnum`	Represent FlexPendant window with a symbolic constant

Communicating using the FlexPendant, function group UI

Instruction	Used to
`UIMsgBox`	Write message to FlexPendant Read pressed button from FlexPendant Type basic

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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

Data type	Used to
icondata	Represent icon with a symbolic constant
buttondata	Represent button with a symbolic constant
listitem	Define menu list items
btnres	Represent selected button with a symbolic constant

Reading from or writing to a character-based I/O device or file

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

Function	Used to
ReadDnum	Read a dnum numeric value
ReadNum	Read a numeric value
ReadStr	Read a text string

Communicating using binary I/O devices or files

Instruction	Used to
Open	Open an I/O device or file for binary transfer of data
WriteBin	Write to a binary I/O device or file
WriteAnyBin	Write to any binary I/O device or file
WriteStrBin	Write a string to a binary I/O device or file
Rewind	Set the file position to the beginning of the file
Close	Close the I/O device or file
ClearIOBuff	Clear input buffer of an I/O device or file
ReadAnyBin	Read from any binary I/O device or file
WriteRawBytes	Write data of type rawbytes to a binary I/O device or file

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to
ReadRawBytes	Read data of type rawbytes from a binary I/O device or file

Function	Used to
ReadBin	Read from a binary I/O device or file
ReadStrBin	Read a string from a binary I/O device or file

Communication using rawbytes

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

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

Function	Used to
RawBytesLen	Get the current length of valid bytes in a rawbyte variable

Data for I/O devices or files

Data type	Used to define
iodev	A reference to an I/O device or a file, which can then be used for reading and writing
rawbytes	A general data container, used for communication with I/O devices

Communicating using sockets

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Function	Used to
SocketGetStatus	Get current socket state
SocketPeek	Test for the presence of data on a socket

Data type	Used to define
socketdev	Socket device
socketstatus	Socket status

Communication using RAPID Message Queues

Data type	Used to define
rmqheader [i]	The rmqheader is a part of the data type rmqmessage and is used to describe the message
rmqmessage	A general data container, used when communicate with RAPID Message Queue functionality
rmqslot	Identity number of a RAPID task or SDK client
IRMQMessage	Order and enable interrupts for a specific data type
RMQFindSlot	Find the identity number of the queue configured for a RAPID task or SDK client
RMQGetMessage	Get the first message from the queue of this task
RMQGetMsgData	Extract the data from a message
RMQGetMsgHeader	Extract header information from a message
RMQSendMessage	Send data to the queue of the queue configured for a RAPID task or SDK client
RMQSendWait	Send a message and wait for the answer
RMQEmptyQueue	Empty the RMQ connected to the task executing instruction.
RMQReadWait	Wait until a message has arrived, or timeout occurs.

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

Function	Used to
RMQGetSlotName [i]	Get the name of a RAPID Message Queue client from a given identity number, that is from a given rmqslot

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

© Copyright 2004-2026 ABB. All rights reserved.

1.11 Interrupts

Introduction

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 (first in, first out). 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 75*.

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

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

- Individual interrupts can be deactivated. Any interrupts occurring during this time are disregarded.

Connecting interrupts to trap routines

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

Ordering interrupts

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

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

Cancelling interrupts

Instruction	Used to
IDelete	Cancel (delete) an interrupt

Enabling/disabling interrupts

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

Interrupt data

Instruction	Used to
GetTrapData	Get interrupt data for current TRAP
ReadErrData	Get information about an error

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Data type of interrupts

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

Safe Interrupt

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 continuos 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.

Interrupt manipulation

Defining an interrupt makes it known to the system. 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:

```
! deactivate
ISleep sig1int;

! activate
IWatch sig1int;
```

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:

```
! disable
IDisable sig1int;

1 enable
IEnable sig1int;
```

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:

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
IDelete sig1int;
```

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.

© Copyright 2004-2026 ABB. All rights reserved.

1.12 Error recovery

Introduction

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.

Error numbers

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.

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.

Creating an error situation from within the program Instruction

Instruction	Used to
`RAISE`	Create an error and call the error handler

Booking an error number instruction

Instruction	Used to
`BookErrNo`	Book a new RAPID system error number.

Restarting/returning from the error handler

Instruction	Used to
`EXIT`	Stop program execution in the event of a fatal error

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Instruction	Used to
RAISE	Call the error handler of the routine that called the current routine
RETRY	Re-execute the instruction that caused the error
TRYNEXT	Execute the instruction following the instruction that caused the error
RETURN	Return to the routine that called the current routine
RaiseToUser	From a NOSTEPIN routine, the error is raised to the error handler at user level.
StartMoveRetry	An instruction that replaces the two instructions StartMove and RETRY. It both resumes movements and re-execute the instruction that caused the error.
SkipWarn	Skip the latest requested warning message.
ResetRetryCount	Reset the number of counted retries.

Function	Used to
RemainingRetries	Remaining retries left to do.

Generate process error

Instruction	Used to
ErrLog	Display an error message on the FlexPendant and write it in the robot message log.
ErrRaise	Create an error in the program and then call the error handler of the routine.

Function	Used to
ProcerrRecovery	Generate process error during robot movement.

Data for error handling

Data type	Used to
errnum	The reason for the error
errstr	Text in an error message

Configuration for error handling

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

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:

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
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.

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.

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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.

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.

© Copyright 2004-2026 ABB. All rights reserved.

1.13 UNDO

Introduction

RAPID routines may contain an UNDO handler. The handler is executed automatically if the program pointer 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.

The code in the UNDO handler shall be executed as fast as possible without any delays, since the user probably is not aware of that any code will be run at that point.

Terminology

The following terms are related to UNDO.

- UNDO: The execution of cleaning 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 routine Main if nothing else is specified.
- UNDO context: When the current routine is part of a call-chain starting in an UNDO handler.

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.

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 program pointer is unexpectedly moved out of an UNDO-routine, for instance if the user moves the program pointer 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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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.

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, for example 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 program pointer is not visible in an UNDO-handler. When UNDO executes, the program pointer 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 program pointer is moved to Main.

Example

The program:

```
PROC B
  TPWrite "In Routine B";
  Exit;
UNDO
  TPWrite "In UNDO of routine B";
ENDPROC

PROC A
  TPWrite "In Routine A";
  B;
ENDPROC

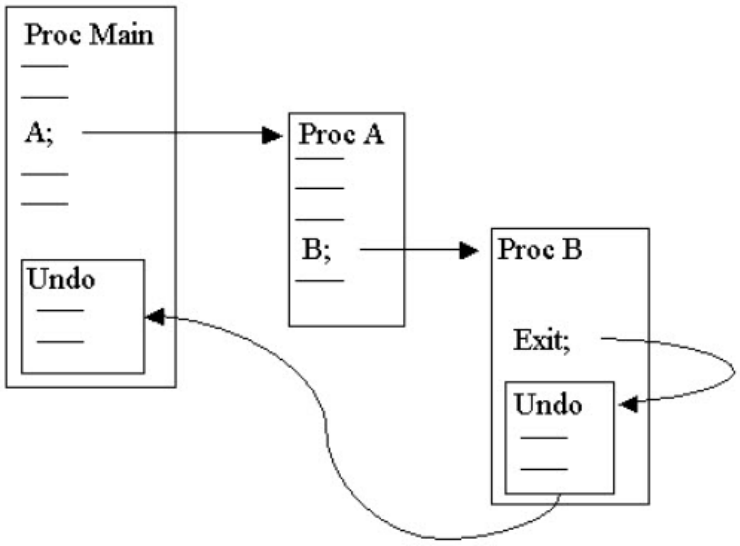
PROC main
  TPWrite "In main";
  A;
UNDO
  TPWrite "In UNDO of main";
ENDPROC
```

The output:

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
In main
In Routine A
In Routine B
In UNDO of routine B
In UNDO of main
```



xx1100000588

© Copyright 2004-2026 ABB. All rights reserved.

1.14 System & time

Description

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

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.

Using a clock to time an event

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

Function	Used to
`ClkRead`	Read a clock used for timing

Data type	Used to
`clock`	Timing – stores a time measurement in seconds

Reading current time and date

Function	Used to
`CDate`	Read the current date as a string
`CTime`	Read the current time as a string
`GetTime`	Read the current time as a numeric value

Retrieve time information from file

Function	Used to
`FileTimeDnum`	Retrieve the last time for modification of a file.
`ModTimeDnum`	Retrieve file modify time for the loaded module.
`ModExist`	Check if program module exist.

Get the size of free program memory

Function	Used to
`ProgMemFree`	Retrieve the size of free program memory.

© Copyright 2004-2026 ABB. All rights reserved.

1.15 Mathematics

Description

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

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.

Simple calculations on numeric data

Instruction	Used to
`Clear`	Clear the value
`Add`	Add or subtract a value
`Incr`	Increment by 1
`Decr`	Decrement by 1
`Rand`	Generate a random number

More advanced calculations

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

Arithmetic functions

Function	Used to
`Abs`	Calculate the absolute value
`AbsDnum`	Calculate the absolute value
`Max`	Returns the largest of two values.
`Min`	Returns the smallest of two values.
`Round`	Round a numeric value
`RoundDnum`	Round a numeric value
`Trunc`	Truncate a numeric value
`TruncDnum`	Truncate a numeric value
`Sqrt`	Calculate the square root
`SqrtDnum`	Calculate the square root
`Exp`	Calculate the exponential value with the base "e"
`Pow`	Calculate the exponential value with an arbitrary base
`PowDnum`	Calculate the exponential value with an arbitrary base
`ACos`	Calculate the arc cosine value
`ACosDnum`	Calculate the arc cosine value

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Function	Used to
ASin	Calculate the arc sine value
ASinDnum	Calculate the arc sine value
ATan	Calculate the arc tangent value in the range [-90,90]
ATanDnum	Calculate the arc tangent value in the range [-90,90]
ATan2	Calculate the arc tangent value in the range [-180,180]
ATan2Dnum	Calculate the arc tangent value in the range [-180,180]
Cos	Calculate the cosine value
CosDnum	Calculate the cosine value
Sin	Calculate the sine value
SinDnum	Calculate the sine value
Tan	Calculate the tangent value
TanDnum	Calculate the tangent value
EulerZYX	Calculate Euler angles from an orientation
OrientZYX	Calculate the orientation from Euler angles
PoseInv	Invert a pose
PoseMult	Multiply a pose
PoseVect	Multiply a pose and a vector
VectMagn	Calculate the magnitude of a pos vector
DotProd	Calculate the dot (or scalar) product of two pos vectors
CrossProd	Calculate the cross (or vector) product of two pos vectors
NOrient	Normalize unnormalized orientation (quaternion)
DnumToNum	Convert a dnum numeric value to a num numeric value
NumToDnum	Convert a num numeric value to a dnum numeric value

String digit functions

Function	Used to
StrDigCmp	Numeric compare of two strings with only digits
StrDigCalc	Arithmetic operations on two strings with only digits

Data type	Used to
stringdig	String with only digits

Bit functions

Instruction	Used to
BitClear	Clear a specified bit in a defined byte or dnum data.
BitSet	Set a specified bit to 1 in a defined byte or dnum data.

Function	Used to
BitCheck	Check if a specified bit in a defined byte data is set to 1.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Function	Used to
BitCheckDnum	Check if a specified bit in a defined dnum data is set to 1.
BitAnd	Execute a logical bitwise *AND* operation on data types byte.
BitAndDnum	Execute a logical bitwise *AND* operation on data types dnum.
BitNeg	Execute a logical bitwise NEGATION operation on data types byte.
BitNegDnum	Execute a logical bitwise NEGATION operation on data types dnum.
BitOr	Execute a logical bitwise *OR* operation on data types byte.
BitOrDnum	Execute a logical bitwise *OR* operation on data types dnum.
BitXOr	Execute a logical bitwise *XOR* operation on data types byte.
BitXOrDnum	Execute a logical bitwise *XOR* operation on data types dnum.
BitLSh	Execute a logical bitwise *LEFT SHIFT* operation on data types byte.
BitLShDnum	Execute a logical bitwise *LEFT SHIFT* operation on data types dnum.
BitRSh	Execute a logical bitwise *RIGHT SHIFT* operation on data types byte.
BitRShDnum	Execute a logical bitwise *RIGHT SHIFT* operation on data types dnum.

Data type	Used to
byte	Used together with instructions and functions that handle bit manipulation (8 bits).
dnum	Used together with instructions and functions that handle bit manipulation (52 bits).

Matrix functions

Instruction	Used to
MatrixAdd	Calculates the sum of two matrices.
MatrixInverse	Inverse a matrix.
MatrixMult	Multiply two matrices, or multiply matrix with scalar.
MatrixReset	Set all elements in a matrix to 0.
MatrixSolve	Solve linear equation systems on the form A*x=b.
MatrixSolveQR	Compute a QR-factorization of an (m x n) matrix A.
MatrixSub	Calculates the difference between two matrices
MatrixSVD	Compute a singular value decomposition (SVD).
MatrixTranspose	Transpose a matrix.

Fitting shapes to points

Instruction	Used to
FitCirle	Fit a circle to 3D-points.
FitLine	Fit a line to a set of points.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

1.15 Mathematics
Continued

Instruction	Used to
FitPlane	Fit a plane to a set of points.
FitSphere	Fit a sphere to a set of points.

© Copyright 2004-2026 ABB. All rights reserved.

1.16 File operation functions

Instructions

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

Functions

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

Data types

Data type	Used to
dir	Traverse directory structures.

© Copyright 2004-2026 ABB. All rights reserved.

1.17 RAPID support instructions

Description

Various functions for supporting of the RAPID language:

- Get system data, system information, memory information
- Read and 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
- Read service information

Get system data

Instruction	Used to
GetSysData	Get data and name of current active tool or work object.
ResetPPMoved	Reset state for the program pointer moved in manual mode.
SetSysData	Activate a specified system data name for a specified data type.

Function	Used to
IsSysID	Test the system identity.
IsStopStateEvent	Get information about the movement of the program pointer (PP).
PPMovedInManMode	Test whether the program pointer is moved in manual mode.
RobOS	Check if the execution is performed on robot controller (RC) or virtual controller (VC).

Get information about the system

Functions to get information about options, products, serial number, software version, robot type, LAN IP address, or controller language.

Function	Used to
GetNextOption	Get name of options installed.
GetNextProduct	Get name of products installed.
GetSysInfo	Get information about the system.

Get information about memory

Function	Used to
ProgMemFree	Get the size of free program memory

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Read configuration data

Instruction to read one attribute of a named system parameter.

Instruction	Used to
ReadCfgData	Read one attribute of a named system parameter.

Write configuration data

Instruction to write one attribute of a named system parameter.

Instruction	Used to
WriteCfgData	Write one attribute of a named system parameter.

Save configuration data

Instruction to save system parameter to file.

Instruction	Used to
SaveCfgData	Save system parameters to file

Restart the controller

Instruction	Used to
WarmStart	Restart the controller for example when you have changed system parameters from RAPID.

Text tables instructions

Instruction	Used to
TextTabInstall	Install a text table in the system.

Function	Used to
TextTabGet	Get the text table number of a user defined text table.
TextGet	Get a text string from the system text tables.
TextTabFreeToUse	Test whether the text table name (text resource string) is free to use or not.

Get object name

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

Instruction	Used to
ArgName	Return the original data object name.

Get information about the tasks

Function	Used to
GetTaskName	Get the identity of the current program task, with its name and number.
MotionPlannerNo	Get the number of the current motion planner.
TaskIsExecuting	Check if a task is executing

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Get current event type, execution handler or execution level

Function	Used to
EventType	Get current event routine type.
ExecHandler	Get type of execution handler.
ExecLevel	Get execution level.

Data type	Used to
event_type	Event routine type.
handler_type	Type of execution handler.
exec_level	Execution level.

Get current task selection panel status for a program task

Function	Used to
GetTSPStatus	Get current task selection panel status
TaskIsActive	Check if a normal task is active

Data type	Used to
tsp_status	Task selection panel status

Search for symbols

Instructions to search for data objects in the system.

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

Function	Used to
GetNextSym	Together with SetDataSearch data objects can be retrieved from the system.

Data type	Used to
datapos	Holds information of where a certain object is defined in the system.

Read service information

Instruction	Used to
GetServiceInfo	Read service information from the system.

© Copyright 2004-2026 ABB. All rights reserved.

1.18 Calibration & service

Description

A number of instructions are available to calibrate and test the robot system.

Calibration of the tool

Instruction	Used to
MToolRotCalib	Calibrate the rotation of a moving tool.
MToolTCPCalib	Calibrate tool center point (TCP) for a moving tool.
SToolRotCalib	Calibrate tool center point (TCP) and rotation of a stationary tool.
SToolTCPCalib	Calibrate tool center point (TCP) for a stationary tool

Various calibration methods

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

Directing a value to the robot's test signal

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

Instruction	Used to
TestSignDefine	Define a test signal
TestSignReset	Reset all test signals definitions

Function	Used to
TestSignRead	Read test signal value

Data type	Used to
testsignal	For programming instruction TestSignDefine

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.

Instruction	Used to
SpyStart	Start the recording of instruction and time data during execution.
SpyStop	Stop the recording of time data during execution.

© Copyright 2004-2026 ABB. All rights reserved.

1.19 String functions

Description

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

Basic operations

Data type	Used to
string	String. Predefined constants STR_DIGIT, STR_UPPER, STR_LOWER, and STR_WHITE.

Instruction/operator	Used to
StrFormat	Format a string
:=	Assign a value (copy of string)
+	String concatenation

Function	Used to
StrLen	Find string length
StrPart	Obtain part of a string

Comparison and searching

Operator	Used to
=	Test if equal to
<>	Test if not equal to

Function	Used to
StrMemb	Check if character belongs to a set
StrFind	Search for character in a string
StrMatch	Search for pattern in a string
StrOrder	Check if strings are in order

Conversion

Function	Used to
ByteToStr	Convert a byte to string data
DecToHex	Convert a number specified in a readable string in the base 10 to the base 16
DnumToStr	Convert a numeric value to a string
HexToDec	Convert a number specified in a readable string in the base 16 to the base 10
NumToStr	Convert a numeric value to a string
StrToVal	Convert a string to a value
StrMap	Map a string

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Function	Used to
`StrToByte`	Convert a string to a byte
`ValToStr`	Convert a value to a string

© Copyright 2004-2026 ABB. All rights reserved.

1.20 Multitasking

Description

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.

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

Limitations

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 offline.

Basics

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, in the same way as the normal program. All the modules 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....).

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

General instructions and functions

Instruction	Used to
WaitSyncTask [i]	Synchronize several program tasks at a special point in each program

[i] If the robot is equipped with the option *MultiTasking*.

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

Data type	Used to
taskid	Identify available program tasks in the system.
syncident	Specify the name of a synchronization point
tasks	Specify several RAPID program tasks

MultiMove system with coordinated robots

Instruction	Used to
SyncMoveOn [i]	Start a sequence of synchronized movements
SyncMoveOff	To end synchronized movements
SyncMoveUndo	Reset synchronized movements

[i] If the robot is equipped with the option *MultiMove Coordinated*.

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

Data type	Used to
syncident [i]	To specify the name of a synchronization point
tasks	Specify several RAPID program tasks
identno	Identity for move instructions

[i] If the robot is equipped with the option *MultiTasking*.

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Synchronizing using polling

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

Synchronizing using an interrupt

The instructions `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
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
ENDMODULE
```

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
    ...
  ENDIF
ENDPROC
ENDMODULE
```

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 that 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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Priorities

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

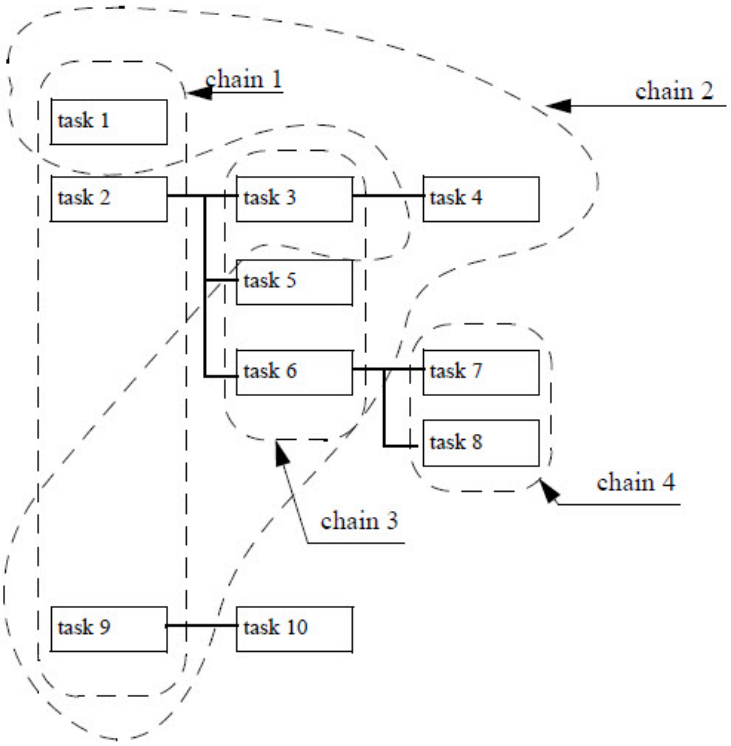
The example below describes some situations where the system has 10 tasks (see *Figure 9*).

Round robbin chain 1: tasks 1, 2, and 9 are busy.

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

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

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



xx1100000589

Figure 9: The tasks can have different priorities

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

- **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*, topic *Controller*, type *Task*.

Recommendation

When specifying task priorities, 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.

© Copyright 2004-2026 ABB. All rights reserved.

1.21 Backward execution

Description

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.
- Most instructions and functions do not do anything when they are executed backwards.
- When executing backward handlers and there is an execution error, it will not be possible to handle the error in an error handler.

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.

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”.

Example 1

```
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:

```
MoveL p1,v500,z10,tool1;
MoveC p2,p3,v500,z10,tool1;
MoveL p4,v500,z10,tool1;
```

Example 2

```
PROC MoveTo ()
  MoveL p1,v500,z10,tool1;
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

```
    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 code is executed (the code of the procedure until the backward handler):

```
MoveL p1,v500,z10,tool1;
MoveC p2,p3,v500,z10,tool1;
MoveL p4,v500,z10,tool1;
```

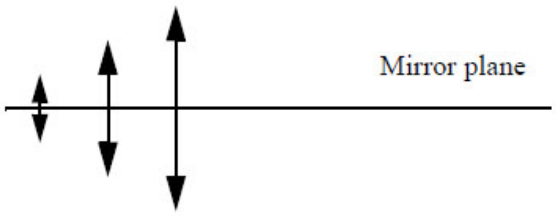
When the procedure is called during backwards execution, the following code is executed (the code in the backward handler):

```
MoveL p4,v500,z10,tool1;
MoveC p2,p3,v500,z10,tool1;
MoveL p1,v500,z10,tool1;
```

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
```



xx1100000633

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

Move instructions includes all instructions that result in some movement of the robot or additional axes, such as `MoveL`, `SearchC`, `TriggJ`, `ArcC`, or `PaintL`.

WARNING

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.

Behavior of the backward execution

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

though 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.

To be able to change from forward to backward execution in a nostepin routine, or vice versa, the system parameter *StepOutNoStepin* must be set.

Target, movement type and speed

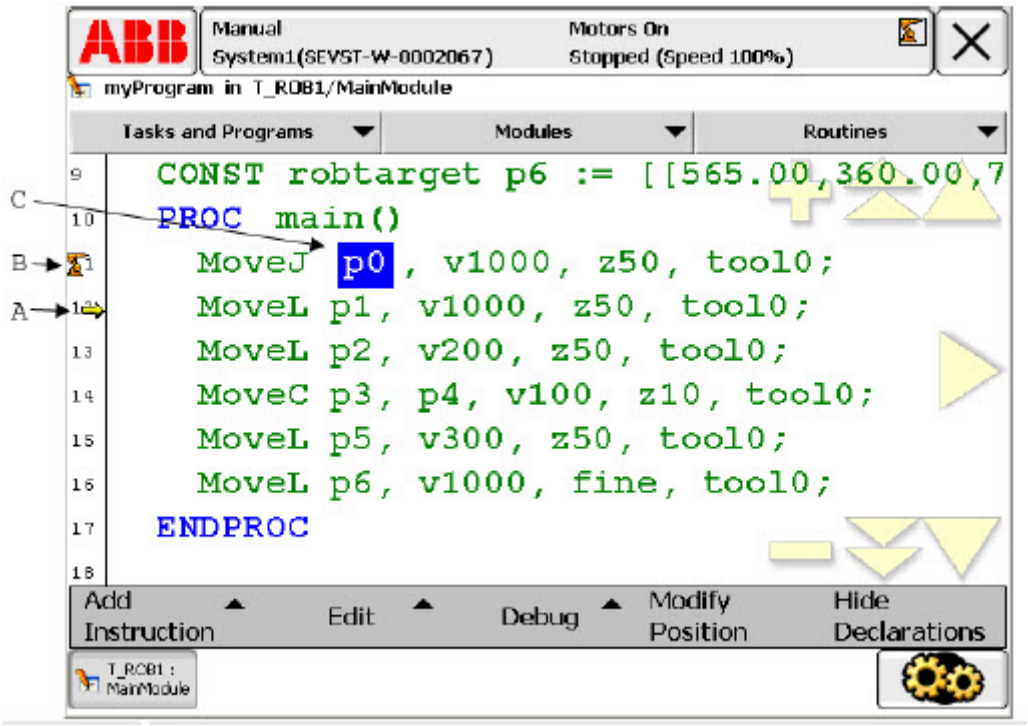
When stepping forward though the program code, a program pointer indicates the next instruction to execute and a motion pointer indicates the move instruction that the robot is performing.

When stepping backward though 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 use the speed related to the `robtarget` for both forward and backward execution.

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.



xx1100000634

A	Program pointer

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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 backward 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 backward button is pressed.
3. The second press of the backward 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 backward 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 backward 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 forward button will not move the robot, but the program pointer will move to the next instruction (`MoveL p2`).
7. The second press of the forward button will move the robot to p2 with the speed v200.

© Copyright 2004-2026 ABB. All rights reserved.

This page is intentionally left blank

2 Motion and I/O programming

2.1 Coordinate systems

2.1.1 The tool center point of the robot (TCP)

Description

The position of the robot and its movements are always related to the tool center point (TCP). 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 *Stationary TCPs on page 115*.

Related information

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

© Copyright 2004-2026 ABB. All rights reserved.

2.1.2 Coordinate systems used to determine the position of the TCP

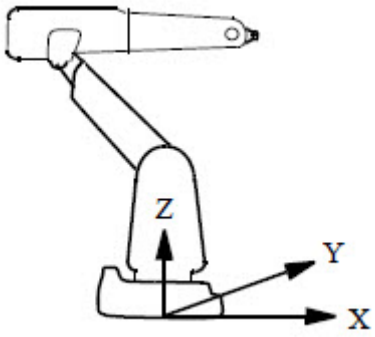
Description

The tool (TCP) position can be specified in different coordinate systems to facilitate programming and adjustment 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.

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.



xx1100000611

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.

World coordinate system

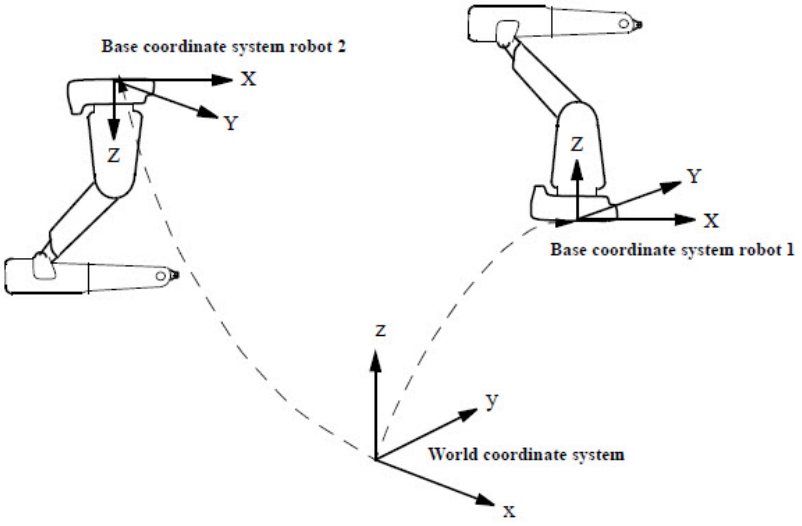
If the robot is floor mounted, programming in the base coordinate system is easy. If, however, the robot is mounted upside down (suspended), programming in the base coordinate system is more difficult because the directions of the axes are not the same as the principal directions in the working space. 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. A common world coordinate system is used in this case to enable the robot programs to communicate with one another. It can also be a good idea to use this when the positions are related to a fixed point in the workshop.

The following figure shows two robots (one suspended) with a common world coordinate system.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.



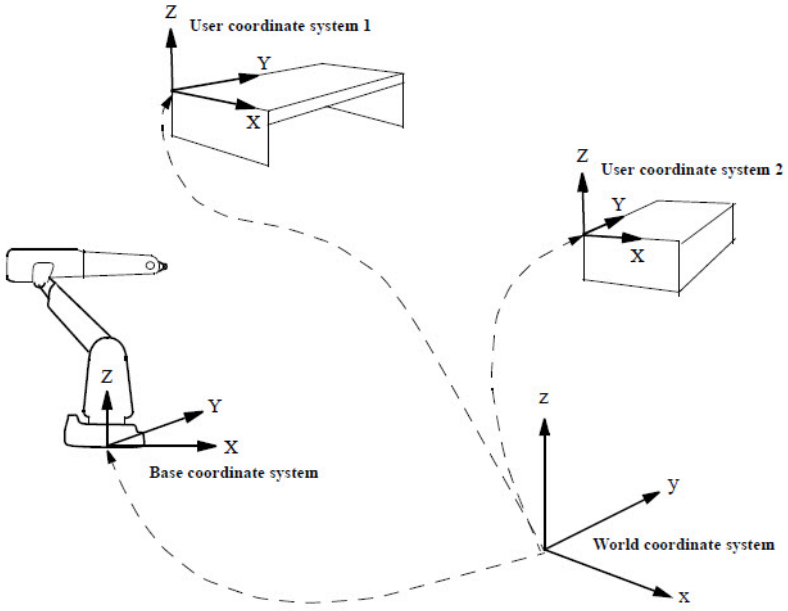
xx1100000612

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.

The following figure shows two user coordinate systems that describe the position of two different fixtures.



xx1100000613

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

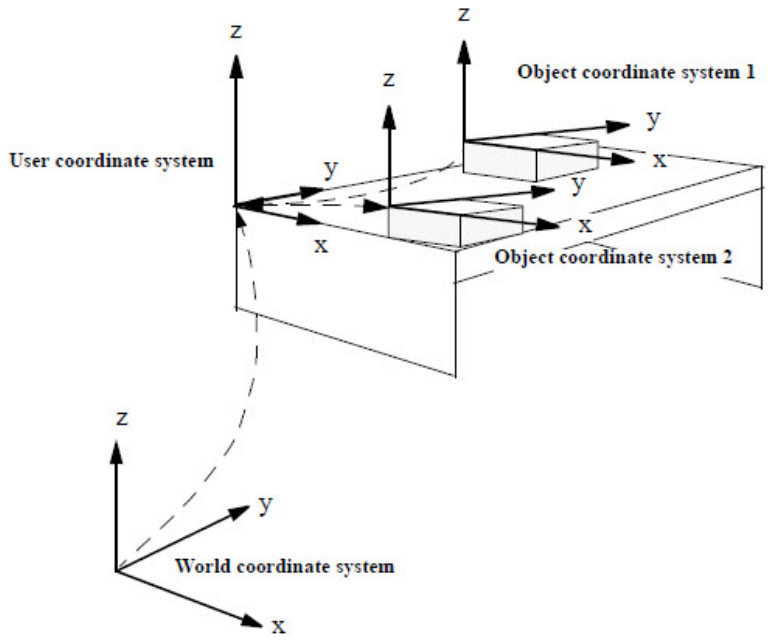
2.1.2 Coordinate systems used to determine the position of the TCP
Continued

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.

The following figure shows two object coordinate systems that describe the position of two different work objects located in the same fixture.



xx1100000614

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.

Displacement coordinate system

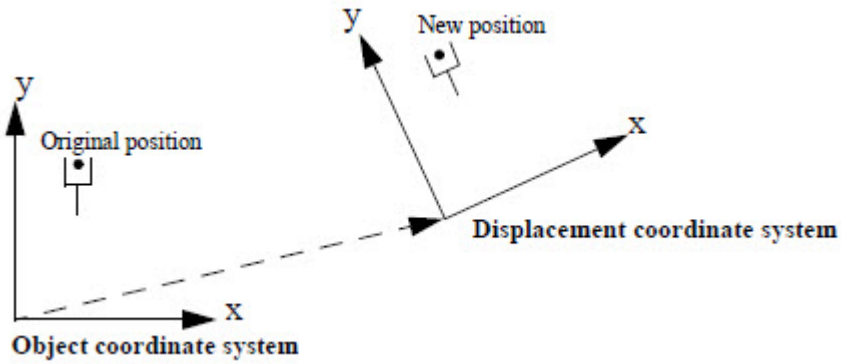
Sometimes, the same path is to be performed at several places on the same object. To avoid having to re-program all positions each time, a coordinate system, known

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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.



xx1100000615

If program displacement is active, all positions are displaced.

Coordinated additional axes

Coordination of user coordinate system

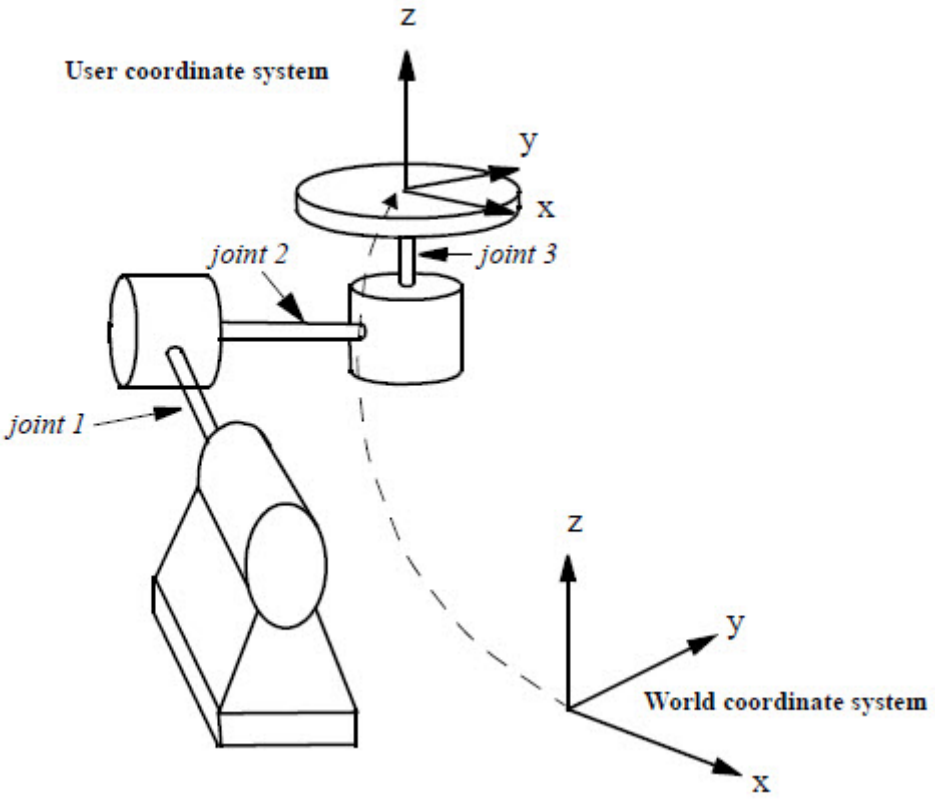
If a work object is placed on an external mechanical unit, that is moved whilst the robot is executing a path defined in the object coordinate system, a movable user coordinate system can be defined. The position and orientation of the user coordinate system will, in this case, be dependent on the axes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

The following figure shows a user coordinate system, defined to follow the movements of a 3-axis external mechanical unit.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

2.1.2 Coordinate systems used to determine the position of the TCP
Continued



xx1100000616

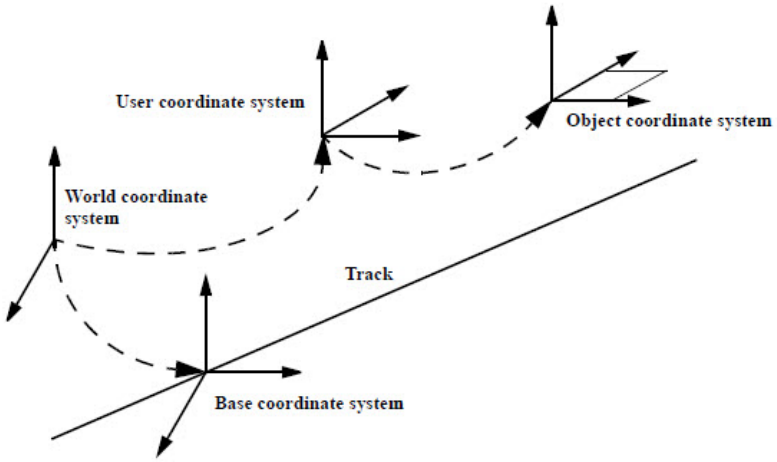
Coordination of base coordinate system

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 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.

The following figure shows coordinated interpolation with a track moving the base coordinate system of the robot.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.



xx1100000617

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.

© Copyright 2004-2026 ABB. All rights reserved.

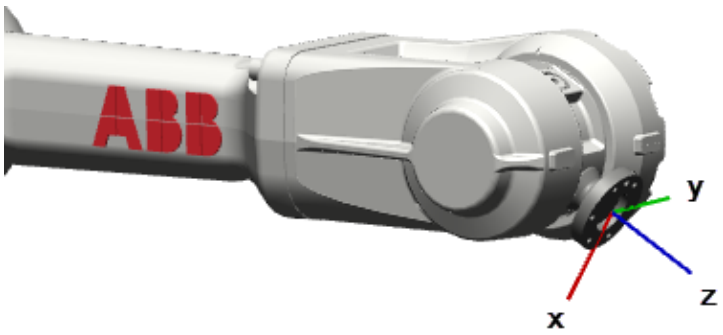
2.1.3 Coordinate systems used to determine the direction of the tool

Description

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.

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.



xx1600000580

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.

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.

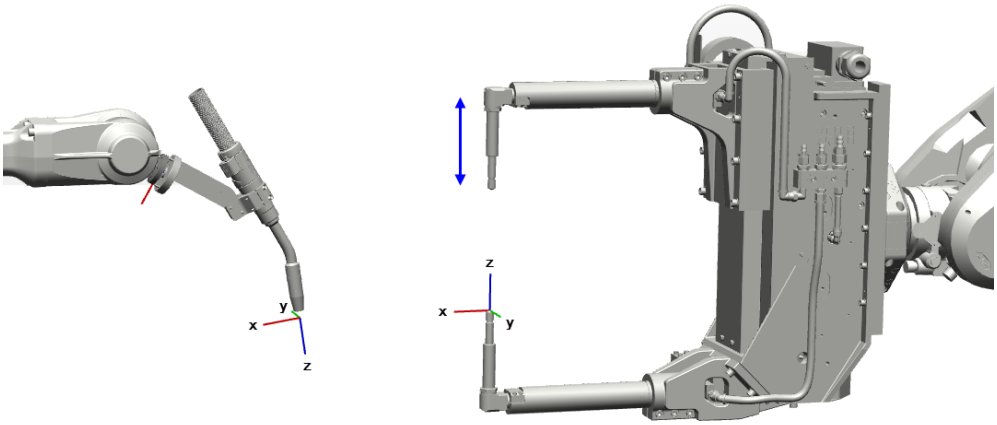
If a tool is damaged or replaced, all you have to do is redefine the tool coordinate system. The program does not normally have to be changed.

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.

The following figure shows a tool coordinate system, as usually defined for an arc-welding gun (left) and a spot welding gun (right).

Continues on next page

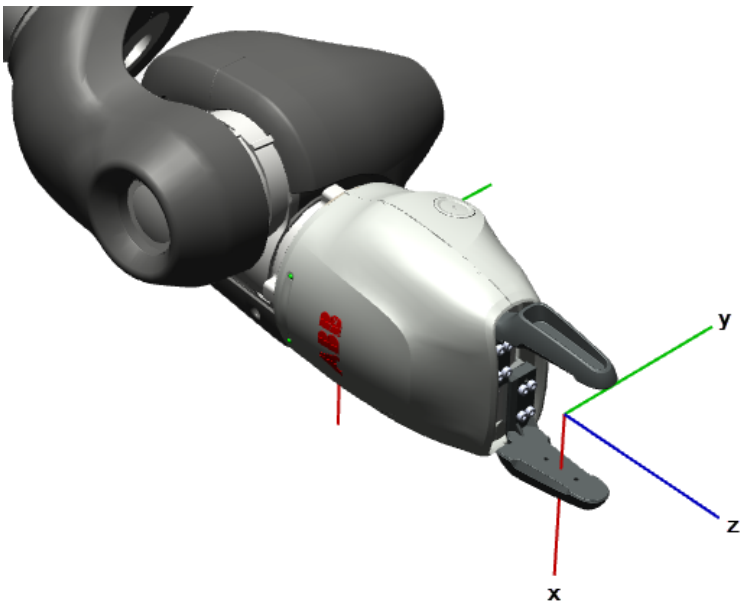
© Copyright 2004-2026 ABB. All rights reserved.



xx1600000581

The tool coordinate system is defined based on the wrist coordinate system.

The following figure shows the tool coordinate system defined relative to the wrist coordinate system, here for a gripper.



xx1600000582

Stationary TCPs

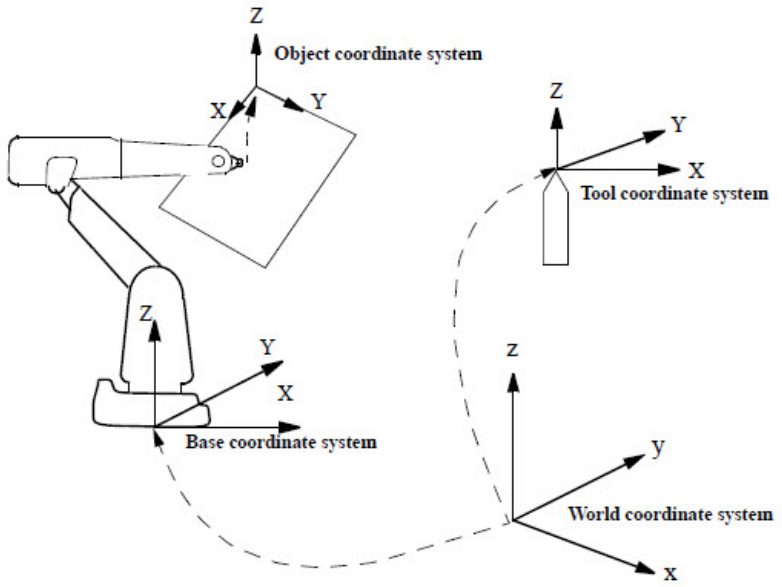
If the robot is holding a work object and working on a stationary tool, a stationary TCP is used. If that tool is active, the programmed path and speed are related to the work object held by the robot. This means that the coordinate systems will be reversed, as in the following figure.

The following figure shows a stationary TCP, where the object coordinate system is based on the wrist coordinate system.

Continues on next page

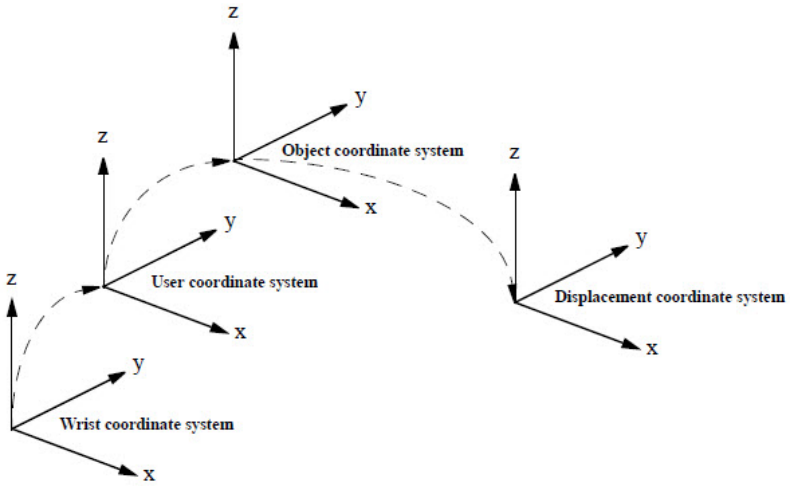
© Copyright 2004-2026 ABB. All rights reserved.

2.1.3 Coordinate systems used to determine the direction of the tool
Continued



xx1100000635

In the following figure, 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.



xx1100000636

Program displacement can also be used together with stationary TCPs

© Copyright 2004-2026 ABB. All rights reserved.

2.2 Positioning during program execution

2.2.1 Introduction

How movements are performed

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, for example stop point or fly-by 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.

WARNING

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.

Related information

	Described in:
Definition of speed	*Technical reference manual - RAPID Instructions, Functions and Data types*
Definition of zones (corner paths)	*Technical reference manual - RAPID Instructions, Functions and Data types*
Instruction for joint interpolation	*Technical reference manual - RAPID Instructions, Functions and Data types*
Instruction for linear interpolation	*Technical reference manual - RAPID Instructions, Functions and Data types*
Instruction for circular interpolation	*Technical reference manual - RAPID Instructions, Functions and Data types*
Instruction for modified interpolation	*Technical reference manual - RAPID Instructions, Functions and Data types*
Singularity	*Singularities on page 149*

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

2.2.1 Introduction

Continued

	Described in:
Concurrent program execution	*Synchronization with logical instructions on page 133*
CPU Optimization	*Technical reference manual - System parameters*

© Copyright 2004-2026 ABB. All rights reserved.

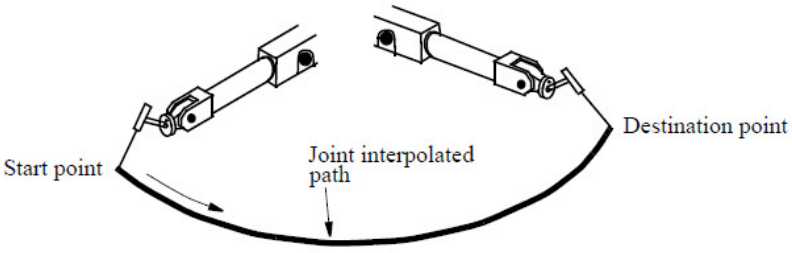
2.2.2 Interpolation of the position and orientation of the tool

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 the following figure.

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).



xx1100000637

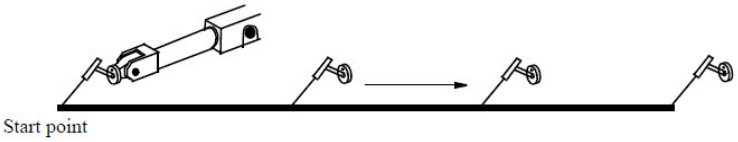
The velocity of the tool center point is expressed in mm/s (in the object coordinate system). As interpolation takes place axis-by-axis, the velocity will not be exactly the programmed value.

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.

Linear interpolation

During linear interpolation, the TCP travels along a straight line between the start and destination points.



xx1100000638

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

2.2.2 Interpolation of the position and orientation of the tool
Continued

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.

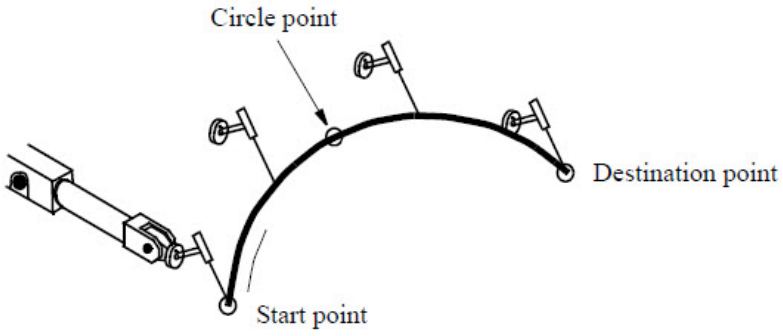
All axes are coordinated in order to obtain a path that is independent of the velocity. Acceleration is optimized automatically.

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. 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.

The following figure shows circular interpolation with a short twist for part of a circle (circle segment) with a start point, circle point and destination point.

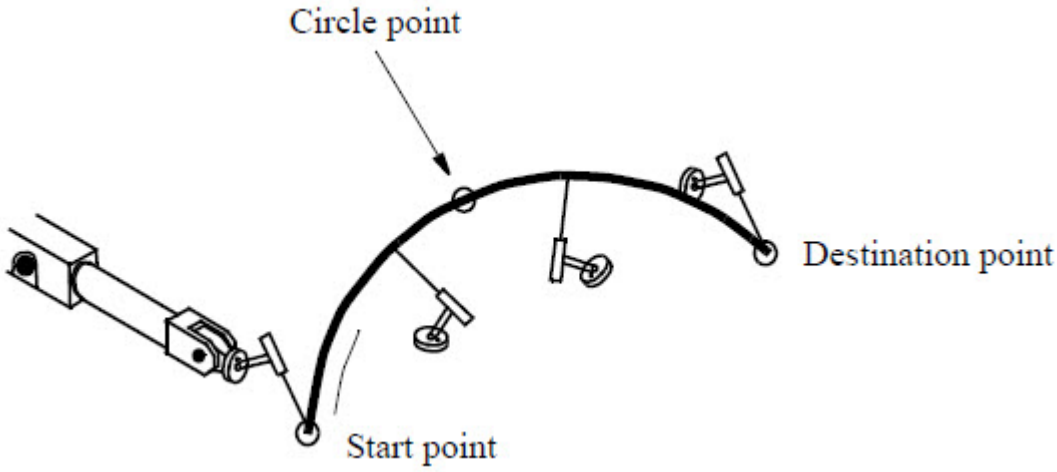


xx1100000639

However, if the orientation at the support point is programmed closer to the orientation rotated 180°, the alternative twist is selected, as shown in the following figure.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.



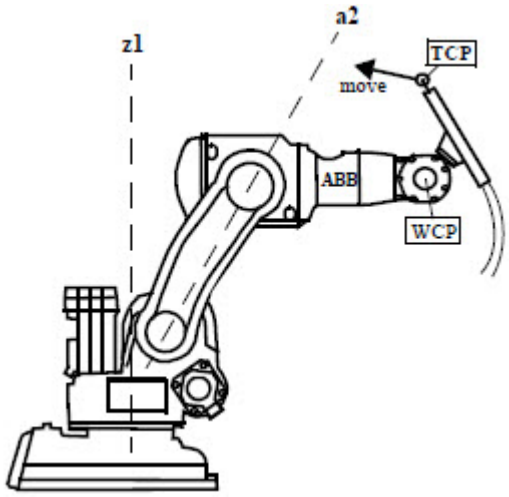
xx1100000640

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.

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.



xx1100000641

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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 is the TCP that makes the singularity, not the WCP as when `SingArea\Off` is used.

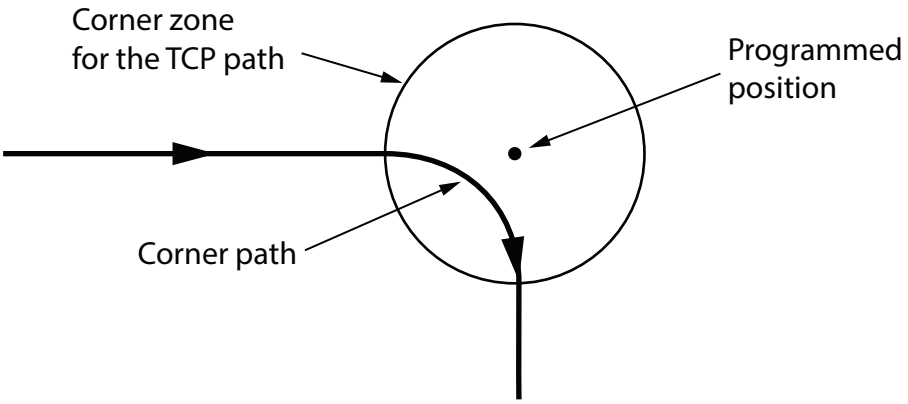
In the `SingArea\Wrist` case the orientation in the circle support point will be the same as programmed. However, the tool will not have a constant direction relative to the circle plane as for normal circular interpolation. If the circle path passes a singularity, the orientation in the programmed positions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

© Copyright 2004-2026 ABB. All rights reserved.

2.2.3 Interpolation of corner paths

Description

Corner paths 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. The beginning and end of this corner path are defined by a zone around the programmed position.



xx1100000643

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

Defining a corner path

The size of the corner zone is defined in the data type `zonedata`, which is used as an argument in move instructions. For more information, see `zonedata` in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Smooth transitions between different speeds

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

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

Fine-tuning the 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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

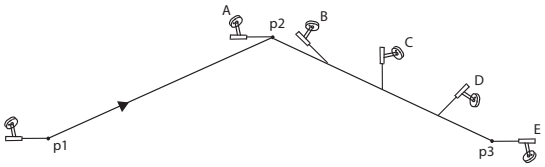
2.2.3 Interpolation of corner paths
Continued

Corner paths with reorientation

A fly-by point can have one zone size for TCP position (position zone) and one zone size for tool reorientation and movement of additional axes (reorientation and additional axis zone).

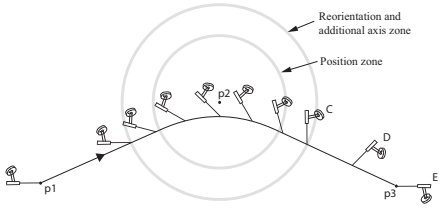
The reorientation and additional axis zone can be defined similar to the position zone. The reorientation and additional axis zone is usually set larger than the position zone. In this case, the reorientation will start interpolating towards the orientation of the next position before the corner path starts. The reorientation will then be smoother.

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 the following figures. Note that the orientation outside the reorientation and additional axis zone is the same for both cases (position C - E) but the orientation inside the zone changes more smoothly than the programmed positions.



xx1100000647

If three positions with different tool orientations are programmed as stop points, program execution will look like this.



xx1100000648

If the middle position is a fly-by point, program execution will look like this.

The reorientation and additional axis zone is normally expressed in mm. In this way, you can determine directly where on the path the 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.

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

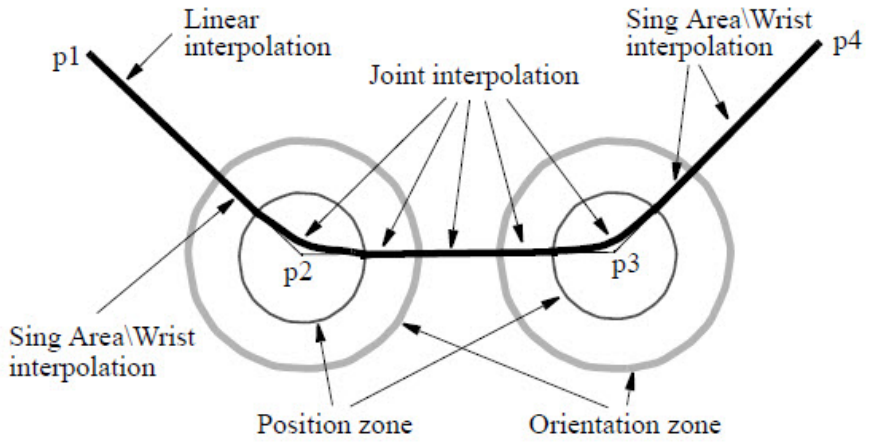
Corner paths when changing the interpolation method

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*).

The following figure shows an example of 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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.



xx1100000649

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.

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.

Size reduction of programmed corner 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. As default, the robot reduces the size of the zone to half the distance from one programmed position to another.

Behavior with default zone settings

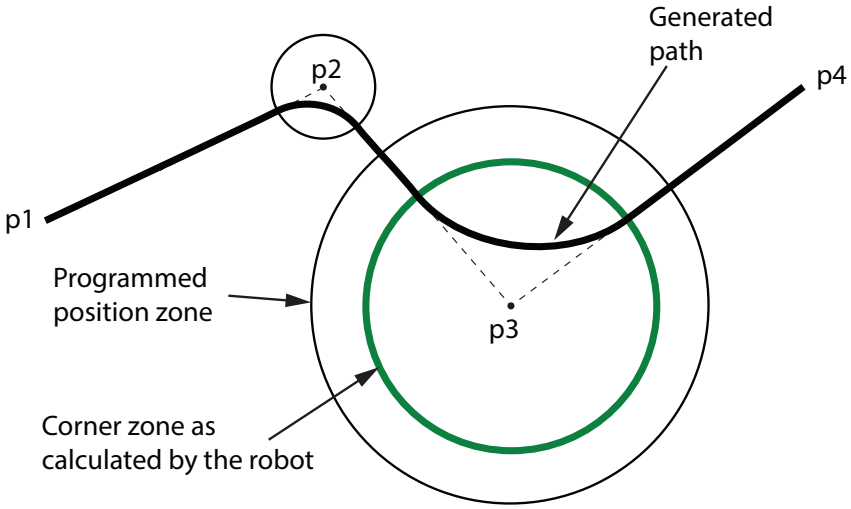
With default settings, programmed zones that reach over half the distance to a neighboring point result in reduced zone (see the following figure). The same zone radius is by default used for inbound path and outbound path to or from a programmed position.

The following figure shows an example of interpolation with too large position zones. The zone around p3 is larger than half the distance from p2 to p3. Thus, the robot reduces the size of the zone to make it equal to half the distance from p2 to p3, thereby generating a symmetrical corner path within the zone.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

2.2.3 Interpolation of corner paths
Continued



xx1800000780

Both the position zone and the reorientation and additional axis zone can overlap. As soon as one of these corner path zones is too big, that zone is reduced.

Note

The corner zone can be allowed to reach more than half the distance to a neighboring fine point by changing the system parameter *Relative zone size with finepoint* (topic *Motion*, type *Motion Planner*).

The corner zone size can be different for inbound and outbound path by setting the system parameter *Allow asymmetric zones* (topic *Motion*, type *Motion Planner*).

See *Technical reference manual - System parameters*.

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.

To suppress warnings from corner path failures, use the instruction `CornerPathWarning`.

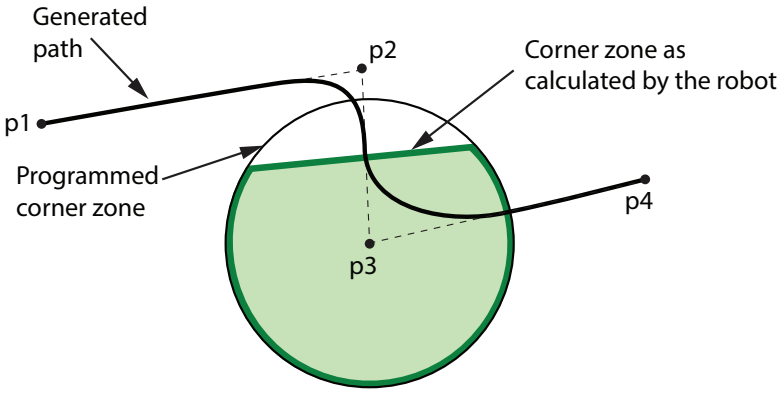
Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Note

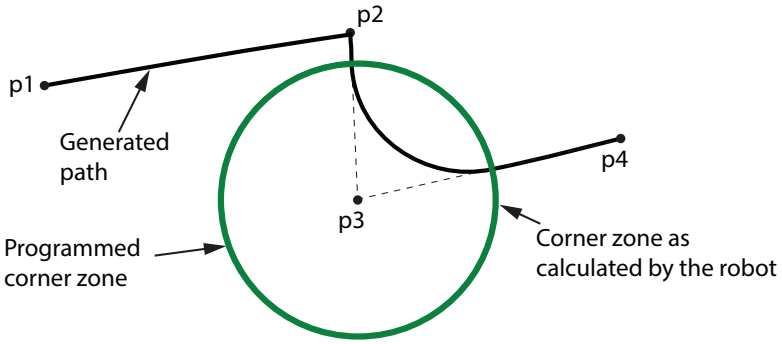
If the fly-by point is changed into a finepoint, the path after the recalculated point can also be affected.

Path when p2 is a fly-by point:



xx1800002286

Path when p2 is a finepoint:



xx1800001538

© Copyright 2004-2026 ABB. All rights reserved.

2.2.4 Independent axes

Description

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.

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.

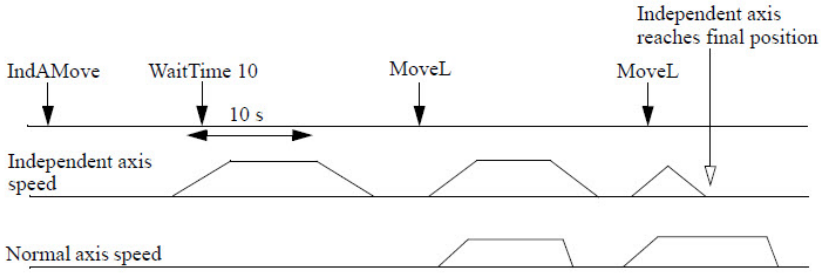
Note that a mechanical unit may not be deactivated when one of its axes is in independent mode.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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.



xx1100000652

Jogging

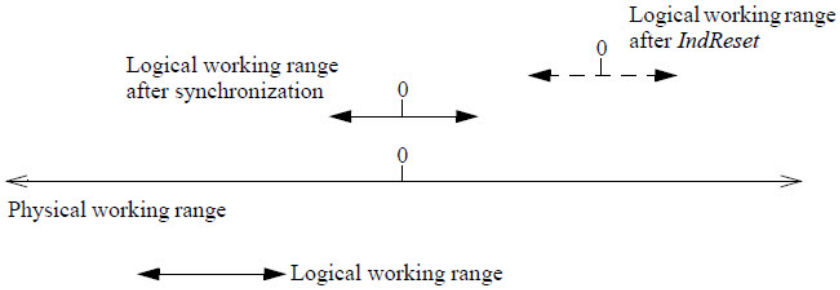
Axes in independent mode cannot be jogged. If an attempt is made to execute the axis manually, the axis does not move and an error message is displayed. Execute an `IndReset` instruction or move the program pointer to main, in order to leave the independent mode.

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 the following figure.



xx1100000653

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

2.2.4 Independent axes
Continued

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` 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` 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.

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 1600, 2600 and 4600 models (not for ID versions). 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.

© Copyright 2004-2026 ABB. All rights reserved.

2.2.5 Soft Servo

Description

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).

Softness

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

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

Note

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.

© Copyright 2004-2026 ABB. All rights reserved.

2.2.6 Stop and restart

Stopping movement

A movement can be stopped in three different ways:

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

  For a stiff stop, the robot will stop in a shorter time than for the soft 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.
- 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.

Starting movement

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`.

© Copyright 2004-2026 ABB. All rights reserved.

2.3 Synchronization with logical instructions

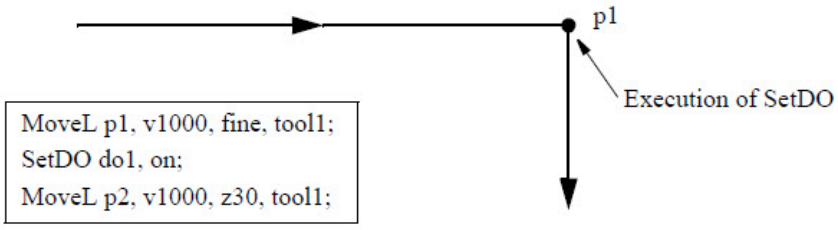
Logical instructions

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.

Sequential program execution at stop points

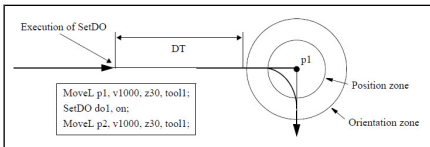
If a positioning instruction has been programmed as a stop point, the next instruction is not executed until the robot and the additional axes have come to a standstill, that is, when the programmed position has been reached.



xx1100000654

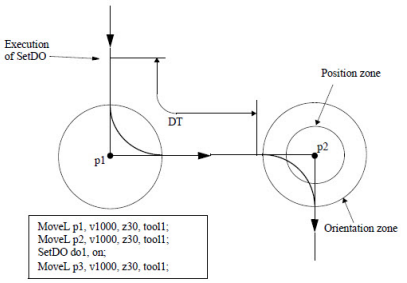
Sequential program execution at fly-by points

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



xx1100000655

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



xx1100000656

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.

Continues on next page

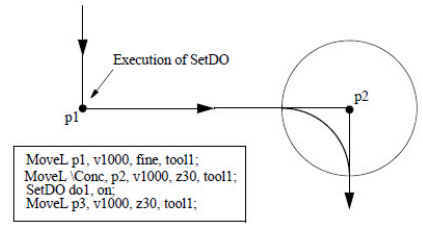
© Copyright 2004-2026 ABB. All rights reserved.

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 I/O device).

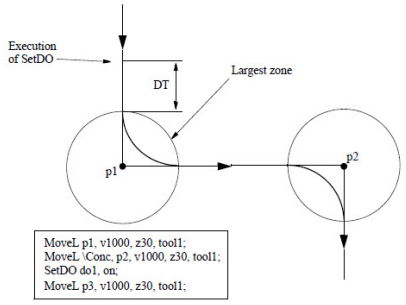
When a positioning instruction with the argument `\Conc` is executed, the following logical instructions are also executed (in sequence).

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).



xx1100000657

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).



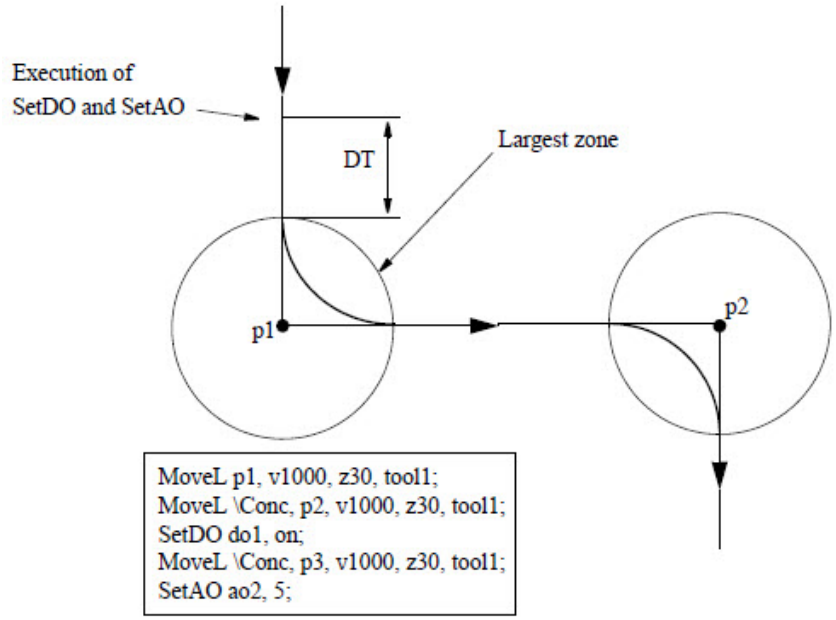
xx1100000658

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.



xx1100000659

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`.

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 +/- 2 ms.

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Related information

	Described in:
Positioning instructions	*Motion on page 56*
Definition of zone size	*Technical reference manual - RAPID Instructions, Functions and Data types*

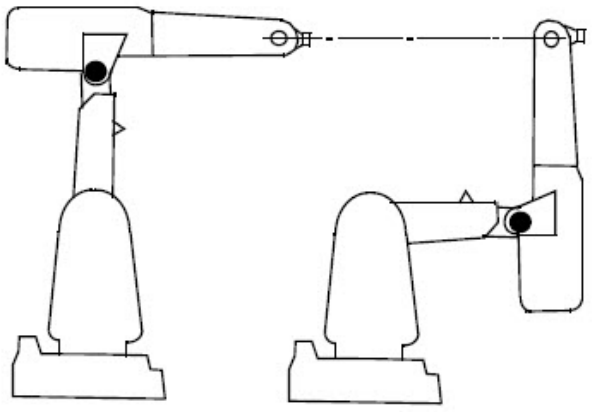
© Copyright 2004-2026 ABB. All rights reserved.

2.4 Robot configuration

Different types of robot configurations

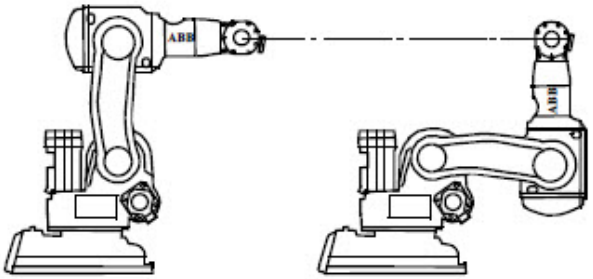
It is usually possible to attain the same robot tool position and orientation in several different ways, using different sets of axis angles. We call these different robot configurations.

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, as shown in the following figure where the configuration to the right is attained by rotating the arm backwards. Axis 1 is rotated 180 degrees.



xx1100000660

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. In the following figure, two different arm configurations are used to attain same position and orientation. The axis 1 angle is the same for both configurations. The configuration on the right is attained by rotating the lower arm forward and the upper arm backwards.



xx1100000661

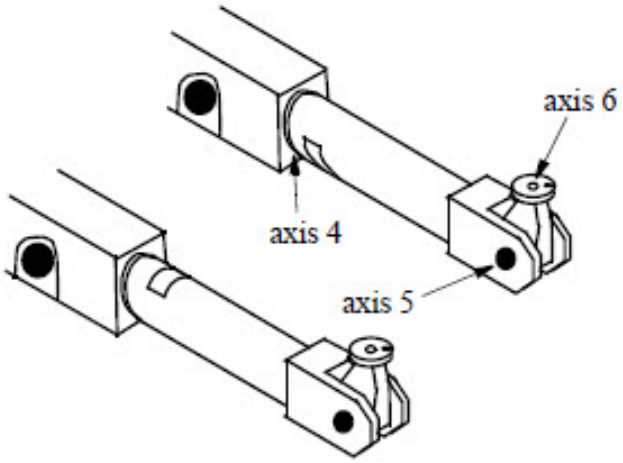
This can also be achieved by turning the front part of the robot upper arm (axis 4) upside down while rotating axes 5 and 6 to the desired position and orientation.

The following figure shows 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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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).

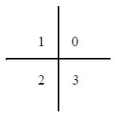
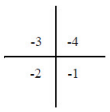


xx1100000662

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 the following figures.

 xx1100000663	 xx1100000664
Quarter revolution for a positive joint angle	Quarter revolution for a negative joint angle

Configuration control and monitoring

To achieve a well-defined robot movement it is usually desirable to let the robot attain the same configuration during program execution as the one specified in the program. To do this, configuration control and monitoring with `ConfL\On` or `ConfJ\On` should be used. The configuration monitoring 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 the configuration monitoring is active with `ConfL\On`, a verification is made in advance to see if it is possible to achieve the programmed configuration. If it is not possible the program is stopped. When the movement is finished (in a zone or in a finepoint), it is also verified that the robot has reached the programmed configuration. If the configuration is different the program is stopped. For a detailed description of the configuration data for a specific robot type see the data type

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

`confdata` in *Technical reference manual - RAPID Instructions, Functions and Data types*.

During axis-by-axis movement the robot always moves to the programmed configuration when `ConfJ\On` is used. No supervision of the axis movements is performed when configuration control is enabled. Depending on the difference between the start point configuration and the end point configuration a large movement, of especially the wrist, can be a result. If the configuration supervision is not active, the robot moves to the specified position and orientation with the configuration that has the closest joint values compared to the start point.

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 offline 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.

Detailed information for ConfJ

`MoveJ` with `ConfJ\Off`:

- The robot is moved to the programmed position, with an axis position that is closest to the axis position of the start. This means that the `confdata` in the instruction is not used. No configuration supervision is done.

`MoveJ` with `ConfJ\On`:

- The robot is moved to the programmed position, with an axis position such that the corresponding configuration is equal to or close to the programmed configuration in the `confdata`.
- If a program displacement or path correction is active, the risk is that the programmed configuration differs from the original position. As a result the robot can perform large movements of the wrist axis to achieve the programmed configuration.

Detailed information for ConfL

`MoveL` with `ConfL\Off`:

- The robot is moved along a straight line to the programmed position, with an axis position that is closest to the axis position of the start. This means that the `confdata` in the instruction is not used and no configuration supervision is done.

`MoveL` with `ConfL\On`:

- First the end position is calculated in joints, using the programmed `confdata` to determine the solution. Then the joint values for the configuration axes in the end position are compared to the corresponding axes for the start position.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

If the new configuration data is OK compared to the start point the movement will be permitted. In other cases the robot will stop in the start position with an error message. For details about the allowed configuration error for different robot types, see description of `confdata`, *Technical reference manual - RAPID Instructions, Functions and Data types*.

- If no error was reported before the movement started the system will check the configuration again when the movement is finished. If it is not the same as the programmed configuration the program will be stopped.

Related information

	Described in:
Definition of robot configuration	*Technical reference manual - RAPID Instructions, Functions and Data types*
Activating/deactivating the configuration supervision	*Motion on page 56*

© Copyright 2004-2026 ABB. All rights reserved.

2.5 Robot kinematic models

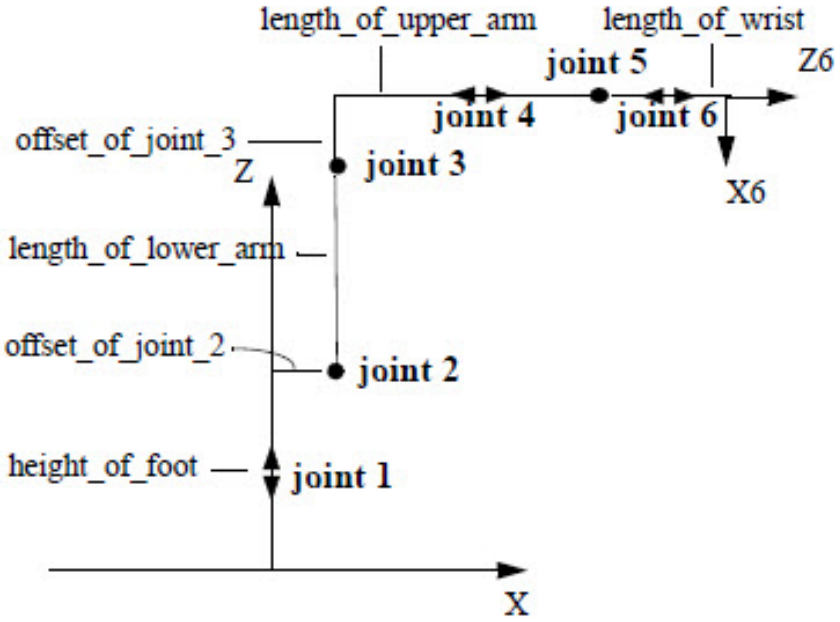
Robot kinematics

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.

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.

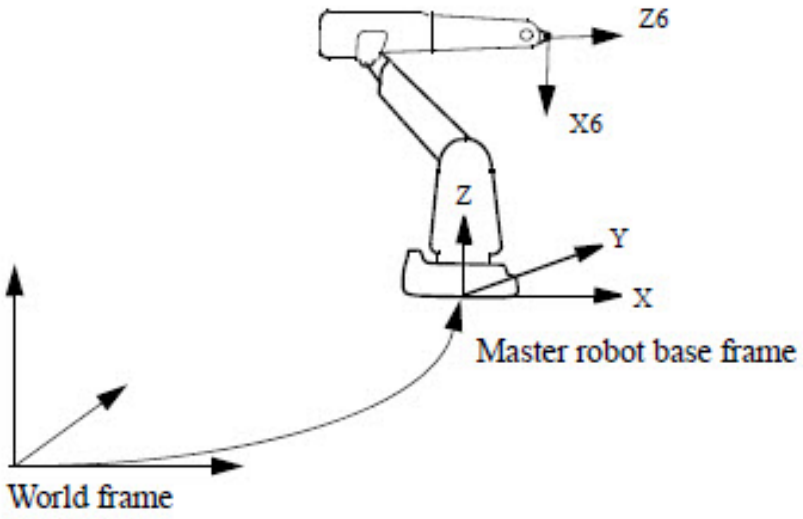


xx1100000666

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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

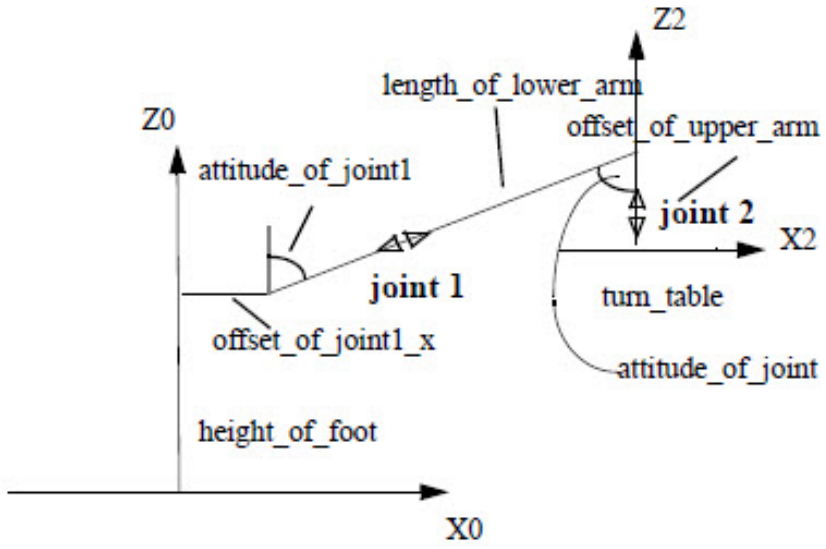


xx1100000667

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.

The following figure shows a kinematic structure of an ORBIT 160B robot using predefined model.

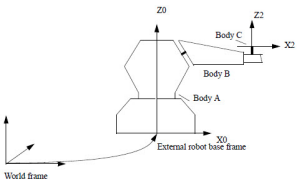


xx1100000668

Continues on next page

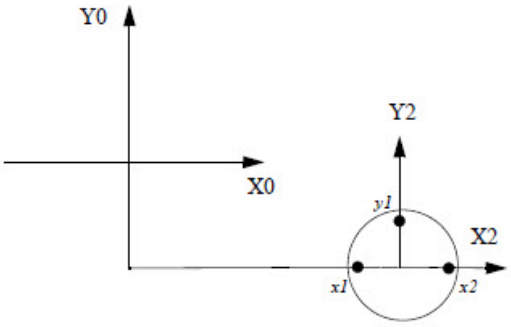
© Copyright 2004-2026 ABB. All rights reserved.

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



xx1100000669

Base frame of an ORBIT_160B robot.



xx1100000670

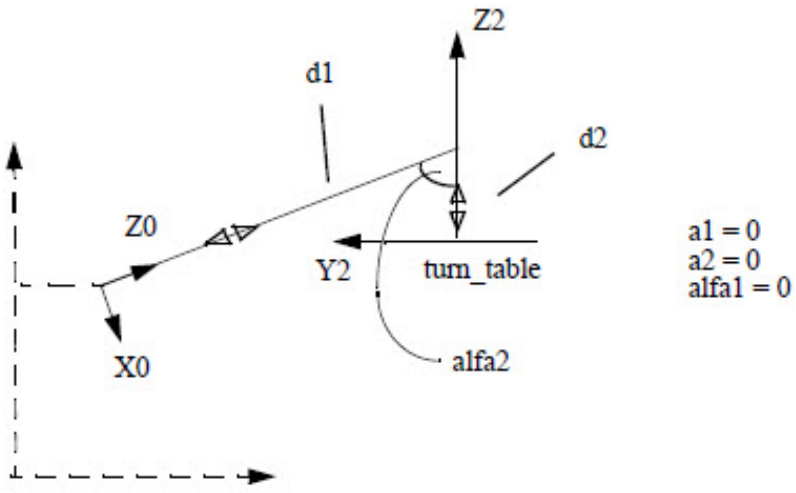
Reference points on turntable for base frame calibration of an ORBIT_160B robot in the home position using pre-defined model.

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*, by John J. Craigh (Addison-Wesley 1986).

The following figure shows a kinematic structure of an ORBIT 160B robot using general kinematics model.



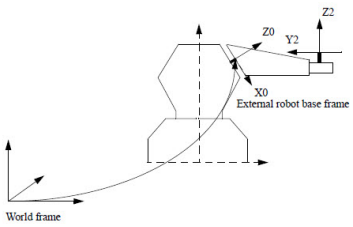
xx1100000671

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

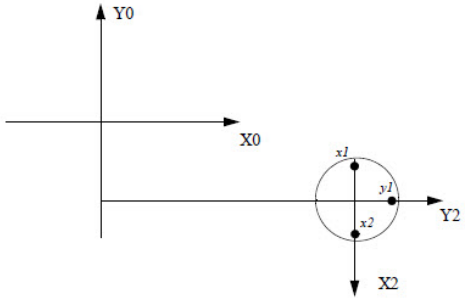
2.5 Robot kinematic models
Continued

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



xx1100000672

Base frame of an ORBIT_160B robot using general kinematics model.



xx1100000673

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

Related information

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

© Copyright 2004-2026 ABB. All rights reserved.

2.6 Motion supervision/collision detection

Introduction

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 (option *Collision Detection*).

The collision detection may trig 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 robot is stopped as quickly as possible. By default, the torques are reversed and the brakes are applied, but it is also possible to configure a stop that does not use the brakes.

The motion supervision is by default active only 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. The system parameter *Collision detection at standstill* enables the detection of any collision even at standstill, see *Technical reference manual - System parameters*.

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 *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 *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%.

Continues on next page

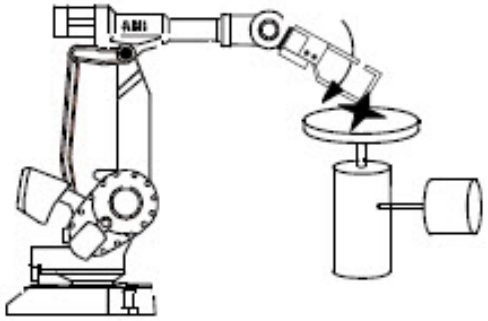
© Copyright 2004-2026 ABB. All rights reserved.

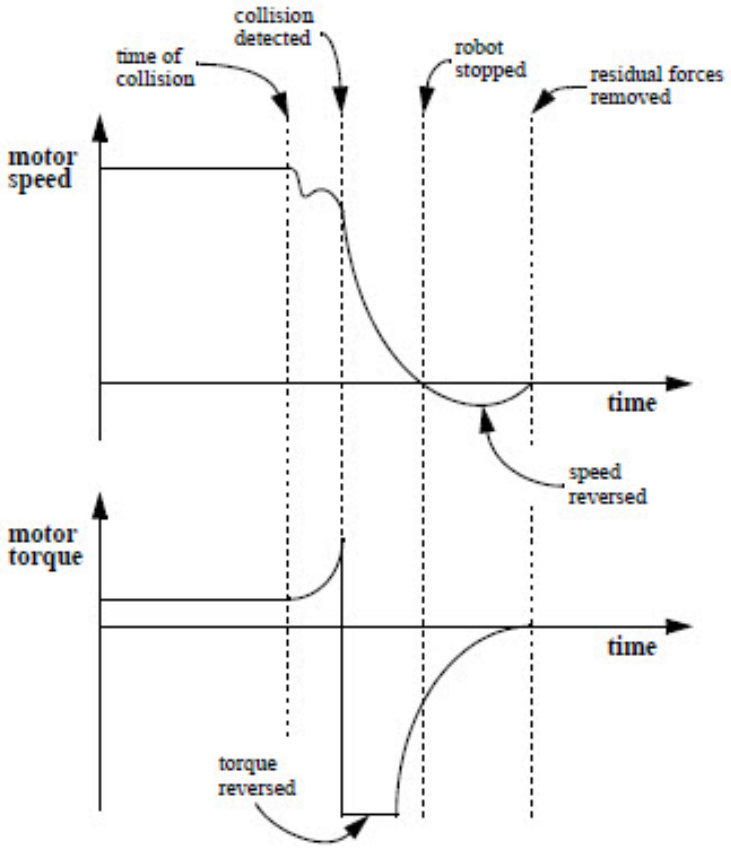
2.6 Motion supervision/collision detection
Continued

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.





xx1100000674

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*.

Modifying motion supervision

To modify modify the motion supervision, open the tasks.

1. 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.
2. 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

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

the option *Collision Detection* installed, path supervision only affects the robot in auto and manually full speed mode.

3 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 - Controller software IRC5*.

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*.

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Related information

	Described in:
RAPID instruction MotionSup	*Motion on page 56*
System parameters for tuning	*Technical reference manual - System parameters*
Motion supervision I/O signals	*Technical reference manual - System parameters*
Collision detection at standstill	*Technical reference manual - System parameters*
Independent collision stop without brake	*Technical reference manual - System parameters*
Load identification	*Operating manual - IRC5 with FlexPendant*

© Copyright 2004-2026 ABB. All rights reserved.

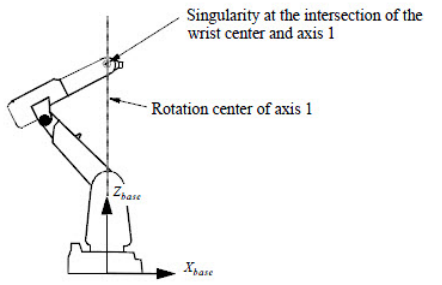
2.7 Singularities

Description

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.

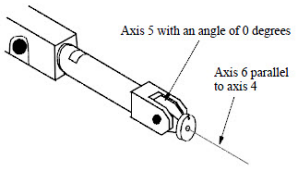
Generally speaking, a robot has two types of singularities; arm singularities or wrist singularities.

Arm singularities are all configurations where the wrist center (the intersection of axes 4, 5, and 6) ends up directly above axis 1. 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.



xx1100000675

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



xx1100000676

Wrist singularity occurs when axis 5 is at 0 degrees.

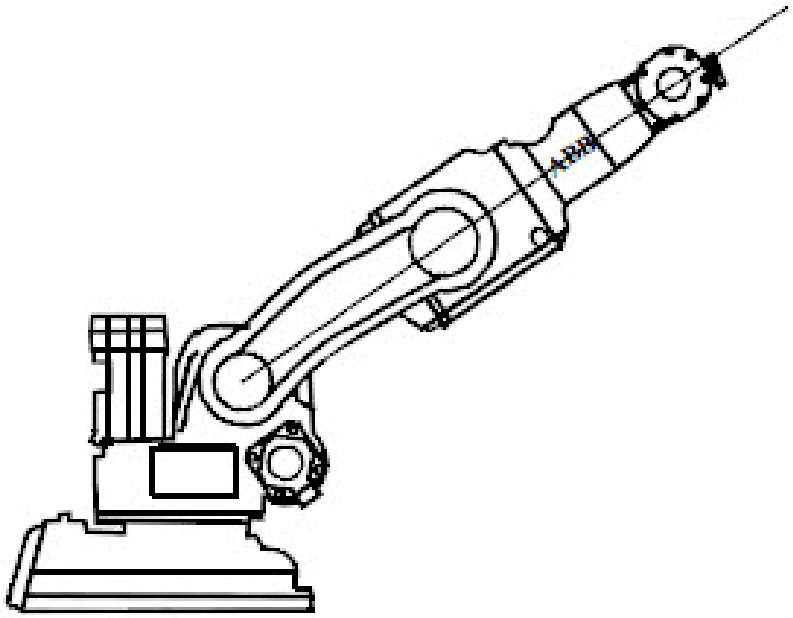
Singularity points of robots without parallel rod

Robots without parallel rod (serial link robots) have the wrist singularity and the arm singularity, and in addition, they have a third kind of singularity. This singularity

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

occurs at robot positions where the wrist center and the rotation centers of axes 2 and 3 are all in a straight line, see the following figure.



xx1100000677

Singularity points of serial link robots with 7 axes

Serial link robots with 7 axes, such as the IRB 14000/14050, have all the singularities of 6-axis robots but also have two additional singularities.

The first of these is when axis 2 has an angle equal to zero so that axis 1 and axis 7 are on the same line. This is similar to the wrist singularity.

The second singularity is related to the calculation of the arm angle and varies depending on which reference direction is configured, since the singularity occurs when the WCP is on the reference direction. For more information, see *Arm-Angle Reference Direction* in *Technical reference manual - System parameters*.

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 (`SingArea\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. To avoid the

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

re-configuration, the first position on the other side of the singularity should be programmed with an orientation that makes the re-configuration 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.

For robots with 7 axes or more, the robot may be able to automatically adjust the arm angle in order to avoid some singularities, see *Operating manual - IRC5 with FlexPendant*.

Jogging through singularities

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

During linear interpolation, the robot cannot pass singular points. Moving close to singularities may lead to a lower TCP speed while having large joint movements.

Related information

	Described in:
Controlling how the robot is to act on execution near singular points	*Technical reference manual - RAPID Instructions, Functions and Data types*

© Copyright 2004-2026 ABB. All rights reserved.

2.8 Optimized acceleration limitation

Description

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).

Load data

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

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.

© Copyright 2004-2026 ABB. All rights reserved.

2.9 World Zones

Description of World Zones

When using world zones (option *World Zones*), 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.

WARNING

For safety reasons, this software shall not be used for protection of personnel. Use hardware protection equipment for that.

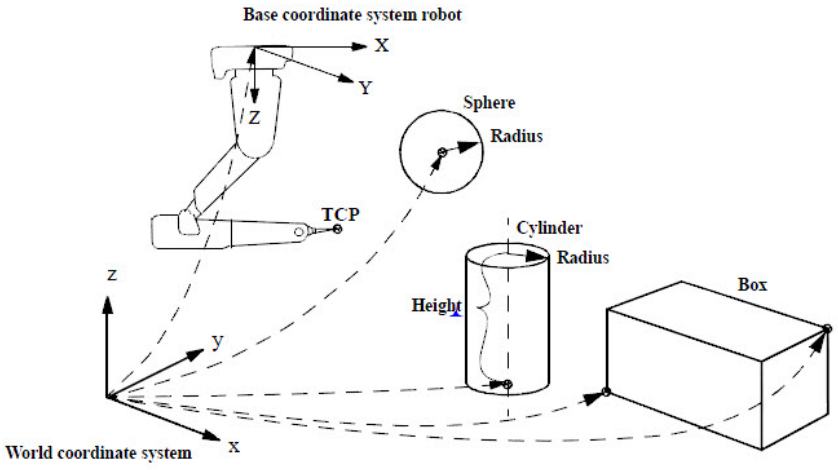
Using World Zones

Use 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.

Definition of World Zones in the world coordinate system

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



xx1100000678

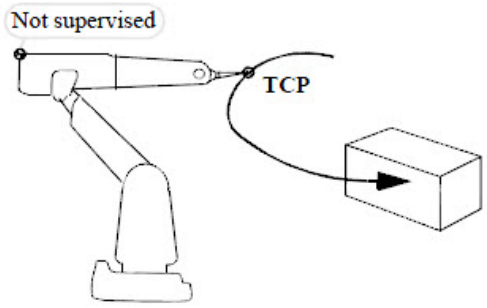
Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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.

Supervision of the robot TCP



xx1100000679

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.

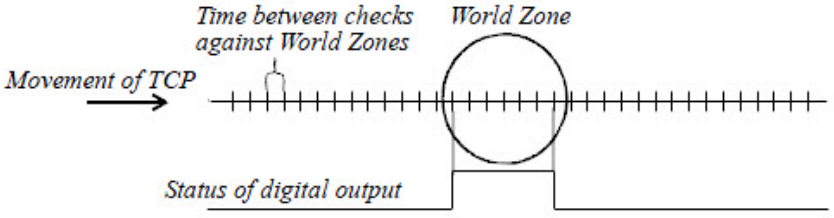
Stationary TCPs

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.

Actions

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.



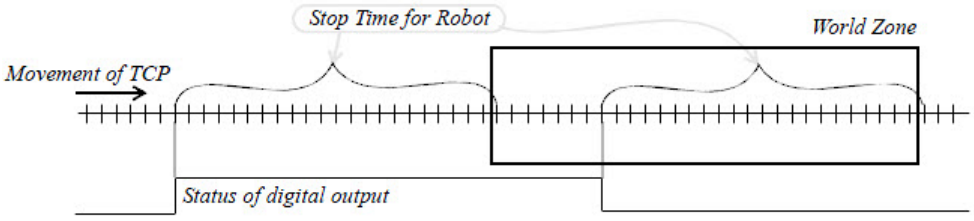
xx1100000680

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Set a digital output before the TCP reaches a World Zone

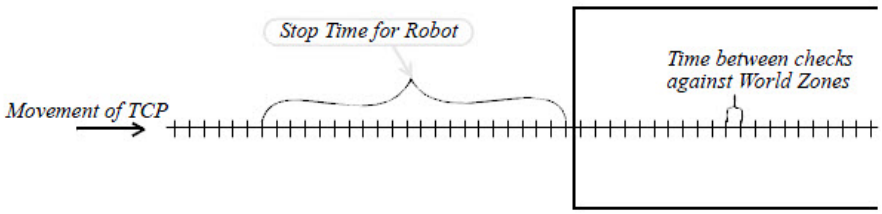
This action sets a digital output before the TCP reaches a World Zone. It can be used to stop the robot just inside a World Zone.



xx1100000681

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.



xx1100000682

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.

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.

Make sure that the zones are a bit larger than the minimum size.

Continues on next page

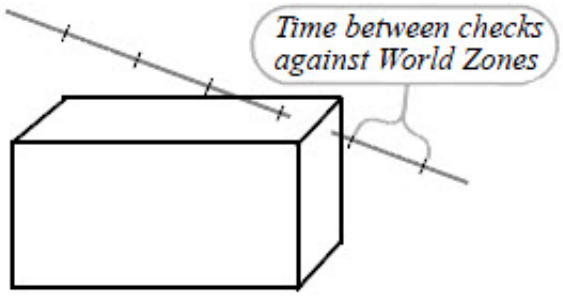
© Copyright 2004-2026 ABB. All rights reserved.

Min. size of zone for used path_resolution and max. speed			
Speed / Resol.	1000 mm/s	2000 mm/s	4000 mm/s
1	40 mm	80 mm	160 mm
2	80 mm	160 mm	320 mm
3	120 mm	240 mm	480 mm

xx1100000683

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.



xx1100000684

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`.

Maximum number of World Zones

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

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

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 (*PP to Main* or *PP to routine*).

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 restart 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.

Related information

Motion and I/O principles	Coordinate Systems
Data types: • `wztemporary` • `wzstationary` • `shapedata`	*Technical reference manual - RAPID Instructions, Functions and Data types*
Instructions: • `WZBoxDef` • `WZSphDef` • `WZCylDef` • `WZHomeJointDef` • `WZLimJointDef` • `WZLimSup` • `WZDOSet` • `WZDisable` • `WZEnable` • `WZFree`	*Technical reference manual - RAPID Instructions, Functions and Data types*

© Copyright 2004-2026 ABB. All rights reserved.

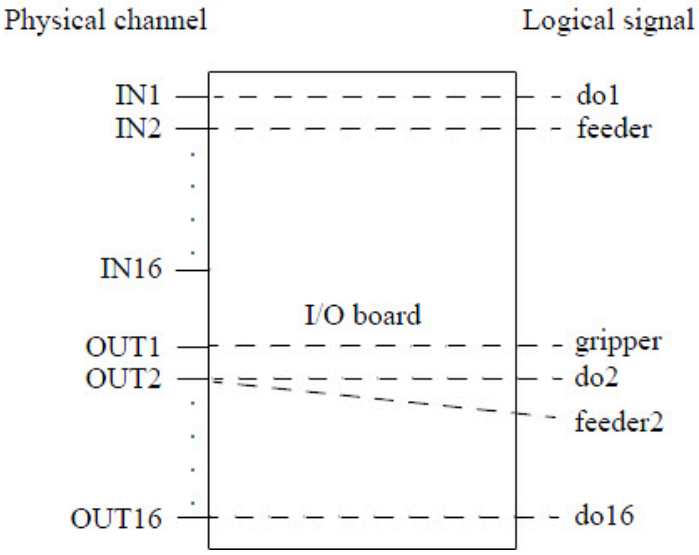
2.10 I/O principles

Description

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 signals must always be used during programming.

A physical channel can be connected to several logical signals, but can also have no logical connections, as shown in the following figure.

To be able to use an I/O board, its channels must be given logical signals. In the following figure, the physical output 2 is connected to two different logical signals. IN16, on the other hand, has no logical signal and thus cannot be used.



xx1100000685

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 for the robot controller). The characteristics, filter times and scaling between programmed and physical values, are defined in the system parameters.

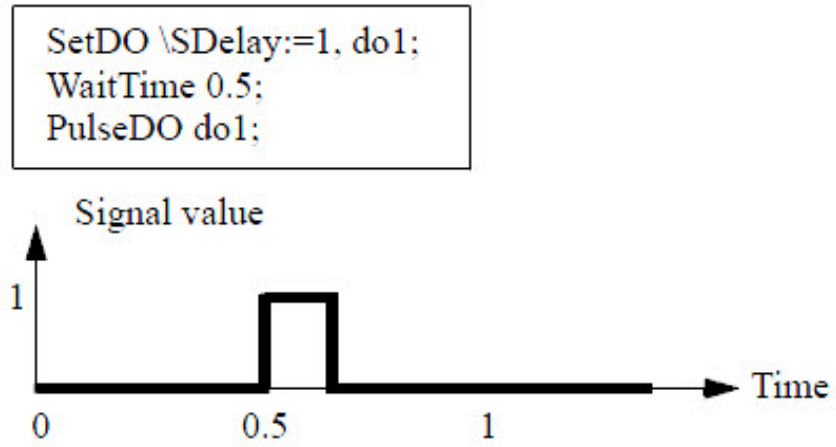
When the power supply to the robot is switched on, all signals are set to zero. They are not, however, affected by emergency stops or similar events.

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, the 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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

The following figure shows an example where the instruction `SetDO` is not carried out at all because a new command is given before the time delay has elapsed.



xx1100000686

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.

For example, if a function is connected to a signal called *do1* and the program code is as follows:

```
SetDO do1,1;
SetDO do1,0;
```

The signal will first go to high (1) and then low (0) in a few milliseconds. In this case the interrupt may be lost. To be sure that the interrupt is not lost, make sure that the output is set before resetting it.

For example:

```
SetDO do1,1;
WaitDO do1 ,1;
SetDO do1,0;
```

With this method, no interrupts will be lost.

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.

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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

- 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`.

Examples	Description

`di2=di1`
`di3=di2`
`do4=di2`
	If `di1` changes, then `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` are set to 1.

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.

Related information

	Described in
Definition of I/O boards and signals	*Technical reference manual - System parameters*
Instructions for handling I/O	*Input and output signals on page 65*
Manual manipulation of I/O	*Operating manual - IRC5 with FlexPendant*

© Copyright 2004-2026 ABB. All rights reserved.

3 Best practice for programming robots

Introduction

This section outlines best practices for programming robots to achieve high performance and reliability. It provides guidelines on task prioritization, RAPID instruction execution, performance optimization, and mathematical efficiency. The recommendations are based on practical experience and aim to help developers structure their programs effectively.

Aim for event driven design

To achieve high responsiveness in both motion execution and background supervision, use an event-driven design instead of continuous polling loops.

Avoid continuous polling loops

A continuous polling loop repeatedly reads inputs, evaluates system states, and determines required actions. This approach can:

- Increase CPU utilization
- Reduce application responsiveness
- Scale poorly when monitoring many signals
- Affect motion task performance and determinism

Use event-driven mechanisms whenever possible so that code is executed only when a relevant condition changes.

Use interrupts and TRAP routines

RAPID provides interrupt handling and TRAP routines for event-driven programming. Execution can be triggered immediately when a signal or variable changes.

Typical applications include:

- Monitoring digital and analog signals
- Detecting changes to PERS variables
- Supervising process or safety conditions

Benefits include:

- Fast response to events
- Low CPU overhead
- Separation of monitoring and action logic

Use available system features

When several conditions must be monitored, avoid implementing complex evaluation loops in RAPID. Instead, use available system features such as:

- TRAP routines for event-driven evaluation
- I/O cross connections for logical relationships in the I/O system
- CyclicBool for periodic evaluation of Boolean expressions

Using these features can reduce application complexity and improve performance.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Continued

Keep TRAP routines short

TRAP routines execute in interrupt context and should be kept short and efficient.

The TRAP routine should:

- Perform only the required checks
- Avoid extensive calculations
- Avoid blocking operations
- Avoid complex program flow

If additional processing is required, transfer the work to normal program execution.

Communicate with normal program execution

When more extensive evaluation or actions are required, use synchronization mechanisms to communicate between the TRAP routine and normal execution.

Example:

```
...rapid
WaitTestAndSet flag;
...
```

This approach allows the TRAP routine to execute quickly while the main program performs the required processing.

Protect shared data

Special consideration is required when shared data is accessed both from a TRAP routine and from normal program execution.

To protect critical sections, interrupts can be temporarily disabled:

```
...rapid
Disable;
! Evaluate shared state safely
Enable;
...
```

Keep protected sections as short as possible to maintain application responsiveness.

Recommended practices

- Use event-driven logic instead of polling loops.
- Use interrupts and TRAP routines for event handling.
- Keep TRAP routines short and efficient.
- Use I/O cross connections and CyclicBool where applicable.
- Perform complex processing in normal program execution.
- Protect shared data when required.

By using an event-driven design, RAPID applications can achieve improved responsiveness, lower CPU utilization, and better scalability.

Task prioritization

One of the key aspects of robot programming is setting task priorities correctly. The most important task, typically the motion task that moves the robot, should be set as the foreground task. This is achieved using the parameter *TaskInForeground*, see *Technical reference manual - System parameters*.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

In a RAPID program, instructions consist of direct execution and waiting instructions. If a foreground task runs direct instructions without waiting time, background tasks stay waiting. One way to solve the problem is to add waiting time to the foreground task. Wait time can also be useful in background tasks to balance the system's resource usage.

RAPID instruction execution

The system parameter *RapidInstructionsPerMs* defines the target RAPID execution rate, expressed as instructions per millisecond.

- Higher value: higher requested RAPID throughput.
- Lower value: more conservative RAPID throughput.

For RobotWare 6, the default is 42.

If *RapidInstructionsPerMs* is set higher than the controller can practically sustain, the system runs at the highest stable execution rate available under current conditions. The configured value remains a target, while actual execution settles at the sustainable level.

Tradeoff with external communication

RapidInstructionsPerMs affects the balance between RAPID progression and responsiveness for external clients, such as:

- FlexPendant
- RobotStudio
- Other external communication services

Typical effect:

- Higher setting: faster RAPID progression, with increased risk of higher external communication latency during load.
- Lower setting: smoother external responsiveness, with lower RAPID progression rate.

Cycle-time stability aspect

Keeping *RapidInstructionsPerMs* at a moderate level improves cycle-time stability.

Temporary disturbances from other system functions are evened out because lost execution time is accounted for and compensated over time (catch-up behavior).

Tuning recommendations

If tuning the setting, the setting must be tested.

1. Start from the default value.
2. Adjust in small increments.
3. Validate with realistic combined load (program execution + external communication).
4. Track both:
   - RAPID cycle/throughput
   - FlexPendant/RobotStudio/external response times
5. Keep the highest value that still gives stable cycle behavior and acceptable external responsiveness.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Continued

RAPID instruction complexity

Compound RAPID instructions execute many simple RAPID instructions in one call. This makes it hard to know how many instructions per millisecond are actually used. In general, all instructions listed in *Technical reference manual – RAPID Instructions, Functions and Data types* are simple instructions. Application specific instructions are often compound instructions.

Structure the program for priority

If a program runs at maximum instruction capacity for the higher-priority domains, lower priority domains may be halted. This behavior should be considered when designing the program structure.

Structure the program to perform only essential tasks at any given moment. Avoid including non-essential instructions in every cycle.

Shared modules save memory instead of duplicating data in many tasks.

Example with cycles

For example, split the program into procedures. In the following example, the high priority procedures are executed in every cycle, while medium priority procedures are executed every 5th cycle, and low priority every 100th cycle.

```
VAR num cycleCounter := 0;
PROC main()
  DoHighPriorityProcs;
  IF (cycleCounter MOD 5) = 0 THEN
    DoMediumPriorityProcs;
  ENDIF
  IF (cycleCounter MOD 100) = 0 THEN
    DoLowPriorityProcs;
  ENDIF
  WaitTime 0.1;
  cycleCounter := cycleCounter + 1;
ENDPROC
```

Performance considerations

Avoid performance-consuming mathematical calculations in RAPID

Avoid performance-consuming mathematical calculations in the RAPID program. Instead, use existing RAPID instructions for mathematical operations, such as matrix calculations.

See *Mathematics on page 85*.

Reset arrays with `ResetData`

Some instructions such as `IF` or `FOR` loops can become very costly. For example, if used for resetting arrays which take a long time. This can be simplified by `ResetData`.

Use constants to initialize values

A good practice is to use constants to initialize the values that you want to use.

```
CONST triggios trigsingle{1}:=[[TRUE,0,TRUE,0,"do6",1]];
TriggLIOs p2, v500, \TriggData1:=trigsingle, z50, tool1;
```

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Use constants for array dimensions

A good practice is to use a constant for an array dimension. There is no need to use the `Dim` function in `FOR` loops.

Use the constant `triggios_init` that can initialize most of the record component. Then initialize values for the specifics.

```
CONST num TRIGGIOS_SIZE:=5;
CONST triggios triggios_init:=[TRUE,0,TRUE,0,stEmpty,1];
VAR triggios trigiosarr{TRIGGIOS_SIZE};
FOR i FROM 1 TO TRIGGIOS_SIZE DO
  trigiosarr{i}:=triggios_init;
ENDFOR
trigiosarr{1}.signalname:="do1";
trigiosarr{1}.distance:=1;
trigiosarr{2}.signalname:="do2";
trigiosarr{2}.distance:=5;
trigiosarr{3}.signalname:="do3";
trigiosarr{3}.distance:=10;
trigiosarr{4}.signalname:="do4";
trigiosarr{4}.distance:=15;
trigiosarr{5}.signalname:="do5";
trigiosarr{5}.distance:=20;
TriggLIOs p2, v500, \TriggData1:=trigiosarr, z50, tool1;
```

Constants consume less memory than VAR, and VAR consumes less than PERS. See *Technical reference manual - RAPID kernel*.

Late-binding instructions

If using late-binding instructions (reference by name) such as `AliasIO`, `SetDataSearch`, `GetNextSym`, the execution time can increase compared with explicit naming.

Position, path and speed programming

When programming positions, avoid setting the fly-by zone to 0 when the speed is higher than 70 mm/s. A higher zone value requires fewer resources and improves performance.

Use 0 only where precision is critical.

Try to program paths with as long segments as possible, as this reduces the load. If short segments are required, then try lowering the speed.

Communication

WebApp UI update frequency should be considered, as this communication takes bandwidth from other communication with the controller.

MultiMove considerations

In a MultiMove setup, declaring variables as TASK PERS is a good idea instead of just using PERS. TASK PERS gives you unique variables per task. If declaring with only PERS then the variables share the content between the tasks.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Continued

Paths with positioners

If a positioner is used in coordinated movements, it is a good idea to increase the size of the positioner's zone in zonedata, so the positioner can start to move earlier. This reduces the risk of forcing the robot to move through the zone at a lower speed than programmed.

Conclusion

By following these best practices, developers can ensure their robot programs are efficient, reliable, and maintain high performance. Proper task prioritization, optimized instruction execution, and thoughtful program structure are key to successful robot programming.

If certain functionality is missing, then contact ABB Robotics to discuss if it is possible to create new RAPID instructions or similar to simplify programming and speed up execution.

© Copyright 2004-2026 ABB. All rights reserved.

4 Glossary

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  xx1100000688
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 must always be acknowledged (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 `reg1+5` or `reg1>5`.
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  xx1100000687
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.

Continues on next page

© Copyright 2004-2026 ABB. All rights reserved.

Continued

Term	Description
Operator panel	The panel located on the front of the controller.
Orientation	The direction of an end effector.
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.
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.
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.
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.

© Copyright 2004-2026 ABB. All rights reserved.

Index

7
7 axes robots, 150

A
additional axes, 59, 111
aggregate, 29
aggregates
  expressions, 41
alias data types, 29
AND, 38
argument, 167
  conditional, 42
arguments
  description, 13
arithmetic expressions, 37
arithmetic functions, 85
arrays
  expressions, 40
  variables, 33
assigning value to data, 50
atomic data type, 29
automatic mode, 167
axis configuration, 137

B
backward execution, 102
backward handler, 102
base coordinate system, 108, 112
binary communication, 70
bit functions, 86
bool, 51

C
calibration, 93
case sensitivity, 14
circular interpolation, 120
circular movement, 56, 120
clock, 84
collision detection, 62, 145
comment, 16
comments, 50
communication instructions, 69
component of a record, 29, 167
concurrent execution, 134
conditional argument, 42
confdata, 138
configuration, 167
  robot, 137
configuration data, 91
ConfJ, 139
ConfL, 139
CONST, 34
constant, 31, 167
constants, 34
  initialization values, 34
conversion, 94
conversions, 51
conveyor tracking, 61
coordinated additional axes, 111
coordinate systems, 107, 141
corner path, 123, 167
cross connections, 159

D
data, 29
  assigning values, 50
  constant, 31
  declarations, 31
  description, 13
  initiating, 34
  persistent, 31
  program, 31
  routine, 31
  scope, 31
  storage class, 35
  used in expressions, 40
  variable, 31
data types, 29
  aggregates, 29
  alias, 29
  atomic, 29
  components, 29
  non-value, 29
  record, 29
  semi value, 29
declaration, 167
  module, 20
  routine, 25
declarations
  constants, 34
  persistents, 33
  variables, 32
dialog box, 167
displacement coordinate system, 110
DIV, 37
dnum, 51

E
equal data types, 29
ERRNO, 79
error handler, 167
error handlers, 78
error numbers, 77
error recovery, 77
event log, 80
event type, 92
execution handler, 92
execution level, 92
expression, 167
expressions
  arithmetic, 37
  logical, 38
  string, 39

F
file header, 17
file instructions, 70
file operation functions, 89
fly-by point, 123, 133, 167
function, 23, 167
function calls, 42
function declaration, 25

G
global
  data, 31
  routine, 23
glossary, 167
group signal, 167

© Copyright 2004-2026 ABB. All rights reserved.

I
I/O, 167
I/O device communication, 70
I/O principles, 158
I/O signals, 65
I/O synchronization, 133
identifiers, 15
independent axes, 60, 128
indexing conveyor, 61
initiating data, 34
INOUT, 24
input signals, 65
instructions
  description, 13
  pick lists, 47
  program flow, 48
interpolation, 119, 123
interrupt, 167
interrupted path, 63
interrupts, 57, 73

J
joint interpolation, 119
joint movement, 56, 119

K
kinematic models, 141

L
linear interpolation, 119
linear movement, 56, 119
load identification, 62
loading modules, 50
local
  data, 31
  routine, 23
logical expressions, 38
logical values, 16

M
main routine, 19, 167
manual mode, 167
mathematical instructions, 85
mechanical unit, 167
memory, 90
MOD, 37
modified linear interpolation, 121
module, 167
module declaration, 20
modules, 19
  description, 19
motion, 56
motion data, 63
motion instructions, 56
motion settings
  instructions, 52
motion supervision, 145
motors on/off, 167
move instructions, 56
MultiMove, 59, 97
Multitasking, 96

N
non-value data types, 29
NOT, 38
num, 51
numeric values, 16

O
object coordinate system, 110
operator panel, 168
operator priority, 44
optional parameter, 24
options, 90
OR, 38
orientation, 168
output signals, 65

P
parameter, 24, 168
path correction, 60
path recorder, 61
path synchronization, 135
PERS, 33
persistent, 31, 168
persistents, 33
  initialization values, 34
pick lists, 47
placeholders, 17
position fix I/O, 135
position functions, 62
positioning instructions, 56
priority
  operators, 44
  tasks, 100
procedure, 23, 168
procedure call, 26
procedure declaration, 25
program, 19, 168
program data, 29, 31, 168
program flow instructions, 48
program module, 19, 168

Q
quarter revolutions, 138

R
RAPID Message Queues, 72
rawbytes communication, 71
record, 29, 168
records
  expressions, 40
reserved words, 15
restart the controller, 91
robot configuration, 137
robot configuration supervision, 138
robot kinematics, 141
routine, 23, 168
routine data, 31, 168
routine declaration, 25
routines
  description, 13

S
scope
  data, 31
  routine, 23
search instructions, 57
semi value data types, 29
sensor synchronization, 62
serial link robot, 149
service, 93
service information, 92
servo tracking, 61
signals, 65, 158

© Copyright 2004-2026 ABB. All rights reserved.

singularities, 121, 149
socket communication, 71
soft servo, 54, 131
start point, 168
stationary TCP, 115
status functions, 63
stop, 132
stopping program execution, 48
stop point, 168
string, 16, 51
string expressions, 39
string functions, 94
supervision
  robot configuration, 138
switch, 24, 51
synchronization, 133
syntax rules, 10
system data, 90
system information, 90
system module, 20, 168
system parameters, 168

T
tasks, 91, 96
task selection panel, 92
TCP, 107, 168
  stationary, 115
terminating routine, 25
time instructions, 84
tool center point, 107, 168
tool coordinate system, 114
trap declaration, 26
trap routine, 23, 168
trap routines, 73, 76

U
UNDO, 81
user coordinate system, 109, 111
User system module, 22

V
VAR, 32
variable, 31, 168
variables, 32
  arrays, 33
  initialization values, 34

W
wait instructions, 50
world coordinate system, 108
world zones, 55, 153
wrist coordinate system, 114

X
XOR, 38

Z
zone, 123, 168

© Copyright 2004-2026 ABB. All rights reserved.

ABB Robotics Sweden AB
S-721 71 VÄSTERÅS, Sweden
Telephone +46 21 372 8009

ABB Robotics Norway AS
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 69 03 33 99

ABB Robotics (Shanghai) Limited
No. 39, 99 Miaoqiao Road,
Pudong New Area,
Shanghai, 201315, China
Telephone: +86 400 920 9090

ABB Robotics LLC
1250 Brown Road
Auburn Hills, MI 48326
USA
Telephone: +1 615 237 6360

abb.com/robotics

3HAC050947-001, Rev Y, en

© Copyright 2004-2026 ABB. All rights reserved.
Specifications subject to change without notice.