

Application manual

Controller software IRC5

Power and productivity
for a better world™



Trace back information:
Workspace R15-2 version a15
Checked in 2015-10-08
Skribenta version 4.6.081

Application manual

Controller software IRC5

RobotWare 6.02

Document ID: 3HAC050798-001

Revision: B

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

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

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

This manual and parts thereof must not be reproduced or copied without ABB's written permission.

Additional copies of this manual may be obtained from ABB.

The original language for this publication is English. Any other languages that are supplied have been translated from English.

© Copyright 2015 ABB. All rights reserved.

ABB AB
Robotics Products
Se-721 68 Västerås
Sweden

Table of contents

Overview of this manual	11
1 Introduction to RobotWare	13
2 RobotWare-OS	15
2.1 Advanced RAPID	15
2.1.1 Introduction to Advanced RAPID	15
2.1.2 Bit functionality	16
2.1.2.1 Overview	16
2.1.2.2 RAPID components	17
2.1.2.3 Bit functionality example	18
2.1.3 Data search functionality	19
2.1.3.1 Overview	19
2.1.3.2 RAPID components	20
2.1.3.3 Data search functionality examples	21
2.1.4 Alias I/O signals	22
2.1.4.1 Overview	22
2.1.4.2 RAPID components	23
2.1.4.3 Alias I/O functionality example	24
2.1.5 Configuration functionality	25
2.1.5.1 Overview	25
2.1.5.2 RAPID components	26
2.1.5.3 Configuration functionality example	27
2.1.6 Power failure functionality	28
2.1.6.1 Overview	28
2.1.6.2 RAPID components and system parameters	29
2.1.6.3 Power failure functionality example	30
2.1.7 Process support functionality	31
2.1.7.1 Overview	31
2.1.7.2 RAPID components	32
2.1.7.3 Process support functionality examples	33
2.1.8 Interrupt functionality	35
2.1.8.1 Overview	35
2.1.8.2 RAPID components	36
2.1.8.3 Interrupt functionality examples	37
2.1.9 User message functionality	38
2.1.9.1 Overview	38
2.1.9.2 RAPID components	39
2.1.9.3 User message functionality examples	40
2.1.9.4 Text table files	41
2.1.10 RAPID support functionality	42
2.1.10.1 Overview	42
2.1.10.2 RAPID components	43
2.1.10.3 RAPID support functionality examples	44
2.2 Analog Signal Interrupt	45
2.2.1 Introduction to Analog Signal Interrupt	45
2.2.2 RAPID components	46
2.2.3 Code example	47
2.3 Auto Acknowledge Input	48
2.4 Electronically Linked Motors	49
2.4.1 Overview	49
2.4.2 Configuration	51
2.4.2.1 System parameters	51
2.4.2.2 Configuration example	53
2.4.3 Managing a follower axis	54
2.4.3.1 Using the service program	54
2.4.3.2 Calibrate follower axis position	55

2.4.3.3	Reset follower axis	57
2.4.4	Tuning a torque follower	58
2.4.4.1	Torque follower descriptions	58
2.4.4.2	Using the service program	59
2.4.5	Data setup	61
2.4.5.1	Set up data for service program	61
2.4.5.2	Example of data setup	63
2.5	Fixed Position Events	65
2.5.1	Overview	65
2.5.2	RAPID components and system parameters	66
2.5.3	Code examples	69
2.6	File and Serial Channel Handling	71
2.6.1	Introduction to File and Serial Channel Handling	71
2.6.2	Binary and character based communication	72
2.6.2.1	Overview	72
2.6.2.2	RAPID components	73
2.6.2.3	Code examples	74
2.6.3	Raw data communication	76
2.6.3.1	Overview	76
2.6.3.2	RAPID components	77
2.6.3.3	Code examples	78
2.6.4	File and directory management	80
2.6.4.1	Overview	80
2.6.4.2	RAPID components	81
2.6.4.3	Code examples	82
2.7	Device Command Interface	84
2.7.1	Introduction to Device Command Interface	84
2.7.2	RAPID components and system parameters	85
2.7.3	Code example	86
2.8	Logical Cross Connections	88
2.8.1	Introduction to Logical Cross Connections	88
2.8.2	Configuring Logical Cross Connections	89
2.8.3	Examples	90
2.8.4	Limitations	92
3	Motion performance	93
3.1	Absolute Accuracy [603-1, 603-2]	93
3.1.1	About Absolute Accuracy	93
3.1.2	When is Absolute Accuracy being used	95
3.1.3	Useful tools	96
3.1.4	Configuration	97
3.1.5	Maintenance	99
3.1.5.1	Maintenance that affect the accuracy	99
3.1.5.2	Loss of accuracy	101
3.1.6	Compensation theory	102
3.1.6.1	Error sources	102
3.1.6.2	Absolute Accuracy compensation	103
3.1.7	Preparation of Absolute Accuracy robot	105
3.1.7.1	ABB calibration process	105
3.1.7.2	Birth certificate	107
3.1.7.3	Compensation parameters	108
3.1.8	Cell alignment	111
3.1.8.1	Overview	111
3.1.8.2	Measure fixture alignment	112
3.1.8.3	Measure robot alignment	113
3.1.8.4	Frame relationships	114
3.1.8.5	Tool calibration	115
3.2	Advanced robot motion [687-1]	116

3.3	Advanced Shape Tuning [included in 687-1]	117
3.3.1	About Advanced Shape Tuning	117
3.3.2	Automatic friction tuning	118
3.3.3	Manual friction tuning	120
3.3.4	System parameters	122
3.3.4.1	System parameters	122
3.3.4.2	Setting tuning system parameters	123
3.3.5	RAPID components	124
3.4	Motion Process Mode [included in 687-1]	125
3.4.1	About Motion Process Mode	125
3.4.2	User-defined modes	126
3.4.3	General information about robot tuning	128
3.4.4	Additional information	130
3.5	Wrist Move [included in 687-1]	131
3.5.1	Introduction to Wrist Move	131
3.5.2	Cut plane frame	133
3.5.3	RAPID components	135
3.5.4	RAPID code, examples	136
3.5.5	Trouble shooting	138
4	Motion coordination	139
4.1	Machine Synchronization [607-1], [607-2]	139
4.1.1	Overview	139
4.1.2	What is needed	141
4.1.3	Synchronization features	143
4.1.4	General description of the synchronization process	144
4.1.5	Limitations	145
4.1.6	Hardware installation for Sensor Synchronization	146
4.1.6.1	Encoder specification	146
4.1.6.2	Encoder description	147
4.1.6.3	Installation recommendations	148
4.1.6.4	Connecting encoder and encoder interface unit	149
4.1.7	Hardware installation for Analog Synchronization	151
4.1.7.1	Required hardware	151
4.1.8	Software installation	152
4.1.8.1	Sensor installation	152
4.1.8.2	Reloading saved Motion parameters	154
4.1.8.3	Installation of several sensors	155
4.1.9	Programming the synchronization	156
4.1.9.1	General issues when programming with the synchronization option	156
4.1.9.2	Programming examples	158
4.1.9.3	Entering and exiting coordinated motion in corner zones	160
4.1.9.4	Use several sensors	161
4.1.9.5	Finepoint programming	162
4.1.9.6	Drop sensor object	163
4.1.9.7	Information on the FlexPendant	164
4.1.9.8	Programming considerations	165
4.1.9.9	Modes of operation	167
4.1.10	Robot to robot synchronization	169
4.1.10.1	Introduction	169
4.1.10.2	The concept of robot to robot synchronization	170
4.1.10.3	Master robot configuration parameters	171
4.1.10.4	Slave robot configuration parameters	174
4.1.10.5	Programming example for master robot	177
4.1.10.6	Programming example for slave robot	179
4.1.11	Synchronize with hydraulic press using recorded profile	180
4.1.11.1	Introduction	180
4.1.11.2	Configuration of system parameters	181
4.1.11.3	Program example	183

Table of contents

4.1.12	Synchronize with molding machine using recorded profile	184
4.1.12.1	Introduction	184
4.1.12.2	Configuration of system parameters	185
4.1.12.3	Program example	187
4.1.13	Supervision	188
4.1.14	System parameters	189
4.1.15	I/O signals	192
4.1.16	RAPID components	193
5	Motion Events	195
5.1	World Zones [608-1]	195
5.1.1	Overview	195
5.1.2	RAPID components	197
5.1.3	Code examples	199
6	Motion functions	201
6.1	Independent Axes [610-1]	201
6.1.1	Overview	201
6.1.2	System parameters	203
6.1.3	RAPID components	204
6.1.4	Code examples	205
6.2	Path Recovery [611-1]	207
6.2.1	Overview	207
6.2.2	RAPID components	208
6.2.3	Store current path	209
6.2.4	Path recorder	215
6.3	Path Offset [612-1]	222
6.3.1	Overview	222
6.3.2	RAPID components	223
6.3.3	Related RAPID functionality	224
6.3.4	Code example	225
7	Motion Supervision	227
7.1	Collision Detection [613-1]	227
7.1.1	Overview	227
7.1.2	Limitations	228
7.1.3	What happens at a collision	229
7.1.4	Additional information	231
7.1.5	Configuration and programming facilities	232
7.1.5.1	System parameters	232
7.1.5.2	RAPID components	234
7.1.5.3	Signals	235
7.1.6	How to use Collision Detection	236
7.1.6.1	Set up system parameters	236
7.1.6.2	Adjust supervision from FlexPendant	237
7.1.6.3	Adjust supervision from RAPID program	238
7.1.6.4	How to avoid false triggering	239
8	Communication	241
8.1	FTP Client [614-1]	241
8.1.1	Introduction to FTP Client	241
8.1.2	System parameters	243
8.1.3	Examples	244
8.2	NFS Client [614-1]	245
8.2.1	Introduction to NFS Client	245
8.2.2	System parameters	247
8.2.3	Examples	248

8.3	PC Interface [616-1]	249
8.3.1	Introduction to PC Interface	249
8.3.2	Send variable from RAPID	250
8.3.3	ABB software using PC Interface	252
8.4	Socket Messaging [616-1]	253
8.4.1	Introduction to Socket Messaging	253
8.4.2	Schematic picture of socket communication	254
8.4.3	Technical facts about Socket Messaging	255
8.4.4	RAPID components	256
8.4.5	Code examples	258
8.5	RAPID Message Queue [included in 616-1, 623-1]	260
8.5.1	Introduction to RAPID Message Queue	260
8.5.2	RAPID Message Queue behavior	261
8.5.3	System parameters	265
8.5.4	RAPID components	266
8.5.5	Code examples	267
9	Engineering tools	271
9.1	Multitasking [623-1]	271
9.1.1	Introduction to Multitasking	271
9.1.2	System parameters	273
9.1.3	RAPID components	275
9.1.4	Task configuration	276
9.1.4.1	Debug strategies for setting up tasks	276
9.1.4.2	Priorities	278
9.1.4.3	Task Panel Settings	280
9.1.4.4	Select which tasks to start with START button	281
9.1.5	Communication between tasks	283
9.1.5.1	Persistent variables	283
9.1.5.2	Waiting for other tasks	285
9.1.5.3	Synchronizing between tasks	287
9.1.5.4	Using a dispatcher	289
9.1.6	Other programming issues	291
9.1.6.1	Share resource between tasks	291
9.1.6.2	Test if task controls mechanical unit	292
9.1.6.3	taskid	293
9.1.6.4	Avoid heavy loops	294
9.2	Sensor Interface [628-1]	295
9.2.1	Introduction to Sensor Interface	295
9.2.2	Configuring sensors	296
9.2.2.1	About the sensors	296
9.2.2.2	Configuring sensors over serial channels	297
9.2.2.3	Configuring sensors over Ethernet channel	298
9.2.3	RAPID	299
9.2.3.1	RAPID components	299
9.2.4	Examples	302
9.2.4.1	Code examples	302
9.3	Externally Guided Motion [689-1]	304
9.3.1	Introduction to EGM	304
9.3.1.1	Overview	304
9.3.1.2	Introduction to EGM Position Guidance	306
9.3.1.3	Introduction to EGM Path Correction	307
9.3.2	Using EGM	308
9.3.2.1	Basic approach	308
9.3.2.2	Execution states	309
9.3.2.3	Input data	310
9.3.2.4	Output data	313
9.3.2.5	Configuration	314
9.3.2.6	Frames	316

Table of contents

9.3.3	The EGM sensor protocol	318
9.3.4	System parameters	322
9.3.5	RAPID components	323
9.3.6	RAPID code examples	325
9.3.6.1	Using EGM Position Guidance with an UdpUc device	325
9.3.6.2	Using EGM Position Guidance with signals as input	327
9.3.6.3	Using EGM Path Correction with different protocol types	331
9.3.7	UdpUc code examples	334
9.4	Robot Reference Interface [included in 689-1]	335
9.4.1	Introduction to Robot Reference Interface	335
9.4.2	Installation	336
9.4.2.1	Connecting the communication cable	336
9.4.2.2	Prerequisites	337
9.4.2.3	Data orchestration	338
9.4.2.4	Supported data types	339
9.4.3	Configuration	340
9.4.3.1	Interface configuration	340
9.4.3.2	Interface settings	341
9.4.3.3	Device description	342
9.4.3.4	Device configuration	345
9.4.4	Configuration examples	348
9.4.4.1	RAPID programming	348
9.4.4.2	Example configuration	349
9.4.5	RAPID components	354
10	Servo motor control	355
10.1	Servo Tool Change [630-1]	355
10.1.1	Overview	355
10.1.2	Requirements and limitations	356
10.1.3	Configuration	358
10.1.4	Connection relay	359
10.1.5	Tool change procedure	361
10.1.6	Jogging servo tools with activation disabled	362
10.2	Servo Tool Control [included in 635-6]	363
10.2.1	Overview	363
10.2.2	Servo tool movements	364
10.2.3	Tip management	365
10.2.4	Supervision	367
10.2.5	RAPID components	368
10.2.6	System parameters	369
10.2.7	Commissioning and service	374
10.2.8	Mechanical unit calibrations	376
10.2.9	RAPID code example	377
10.2.10	Service routines	378
	Index	381

Overview of this manual

About this manual

This manual explains the basics of when and how to use various RobotWare options and functions.

Usage

This manual can be used either as a reference to find out if an option is the right choice for solving a problem, or as a description of how to use an option. Detailed information regarding syntax for RAPID routines, and similar, is not described here, but can be found in the respective reference manual.

Who should read this manual?

This manual is intended for robot programmers.

Prerequisites

The reader should...

- be familiar with industrial robots and their terminology.
- be familiar with the RAPID programming language.
- be familiar with system parameters and how to configure them.

References

Reference	Document ID
<i>Product specification - Controller software IRC5</i> IRC5 with main computer DSQC1000 and RobotWare 6.	3HAC050945-001
<i>Product specification - Controller IRC5</i> IRC5 with main computer DSQC1000.	3HAC047400-001
<i>Operating manual - RobotStudio</i>	3HAC032104-001
<i>Operating manual - IRC5 with FlexPendant</i>	3HAC050941-001
<i>Technical reference manual - RAPID Instructions, Functions and Data types</i>	3HAC050917-001
<i>Technical reference manual - RAPID overview</i>	3HAC050947-001
<i>Technical reference manual - System parameters</i>	3HAC050948-001

Revisions

Revision	Description
-	Released with RobotWare 6.0. First release.
A	Released with RobotWare 6.01. <ul style="list-style-type: none"> • Added Auto Acknowledge Input, see Auto Acknowledge Input on page 48. • The functionality of RAPID Message Queue is corrected, see RAPID Message Queue [included in 616-1, 623-1] on page 260. • Minor corrections.

Continues on next page

Revision	Description
B	<p>Released with RobotWare 6.02.</p> <ul style="list-style-type: none">• Updated the path to the template files, see UdpUc code examples on page 334 and Commissioning and service on page 374.• The TCP ports and protocols are updated for the option Sensor Interface [628-1], see Configuring sensors over Ethernet channel on page 298.• Added the functionality EGM Path Correction with corresponding RAPID instructions, see Externally Guided Motion [689-1] on page 304.• Bundled options are reordered in the manual according to the parent option.• Updated the LTAPP variable list available for optical tracking, see Constants on page 300.


1 Introduction to RobotWare

Software products

RobotWare is a family of software products from ABB Robotics. The products are designed to make you more productive and lower your cost of owning and operating a robot. ABB Robotics has invested many years into the development of these products and they represent knowledge and experience based on several thousands of robot installations.

Product classes

Within the RobotWare family, there are different classes of products:

Product classes	Description
RobotWare-OS	<p>This is the operating system of the robot. RobotWare-OS provides all the necessary features for fundamental robot programming and operation. It is an inherent part of the robot, but can be provided separately for upgrading purposes.</p> <p>For a description of RobotWare-OS, see <i>Product specification - Controller IRC5</i>.</p>
RobotWare options	<p>These products are options that run on top of RobotWare-OS. They are intended for robot users that need additional functionality for motion control, communication, system engineering, or applications.</p> <div>  Note </div> <p>Not all RobotWare options are described in this manual. Some options are more comprehensive and are therefore described in separate manuals. For more information see <i>Product specification - Controller software IRC5</i>.</p>
Process application options	<p>These are extensive packages for specific process application like spot welding, arc welding, and dispensing. They are primarily designed to improve the process result and to simplify installation and programming of the application.</p> <p>The process application options are all described in separate manuals. For more information see <i>Product specification - Controller software IRC5</i>.</p>
RobotWare Add-ins	<p>A RobotWare Add-in is a self-contained package that extends the functionality of the robot system.</p> <p>Some software products from ABB Robotics are delivered as Add-ins. For example track motion IRBT, positioner IRBP, and stand alone controller. For more information see <i>Product specification - Controller software IRC5</i>.</p> <p>The purpose of RobotWare Add-ins is also that a robot program developer outside of ABB can create options for the ABB robot systems, and sell the options to their customers. For more information on creating RobotWare Add-ins, contact your local ABB Robotics representative at www.abb.com/contacts.</p>

Continues on next page

Option groups

For IRC5, the RobotWare options have been gathered in groups, depending on the customer benefit. The goal is to make it easier to understand the customer value of the options. However, all options are purchased individually. The groups are as follows:

Option groups	Description
Motion performance	Options that optimize the performance of your robot.
Motion coordination	Options that make your robot coordinated with external equipment or other robots.
Motion Events	Options that supervises the position of the robot.
Motion functions	Options that controls the path of the robot.
Motion Supervision	Options that supervises the movement of the robot.
Communication	Options that make the robot communicate with other equipment. (External PCs etc.)
Engineering tools	Options for the advanced robot integrator.
Servo motor control	Options that make the robot controller operate external motors, independent of the robot.



Note

Not all RobotWare options are described in this manual. Some options are more comprehensive and are therefore described in separate manuals. For more information see *Product specification - Controller software IRC5*.

2 RobotWare-OS

2.1 Advanced RAPID

2.1.1 Introduction to Advanced RAPID

Introduction to Advanced RAPID

The RobotWare base functionality *Advanced RAPID* is intended for robot programmers who develop applications that require advanced functionality.

Advanced RAPID includes many different types of functionality, which can be divided into these groups:

Functionality group	Description
Bit functionality	Bitwise operations on a byte.
Data search functionality	Search and get/set data objects (e.g. variables).
Alias I/O functionality	Give an I/O signal an optional alias name.
Configuration functionality	Get/set system parameters.
Power failure functionality	Restore signals after power failure.
Process support functionality	Useful when creating process applications.
Interrupt functionality	More interrupt functionality than included in RobotWare base functionality.
User message functionality	Error messages and other texts.
RAPID support functionality	Miscellaneous support for the programmer.

2 RobotWare-OS

2.1.2.1 Overview

2.1.2 Bit functionality

2.1.2.1 Overview

Purpose

The purpose of the bit functionality is to be able to make operations on a byte, seen as 8 digital bits. It is possible to get or set a single bit, or make logical operations on a byte. These operations are useful, for example, when handling serial communication or group of digital I/O signals.

What is included

Bit functionality includes:

- The data type `byte`.
- Instructions used set a bit value: `BitSet` and `BitClear`.
- Function used to get a bit value: `BitCheck`.
- Functions used to make logical operations on a byte: `BitAnd`, `BitOr`, `BitXOr`, `BitNeg`, `BitLSh`, and `BitRSh`.

2.1.2.2 RAPID components

Data types

This is a brief description of each data type used for the bit functionality. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
byte	The data type <code>byte</code> represent a decimal value between 0 and 255.

Instructions

This is a brief description of each instruction used for the bit functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
BitSet	BitSet is used to set a specified bit to 1 in a defined byte data.
BitClear	BitClear is used to clear (set to 0) a specified bit in a defined byte data.

Functions

This is a brief description of each function used for the bit functionality. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
BitAnd	BitAnd is used to execute a logical bitwise AND operation on data types <code>byte</code> .
BitOr	BitOr is used to execute a logical bitwise OR operation on data types <code>byte</code> .
BitXOr	BitXOr (Bit eXclusive Or) is used to execute a logical bitwise XOR operation on data types <code>byte</code> .
BitNeg	BitNeg is used to execute a logical bitwise negation operation (one's complement) on data types <code>byte</code> .
BitLSh	BitLSh (Bit Left Shift) is used to execute a logical bitwise left shift operation on data types <code>byte</code> .
BitRSh	BitRSh (Bit Right Shift) is used to execute a logical bitwise right shift operation on data types <code>byte</code> .
BitCheck	BitCheck is used to check if a specified bit in a defined byte data is set to 1.



Tip

Even though not part of the option, the functions for conversion between a byte and a string, `StrToByte` and `ByteToStr`, are often used together with the bit functionality.

2.1.2.3 Bit functionality example

Program code

```
CONST num parity_bit := 8;

!Set data1 to 00100110
VAR byte data1 := 38;

!Set data2 to 00100010
VAR byte data2 := 34;

VAR byte data3;

!Set data3 to 00100010
data3 := BitAnd(data1, data2);

!Set data3 to 00100110
data3 := BitOr(data1, data2);

!Set data3 to 00000100
data3 := BitXOr(data1, data2);

!Set data3 to 11011001
data3 := BitNeg(data1);

!Set data3 to 10011000
data3 := BitLSh(data1, 2);

!Set data3 to 00010011
data3 := BitRSh(data1, 1);

!Set data1 to 10100110
BitSet data1, parity_bit;

!Set data1 to 00100110
BitClear data1, parity_bit;

!If parity_bit is 0, set it to 1
IF BitCheck(data1, parity_bit) = FALSE THEN
  BitSet data1, parity_bit;
ENDIF
```

2.1.3 Data search functionality

2.1.3.1 Overview

Purpose

The purpose of the data search functionality is to search and get/set values for data objects of a certain type.

Here are some examples of applications for the data search functionality:

- Setting a value to a variable, when the variable name is only available in a string.
- List all variables of a certain type.
- Set a new value for a set of similar variables with similar names.

What is included

Data search functionality includes:

- The data type `datapos`.
- Instructions used to find a set of data objects and get or set their values: `SetDataSearch`, `GetDataVal`, `SetDataVal`, and `SetAllDataVal`.
- A function for traversing the search result: `GetNextSym`.

2.1.3.2 RAPID components

Data types

This is a brief description of each data type used for the data search functionality. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
datapos	datapos is the enclosing block to a data object (internal system data) retrieved with the function <code>GetNextSym</code> .

Instructions

This is a brief description of each instruction used for the data search functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
<code>SetDataSearch</code>	<code>SetDataSearch</code> is used together with <code>GetNextSym</code> to retrieve data objects from the system.
<code>GetDataVal</code>	<code>GetDataVal</code> makes it possible to get a value from a data object that is specified with a string variable, or from a data object retrieved with <code>GetNextSym</code> .
<code>SetDataVal</code>	<code>SetDataVal</code> makes it possible to set a value for a data object that is specified with a string variable, or from a data object retrieved with <code>GetNextSym</code> .
<code>SetAllDataVal</code>	<code>SetAllDataVal</code> make it possible to set a new value to all data objects of a certain type that match the given grammar.

Functions

This is a brief description of each function used for the data search functionality. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
<code>GetNextSym</code>	<code>GetNextSym</code> (Get Next Symbol) is used together with <code>SetDataSearch</code> to retrieve data objects from the system.

2.1.3.3 Data search functionality examples

Set unknown variable

This is an example of how to set the value of a variable when the name of the variable is unknown when programming, and only provided in a string.

```
VAR string my_string;
VAR num my_number;
VAR num new_value:=10;
my_string := "my_number";
!Set value to 10 for variable specified by my_string
SetDataVal my_string,new_value;
```

Reset a range of variables

This is an example where all numeric variables starting with "my" is reset to 0.

```
VAR string my_string:="my.*";
VAR num zeroval:=0;
SetAllDataVal "num"\Object:=my_string,zeroval;
```

List/set certain variables

In this example, all numeric variables in the module "mymod" starting with "my" are listed on the FlexPendant and then reset to 0.

```
VAR datapos block;
VAR string name;
VAR num valuevar;
VAR num zeroval:=0;

!Search for all num variables starting with "my" in the module
"mymod"
SetDataSearch "num"\Object:="my.*"\InMod:="mymod";

!Loop through the search result
WHILE GetNextSym(name,block) DO
  !Read the value from each found variable
  GetDataVal name\Block:=block,valuevar;

  !Write name and value for each found variable
  TPWrite name+" = "\Num:=valuevar;

  !Set the value to 0 for each found variables
  SetDataVal name\Block:=block,zeroval;
ENDWHILE
```

2 RobotWare-OS

2.1.4.1 Overview

2.1.4 Alias I/O signals

2.1.4.1 Overview

Purpose

The Alias I/O functionality gives the programmer the ability to use any name on a signal and connect that name to a configured I/O signal.

This is useful when a RAPID program is reused between different systems. Instead of rewriting the code, using a signal name that exist on the new system, the signal name used in the program can be defined as an alias name.

What is included

Alias I/O functionality consists of the instruction `AliasIO`.

2.1.4.2 RAPID components

Data types

There are no RAPID data types for the Alias I/O functionality.

Instructions

This is a brief description of each instruction used for the Alias I/O functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
AliasIO	AliasIO is used to define a signal of any type with an alias name, or to use signals in built-in task modules. The alias name is connected to a configured I/O signal. The instruction AliasIO must be run before any use of the actual signal.

Functions

There are no RAPID functions for the Alias I/O functionality.

2.1.4.3 Alias I/O functionality example

Assign alias name to signal

This example shows how to define the digital output signal `alias_do` to be connected to the configured digital output I/O signal `config_do`.

The routine `prog_start` is connected to the START event.

This will ensure that "alias_do" can be used in the RAPID code even though there is no configured signal with that name.

```
VAR signaldo alias_do;
PROC prog_start()
  AliasIO config_do, alias_do;
ENDPROC
```


2.1.5 Configuration functionality

2.1.5.1 Overview

Purpose

The configuration functionality gives the programmer access to the system parameters at run time. The parameter values can be read and edited. The controller can be restarted in order for the new parameter values to take effect.

What is included

Configuration functionality includes the instructions: `ReadCfgData`, `WriteCfgData`, and `WarmStart`.

2.1.5.2 RAPID components

Data types

There are no RAPID data types for the configuration functionality.

Instructions

This is a brief description of each instruction used for the configuration functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
ReadCfgData	ReadCfgData is used to read one attribute of a named system parameter (configuration data).
WriteCfgData	WriteCfgData is used to write one attribute of a named system parameter (configuration data).
WarmStart	WarmStart is used to restart the controller at run time. This is useful after changing system parameters with the instruction WriteCfgData.

Functions

There are no RAPID functions for the configuration functionality.

2.1.5.3 Configuration functionality example

Configure system parameters

This is an example where the system parameter *cal_offset* for rob1_1 is read, increased by 0.2 mm and then written back. To make this change take effect, the controller is restarted.

```
VAR num old_offset;  
VAR num new_offset;  
  
ReadCfgData "/MOC/MOTOR_CALIB/rob1_1", "cal_offset",old_offset;  
new_offset := old_offset + (0.2/1000);  
WriteCfgData "/MOC/MOTOR_CALIB/rob1_1", "cal_offset",new_offset;  
WarmStart;
```

2.1.6 Power failure functionality

2.1.6.1 Overview

Purpose

If the robot was in the middle of a path movement when the power fail occurred, some extra actions may need to be taken when the robot motion is resumed. The power failure functionality helps you detect if the power fail occurred during a path movement.



Note

For more information see the type *Signal Safe Level*, which belongs to the topic *I/O System*, in *Technical reference manual - System parameters*.

What is included

The power failure functionality includes a function that checks for interrupted path:
`PFRestart`

2.1.6.2 RAPID components and system parameters

Data types

There are no RAPID data types in the power failure functionality.

Instructions

There are no RAPID instructions in the power failure functionality.

Functions

This is a brief description of each function in the power failure functionality. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
PFRestart	PFRestart (Power Failure Restart) is used to check if the path was interrupted at power failure. If so it might be necessary to make some specific actions. The function checks the path on current level, base level or on interrupt level.

System parameters

There are no system parameters in the power failure functionality. However, regardless of whether you have any options installed, you can use the parameter *Store signal at power fail*.

For more information, see *Technical reference manual - System parameters*.

2.1.6.3 Power failure functionality example

Test for interrupted path

When resuming work after a power failure, this example tests if the power failure occurred during a path (i.e. when the robot was moving).

```
!Test if path was interrupted
IF PFRestart() = TRUE THEN
  SetDO do5,1;
ELSE
  SetDO do5,0;
ENDIF
```

2.1.7 Process support functionality

2.1.7.1 Overview

Purpose

Process support functionality provides some RAPID instructions that can be useful when creating process applications. Examples of its use are:

- Analog output signals, used in continuous process application, can be set to be proportional to the robot TCP speed.
- A continuous process application that is stopped with program stop or emergency stop can be continued from where it stopped.

What is included

The process support functionality includes:

- The data type `restartdata`.
- Instruction for setting analog output signal: `TriggSpeed`.
- Instructions used in connection with restart: `TriggStopProc` and `StepBwdPath`.

Limitations

The instruction `TriggSpeed` can only be used if you have the base functionality *Fixed Position Events*.

2.1.7.2 RAPID components

Data types

This is a brief description of each data type used for the process support functionality. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
restartdata	restartdata can contain the pre- and post-values of specified I/O signals (process signals) at the stop sequence of the robot movements. restartdata, together with the instruction TriggStopProc is used to preserve data for the restart after program stop or emergency stop of self-developed process instructions.

Instructions

This is a brief description of each instruction used for the process support functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
TriggSpeed	TriggSpeed is used to define the setting of an analog output to a value proportional to the TCP speed. TriggSpeed can only be used together with the option Fixed Position Events.
TriggStopProc	TriggStopProc is used to store the pre- and post-values of all used process signals. TriggStopProc and the data type restartdata are used to preserve data for the restart after program stop or emergency stop of self-developed process instructions.
StepBwdPath	StepBwdPath is used to move the TCP backwards on the robot path from a RESTART event routine.

Functions

There are no RAPID functions for the process support functionality.

2.1.7.3 Process support functionality examples

Signal proportional to speed

In this example, the analog output signal that controls the amount of glue is set to be proportional to the speed.

Any speed dip by the robot is time compensated in such a way that the analog output signal `glue_ao` is affected 0.04 s before the TCP speed dip occurs. If overflow of the calculated logical analog output value in `glue_ao`, the digital output signal `glue_err` is set.

```
VAR triggdata glueflow;

!The glue flow is set to scale value 0.8 0.05 s before point p1
TriggSpeed glueflow, 0, 0.05, glue_ao, 0.8 \DipLag=:0.04,
\ErrDO:=glue_err;
TriggL p1, v500, glueflow, z50, gun1;

!The glue flow is set to scale value 1 10 mm plus 0.05 s before
point p2
TriggSpeed glueflow, 10, 0.05, glue_ao, 1;
TriggL p2, v500, glueflow, z10, gun1;

!The glue flow ends (scale value 0) 0.05 s before point p3
TriggSpeed glueflow, 0, 0.05, glue_ao, 0;
TriggL p3, v500, glueflow, z50, gun1;
```



Tip

Note that it is also possible to create self-developed process instructions with `TriggSpeed` using the `NOSTEPIN` routine concept.

Resume signals after stop

In this example, an output signal resumes its value after a program stop or emergency stop.

The procedure `supervise` is defined as a `POWER ON` event routine and `resume_signals` as a `RESTART` event routine.

```
PERS restartdata myproc_data :=
[FALSE,FALSE,0,0,0,0,0,0,0,0,0,0,0,0,0];
...
PROC myproc()
MoveJ p1, vmax, fine, my_gun;
SetDO do_close_gun, 1;
MoveL p2,v1000,z50,my_gun;
MoveL p3,v1000,fine,my_gun;
SetDO do_close_gun, 0;
ENDPROC
...
PROC supervise()
TriggStopProc myproc_data \DO1:=do_close_gun, do_close_gun;
```

Continues on next page

2 RobotWare-OS

2.1.7.3 Process support functionality examples

Continued

```
ENDPROC

PROC resume_signals()
  IF myproc_data.preshadowval = 1 THEN
    SetDO do_close_gun,1;
  ELSE
    SetDO do_close_gun,0;
  ENDIF
ENDPROC
```

Move TCP backwards

In this example, the TCP is moved backwards 30 mm in 1 second, along the same path as before the restart.

The procedure `move_backward` is defined as a RESTART event routine.

```
PROC move_backward()
  StepBwdPath 30, 1;
ENDPROC
```

2.1.8 Interrupt functionality

2.1.8.1 Overview

Purpose

The interrupt functionality in Advanced RAPID has some extra features, in addition to the interrupt features always included in RAPID. For more information on the basic interrupt functionality, see *Technical reference manual - RAPID overview*.

Here are some examples of interrupt applications that Advanced RAPID facilitates:

- Generate an interrupt when a persistent variable change value.
- Generate an interrupt when an error occurs, and find out more about the error.

What is included

The interrupt functionality in Advanced RAPID includes:

- **Data types for error interrupts:** `trapdata`, `errdomain`, and `errtype`.
- **Instructions for generating interrupts:** `IPers` and `IError`.
- **Instructions for finding out more about an error interrupt:** `GetTrapData` and `ReadErrData`.

2.1.8.2 RAPID components

Data types

This is a brief description of each data type in the interrupt functionality. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
trapdata	trapdata represents internal information related to the interrupt that caused the current trap routine to be executed.
errdomain	errdomain is used to specify an error domain. Depending on the nature of the error, it is logged in different domains.
errtype	errtype is used to specify an error type (error, warning, state change).

Instructions

This is a brief description of each instruction in the interrupt functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
IPers	IPers (Interrupt Persistent) is used to order an interrupt to be generated each time the value of a persistent variable is changed.
IError	IError (Interrupt Errors) is used to order an interrupt to be generated each time an error occurs.
GetTrapData	GetTrapData is used in trap routines generated by the instruction IError. GetTrapData obtains all information about the interrupt that caused the trap routine to be executed.
ReadErrData	ReadErrData is used in trap routines generated by the instruction IError. ReadErrData read the information obtained by GetTrapData.
ErrRaise	ErrRaise is used to create an error in the program and the call the error handler of the routine. ErrRaise can also be used in the error handler to propagate the current error to the error handler of the calling routine.

Functions

There are no RAPID functions for the interrupt functionality.

2.1.8.3 Interrupt functionality examples

Interrupt when persistent variable changes

In this example, a trap routine is called when the value of the persistent variable **counter** changes.

```
VAR intnum int1;
PERS num counter := 0;

PROC main()
  CONNECT int1 WITH iroutinel;
  IPers counter, int1;
  ...
  counter := counter + 1;
  ...
  Idelete int1;
ENDPROC

TRAP iroutinel
  TPWrite "Current value of counter = " \Num:=counter;
ENDTRAP
```

Error interrupt

In this example, a trap routine is called when an error occurs. The trap routine determines the error domain and the error number and communicates them via output signals.

```
VAR intnum err_interrupt;
VAR trapdata err_data;
VAR errdomain err_domain;
VAR num err_number;
VAR errtype err_type;

PROC main()
  CONNECT err_interrupt WITH trap_err;
  IError COMMON_ERR, TYPE_ERR, err_interrupt;
  ...
  a:=3;
  b:=0;
  c:=a/b;
  ...
  IDelete err_interrupt;
ENDPROC

TRAP trap_err
  GetTrapData err_data;
  ReadErrData err_data, err_domain, err_number, err_type;
  SetGO go_err1, err_domain;
  SetGO go_err2, err_number;
ENDTRAP
```

2.1.9 User message functionality

2.1.9.1 Overview

Purpose

The user message functionality is used to set up event numbers and facilitate the handling of event messages and other texts to be presented in the user interface.

Here are some examples of applications:

- Get user messages from a text table file, which simplifies updates and translations.
- Add system error number to be used as error recovery constants in RAISE instructions and for test in ERROR handlers.

What is included

The user message functionality includes:

- Text table operating instruction `TextTabInstall`.
- Text table operating functions: `TextTabFreeToUse`, `TextTabGet`, and `TextGet`.
- Instruction for error number handling: `BookErrNo`.

2.1.9.2 RAPID components

Data types

There are no RAPID data types for the user message functionality.

Instructions

This is a brief description of each instruction used for the user message functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
BookErrNo	BookErrNo is used to define a new RAPID system error number.
TextTabInstall	TextTabInstall is used to install a text table in the system.

Functions

This is a brief description of each function used for the user message functionality. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
TextTabFreeToUse	TextTabFreeToUse is used to test whether the text table name is free to use (not already installed in the system).
TextTabGet	TextTabGet is used to get the text table number of a user defined text table.
TextGet	TextGet is used to get a text string from the system text tables.

2.1.9.3 User message functionality examples

Book error number

This example shows how to add a new error number.

```
VAR intnum siglint;

!Introduce a new error number in a glue system.
!Note: The new error variable must be declared with the initial
      value -1
VAR errnum ERR_GLUEFLOW := -1;

PROC main()
  !Book the new RAPID system error number
  BookErrNo ERR_GLUEFLOW;

  !Raise glue flow error if dil=1
  IF dil=1 THEN
    RAISE ERR_GLUEFLOW;
  ENDIF
ENDPROC

!Error handling
ERROR
IF ERRNO = ERR_GLUEFLOW THEN
  ErrWrite "Glue error", "There is a problem with the glue flow";
ENDIF
```

Error message from text table file

This example shows how to get user messages from a text table file.

There is a text table named `text_table_name` in a file named `HOME:/language/en/text_file.xml`. This table contains error messages in english. The procedure `install_text` is executed at event **POWER ON**. The first time it is executed, the text table file `text_file.xml` is installed. The next time it is executed, the function `TextTabFreeToUse` returns **FALSE** and the installation is not repeated. The table is then used for getting user interface messages.

```
VAR num text_res_no;

PROC install_text()
  !Test if text_table_name is already installed
  IF TextTabFreeToUse("text_table_name") THEN
    !Install the table from the file HOME:/language/en/text_file.xml
    TextTabInstall "HOME:/language/en/text_file.xml";
  ENDIF
  !Assign the text table number for text_table_name to text_res_no
  text_res_no := TextTabGet("text_table_name");
ENDPROC

...
!Write error message with two strings from the table text_res_no
ErrWrite TextGet(text_res_no, 1), TextGet(text_res_no, 2);
```


2.1.9.4 Text table files

Overview

A text table is stored in an XML file (each file can contain one table in one language). This table can contain any number of text strings.

Explanation of the text table file

This is a description of the XML tags and arguments used in the text table file.

Tag	Argument	Description
Resource		Represents a text table. A file can only contain one instance of Resource.
	Name	The name of the text table. Used by the RAPID instruction TextTabGet.
	Language	Language code for the language of the text strings. Currently this argument is not being used. The RAPID instruction TextTabInstall can only handle English texts.
Text		Represents a text string.
	Name	The text string's number in the table.
Value		The text string to be used.
Comment		Comments about the text string and its usage.

Example of text table file

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<Resource Name="text_table_name" Language="en">
  <Text Name="1">
    <Value>This is a text that is </Value>
    <Comment>The first part of my text</Comment>
  </Text>
  <Text Name="2">
    <Value>displayed in the user interface.</Value>
    <Comment>The second part of my text</Comment>
  </Text>
</Resource>
```

2.1.10 RAPID support functionality

2.1.10.1 Overview

Purpose

The RAPID support functionality consists of miscellaneous routines that might be helpful for an advanced robot programmer.

Here are some examples of applications:

- Activate a new tool, work object or payload.
- Find out what an argument is called outside the current routine.
- Test if the program pointer has been moved during the last program stop.

What is included

RAPID support functionality includes:

- Instruction for activating specified system data: `SetSysData`.
- Function that gets original data object name: `ArgName`.
- Function for information about program pointer movement:
`IsStopStateEvent`.

2.1.10.2 RAPID components

Data types

There are no data types for RAPID support functionality.

Instructions

This is a brief description of each instruction used for RAPID support functionality. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
SetSysData	SetSysData activates (or changes the current active) tool, work object, or payload for the robot.

Functions

This is a brief description of each function used for RAPID support functionality. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
ArgName	ArgName is used to get the name of the original data object for the current argument or the current data.
IsStopStateEvent	IsStopStateEvent returns information about the movement of the program pointer.

2.1.10.3 RAPID support functionality examples

Activate tool

This is an example of how to activate a known tool:

```
!Activate tool1  
SetSysData tool1;
```

This is an example of how to activate a tool when the name of the tool is only available in a string:

```
VAR string tool_string := "tool2";  
!Activate the tool specified in tool_string  
SetSysData tool0 \ObjectName := tool_string;
```

Get argument name

In this example, the original name of par1 is fetched. The output will be "Argument name my_nbr with value 5".

```
VAR num my_nbr :=5;  
procl my_nbr;  
  
PROC procl (num par1)  
  VAR string name;  
  name:=ArgName(par1);  
  TPWrite "Argument name "+name+" with value " \Num:=par1;  
ENDPROC
```

Test if program pointer has been moved

This example tests if the program pointer was moved during the last program stop.

```
IF IsStopStateEvent (\PPMoved) = TRUE THEN  
  TPWrite "The program pointer has been moved.";  
ENDIF
```

2.2 Analog Signal Interrupt

2.2.1 Introduction to Analog Signal Interrupt

Purpose

The purpose of Analog Signal Interrupt is to supervise an analog signal and generate an interrupt when a specified value is reached.

Analog Signal Interrupt is faster, easier to implement, and require less computer capacity than polling methods.

Here are some examples of applications:

- Save cycle time with better timing (start robot movement exactly when a signal reach the specified value, instead of waiting for polling).
- Show warning or error messages if a signal value is outside its allowed range.
- Stop the robot if a signal value reaches a dangerous level.

What is included

The RobotWare base functionality Analog Signal Interrupt gives you access to the instructions:

- `ISignalAI`
- `ISignalAO`

Basic approach

This is the general approach for using Analog Signal Interrupt. For a more detailed example of how this is done, see [Code example on page 47](#).

- 1 Create a trap routine.
- 2 Connect the trap routine using the instruction `CONNECT`.
- 3 Define the interrupt conditions with the instruction `ISignalAI` or `ISignalAO`.

Limitations

Analog signals can only be used if you have an industrial network option (for example DeviceNet or PROFIBUS).

2.2.2 RAPID components

Data types

Analog Signal Interrupt includes no data types.

Instructions

This is a brief description of each instruction in Analog Signal Interrupt. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
ISignalAI	Defines the values of an analog input signal, for which an interrupt routine shall be called. An interrupt can be set to occur when the signal value is above or below a specified value, or inside or outside a specified range. It can also be specified if the interrupt shall occur once or repeatedly.
ISignalAO	Defines the values of an analog output signal, for which an interrupt routine shall be called. An interrupt can be set to occur when the signal value is above or below a specified value, or inside or outside a specified range. It can also be specified if the interrupt shall occur once or repeatedly.

Functions

Analog Signal Interrupt includes no RAPID functions.

2.2.3 Code example

Temperature surveillance

In this example a temperature sensor is connected to the signal `ail`.

An interrupt routine with a warning is set to execute every time the temperature rises 0.5 degrees in the range 120-130 degrees. Another trap routine, stopping the robot, is set to execute as soon as the temperature rise above 130 degrees.

```
VAR intnum ail_warning;
VAR intnum ail_exceeded;

PROC main()
  CONNECT ail_warning WITH temp_warning;
  CONNECT ail_exceeded WITH temp_exceeded;
  ISignalAI ail, AIO_BETWEEN, 130, 120, 0.5, \DPos, ail_warning;
  ISignalAI \Single, ail, AIO_ABOVE_HIGH, 130, 120, 0, ail_exceeded;
  ...
  IDelete ail_warning;
  IDelete ail_exceeded;
ENDPROC

TRAP temp_warning
  TPWrite "Warning: Temperature is "\Num:=ail;
ENDTRAP

TRAP temp_exceeded
  TPWrite "Temperature is too high";
  Stop;
ENDTRAP
```

2.3 Auto Acknowledge Input

Description

Auto Acknowledge Input is a system input which will acknowledge the dialog presented on the FlexPendant when switching from operator mode manual to auto with the key switch on the robot controller.



WARNING

Note that using such an input will be contrary to the regulations in the safety standard ISO 10218-1 chapter 5.3.5 Single point of control with following text: "The robot control system shall be designed and constructed so that when the robot is placed under local pendant control or other teaching device control, initiation of robot motion or change of local control selection from any other source shall be prevented."

Thus it is absolutely necessary to use other means of safety to maintain the requirements of the standard and the machinery directive and also to make a risk assessment of the completed cell. Such additional arrangements and risk assessment is the responsibility of the system integrator and the system must not be put into service until these actions have been completed

Limitations

The system parameter cannot be defined using the FlexPendant or RobotStudio, only with a text string in the I/O configuration file.

Activate Auto Acknowledge Input

Use the following procedure to activate the system input for *Auto Acknowledge Input*.

	Action
1	Save a copy of the I/O configuration file, <i>eio.cfg</i> , using the FlexPendant or RobotStudio.
2	Edit the I/O configuration file, <i>eio.cfg</i> , using a text editor. Add the following line in the group SYSSIG_IN: -Signal "my_signal_name" -Action "AckAutoMode" my_signal_name is the name of the configured digital input signal that should be used as the system input.
3	Save the file and reload it to the controller.
4	Restart the system to activate the signal.

2.4 Electronically Linked Motors

2.4.1 Overview

Description

Electronically Linked Motors makes a master/follower configuration of motors (for example two additional axes). The follower axis will continuously follow the master axis in terms of position, velocity, and acceleration.

For stiff mechanical connection between the master and followers, the torque follower function can be used. Instead of regulating to exactly the same position for the master and follower, the torque is distributed between the axes. A small position error between master and follower will occur depending on backlash and mechanical misalignment.

Purpose

The primary purpose of Electronically Linked Motors is to replace driving shafts of gantry machines, but the base functionality can be used to control any other set of motors as well.

What is included

The RobotWare base functionality Electronically Linked Motors gives you access to:

- a service program for defining linked motor groups and trimming the axis positions
- system parameters used to configure a follower axis

Basic approach

This is the general approach for setting up Electronically Linked Motors. For a more detailed description of how this is done, see the respective section.

- 1 Configure the additional axes that you want to use. See *Application manual - Additional axes and stand alone controller*.
- 2 Configure tolerance limits in the system parameters, in the types *Linked M Process*, *Process*, and *Joint*.
- 3 Restart the controller for the changes to take effect.
- 4 Set values to data variables, defining the linked motor group and connecting follower and master axes.
- 5 Use the service program to trim positions or reset follower after position error.

Limitations

There can be up to 5 follower axes. The follower axes can be configured to follow one master each, or several followers can follow one master, but the total number of follower axes cannot be more than 5.

The follower axis cannot be an ABB robot (IRB robot). The master axis can be either an additional axis or a robot axis.

Continues on next page

The torque follower function can only be used if the follower axis is connected to the same drive module as the master axis.

Using the torque follower functionality might reduce the number of follower axes depending on the number of axes that are available in the drive module where master axis is configured.

The RAPID instruction `IndReset` (*Independent Reset*) cannot be used in combination with Electronically Linked Motors.

2.4.2 Configuration

2.4.2.1 System parameters

About the system parameters

This is a brief description of each parameter used for Electronically Linked Motors. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Joint

These parameters belong to the topic *Motion* and the type *Joint*.

Parameter	Description
Follower to Joint	Specifies which master axis this axis shall follow. Refers to the parameter <i>Name</i> in the type <i>Joint</i> . Robot axes are referred to as rob1 followed by underscore and the axis number (for example rob1_6).
Use Process	Id name of the process that is called. Refers to the parameter <i>Name</i> in the type <i>Process</i> .
Lock Joint in lpol	A flag that locks the axis so it is not used in the path interpolation. This parameter must be set to TRUE when the axis is electronically linked to another axis.

Process

These parameters belong to the topic *Motion* and the type *Process*.

Parameter	Description
Name	Id name of the process.
Use Linked Motor Process	Id name of electronically linked motor process. Refers to the parameter <i>Name</i> in the type <i>Linked M Process</i> .

Linked M Process

These parameters belong to the topic *Motion* and the type *Linked M Process*.

Parameter	Description
Name	Id name for the linked motor process.
Offset Adjust Delay Time	Time delay from control on until the follower starts to follow the master. This can be used to give the master time to stabilize before the follower starts following.
Max Follower Offset	The maximum allowed difference in distance (in radians or meters) between master and follower. If <i>Max Follower Offset</i> is exceeded, emergency stop is activated.
Max Offset Speed	The maximum allowed difference in speed (in rad/s or m/s) between master and follower. If <i>Max Offset Speed</i> is exceeded, emergency stop is activated.
Offset Speed Ratio	Defines how large part of the <i>Max Offset Speed</i> that can be used to compensate for position error.

Continues on next page

2 RobotWare-OS

2.4.2.1 System parameters

Continued

Parameter	Description
Ramp Time	Time for acceleration up to <i>Max Offset Speed</i> . The proportion constant for position regulation is ramped from zero up to its final value (<i>Master Follower kp</i>) during <i>Ramp Time</i> .
Master Follower kp	The proportion constant for position regulation. Determines how fast the position error is compensated.
Torque follower	Set to True if the follower and master should share torque instead of regulating on exact position. This can only be used if the follower axis is connected to the same drive module as the master axis.
Torque distribution	The ratio (of the total torque) that should be applied to the follower (for example 0.3 result in 30% on follower and 70% on master). If drive and motors are equal this is normally set to 0.5.
Follower axis pos. acc. reduction	This value is set to reduce the accuracy of the follower position loop. This is needed in cases where the mechanical structure gives high torques between the motors due to large position mismatch in a stiff mechanical connection etc. <ul style="list-style-type: none">• 0: accuracy reduction not active• 10-30 typical values

2.4.2.2 Configuration example

About this example

This is an example of how to configure the additional axis M8DM1 to be a follower to the axis M7DM1 and axis M9DM1 to be a follower to robot axis 6.

Joint

Name	Follower to Joint	Use Process	Lock Joint in Ipol
M7DM1			
M8DM1	M7DM1	ELM_1	True
M9DM1	rob1_6	ELM_2	True

Process

Name	Use Linked Motor Process
ELM_1	Linked_m_1
ELM_2	Linked_m_2

Linked M Process

Name	Offset Adjust Delay Time	Max Follow-er Offset	Max Offset Speed	Offset Speed Ratio	Ramp Time	Master Follower kp
Linked_m_1	0.2	0.05	0.05	0.33	1	0.05
Linked_m_2	0.1	0.1	0.1	0.4	1.5	0.08

2.4.3 Managing a follower axis

2.4.3.1 Using the service program

About the service program

The service program is used when you need to:

- calibrate the follower axis
- reset follower after a position error
- tune a torque follower axis, see [Tuning a torque follower on page 58](#).

Data variables

At start up the service routine will read values from system parameters and set the values for a set of data variables used by the service routine. These variables only need to be set manually if something goes wrong, see [Data setup on page 61](#).

Start service program



Note

The controller must be in manual or auto mode to run this service program.

Step	Action
1	In the program view, tap Debug and select Call Routine....
2	Select Linked_m and tap Go to .
3	Press the RUN button to start the service program. The service program is shown on the screen.
4	Tap Menu 1 . The follower axes that are set up in the system are shown in the task bar.
5	Tap the follower axis you want to use the service program for. The main menu of the service program is now shown.

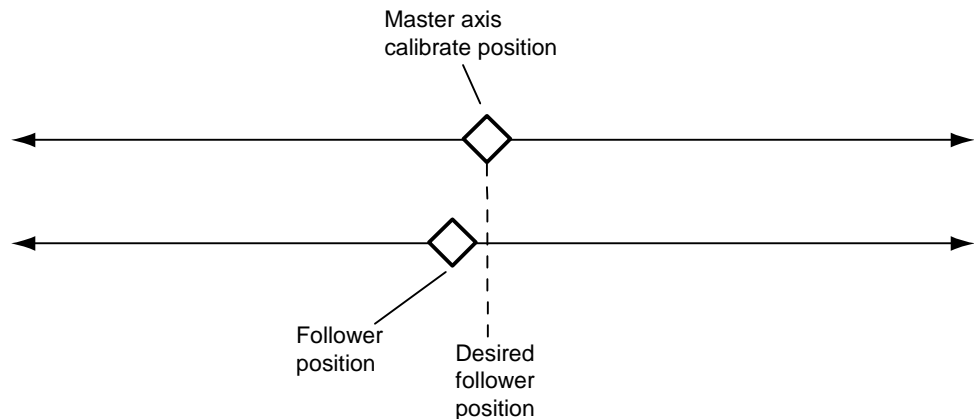
Menu buttons

Button	Description
AUTO	Automatically moves the follower axis to the position corresponding to the master axis, see Reset follower automatically on page 57 .
STOP	Stops the movement of the follower axis. Can be used when jogging or using AUTO and the movement must be stopped immediately.
JOG	Manual stepwise movement of the follower axis, see Jog follower axis on page 55 . If the follower axis is synchronized with the master axis, it will resume its position when you tap AUTO or when you exit the service program.
UNSYNC	Used to suspend the synchronization between follower axis and master axis, see Unsyncronize on page 55 .
HELP	Show some help for how to use the service program. The button Next shows the next help subject.

2.4.3.2 Calibrate follower axis position

Overview

Before the follower axis can follow the master axis, you must define the calibration positions for both master and follower.



en0400000963

This calibration is done by following the procedures below:

- 1 Jog the master axis to its calibration position.
- 2 Unsynchronize the follower and master axes. See [Unsynchronize on page 55](#).
- 3 Jog the follower to the desired position. See [Jog follower axis on page 55](#).
- 4 Fine calibrate follower axis. See [Fine calibrate on page 56](#).

Unsynchronize

Step	Action
1	In the main menu of the service program, tap UNSYNC .
2	Confirm that you want to unsynchronize the axes by tapping YES .
3	Restart the controller when an information text tells you to do it. After the restart the follower axis is no longer synchronized with the master axis.

Jog follower axis

Step	Action
1	In the main menu of the service program, tap JOG .
2	Select the speed with which the follower axis should move when you jog it.
3	Select the step size with which the follower axis should move for each step you jog it.
4	Tap on Positive or Negative , depending on in which direction you want to move the follower axis. Jog the follower axis until it is exactly in the calibration position (the position that corresponds to the master axis calibration position).

Continues on next page

2 RobotWare-OS

2.4.3.2 Calibrate follower axis position

Continued

Fine calibrate

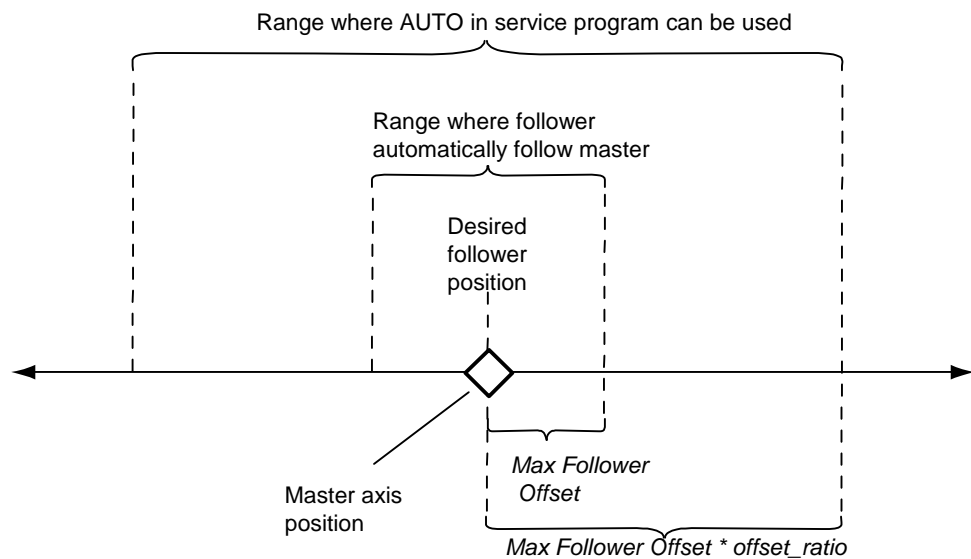
Step	Action
1	In the ABB menu, select Calibration .
2	Select the mechanical unit that the follower axis belongs to.
3	Tap the button Calib. Parameters .
4	Tap Fine Calibration....
5	In the warning dialog that appears, tap Yes .
6	Select the axis that is used as follower axis and tap Calibrate .
7	In the warning dialog that appears, tap Calibrate . The follower axis is now calibrated. As soon as the follower is calibrated, it is also synchronized with the master again.

2.4.3.3 Reset follower axis

Overview

If the follower offset exceeds its tolerance limits (configured with the system parameter *Max follower offset*), the service program must be used to move the follower back within the tolerance limits. This can be done automatically in the service program if the follower is within the AUTO range. Otherwise the follower must be manually jogged.

The range where AUTO can be used is determined by the system parameter *Max Follower Offset* multiplied with the data variable *offset_ratio*.



en0400000962

Reset follower automatically

Step	Action
1	In the main menu of the service program, tap AUTO .
2	Select the speed with which the follower axis should move to its desired position.

Reset follower by manual jogging

Step	Action
1	In the main menu of the service program, tap JOG .
2	Select the speed with which the follower axis should move when you jog it.
3	Select the step size with which the follower axis should move for each step you jog it.
4	Tap on Positive or Negative , depending on which direction you want to move the follower axis. Jog the follower until it is within the tolerance of <i>Max Follower Offset</i> (or use AUTO when you are close enough).

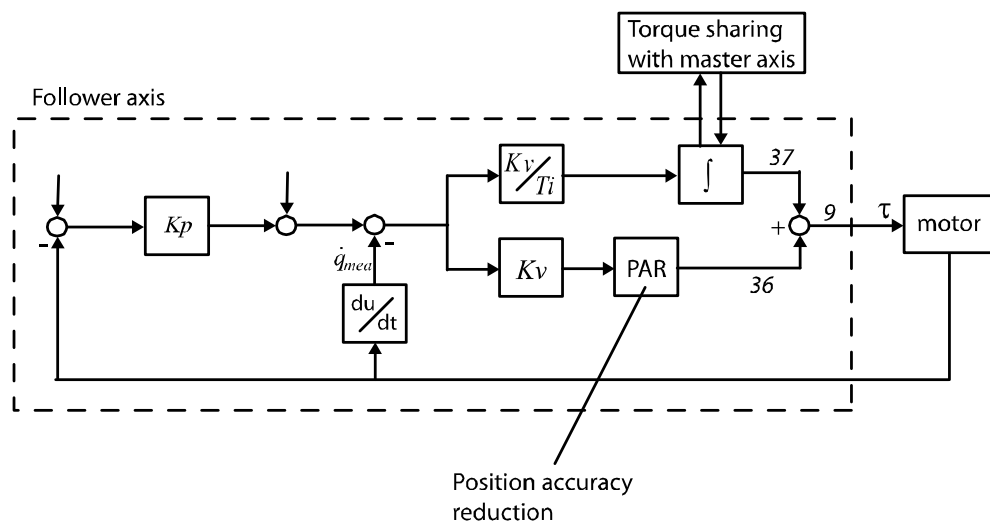
2.4.4 Tuning a torque follower

2.4.4.1 Torque follower descriptions

About torque followers

The follower axis can be setup so the torque is shared between the master and the follower. This is only allowed if the follower axis is connected to the same drive module as the master axis.

Below is a simplified picture of the control loop of the follower axis.



en0900000679

Torque distribution

The sharing of torque will be done on the integral part of the control loops. By setting torque distribution to 0.5, the master and follower will have equal part of the integral part of the total torque. A value of 0.3 will make the follower axis have 30% of the integral torque and the master axis 70%.

Position accuracy reduction

If the mechanical structure is very stiff and has a mechanical misalignment or a large backlash, the proportional part will be a major part of the total torque. If this becomes a problem with too high torque difference between the master and the follower the *position accuracy reduction* function (PAR in the illustration) can be used. This will make the follower axis less accurate when it comes in to a position. This will make the follower act more like a true torque follower.

Test signals that can be useful to check the behavior of this is:

Test signal	Test signal number
Integral part of torque	37
Proportional part of torque	36
Total torque ref (also including any feed forward torque)	9

2.4.4.2 Using the service program

About the service program for torque follower

The part of the service program for torque follower is used to find the suitable values of some parameters. Once the values are found, system parameters are updated and a new fine calibration is made. After that, there is no need for any tuning of the torque follower.

Opening the tune torque follower menu

	Action	Illustration
1	Start the service program (as described by the first steps in Start service program on page 54 .	
2	Tap Menu 2 .	
3	Tap on the name of the follower axis to tune.	
4	Use the tune torque follower menu as described below.	

Tuning the torque distribution

Use this procedure to change the distribution of torque between the master and the follower axis.

	Action	Illustration
1	Tap Torque distribution .	
2	Type a number (between 0 and 1) for the follower's share of the total torque. For example, 0.3 will result in 30% of the torque on the follower and 70% on the master.	
3	To update the system parameters using the new value, tap Store to cfg . If not saved to cfg, the new value will be used until the robot controller is restarted, but the value will be lost at restart.	

Tuning the position accuracy reduction

Use this procedure to set the position accuracy reduction of the torque follower axis.

	Action	Illustration
1	Tap Position accuracy reduction .	
2	Type a number for reduced position accuracy. 0 means no position accuracy reduction. 10 -30 is typically used for a torque follower to reduce the torque tension between the master and the follower.	

Continues on next page

2 RobotWare-OS

2.4.4.2 Using the service program

Continued

	Action	Illustration
3	To update the system parameters using the new value, tap Store to cfg . If not saved to cfg, the new value will be used until the robot controller is restarted, but the value will be lost at restart.	

Tuning the temporary position delta

Use this procedure to tune the position delta of the torque follower axis. This delta value is then used to adjust the fine calibration of the follower axis.

	Action	Illustration
1	Tap Temp. position delta .	
2	Type a number (degrees on motor side) that will be added to the position reference for the follower axis.	
3	Test which value results in the lowest torque tension and make a fine calibration of the master axis. This will update the follower axis with the current position delta.	

2.4.5 Data setup

2.4.5.1 Set up data for service program

Overview

At start of the service routine for Electronically Linked Motors, some data variables are read from the linked motor configuration. These variables are used by the service program. If they are not read correctly, the variables need to be edited in the service program.

Data descriptions

Data variable	Description
<code>l_f_axis_name</code>	A name for the follower axis that will be displayed on the FlexPendant. String array with 5 elements, one for each follower axis. If you only have one linked motor, use only the first element.
<code>l_f_mecunt_n</code>	The name of the mechanical unit for the follower axis. Refers to the system parameter <i>Name</i> in the type <i>Mechanical Unit</i> . String array with 5 elements, one for each follower axis. If you only have one linked motor, use only the first element.
<code>l_f_axis_no</code>	Defines which axis in the mechanical unit (<code>l_f_mecunt_n</code>) is the follower axis. Num array with 5 elements, one for each follower axis. If you only have one linked motor, use only the first element.
<code>l_m_mecunt_n</code>	The name of the mechanical unit for the master axis. Refers to the system parameter <i>Name</i> in the type <i>Mechanical Unit</i> . String array with 5 elements, one for each master axis. If you only have one linked motor, use only the first element.
<code>l_m_axis_no</code>	Defines which axis in the mechanical unit (<code>l_m_mecunt_n</code>) is the master axis. Num array with 5 elements, one for each master axis. If you only have one linked motor, use only the first element.
<code>offset_ratio</code>	Defines the range where the AUTO function in the service program reset the follower axis. <code>offset_ratio</code> defines this range as a multiple of the range where the follower automatically follow the master (defined with the parameter <i>Max Follow Offset</i>). If the follower has a position error that is larger than <i>Max Follower Offset</i> * <code>offset_ratio</code> , the follower must be reset manually. For more information, see Reset follower axis on page 57 .
<code>speed_ratio</code>	Defines the speed of the follower axis when controlled by the service program. The values are given as a part of the maximum allowed manual speed (that is, the value 0.5 means half the max manual speed). Num array with 20 elements. Elements 1-5 define the speed "very slow" for each follower axis. Elements 6-10 define "slow", elements 11-15 define "normal" and elements 16-20 define "fast". If you only have one linked motor, use only elements 1, 6, 11 and 16.

Continues on next page

2 RobotWare-OS

2.4.5.1 Set up data for service program

Continued

Data variable	Description
displacement	<p>Defines the distance the follower axis will move for each tap on Positive or Negative when jogging the follower axis from the service program. The values are given in degrees or meters, depending on if the follower axis is circular or linear.</p> <p>Num array with 20 elements. Elements 1-5 define the displacement "very short" for each follower axis. Elements 6-10 define "short", elements 11-15 define "normal" and elements 16-20 define "long". If you only have one linked motor, use only elements 1, 6, 11 and 16.</p>

Edit data variables

This is a description of how to set values for the data variables from the FlexPendant.

Step	Action
1	In the ABB menu, select Program Data .
2	Select string and tap Show Data .
3	Select I_f_axis_name and tap Edit Value .
4	Tap the first element.
5	Tap the line to edit it.
6	Enter the name you want to give your first follower axis.
7	If you have more than one follower axis, repeat step 4-6 for the next elements.
8	Repeat step 3-7 for I_f_mecunt_n and I_m_mecunt_n .
9	In the Program Data menu, select num and repeat step 3-7 for I_f_axis_no , I_m_axis_no , offset_ratio , speed_ratio and displacement .

2.4.5.2 Example of data setup

About this example

This is an example of how to set up the data variables for two follower axis. The first follower axis is M8C1B1, which is a follower to the additional axis M7C1B1. The second follower axis is M9C1B1, which is a follower to robot axis 6.

I_f_axis_name

Represented axis	Element and value in I_f_axis_name
Follower 1	{1}: "follow_external"
Follower 2	{2}: "follow_axis6"
Follower 3	{3}: ""
Follower 4	{4}: ""
Follower 5	{5}: ""

I_f_mecunt_n

Represented axis	Element and value in I_f_mecunt_n
Follower 1	{1}: "M8DM1"
Follower 2	{2}: "M9DM1"
Follower 3	{3}: ""
Follower 4	{4}: ""
Follower 5	{5}: ""

I_f_axis_no

Represented axis	Element and value in I_f_axis_no
Follower 1	{1}: 1
Follower 2	{2}: 1
Follower 3	{3}: 0
Follower 4	{4}: 0
Follower 5	{5}: 0

I_m_mecunt_n

Represented axis	Element and value in I_m_mecunt_n
Master 1	{1}: "M7DM1"
Master 2	{2}: "rob1"
Master 3	{3}: ""
Master 4	{4}: ""
Master 5	{5}: ""

Continues on next page

2 RobotWare-OS

2.4.5.2 Example of data setup

Continued

I_m_axis_no

Represented axis	Element and value in I_m_axis_no
Master 1	{1}: 1
Master 2	{2}: 6
Master 3	{3}: 0
Master 4	{4}: 0
Master 5	{5}: 0

offset_ratio

Represented axis	Element and value in offset_ratio
Follower 1	{1}: 10
Follower 2	{2}: 15
Follower 3	{3}: 0
Follower 4	{4}: 0
Follower 5	{5}: 0

speed_ratio

Represented axis	very slow	slow	normal	fast
Follower 1	{1}: 0.01	{6}: 0.05	{11}: 0.2	{16}: 1
Follower 2	{2}: 0.01	{7}: 0.05	{12}: 0.2	{17}: 1
Follower 3	{3}: 0	{8}: 0	{13}: 0	{18}: 0
Follower 4	{4}: 0	{9}: 0	{14}: 0	{19}: 0
Follower 5	{5}: 0	{10}: 0	{15}: 0	{20}: 0

displacement

Represented axis	very short	short	normal	long
Follower 1	{1}: 0.001	{6}: 0.005	{11}: 0.02	{16}: 0.1
Follower 2	{2}: 0.01	{7}: 0.1	{12}: 1	{17}: 10
Follower 3	{3}: 0	{8}: 0	{13}: 0	{18}: 0
Follower 4	{4}: 0	{9}: 0	{14}: 0	{19}: 0
Follower 5	{5}: 0	{10}: 0	{15}: 0	{20}: 0

2.5 Fixed Position Events

2.5.1 Overview

Purpose

The purpose of Fixed Position Events is to make sure a program routine is executed when the position of the TCP is well defined.

If a move instruction is called with the zone argument set to `fine`, the next routine is always executed once the TCP has reached its target. If a move instruction is called with the zone argument set to a distance (for example `z20`), the next routine may be executed before the TCP is even close to the target. This is because there is always a delay between the execution of RAPID instructions and the robot movements.

Calling the move instruction with zone set to `fine` will slow down the movements. With Fixed Position Events, a routine can be executed when the TCP is at a specified position anywhere on the TCP path without slowing down the movement.

What is included

The RobotWare base functionality Fixed Position Events gives you access to:

- instructions used to define a position event
- instructions for moving the robot and executing the position event at the same time
- instructions for moving the robot and calling a procedure while passing the target, without first defining a position event

Basic approach

Fixed Position Events can either be used with one simplified instruction calling a procedure or it can be set up following these general steps. For more detailed examples of how this is done, see [Code examples on page 69](#).

- 1 Declare the position event.
- 2 Define the position event:
 - when it shall occur, compared to the target position
 - what it shall do
- 3 Call a move instruction that uses the position event. When the TCP is as close to the target as defined, the event will occur.

2.5.2 RAPID components and system parameters

Data types

This is a brief description of each data type in Fixed Position Events. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
triggdata	triggdata is used to store data about a position event. A position event can take the form of setting an output signal or running an interrupt routine at a specific position along the movement path of the robot. triggdata also contains information on when the action shall occur, for example when the TCP is at a defined distance from the target. triggdata is a non-value data type.
triggios	triggios is used to store data about a position event used by the instruction TriggLIOS. triggios sets the value of an output signal using a num value.
triggiosdnum	triggiosdnum is used to store data about a position event used by the instruction TriggLIOS. triggiosdnum sets the value of an output signal using a dnum value.
triggstrgo	triggstrgo is used to store data about a position event used by the instruction TriggLIOS. triggstrgo sets the value of an output signal using a stringdig value (string containing a number).

Instructions

This is a brief description of each instruction in Fixed Position Events. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
TriggIO	TriggIO defines the setting of an output signal and when to set that signal. The definition is stored in a variable of type triggdata. TriggIO can define the setting of the signal to occur at a certain distance (in mm) from the target, or a certain time from the target. It is also possible to set the signal at a defined distance or time from the starting position. By setting the distance to 0 (zero), the signal will be set when the TCP is as close to the target as it gets (the middle of the corner path).
TriggEquip	TriggEquip works like TriggIO, with the difference that TriggEquip can compensate for the internal delay of the external equipment. For example, the signal to a glue gun must be set a short time before the glue is pressed out and the gluing begins.
TriggInt	TriggInt defines when to run an interrupt routine. The definition is stored in a variable of type triggdata. TriggInt defines at what distance (in mm) from the target (or from the starting position) the interrupt routine shall be called. By setting the distance to 0 (zero), the interrupt will occur when the TCP is as close to the target as it gets (the middle of the corner path).

Continues on next page

Instruction	Description
TriggCheckIO	<p>TriggCheckIO defines a test of an input or output signal, and when to perform that test. The definition is stored in a variable of type <code>triggdata</code>.</p> <p>TriggCheckIO defines a test, comparing an input or output signal with a value. If the test fails, an interrupt routine is called. As an option the robot movement can be stopped when the interrupt occurs.</p> <p>TriggCheckIO can define the test to occur at a certain distance (in mm) from the target, or a certain time from the target. It is also possible to perform the test at a defined distance or time from the starting position.</p> <p>By setting the distance to 0 (zero), the interrupt routine will be called when the TCP is as close to the target as it gets (the middle of the corner path).</p>
TriggRampAO	<p>TriggRampAO defines the ramping up or down of an analog output signal and when this ramping is performed. The definition is stored in a variable of type <code>triggdata</code>.</p> <p>TriggRampIO defines where the ramping of the signal is to start and the length of the ramping.</p>
TriggL	<p>TriggL is a move instruction, similar to MoveL. In addition to the movement the TriggL instruction can set output signals, run interrupt routines and check input or output signals at fixed positions.</p> <p>TriggL executes up to 8 position events stored as <code>triggdata</code>. These must be defined before calling TriggL.</p>
TriggC	<p>TriggC is a move instruction, similar to MoveC. In addition to the movement the TriggC instruction can set output signals, run interrupt routines and check input or output signals at fixed positions.</p> <p>TriggC executes up to 8 position events stored as <code>triggdata</code>. These must be defined before calling TriggC.</p>
TriggJ	<p>TriggJ is a move instruction, similar to MoveJ. In addition to the movement the TriggJ instruction can set output signals, run interrupt routines and check input or output signals at fixed positions.</p> <p>TriggJ executes up to 8 position events stored as <code>triggdata</code>. These must be defined before calling TriggJ.</p>
TriggLIOS	<p>TriggLIOS is a move instruction, similar to MoveL. In addition to the movement the TriggLIOS instruction can set output signals at fixed positions.</p> <p>TriggLIOS is similar to the combination of TriggEquip and TriggL. The difference is that TriggLIOS can handle up to 50 position events stored as an array of datatype <code>triggios</code>, <code>triggiosdnum</code>, or <code>triggstrgo</code>.</p>
MoveLSync	MoveLSync is a linear move instruction that calls a procedure in the middle of the corner path.
MoveCSync	MoveCSync is a circular move instruction that calls a procedure in the middle of the corner path.
MoveJSync	MoveJSync is a joint move instruction that calls a procedure in the middle of the corner path.

Functions

Fixed Position Events includes no RAPID functions.

Continues on next page

2 RobotWare-OS

2.5.2 RAPID components and system parameters

Continued

System parameters

This is a brief description of each parameter in Fixed Position Events. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Parameter	Description
Event Preset Time	TriggEquip takes advantage of the delay between the RAPID execution and the robot movement, which is about 70 ms. If the delay of the equipment is longer than 70 ms, then the delay of the robot movement can be increased by configuring <i>Event preset time</i> . <i>Event preset time</i> belongs to the type <i>Motion System</i> in the topic <i>Motion</i> .

2.5.3 Code examples

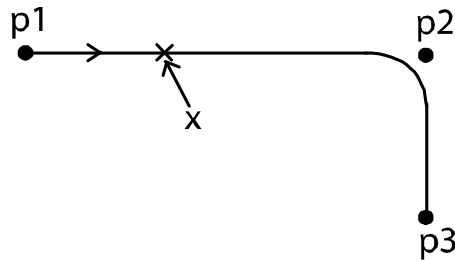
Example without Fixed Position Events

Without the use of Fixed Position Events, the code can look like this:

```
MoveJ p1, vmax, fine, tool1;
MoveL p2, v1000, z20, tool1;
SetDO do1, 1;
MoveL p3, v1000, fine, tool1;
```

Result

The code specifies that the TCP should reach p2 before setting do1. Because the robot path is delayed compared to instruction execution, do1 is set when the TCP is at the position marked with X (see illustration).



xx0300000151

Example with TriggIO and TriggL instructions

Setting the output signal 30 mm from the target can be arranged by defining the position event and then moving the robot while the system is executing the position event.

```
VAR triggdata do_set;
!Define that do1 shall be set when 30 mm from target
TriggIO do_set, 30 \DOp:=do1, 1;
MoveJ p1, vmax, fine, tool1;
!Move to p2 and let system execute do_set
TriggL p2, v1000, do_set, z20, tool1;
MoveL p3, v1000, fine, tool1;
```

Continues on next page

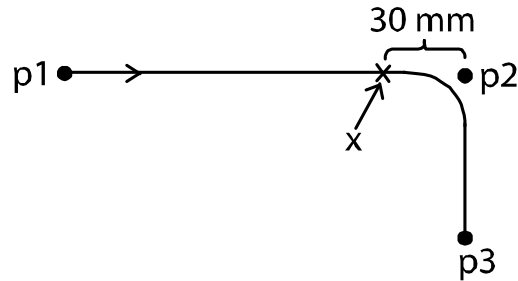
2 RobotWare-OS

2.5.3 Code examples

Continued

Result

The signal `do1` will be set when the TCP is 30 mm from `p2`. `do1` is set when the TCP is at the position marked with X (see illustration).



xx0300000158

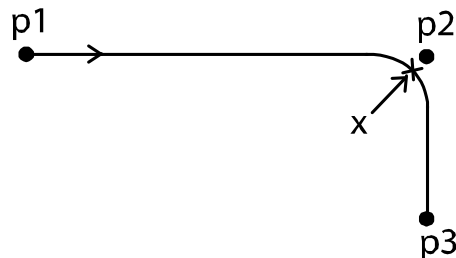
Example with MoveLSync instruction

Calling a procedure when the robot path is as close to the target as possible can be done with one instruction call.

```
MoveJ p1, vmax, fine, tool1;  
!Move to p2 while calling a procedure  
MoveLSync p2, v1000, z20, tool1, "proc1";  
MoveL p3, v1000, fine, tool1;
```

Result

The procedure will be called when the TCP is at the position marked with X (see illustration).



xx0300000165

2.6 File and Serial Channel Handling

2.6.1 Introduction to File and Serial Channel Handling

About File and Serial Channel Handling

The RobotWare base functionality File and Serial Channel Handling gives the robot programmer control of files, fieldbuses, and serial channels from the RAPID code. This can, for example, be useful for:

- Reading from a bar code reader.
- Writing production statistics to a log file or to a printer.
- Transferring data between the robot and a PC.

The functionality included in File and Serial Channel Handling can be divided into groups:

Functionality group	Description
Binary and character based communication	Basic communication functionality. Communication with binary or character based files or serial channels.
Raw data communication	Data packed in a container. Especially intended for fieldbus communication.
File and directory management	Browsing and editing of file structures.

2.6.2 Binary and character based communication

2.6.2.1 Overview

Purpose

The purpose of binary and character based communication is to:

- store information in a remote memory or on a remote disk
- let the robot communicate with other devices

What is included

To handle binary and character based communication, the RobotWare base functionality File and Serial Channel Handling gives you access to:

- instructions for manipulations of a file or serial channel
- instructions for writing to file or serial channel
- instruction for reading from file or serial channel
- functions for reading from file or serial channel.

Basic approach

This is the general approach for using binary and character based communication. For a more detailed example of how this is done, see [Code examples on page 74](#).

- 1 Open a file or serial channel.
- 2 Read or write to the file or serial channel.
- 3 Close the file or serial channel.

Limitations

Access to files, serial channels and field busses cannot be performed from different RAPID tasks simultaneously. Such an access is performed by all instruction in binary and character based communication, as well as `WriteRawBytes` and `ReadRawBytes`. E.g. if a `ReadBin` instruction is executed in one task, it must be ready before a `WriteRawBytes` can execute in another task.

2.6.2.2 RAPID components

Data types

This is a brief description of each data type used for binary and character based communication. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
iodev	iodev contains a reference to a file or serial channel. It can be linked to the physical unit with the instruction <code>Open</code> and then used for reading and writing.

Instructions

This is a brief description of each instruction used for binary and character based communication. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
Open	Open is used to open a file or serial channel for reading or writing.
Close	Close is used to close a file or serial channel.
Rewind	Rewind sets the file position to the beginning of the file.
ClearIOBuff	ClearIOBuff is used to clear the input buffer of a serial channel. All buffered characters from the input serial channel are discarded.
Write	Write is used to write to a character based file or serial channel.
WriteBin	WriteBin is used to write a number of bytes to a binary serial channel or file.
WriteStrBin	WriteStrBin is used to write a string to a binary serial channel or file.
WriteAnyBin	WriteAnyBin is used to write any type of data to a binary serial channel or file.
ReadAnyBin	ReadAnyBin is used to read any type of data from a binary serial channel or file.

Functions

This is a brief description of each function used for binary and character based communication. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
ReadNum	ReadNum is used to read a number from a character based file or serial channel.
ReadStr	ReadStr is used to read a string from a character based file or serial channel.
ReadBin	ReadBin is used to read a byte (8 bits) from a file or serial channel. This function works on both binary and character based files or serial channels.
ReadStrBin	ReadStrBin is used to read a string from a binary serial channel or file.

2.6.2.3 Code examples

Communication with character based file

This example show writing and reading to and from a character based file. The line "The number is :8" is written to FILE1.DOC. The contents of FILE1.DOC is then read and the output to the FlexPendant is "The number is :8" followed by "The number is 8".

```
PROC write_to_file()
  VAR iodev file;
  VAR num number:= 8;
  Open "HOME:" \File:= "FILE1.DOC", file;
  Write file, "The number is :"\Num:=number;
  Close file;
ENDPROC

PROC read_from_file()
  VAR iodev file;
  VAR num number;
  VAR string text;

  Open "HOME:" \File:= "FILE1.DOC", file \Read;
  TPWrite ReadStr(file);
  Rewind file;
  text := ReadStr(file\Delim:=":");
  number := ReadNum(file);
  Close file;
  TPWrite text \Num:=number;
ENDPROC
```

Communication with binary serial channel

In this example, the string "Hello", the current robot position and the string "Hi" is written to the binary serial channel com1.

```
PROC write_bin_chan()
  VAR iodev channel;
  VAR num out_buffer{20};
  VAR num input;
  VAR robtarg target;

  Open "com1:", channel\Bin;

  ! Write control character enq
  out_buffer{1} := 5;
  WriteBin channel, out_buffer, 1;

  ! Wait for control character ack
  input := ReadBin (channel \Time:= 0.1);
  IF input = 6 THEN
    ! Write "Hello" followed by new line
    WriteStrBin channel, "Hello\0A";
```

Continues on next page

```
! Write current robot position
target := CRobT(\Tool:= tool1\WObj:= wobj1);
WriteAnyBin channel, target;

! Set start text character (2=start text)
out_buffer{1} := 2;

! Set character "H" (72="H")
out_buffer{2} := 72;
! Set character "i"
out_buffer{3} := StrToByte("i"\Char);
! Set new line character (10=new line)

out_buffer{4} := 10;
! Set end text character (3=end text)

out_buffer{5} := 3;
! Write the buffer with the line "Hi"

! to the channel
WriteBin channel, out_buffer, 5;
ENDIF
Close channel;
ENDPROC
```

2.6.3 Raw data communication

2.6.3.1 Overview

Purpose

The purpose of raw data communication is to pack different type of data into a container and send it to a file or serial channel, and to read and unpack data. This is particularly useful when communicating via a fieldbus, such as DeviceNet or Profibus.

What is included

To handle raw data communication, the RobotWare base functionality File and Serial Channel Handling gives you access to:

- instructions used for handling the contents of a `rawbytes` variable
- instructions for reading and writing raw data
- a function to get the valid data length of a `rawbytes` variable.

Basic approach

This is the general approach for raw data communication. For a more detailed example of how this is done, see [Write and read rawbytes on page 78](#).

- 1 Pack data into a `rawbytes` variable (data of type `num`, `byte` or `string`).
- 2 Write the `rawbytes` variable to a file or serial channel.
- 3 Read a `rawbytes` variable from a file or serial channel.
- 4 Unpack the `rawbytes` variable to `num`, `byte` or `string`.

Limitations

Device command communication also require the base functionality Device Command Interface and the option for the industrial network in question.

Access to files, serial channels and field busses cannot be performed from different RAPID tasks simultaneously. Such an access is performed by all instruction in binary and character based communication, as well as `WriteRawBytes` and `ReadRawBytes`. For example, if a `ReadBin` instruction is executed in one task, then it must be ready before a `WriteRawBytes` instruction can execute in another task.

2.6.3.2 RAPID components

Data types

This is a brief description of each data type used for raw data communication. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
rawbytes	<p>rawbytes is used as a general data container. It can be filled with any data of type num, byte, or string. It also store the length of the valid data (in bytes).</p> <p>rawbytes can contain up to 1024 bytes of data. The supported data formats are:</p> <ul style="list-style-type: none"> • Hex (1 byte) • long (4 bytes) • float (4 bytes) • ASCII (1-80 characters)

Instructions

This is a brief description of each instruction used for raw data communication. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
ClearRawBytes	<p>ClearRawBytes is used to set all the contents of a rawbytes variable to 0. The length of the valid data in the rawbytes variable is set to 0.</p> <p>ClearRawBytes can also be used to clear only the last part of a rawbytes variable.</p>
PackRawBytes	PackRawBytes is used to pack the contents of variables of type num, byte or string into a variable of type rawbytes.
UnpackRawBytes	UnpackRawBytes is used to unpack the contents of a variable of type rawbytes to variables of type byte, num or string.
CopyRawBytes	CopyRawBytes is used to copy all or part of the contents from one rawbytes variable to another.
WriteRawBytes	WriteRawBytes is used to write data of type rawbytes to any binary file, serial channel or fieldbus.
ReadRawBytes	ReadRawBytes is used to read data of type rawbytes from any binary file, serial channel or fieldbus.

Functions

This is a brief description of each function used for raw data communication. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
RawBytesLen	RawBytesLen is used to get the valid data length in a rawbytes variable.

2.6.3.3 Code examples

About the examples

These examples are simplified demonstrations of how to use `rawbytes`. For a more realistic example of how to use `rawbytes` in DeviceNet communication, see [Write rawbytes to DeviceNet on page 86](#).

Write and read rawbytes

This example shows how to pack data into a `rawbytes` variable and write it to a device. It also shows how to read and unpack a `rawbytes` variable.

```
VAR iodev io_device;
VAR rawbytes raw_data;

PROC write_rawbytes()
  VAR num length := 0.2;
  VAR string length_unit := "meters";

  ! Empty contents of raw_data
  ClearRawBytes raw_data;

  ! Add contents of length as a 4 byte float
  PackRawBytes length, raw_data, (RawBytesLen(raw_data)+1) \Float4;

  ! Add the string length_unit
  PackRawBytes length_unit, raw_data, (RawBytesLen(raw_data)+1)
    \ASCII;

  Open "HOME:" \File:= "FILE1.DOC", io_device \Bin;

  ! Write the contents of raw_data to io_device
  WriteRawBytes io_device, raw_data;

  Close io_device;
ENDPROC

PROC read_rawbytes()
  VAR string answer;

  ! Empty contents of raw_data
  ClearRawBytes raw_data;

  Open "HOME:" \File:= "FILE1.DOC", io_device \Bin;

  ! Read from io_device into raw_data
  ReadRawBytes io_device, raw_data \Time:=1;

  Close io_device;

  ! Unpack raw_data to the string answer
```

Continues on next page

```
UnpackRawBytes raw_data, 1, answer \ASCII:=10;  
ENDPROC
```

Copy rawbytes

In this example, all data from raw_data_1 and raw_data_2 is copied to raw_data_3.

```
VAR rawbytes raw_data_1;  
VAR rawbytes raw_data_2;  
VAR rawbytes raw_data_3;  
VAR num my_length:=0.2;  
VAR string my_unit:=" meters";  
  
PackRawBytes my_length, raw_data_1, 1 \Float4;  
PackRawBytes my_unit, raw_data_2, 1 \ASCII;  
  
! Copy all data from raw_data_1 to raw_data_3  
CopyRawBytes raw_data_1, 1, raw_data_3, 1;  
  
! Append all data from raw_data_2 to raw_data_3  
CopyRawBytes raw_data_2, 1, raw_data_3, (RawBytesLen(raw_data_3)+1);
```

2.6.4 File and directory management

2.6.4.1 Overview

Purpose

The purpose of the file and directory management is to be able to browse and edit file structures (directories and files).

What is included

To handle file and directory management, the RobotWare base functionality File and Serial Channel Handling gives you access to:

- instructions for handling directories
- a function for reading directories
- instructions for handling files on a file structure level
- functions to retrieve size and type information.

Basic approach

This is the general approach for file and directory management. For more detailed examples of how this is done, see [Code examples on page 82](#).

- 1 Open a directory.
- 2 Read from the directory and search until you find what you are looking for.
- 3 Close the directory.

2.6.4.2 RAPID components

Data types

This is a brief description of each data type used for file and directory management. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
dir	dir contains a reference to a directory on disk or network. It can be linked to the physical directory with the instruction <code>OpenDir</code> .

Instructions

This is a brief description of each instruction used for file and directory management. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
<code>OpenDir</code>	<code>OpenDir</code> is used to open a directory.
<code>CloseDir</code>	<code>CloseDir</code> is used to close a directory.
<code>MakeDir</code>	<code>MakeDir</code> is used to create a new directory.
<code>RemoveDir</code>	<code>RemoveDir</code> is used to remove an empty directory.
<code>CopyFile</code>	<code>CopyFile</code> is used to make a copy of an existing file.
<code>RenameFile</code>	<code>RenameFile</code> is used to give a new name to an existing file. It can also be used to move a file from one place to another in the directory structure.
<code>RemoveFile</code>	<code>RemoveFile</code> is used to remove a file.

Functions

This is a brief description of each function used for file and directory management. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
<code>ReadDir</code>	<code>ReadDir</code> is used to retrieve the name of the next file or subdirectory under a directory that has been opened with the instruction <code>OpenDir</code> . Note that the first items read by <code>ReadDir</code> are <code>.</code> (full stop character) and <code>..</code> (double full stop characters) symbolizing the current directory and its parent directory.
<code>FileSize</code>	<code>FileSize</code> is used to retrieve the size (in bytes) of the specified file.
<code>FSSize</code>	<code>FSSize</code> (File System Size) is used to retrieve the size (in bytes) of the file system in which a specified file resides. <code>FSSize</code> can either retrieve the total size or the free size of the system.
<code>IsFile</code>	<code>IsFile</code> test if the specified file is of the specified type. It can also be used to test if the file exist at all.

2.6.4.3 Code examples

List files

This example shows how to list the files in a directory, excluding the directory itself and its parent directory (. and ..).

```
PROC lsdire(string dirname)
  VAR dir directory;
  VAR string filename;

  ! Check that dirname really is a directory
  IF IsFile(dirname \Directory) THEN
    ! Open the directory
    OpenDir directory, dirname;

    ! Loop through the files in the directory
    WHILE ReadDir(directory, filename) DO
      IF (filename <> "." AND filename <> "..") THEN
        TPWrite filename;
      ENDIF
    ENDWHILE

    ! Close the directory
    CloseDir directory;
  ENDIF
ENDPROC
```

Move file to new directory

This is an example where a new directory is created, a file renamed and moved to the new directory and the old directory is removed.

```
VAR dir directory;
VAR string filename;

! Create the directory newdir
MakeDir "HOME:/newdir";

! Rename and move the file
RenameFile "HOME:/olddir/myfile", "HOME:/newdir/yourfile";

! Remove all files in olddir
OpenDir directory, "HOME:/olddir";
WHILE ReadDir(directory, filename) DO
  IF (filename <> "." AND filename <> "..") THEN
    RemoveFile "HOME:/olddir/" + filename;
  ENDIF
ENDWHILE
CloseDir directory;

! Remove the directory olddir (which must be empty)
RemoveDir "HOME:/olddir";
```

Continues on next page

Check sizes

In this example, the size of the file is compared with the remaining free space on the file system. If there is enough space, the file is copied.

```
VAR num freesyssize;
VAR num f_size;

! Get the size of the file
f_size := FileSize("HOME:/myfile");

! Get the free size on the file system
freesyssize := FSSize("HOME:/myfile" \Free);

! Copy file if enough space free
IF f_size < freesyssize THEN
  CopyFile "HOME:/myfile", "HOME:/yourfile";
ENDIF
```

2.7 Device Command Interface

2.7.1 Introduction to Device Command Interface

Purpose

Device Command Interface provides an interface to communicate with I/O devices on industrial networks.

This interface is used together with raw data communication, see [Raw data communication on page 76](#).

What is included

The RobotWare base functionality Device Command Interface gives you access to:

- Instruction used to create a DeviceNet header.

Basic approach

This is the general approach for using Device Command Interface. For a more detailed example of how this is done, see [Write rawbytes to DeviceNet on page 86](#).

- 1 Add a DeviceNet header to a `rawbytes` variable.
- 2 Add the data to the `rawbytes` variable.
- 3 Write the `rawbytes` variable to the DeviceNet I/O.
- 4 Read data from the DeviceNet I/O to a `rawbytes` variable.
- 5 Extract the data from the `rawbytes` variable.

Limitations

Device command communication also require the base functionality File and Serial Channel Handling and the option for the industrial network in question.

Device Command Interface is supported by the following type of industrial networks:

- DeviceNet
- EtherNet/IP

2.7.2 RAPID components and system parameters

Data types

There are no RAPID data types for Device Command Interface.

Instructions

This is a brief description of each instruction in Device Command Interface. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
PackDNHeader	PackDNHeader adds a DeviceNet header to a rawbytes variable. The header specifies a service to be done (e.g. set or get) and a parameter on a DeviceNet I/O device.

Functions

There are no RAPID functions for Device Command Interface.

System parameters

There are no specific system parameters in Device Command Interface. For information on system parameters in general, see *Technical reference manual - System parameters*.

2.7.3 Code example

Write rawbytes to DeviceNet

In this example, data packed as a `rawbytes` variable is written to a DeviceNet I/O device. For more details regarding `rawbytes`, see [Raw data communication on page 76](#).

```
PROC set_filter_value()
  VAR iodev dev;
  VAR rawbytes rawdata_out;
  VAR rawbytes rawdata_in;
  VAR num input_int;
  VAR byte return_status;
  VAR byte return_info;
  VAR byte return_errcode;
  VAR byte return_errcode2;

  ! Empty contents of rawdata_out and rawdata_in
  ClearRawBytes rawdata_out;
  ClearRawBytes rawdata_in;

  ! Add DeviceNet header to rawdata_out with service
    "SET_ATTRIBUTE_SINGLE" and path to filter attribute on
    DeviceNet I/O device
  PackDNHeader "10", "6,20 1D 24 01 30 64,8,1", rawdata_out;

  ! Add filter value to send to DeviceNet I/O device
  input_int:= 5;
  PackRawBytes input_int, rawdata_out, (RawBytesLen(rawdata_out) +
    1) \IntX := USINT;

  ! Open I/O device
  Open "/FCI1:" \File:="board328", dev \Bin;

  ! Write the contents of rawdata_out to the I/O device
  WriteRawBytes dev, rawdata_out \NoOfBytes :=
    RawBytesLen(rawdata_out);

  ! Read the answer from the I/O device
  ReadRawBytes dev, rawdata_in;

  ! Close the I/O device
  Close dev;

  ! Unpack rawdata_in to the variable return_status
  UnpackRawBytes rawdata_in, 1, return_status \Hex1;

  IF return_status = 144 THEN
    TPWrite "Status OK from device. Status code:
      "\Num:=return_status;
```

Continues on next page

```
ELSE
    ! Unpack error codes from device answer
    UnpackRawBytes rawdata_in, 2, return_errcode \Hex1;
    UnpackRawBytes rawdata_in, 3, return_errcode2 \Hex1;
    TPWrite "Error code from device: " \Num:=return_errcode;
    TPWrite "Additional error code from device: "
        \Num:=return_errcode2;
ENDIF
ENDPROC
```

2.8 Logical Cross Connections

2.8.1 Introduction to Logical Cross Connections

Purpose

The purpose of Logical Cross Connections is to check and affect combinations of digital I/O signals (DO, DI) or group I/O signals (GO, GI). This can be used to verify or control process equipment that are external to the robot. The functionality can be compared to the one of a simple PLC.

By letting the I/O system handle logical operations with I/O signals, a lot of RAPID code execution can be avoided. Logical Cross Connections can replace the process of reading I/O signal values, calculate new values and writing the values to I/O signals.

Here are some examples of applications:

- Interrupt program execution when either of three input signals is set to 1.
- Set an output signal to 1 when both of two input signals are set to 1.

Description

Logical Cross Connections are used to define the dependencies of an I/O signal to other I/O signals. The logical operators AND, OR, and inverted signal values can be used to configure more complex dependencies.

The I/O signals that constitute the logical expression (actor I/O signals) and the I/O signal that is the result of the expression (resultant I/O signal) can be either digital I/O signals (DO, DI) or group I/O signals (GO, GI).

What is included


Logical Cross Connections allows you to build logical expressions with up to 5 actor I/O signals and the logical operations AND, OR, and inverted signal values.

2.8.2 Configuring Logical Cross Connections

System parameters

This is a brief description of the parameters for cross connections. For more information, see the respective parameter in [Configuring Logical Cross Connections on page 89](#).

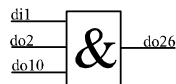
These parameters belong to the type *Cross Connection* in the topic *I/O System*.

Parameter	Description
<i>Name</i>	Specifies the name of the cross connection.
<i>Resultant</i>	The I/O signal that receive the result of the cross connection as its new value.
<i>Actor 1</i>	The first I/O signal to be used in the evaluation of the <i>Resultant</i> .
<i>Invert actor 1</i>	If <i>Invert actor 1</i> is set to <i>Yes</i> , then the inverted value of <i>Actor 1</i> is used in the evaluation of the <i>Resultant</i> .
<i>Operator 1</i>	<p>Operand between <i>Actor 1</i> and <i>Actor 2</i>. Can be either of the operands:</p> <ul style="list-style-type: none"> • AND - Results in the value 1 if both input values are 1. • OR - Results in the value 1 if at least one of the input values are 1. <div>  Note The operators are calculated left to right (<i>Operator 1</i> first and <i>Operator 4</i> last). </div>
<i>Actor 2</i>	The second I/O signal (if more than one) to be used in the evaluation of the <i>Resultant</i> .
<i>Invert actor 2</i>	If <i>Invert actor 2</i> is set to <i>Yes</i> , then the inverted value of <i>Actor 2</i> is used in the evaluation of the <i>Resultant</i> .
<i>Operator 2</i>	<p>Operand between <i>Actor 2</i> and <i>Actor 3</i>. See <i>Operator 1</i>.</p>
<i>Actor 3</i>	The third I/O signal (if more than two) to be used in the evaluation of the <i>Resultant</i> .
<i>Invert actor 3</i>	If <i>Invert actor 3</i> is set to <i>Yes</i> , then the inverted value of <i>Actor 3</i> is used in the evaluation of the <i>Resultant</i> .
<i>Operator 3</i>	<p>Operand between <i>Actor 3</i> and <i>Actor 4</i>. See <i>Operator 1</i>.</p>
<i>Actor 4</i>	The fourth I/O signal (if more than three) to be used in the evaluation of the <i>Resultant</i> .
<i>Invert actor 4</i>	If <i>Invert actor 4</i> is set to <i>Yes</i> , then the inverted value of <i>Actor 4</i> is used in the evaluation of the <i>Resultant</i> .
<i>Operator 4</i>	<p>Operand between <i>Actor 4</i> and <i>Actor 5</i>. See <i>Operator 1</i>.</p>
<i>Actor 5</i>	The fifth I/O signal (if all five are used) to be used in the evaluation of the <i>Resultant</i> .
<i>Invert actor 5</i>	If <i>Invert actor 5</i> is set to <i>Yes</i> , then the inverted value of <i>Actor 5</i> is used in the evaluation of the <i>Resultant</i> .

2.8.3 Examples

Logical AND

The following logical structure...



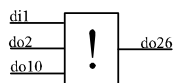
xx0300000457

... is created as shown below.

Resultant	Actor 1	Invert actor 1	Operator 1	Actor 2	Invert actor 2	Operator 2	Actor 3	Invert actor 3
do26	di1	No	AND	do2	No	AND	do10	No

Logical OR

The following logical structure...



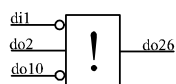
xx0300000459

... is created as shown below.

Resultant	Actor 1	Invert actor 1	Operator 1	Actor 2	Invert actor 2	Operator 2	Actor 3	Invert actor 3
do26	di1	No	OR	do2	No	OR	do10	No

Inverted signals

The following logical structure (where a ring symbolize an inverted signal)...



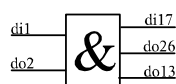
xx0300000460

... is created as shown below.

Resultant	Actor 1	Invert actor 1	Operator 1	Actor 2	Invert actor 2	Operator 2	Actor 3	Invert actor 3
do26	di1	Yes	OR	do2	No	OR	do10	Yes

Several resultants

The following logical structure can not be implemented with one cross connection...



xx0300000462

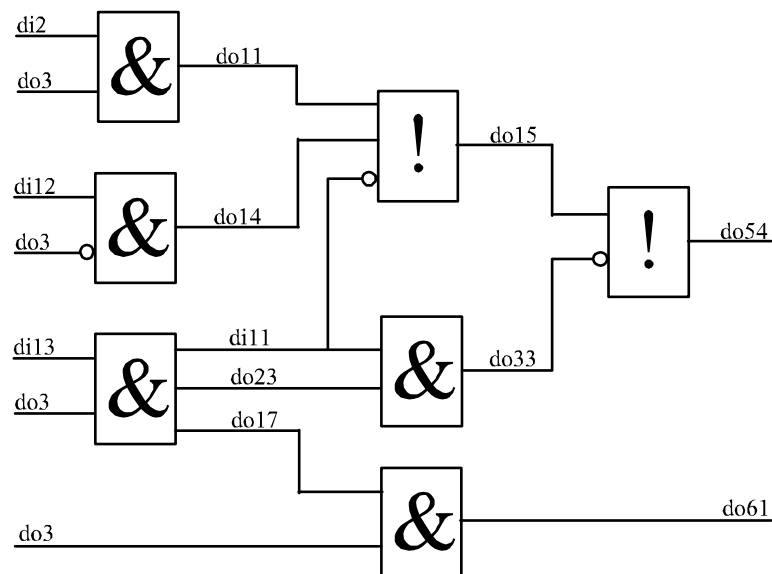
Continues on next page

... but with three cross connections it can be implemented as shown below.

Resultant	Actor 1	Invert actor 1	Operator 1	Actor 2	Invert actor 2
di17	di1	No	AND	do2	No
do26	di1	No	AND	do2	No
do13	di1	No	AND	do2	No

Complex conditions

The following logical structure...



xx0300000461

... is created as shown below.

Resultant	Actor 1	Invert actor 1	Operator 1	Actor 2	Invert actor 2	Operator 2	Actor 3	Invert actor 3
do11	di2	No	AND	do3	No			
do14	di12	No	AND	do3	Yes			
di11	di13	No	AND	do3	No			
do23	di13	No	AND	do3	No			
do17	di13	No	AND	do3	No			
do15	do11	No	OR	do14	No	OR	di11	Yes
do33	di11	No	AND	do23	No			
do61	do17	No	AND	do3	No			
do54	do15	No	OR	do33	Yes			

2.8.4 Limitations

Evaluation order

If more than two actor I/O signals are used in one cross connection, the evaluation is made from left to right. This means that the operation between *Actor 1* and *Actor 2* is evaluated first and the result from that is used in the operation with *Actor 3*.

If all operators in one cross connection are of the same type (only AND or only OR) the evaluation order has no significance. However, mixing AND and OR operators, without considering the evaluation order, may give an unexpected result.



Tip

Use several cross connections instead of mixing AND and OR in the same cross connection.

Maximum number of actor I/O signals

A cross connection may not have more than five actor I/O signals. If more actor I/O signals are required, use several cross connections.

Maximum number of cross connections

The maximum number of cross connections handled by the robot system is 300.

Maximum depth

The maximum allowed depth of cross connection evaluations is 20.

A resultant from one cross connection can be used as an actor in another cross connection. The resultant from that cross connection can in its turn be used as an actor in the next cross connection. However, this type of chain of dependent cross connections cannot be deeper than 20 steps.

Do not create a loop

Cross connections must not form closed chains since that would cause infinite evaluation and oscillation. A closed chain appears when cross connections are interlinked so that the chain of cross connections forms a circle.

Do not have the same resultant more than once

Ambiguous resultant I/O signals are not allowed since the outcome would depend on the order of evaluation (which cannot be controlled). Ambiguous resultant I/O signals occur when the same I/O signal is resultant in several cross connections.

Overlapping device maps

The resultant I/O signal in a cross connection must not have an overlapping device map with any inverted actor I/O signals defined in the cross connection. Using I/O signals with overlapping device map in a cross connection can cause infinity signal setting loops.

3 Motion performance

3.1 Absolute Accuracy [603-1, 603-2]

3.1.1 About Absolute Accuracy

Purpose

Absolute Accuracy is a calibration concept that ensures that the TCP accuracy in most cases is better than ± 1 mm in the entire working range.

The difference between an ideal robot and a real robot can be several millimeters, resulting from mechanical tolerances and deflection in the robot structure. Absolute Accuracy compensates for these differences, ensuring that the given coordinates coincide with the actual robot position.

Here are some examples of when this accuracy is important:

- Off-line programming with minimum touch-up.
- Exchangeability of robots
- On-line programming with accurate reorientation of tool
- Re-use of programs between applications

What is included

Every Absolute Accuracy robot is delivered with:

- compensation parameters saved on the robot's serial measurement board
- a birth certificate representing the Absolute Accuracy measurement protocol for the calibration and verification sequence.

Recognizing an Absolute Accuracy robot

A robot with Absolute Accuracy calibration is marked with a sign on the manipulator (close to the identification plate) that looks like this:



xx0300000314

Basic approach

These are the basic steps to set up Absolute Accuracy on your robot. For a more detailed description, see [Activate Absolute Accuracy on page 97](#).

- 1 Activate Absolute Accuracy.
- 2 Restart the controller.

Limitations

Absolute Accuracy works on a robot target in Cartesian coordinates, not on the individual joints. Therefore, joint based movements (e.g. MoveAbsJ) will not be affected. See [When is Absolute Accuracy being used on page 95](#).

Continues on next page

3 Motion performance

3.1.1 About Absolute Accuracy

Continued

If the robot is suspended, the Absolute Accuracy calibration must be performed when the robot is suspended. The compensation parameters differ depending on if the robot is floor mounted or suspended.

3.1.2 When is Absolute Accuracy being used

General

When Absolute Accuracy is activated the robot is used as normal, but with Absolute Accuracy active. However, Absolute Accuracy is only used in connection with Cartesian coordinates (i.e. robtargets). Joint based movement (i.e. jointtargets) is not affected by Absolute Accuracy.

The list below for Absolute Accuracy active defines when it is active. To further explain it is followed by some examples of when it is not active.

Absolute Accuracy active

Absolute Accuracy will be active in the following cases:

- Any motion function based on robtargets (e.g. MoveL) and ModPos on robtargets
- Reorientation jogging
- Linear jogging
- Tool definition (4, 5, 6 point tool definition, room fixed TCP, stationary tool)
- Work object definition

Absolute Accuracy not active

The following are examples of when Absolute Accuracy is not active:

- Any motion function based on a jointtarget (MoveAbsJ)
- Independent joint
- Joint based jogging
- Additional axes
- Track motion

3 Motion performance

3.1.3 Useful tools

3.1.3 Useful tools

Overview

The following products are recommended for operation and maintenance of Absolute Accurate robots:

- Load Identification
- Calibration Pendulum (standard robot calibration tool)
- CalibWare (Absolute Accuracy calibration tool)

Load Identification

Absolute Accuracy calculates the robot's deflection depending on payload. It is very important to have an accurate description of the load.

Load Identification is a tool that determines the mass, center of gravity, and inertia of the payload.

For more information, see *Operating manual - IRC5 with FlexPendant*.

Calibration Pendulum

Calibration Pendulum is used to calibrate the robot's resolver offset. This means that the robot is in its home position (all axes angles set to zero) and the resolver angles are calibrated.

There are different recommended resolver offset calibration tools, depending on the robot model. The most commonly used is Calibration Pendulum. Information about calibration for a specific robot is found in the product manual for the respective robot, and in *Operating manual - Calibration Pendulum*.

Calibration Pendulum is used at initial calibration and when servicing the robot.

CalibWare

CalibWare, provided by ABB, is a tool for calibrating Absolute Accuracy. The documentation to CalibWare describes the Absolute Accuracy calibration procedure in detail.

CalibWare is used at initial calibration and when servicing the robot.

3.1.4 Configuration

Activate Absolute Accuracy

Use RobotStudio and follow these steps (see *Operating manual - RobotStudio* for more information):

	Action
1	If you do not already have write access, click Request Write Access and wait for grant from the FlexPendant.
2	Click Configuration Editor and select Motion .
3	Click the type Robot .
4	Configure the parameter <i>Use Robot Calibration</i> and change the value to "r1_calib".
5	For a MultiMove system, repeat step 3 and 4 for each robot. <i>Use Robot Calibration</i> is then set to "r2_calib" for robot 2, "r3_calib" for robot 3 and "r4_calib" for robot 4.
6	Restart the controller for the changes to take effect.



Tip

To verify that Absolute Accuracy is active, look at the Jogging window on the FlexPendant. When Absolute Accuracy is active, the text "Absolute Accuracy On" is shown in the left window. In a MultiMove system, check this status for all mechanical units.

Deactivate Absolute Accuracy

Use RobotStudio and follow these steps (see *Operating manual - RobotStudio* for more information):

	Action
1	If you do not already have write access, click Request Write Access and wait for grant from the FlexPendant.
2	Click Configuration Editor and select the topic Motion .
3	Click the type Robot .
4	Configure the parameter <i>Use Robot Calibration</i> and change the value to "r1_uncalib".
5	For a MultiMove system, repeat step 3 and 4 for each robot. <i>Use Robot Calibration</i> is then set to "r2_uncalib" for robot 2, "r3_uncalib" for robot 3 and "r4_uncalib" for robot 4.
6	Restart the controller for the changes to take effect.

Continues on next page

3 Motion performance

3.1.4 Configuration

Continued

Change calibration data

If you exchange the manipulator, the calibration data for the new manipulator must be loaded. This is done by copying the calibration data from the robot's serial measurement board to the robot controller.

Use the FlexPendant and follow these steps (for more information, see *Operating manual - IRC5 with FlexPendant*):

	Action
1	Tap the ABB menu and then Calibration .
2	Tap on the robot you wish to update.
3	Tap the tab SMB Memory .
4	Tap Advanced .
5	Tap Clear Cabinet Memory .
6	Tap Clear and then confirm by tapping Yes .
7	Tap Close .
8	Tap Update .
9	Tap Cabinet or manipulator has been exchanged and confirm by tapping Yes .

3.1.5 Maintenance

3.1.5.1 Maintenance that affect the accuracy

Overview

This section will focus on those maintenance activities that directly affect the accuracy of the robot, summarized as follows:

- Tool recalibration
- Motor replacement
- Wrist replacement (large robots)
- Arm replacement (lower arm, upper arm, gearbox, foot)
- Manipulator replacement
- Loss of accuracy

Tool recalibration

For information about tool recalibration, see [Tool calibration on page 115](#).

Motor replacement

Replacement of all motors on small robots and motors for axes 1 through 4 on large robots (for example IRB 6700) requires a re-calibration of the corresponding resolver offset parameter using Calibration Pendulum.

For description of the calibration process, see the product manual for the respective robot.

Wrist replacement

Replacement of the wrist unit on large robots (for example IRB 6700) requires a re-calibration of the resolver offsets for axes 5 and 6 using Calibration Pendulum.

For description of the calibration process, see the product manual for the respective robot.

Arm replacement

Replacement of any of the robot arms, or other mechanical structure (excluding wrist), changes the structure of the robot to the extent that a robot recalibration is required. It is recommended that, after an arm replacement, the entire robot should be recalibrated to ensure optimal Absolute Accuracy functionality. This is typically performed with CalibWare and a separate measurement system. CalibWare can be used together with any generic 3Dmeasurement system.

For more information about the calibration process, see documentation for CalibWare.

A summary of the calibration process is presented as follows:


	Action
1	Replace the affected component.
2	Perform a resolver offset calibration for all axes. See the product manual for the respective robot.

Continues on next page

3 Motion performance

3.1.5.1 Maintenance that affect the accuracy

Continued

	Action
3	Recalibrate the TCP.
4	Check the accuracy by comparison to a fixed reference point in the cell.
5	Check the accuracy of the work objects. <div> Note An update of the defined work objects will make the deviation less in positioning.</div>
6	Check the accuracy of the positions in the current application.
7	If the accuracy still is unsatisfactory, perform an Absolute Accuracy calibration of the entire robot. See documentation for CalibWare.

Manipulator replacement

When a robot manipulator is replaced without replacing the controller cabinet, it is necessary to update the Absolute Accuracy parameters in the controller cabinet and realign the robot to the cell. The Absolute Accuracy parameters are updated by loading the replacement robot's calibration parameters into the controller as described in [Change calibration data on page 98](#). Ensure that the calibration data is loaded and that Absolute Accuracy is activated.

The alignment of the replacement robot to the cell depends on the robot alignment technique chosen at installation. If the robot mounting pins are aligned to the cell then the robot need only be placed on the pins - no further alignment is necessary. If the robot was aligned using a robot program then it is necessary to measure the cell fixture(s) and measure the robot in several positions (for best results use the same program as the original robot). See [Measure robot alignment on page 113](#).

3.1.5.2 Loss of accuracy

Cause and action

Loss of accuracy usually occur after robot collision or large temperature variations.

It is necessary to determine the cause of the errors, and take adequate action.

If...	...then...
the tool is not properly calibrated	recalibrate if the TCP has changed.
the tool load is not correctly defined	run Load Identification to ensure correct mass, centre of gravity and inertia for the active tool.
the resolver offsets are no longer valid	<ol style="list-style-type: none"> 1 Check that the axis scales show that the robot stands correctly in the home position. 2 If the indicators are not aligned, move the robot to correct position and update the revolution counters. 3 If the indicators are close to aligned but not correct, re-calibrate with Calibration Pendulum.
the robot's relationship to the fixture(s) has changed	<ol style="list-style-type: none"> 1 Check by moving the robot to a predefined position on the fixture(s). 2 Visually assessing whether the deviation is excessive. 3 If excessive, realign robot to fixture(s).
the robot structure has changed	<ol style="list-style-type: none"> 1 Visually assess whether the robot is damaged. 2 If damaged then replace entire manipulator -or- replace affected arm(s) -or- recalibrate affected arm(s).

3 Motion performance

3.1.6.1 Error sources

3.1.6 Compensation theory

3.1.6.1 Error sources

Types of errors

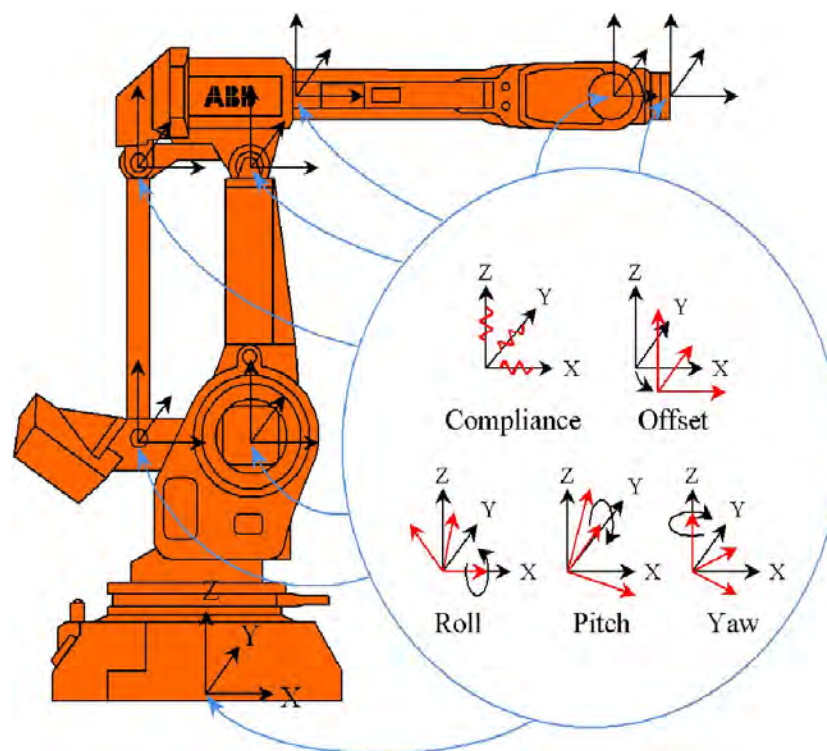
The errors compensated for in the controller derive from the mechanical tolerances of the constituent robot parts. A subset of these are detailed in the illustration below.

Compliance errors are due to the effect of the robot's own weight together with the weight of the current payload. These errors depend on gravity and the characteristics of the load. The compensation of these errors is most efficient if you use Load Identification (see *Operating manual - IRC5 with FlexPendant*).

Kinematic errors are caused by position or orientational deviations in the robot axes. These are independent of the load.

Illustration

There are several types of errors that can occur in each joint.



en0300000232

3.1.6.2 Absolute Accuracy compensation

Introduction

Both compliance and kinematic errors are compensated for with "fake targets". Knowing the deflection of the robot (i.e. deviation from ordered position), *Absolute Accuracy* can compensate by ordering the robot to a fake target.

The compensation works on a robot target in cartesian coordinates, not on the individual joints. This means that it is the position of the TCP (marked with an arrow in the following illustrations) that is correctly compensated.

Desired position

The following illustration shows the position you want the robot to have.



xx0300000225

Position due to deflection

The following illustration shows the position the robot will get without *Absolute Accuracy*. The weight of the robot arms and the load will make a deflection on the robot. Note that the deflection is exaggerated.



xx0300000227

Fake target

In order to get the desired position, *Absolute Accuracy* calculates a fake target. When you enter a desired position, the system recalculates it to a fake target that after the deflection will result in the desired position.



xx0300000226

Continues on next page

3 Motion performance

3.1.6.2 Absolute Accuracy compensation

Continued

Compensated position

The actual position will be the same as your desired position. As a user you will not notice the fake target or the deflection. The robot will behave as if it had no deflection.



xx0300000224

3.1.7 Preparation of Absolute Accuracy robot

3.1.7.1 ABB calibration process

Overview

This section describes the calibration process that ABB performs on each Absolute Accuracy robot, regardless of robot type or family, before it is delivered.

The process can be divided in four steps:

- 1 Resolver offset calibration
- 2 Absolute Accuracy calibration
- 3 Calibration data stored on the serial measurement board
- 4 Absolute Accuracy verification
- 5 Generation of a birth certificate

Resolver offset calibration

The resolver offset calibration process is used to calibrate the resolver offset parameters.

For information on how to do this, see the product manual for the respective robot.

Absolute Accuracy calibration

The Absolute Accuracy calibration is performed on top of the resolver offset calibration, hence the importance of having repeatable methods for both processes. Each robot is calibrated with maximum load to ensure that the correct compensation parameters are detected (calibration at lower load might not result in a correct determination of the robot flexibility parameters.) The process runs the robot to 100 jointtarget poses and measures each corresponding measurement point coordinate. The list of poses and measurements are fed into the CalibWare calibration core and a set of robot compensation parameters are created.

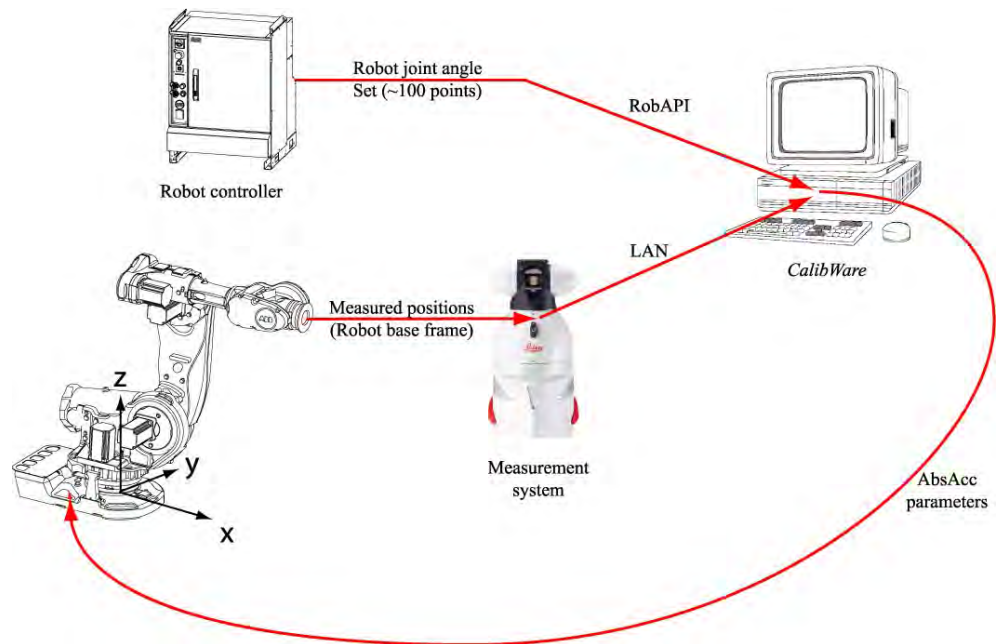
Continues on next page

3 Motion performance

3.1.7.1 ABB calibration process

Continued

For information on how to do this, see documentation for CalibWare.



en0300000248

Absolute Accuracy verification

The parameters are loaded onto the controller and activated. The robot is then run to a set of 50 robtarget poses. Each pose is measured and the deviation from nominal determined.

For information on how to do this, see documentation for CalibWare.

The requirements for acceptance vary between robot types, but typically 90% of the poses (all non-singular) have to show an absolute deviation of less than 1 mm.

Compensation parameters and birth certificate

The compensation parameters are saved on the robot's serial measurement board (see [Compensation parameters on page 108](#)).

A birth certificate is created representing the Absolute Accuracy measurement protocol for the calibration and verification sequence (see [Birth certificate on page 107](#)).

3.1.7.2 Birth certificate

About the birth certificate

All Absolute Accuracy robots are shipped with a birth certificate. It represents the Absolute Accuracy measurement protocol for the calibration and verification sequence.

The birth certificate comprises the following key information:

- Robot information (robot type, serial number)
- Accuracy information (maximum, average and standard deviation for finepoint error distribution)
- Tool information (TCP, mass, center of gravity)
- Description of measurement protocol (measurement and calibration system, number of points, measurement point location)

Example of birth certificate

ABSOLUTE ACCURACY BIRTH CERTIFICATE					
Calibration Date	January 7, 2002			Robot Version	IRB6400R 25 150
				Robot Serial Number	64-15677
Calibration and Verification Information					
Accuracy Information		GENERAL ABSOLUTE ACCURACY INFORMATION			
Measured verification points	50	1) Default values			
Average Absolute Error (mm)	0.47	Default number of Calibration Points = 100			
Maximum Absolute Error (mm)	1.27	Default number of Verification Points = 50			
Standard Deviation (mm)	0.23	2) Measurement system			
Within Specification (≤ 1 mm) (%)	98%	Leica LTD500. Maximum error = 10 μ m/m.			
Tool Information		4) Calibration software			
		ABB CalibWare (PC), ABB S4ident (S4Cplus)			
TCP (mm)	X: 185.7979736 Y: -272.594 Z: 545.146	5) Calibration process			
COG (mm)	X: 106.4999998 Y: 29.1 Z: 351.9	Calibration performed via measurement of the default number of calibration points by the stated measurement system. The stated calibration software is used to generate robot error parameters, robot base to measurement system relationship, and TCP coincidental with Measurement Point (MP) location.			
Mass (kg)	144.6000061	6) TCP/MP information			
Signature of acceptance (ABB SEROP)		TCP pre-defined by CMM measurement. Tool interface parameters determined for tool to robot flange mounting. Customer TCP should be calibrated before production start. A CMM or similar measured TCP can be used in conjunction with a tool interface calibration.			
Signature of acceptance (ABB FAC)		7) Base Frame Alignment			
		Measurement system pre-aligned to physical mounting points.			
www.abb.com/robots					

xx0300000230

3 Motion performance

3.1.7.3 Compensation parameters

3.1.7.3 Compensation parameters

About the compensation parameters

All Absolute Accuracy robots are shipped with a set of compensation parameters. As the resolver offset calibration is integral in the Absolute Accuracy calibration, the resolver offset parameters are also stored on the robot's serial measurement board.

The compensation parameters

The compensation parameters contains the following sections:

- ROBOT_CALIB
- ARM_CALIB
- JOINT_CALIB
- PARALLEL_ARM_CALIB
- TOOL_INTERFACE
- MOTOR_CALIB

The ROBOT_CALIB section defines the top level of the calibration structure. Default for the system is "uncalib", which results in Absolute Accuracy being deactivated. The "r1_calib" instance activates the Absolute Accuracy functionality by specifying the "-absacc" flag. Furthermore, a tool interface is chosen, in this case "r1_tool". Note that Absolute Accuracy must be activated manually, see [Activate Absolute Accuracy on page 97](#).

The sections ARM_CALIB, JOINT_CALIB, PARALLEL_ARM_CALIB and MOTOR_CALIB are reserved by the system and are chosen automatically when the Absolute Accuracy functionality is activated. The parameter values can be changed by importing a new configuration file, however the keywords must remain as stated. Alteration of the keywords will result in a corrupted configuration file.

The compensation parameters can be viewed by creating a backup and reading the *moc.cfg* file.

Example of compensation parameters (as found in the backup moc.cfg)

```
MOC:CFG_1.0::
# ROBOT_CALIB - ?
ROBOT_CALIB:
-name "r1_calib"
-use_tool_interface "r1_tool" -absacc

# ARM_CALIB - ?
ARM_CALIB:
-name "robl_1"
-error_offset_x 0.0000000 -error_offset_y 0.0000000 -error_offset_z
0.0000000 \
-error_roll 0.0000000 -error_pitch 0.0000000 -error_jaw 0.0000000
-arm_compliance_y 0.0000000

-name "robl_2" \
```

Continues on next page

```
-error_offset_x 0.0002967 -error_offset_y 0.0000000 -error_offset_z
0.0000000 \
-error_roll 0.0001903 -error_pitch -0.0003469 -error_jaw 0.0000000

-name "robl_3" \
-error_offset_x 0.0000000 -error_offset_y 0.0000000 -error_offset_z
0.0005485 \
-error_roll 0.0000537 -error_pitch 0.0006959 -error_jaw 0.0003361
-arm_compliance_x 0.0000000 -arm_compliance_z 0.0000000

-name "robl_4" \
-error_offset_x 0.0000000 -error_offset_y -0.0003586 -error_offset_z
0.0004580 \
-error_roll 0.0000965 -error_pitch 0.0000000 -error_jaw -0.0002578

-name "robl_5" \
-error_offset_x -0.0005467 -error_offset_y 0.0000000 -error_offset_z
0.0000032 \
-error_roll 0.0000000 -error_pitch 0.0009360 -error_jaw -0.0002367

-name "robl_6" \
-error_offset_x 0.0000000 -error_offset_y -0.0000449 -error_offset_z
-0.0000365 \
-error_roll 0.0000000 -error_pitch 0.0000000 -error_jaw -0.0002168

# JOINT_CALIB - ?
JOINT_CALIB:
-name "robl_1" -compl 0.00000000
-name "robl_2" -compl 0.00000004
-name "robl_3" -compl 0.00000107
-name "robl_4" -compl 0.00000257
-name "robl_5" -compl 0.00000490
-name "robl_6" -compl 0.00000941

# PARALLEL_ARM_CALIB - ?
PARALLEL_ARM_CALIB:
-name "robl_2" -error_length 0.0004324
-name "robl_3" -error_length -0.0000744

# TOOL_INTERFACE - ?
TOOL_INTERFACE:
-name "rl_tool" -compl 0.0 -mass 0.0 -mass_centre_x 0.0 \
-offset_x -0.0000465 -offset_y 0.0011064 -offset_z -0.0005255 \
-orient_u0 1.0 -orient_u1 0.0 -orient_u2 0.0 -orient_u3 0.0

# MOTOR_CALIB - ?
MOTOR_CALIB:
-name "robl_1" -valid_com_offset -cal_offset 1.301100
-valid_cal_offset
-name "robl_2" -valid_com_offset -cal_offset 3.422110
-valid_cal_offset
```

Continues on next page

3 Motion performance

3.1.7.3 Compensation parameters

Continued

```
-name "robl_3" -valid_com_offset -cal_offset 5.057730  
-valid_cal_offset  
-name "robl_4" -valid_com_offset -cal_offset 3.584140  
-valid_cal_offset  
-name "robl_5" -valid_com_offset -cal_offset 3.556740  
-valid_cal_offset  
-name "robl_6" -valid_com_offset -cal_offset 4.180770  
-valid_cal_offset
```

3.1.8 Cell alignment

3.1.8.1 Overview

About cell alignment

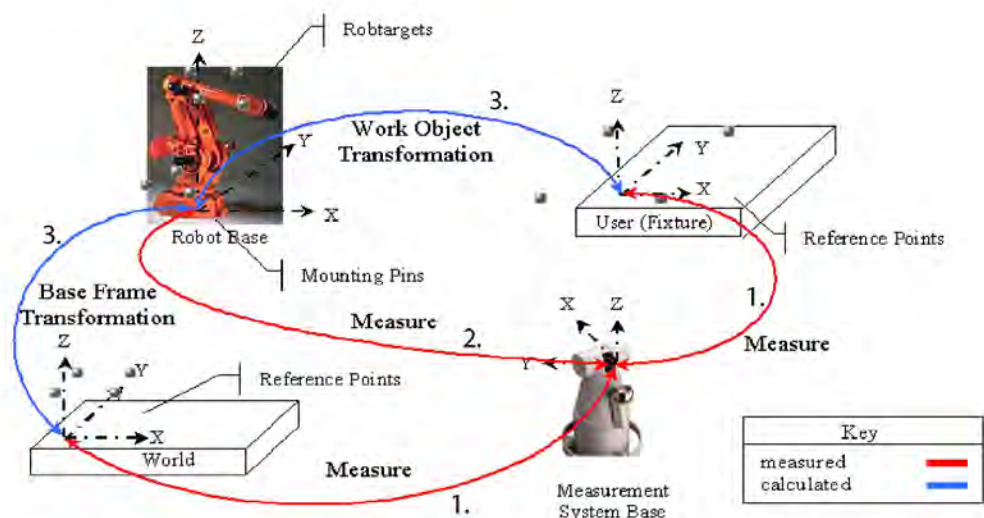
The compensation parameters for the Absolute Accuracy robot are determined from the physical base plate to the robot tool. For many applications this is enough, the robot can be used as any other robot. However, it is common that Absolute Accuracy robots are aligned to the coordinates in their cells. This section describes this alignment procedure. For a more detailed description, see documentation for CalibWare.

Alignment procedure

In order for the robot to be accurate with respect to the entire robot cell, it is necessary to install the robot correctly. In summary, this involves:

	Action	Description
1	Measure fixture alignment	Determine the relationship between the measurement system and the fixture. See Measure fixture alignment on page 112 .
2	Measure robot alignment	Determine the relationship between the measurement system and the robot. See Measure robot alignment on page 113 .
3	Calculate frame relationships	Determine the relationship between, for example, the robot and the fixture. See Frame relationships on page 114 .
4	Calibrate tool	Determine the relationship between the robot tool and other cell components. See Tool calibration on page 115 .

Illustration



en0300000239

3 Motion performance

3.1.8.2 Measure fixture alignment

3.1.8.2 Measure fixture alignment

About fixture alignment

A fixture is defined as a cell component that is associated with a particular coordinate system. The interaction between the robot and the fixture requires an accurate relationship in order to ensure Absolute Accuracy.

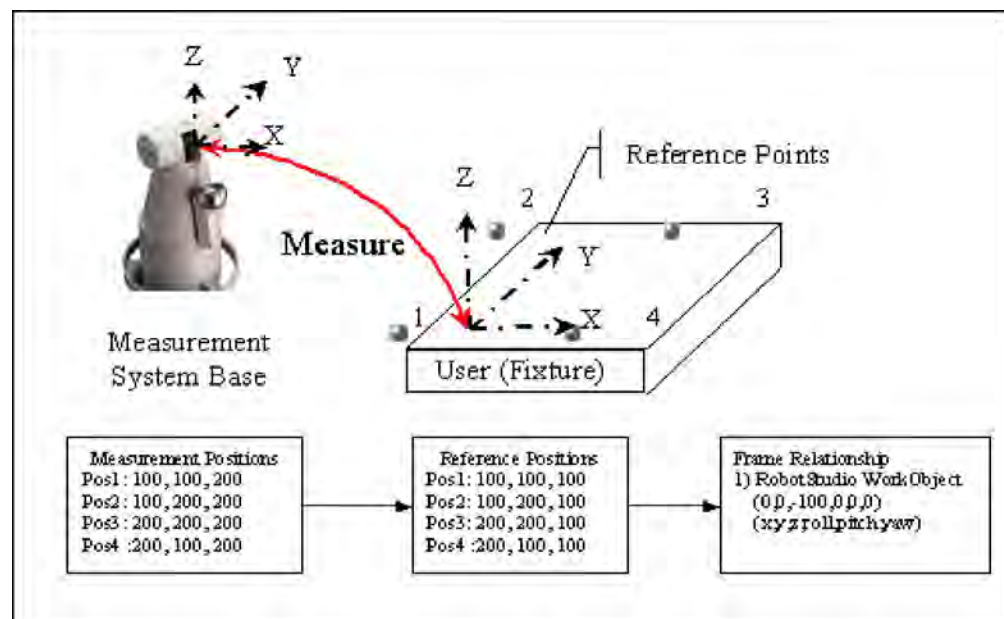
Absolute Accuracy fixtures must be equipped with at least three (preferably four) reference points, each with clearly marked position information.

Fixture measurement procedure

The alignment of the fixture is done in the following steps:

- 1 Enter the reference point names and positions into the alignment software (e.g. CalibWare).
- 2 Measure the reference points and assign the same names.
- 3 Use the alignment software to match the reference to measured points and determine the relationship frame. All measurement systems support this form of transformation.

Illustration



en0300000237

3.1.8.3 Measure robot alignment

Select method

The relationship between the measurement system and the robot can be determined in two separate ways:

Alignment procedure	Description
Alignment to physical base	The equivalent to the fixture alignment in which the physical base pins are measured and aligned with respect to the reference positions detailed in that particular robot's User Manual.
Alignment to theoretical base	Measuring several robot poses and letting the alignment software determine the robot alignment.

Alignment to physical base

The advantage of aligning the robot as a fixture is in its simplicity - the robot is treated as another fixture in the cell and its base points measured accordingly. The disadvantage is that small errors in the subsequent placement of the robot on the pins can result in large TCP errors due to the reach of the robot (i.e. the placement of the robot is not calibrated.)

In order to determine the reference point coordinates, it is necessary to consult the product manual for that robot type.

Once the correct point has been measured, the alignment software is used to determine the frame relationship between the measurement system and robot base.

Alignment to theoretical base

The advantage of aligning the robot to a theoretical base is that any errors resulting from mounting the robot can be eliminated. Furthermore, the alignment process details the robot accuracy at the measured points, confirming correct Absolute Accuracy functionality. The disadvantage is that a robot program must be created (either manually or automatically from CalibWare) and the robot measured (ideally with correct tool however the TCP can also be calibrated as a part of this procedure.)

Once the correct point is measured, the alignment software is used to determine the frame relationship between the measurement system and robot base.

3 Motion performance

3.1.8.4 Frame relationships

3.1.8.4 Frame relationships

About frame relationships

Once the relationships between the measurement system and all other cell components are measured, the relationships between cell components can be determined.

The relationship between the world coordinate system and the robot shall be stored in the robot base. The relationship between the robot and the fixture shall be stored in the `workobject` data type.

The measurement system is initially the active coordinate system as both world and robot are measured relative to the measurement system.

Determine robot base

Use a standard measurement system software to determine the robot base in world coordinates:

- 1 Set the world coordinate system to be active (the origin).
- 2 Read the coordinates of the robot base frame (now relative to the world).

The fixture relationship is similarly determined by setting the robot to be active and reading the coordinates of the fixture frame.

3.1.8.5 Tool calibration

About tool calibration

The Absolute Accuracy robot compensation parameters are calculated to be tool independent. This allows any tool with a correctly pre-defined TCP to be connected to the robot flange and used without requiring a tool re-calibration. In practice, however, it is difficult to perform a correct TCP calibration with, for example, a Coordinate Measurement Machine (CMM) as this does not take into account the connection of the tool to the robot nor the tool flexibility.

Each tool should be calibrated on a regular basis to ensure optimal robot accuracy.

Tool calibration procedures

Suggested tool recalibration procedures are detailed as follows:

- SBCU (Single Beam Calibration Unit) such as the ABB BullsEye for arc-welding or spot-welding applications.
- Geometry calibration such as the 4, 5 or 6 Point tool center point calibration routine available in the controller. A measurement system can be used to ensure that the single point used is accurate.
- RAPID tool calibration routines: MToolTCPCalib (calibration of TCP for moving tool), SToolTCPCalib (calibration of TCP for stationary tool), MToolRotCalib (calibration of rotation for moving tool), SToolRotCalib (calibration of TCP and rotation for stationary tool.)
- Using theoretical data, for example from a CAD model.



Tip

As the tool load characteristics are used in the Absolute Accuracy models, it is essential that all parameters be as accurate as possible. Use of Load Identification is an efficient method of determining tool load characteristics.

3 Motion performance

3.2 Advanced robot motion [687-1]

3.2 Advanced robot motion [687-1]

About Advanced robot motion

The option *Advanced robot motion* gives you access to:

- *Advanced Shape Tuning*, see [Advanced Shape Tuning \[included in 687-1\] on page 117](#).
- Changing *Motion Process Mode* from RAPID, see [Motion Process Mode \[included in 687-1\] on page 125](#).
- *Wrist Move*, see [Wrist Move \[included in 687-1\] on page 131](#).

3.3 Advanced Shape Tuning [included in 687-1]

3.3.1 About Advanced Shape Tuning

Purpose

The purpose of *Advanced Shape Tuning* is to reduce the path deviation caused by joint friction of the robot.

Advanced Shape Tuning is useful for low speed cutting (10-100 mm/s) of, for example, small circles. Effects of robot joint friction can cause path deviation of typically 0.5 mm in these cases. By tuning parameters of a friction model in the controller, the path deviation can be reduced to the repeatability level of the robot, for example, 0.1 mm for a medium sized robot.

What is included

Advanced Shape Tuning is included in the RobotWare option *Advanced robot motion* and gives you access to:

- Instructions `FricIdInit`, `FricIdEvaluate` and `FricIdSetFricLevels` that automatically optimize the joint friction model parameters for a programmed path.
- The system parameters *Friction FFW On*, *Friction FFW level* and *Friction FFW Ramp* for manual tuning of the joint friction parameters.
- The tune types `tune_fric_lev` and `tune_fric_ramp` that can be used with the instruction `TuneServo`.

Basic approach

This is a brief description of how *Advanced Shape Tuning* is most commonly used:

- 1 Set system parameter *Friction FFW On* to TRUE. See [System parameters on page 122](#).
- 2 Perform automatic tuning of the joint friction levels using the instructions `FricIdInit` and `FricIdEvaluate`. See [Automatic friction tuning on page 118](#).
- 3 Compensate for the friction using the instruction `FricIdSetFricLevels`.

3 Motion performance

3.3.2 Automatic friction tuning

3.3.2 Automatic friction tuning

About automatic friction tuning

A robot's joint friction levels are automatically tuned with the instructions `FricIdInit` and `FricIdEvaluate`. These instructions will tune each joint's friction level for a specific sequence of movements.

The automatically tuned levels are applied for friction compensation with the instruction `FricIdSetFricLevels`.

Program execution

To perform automatic tuning for a sequence of movements, the sequence must begin with the instruction `FricIdInit` and end with the instruction `FricIdEvaluate`. When program execution reaches `FricIdEvaluate`, the robot will repeat the movement sequence until the best friction level for each joint axis is found. Each iteration consists of a backward and a forward motion, both following the programmed path. Typically the sequence has to be repeated approximately 20-30 times, in order to iterate to correct joint friction levels.

If the program execution is stopped in any way while the program pointer is on the instruction `FricIdEvaluate` and then restarted, the results will be invalid. After a stop, friction identification must therefore be restarted from the beginning.

Once the correct friction levels are found they have to be set with the instruction `FricIdSetFricLevels`, otherwise they will not be used. Note that the friction levels are tuned for the particular movement between `FricIdInit` and `FricIdEvaluate`. For movements in another region in the robot's working area, a new tuning is needed to obtain the correct friction levels.

For a detailed description of the instructions, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Limitations

There are the following limitations for friction tuning:

- Friction tuning cannot be combined with synchronized movement. That is, `SyncMoveOn` is not allowed between `FricIdInit` and `FricIdEvaluate`.
- The movement sequence for which friction tuning is done must begin and end with a finepoint. If not, finepoints will automatically be inserted during the tuning process.
- Automatic friction tuning works only for TCP robots.
- Automatic joint friction tuning can only be done for one robot at a time.
- Tuning can be made to a maximum of 500%. If that is not enough, set a higher value for the parameter *Friction FFW Level*, see [Starting with an estimated value on page 123](#).
- It is not possible to view any test signals with Test Signal Viewer during automatic friction tuning.
- The movement sequence between `FricIdInit` and `FricIdEvaluate` cannot be longer than 10 seconds.

Continues on next page

**Note**

To use Advanced Shape Tuning, the parameter *Friction FFW On* must be set to TRUE.

Example

This example shows how to program a cutting instruction that encapsulates the friction tuning. When the instruction is run the first time, without calculated friction parameters, the friction tuning is done. During the tuning process, the robot will repeatedly move back and forth along the programmed path. Approximately 25 iterations are needed.

At all subsequent runs the friction levels are set to the tuned values identified in the first run. By using the instruction `CutHole`, the friction can be tuned individually for each hole.

```
PERS num friction_levels1{6} := [9E9,9E9,9E9,9E9,9E9,9E9];
PERS num friction_levels2{6} := [9E9,9E9,9E9,9E9,9E9,9E9];

CutHole p1,20,v50,tool1,friction_levels1;
CutHole p2,15,v50,tool1,friction_levels2;

PROC CutHole(robtarget Center, num Radius, speeddata Speed, PERS
  tooldata Tool, PERS num FricLevels{*})
  VAR bool DoTuning := FALSE;

  IF (FricLevels{1} >= 9E9) THEN
    ! Variable is uninitialized, do tuning
    DoTuning := TRUE;
    FricIdInit;
  ELSE
    FricIdSetFricLevels FricLevels;
  ENDIF

  ! Execute the move sequence
  MoveC p10, p20, Speed, z0, Tool;
  MoveC p30, p40, Speed, z0, Tool;

  IF DoTuning THEN
    FricIdEvaluate FricLevels;
  ENDIF
ENDPROC
```

**Note**

A real program would include deactivating the cutting equipment before the tuning phase.

3 Motion performance

3.3.3 Manual friction tuning

3.3.3 Manual friction tuning

Overview

It is possible to make a manual tuning of a robot's joint friction (instead of automatic friction tuning). The friction level for each joint can be tuned using the instruction `TuneServo`. How to do this is described in this section.

There is usually no need to make changes to the friction ramp.



Note

To use Advanced Shape Tuning, the parameter *Friction FFW On* must be set to TRUE.

Tune types

A tune type is used as an argument to the instruction `TuneServo`. For more information, see *tunetype* in *Technical reference manual - RAPID Instructions, Functions and Data types*.

There are two tune types that are used expressly for Advanced Shape Tuning:

Tune type	Description
TUNE_FRIC_LEV	By calling the instruction <code>TuneServo</code> with the argument <code>TUNE_FRIC_LEV</code> the friction level for a robot joint can be adjusted during program execution. A value is given in percent (between 1 and 500) of the friction level defined by the parameter <i>Friction FFW Level</i> .
TUNE_FRIC_RAMP	By calling the instruction <code>TuneServo</code> with the argument <code>TUNE_FRIC_RAMP</code> the motor shaft speed at which full friction compensation is reached can be adjusted during program execution. A value is given in percent (between 1 and 500) of the friction ramp defined by the parameter <i>Friction FFW Ramp</i> . There is normally no need to tune the friction ramp.

Configure friction level

The friction level is set for each robot joint. Perform the following steps for one joint at a time:

	Action
1	Test the robot by running it through the most demanding parts of its tasks (the most advanced shapes). If the robot shall be used for cutting, then test it by cutting with the same tool as at manufacturing. Observe the path deviations and test if the joint friction levels need to be increased or decreased.
2	Tune the friction level with the RAPID instruction <code>TuneServo</code> and the tune type <code>TUNE_FRIC_LEV</code> . The level is given in percent of the <i>Friction FFW Level</i> value. Example: The instruction for increasing the friction level with 20% looks like this: <code>TuneServo MHA160R1, 1, 120 \Type:= TUNE_FRIC_LEV;</code>
3	Repeat step 1 and 2 until you are satisfied with the path deviation.

Continues on next page

Action	
4	The final tuning values can be transferred to the system parameters. Example: The <i>Friction FFW Level</i> is 0.5 and the final tune value (<code>TUNE_FRIC_LEV</code>) is 120%. Set <i>Friction FFW Level</i> to 0.6 and tune value to 100% (default value), which is equivalent.



Tip

Tuning can be made to a maximum of 500%. If that is not enough, set a higher value for the parameter *Friction FFW Level*, see [Setting tuning system parameters on page 123](#).

3 Motion performance

3.3.4.1 System parameters

3.3.4 System parameters

3.3.4.1 System parameters

About the system parameters

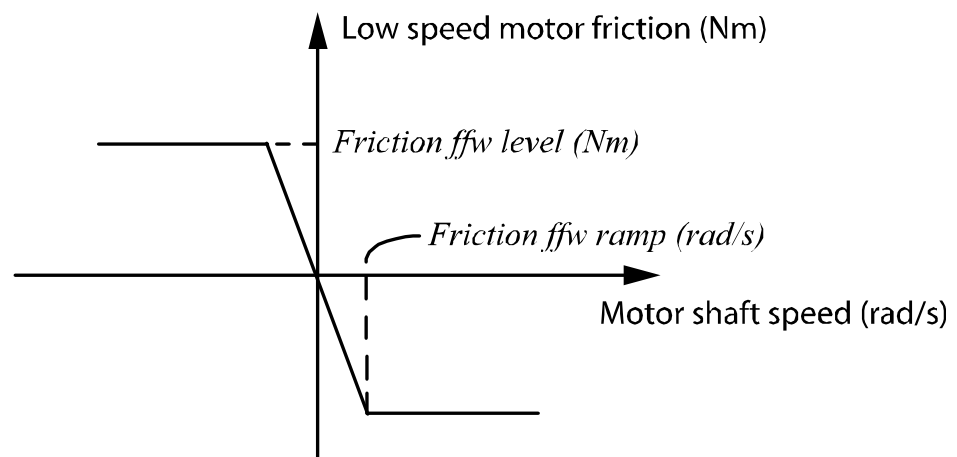
This is a brief description of each parameter in Advanced Shape Tuning. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Friction Compensation / Control Parameters

These parameters belong to the type *Friction Compensation* in the topic *Motion*, except for the robots IRB 1400 and IRB 1410 where they belong to the type *Control Parameters* in the topic *Motion*.

Parameter	Description
Friction FFW On	Advanced Shape Tuning is active when <i>Friction FFW On</i> is set to TRUE.
Friction FFW Level	<i>Friction FFW Level</i> is the friction level for the robot joint. See illustration below.
Friction FFW Ramp	<i>Friction FFW Ramp</i> is the speed of the robot motor shaft, at which the friction has reached the friction level defined by <i>Friction FFW Level</i> . See illustration below. There is normally no need to make changes to <i>Friction FFW Ramp</i> .

Illustration



en0900000117

3.3.4.2 Setting tuning system parameters

Automatic tuning rarely requires changes in system parameters

For automatic tuning, if the friction levels are saved in a persistent array, the tuning is maintained after a power failure. The automatic tuning can also be used to set different tuning levels for different robot movement sequences, which cannot be achieved with system parameters. When using automatic tuning, there is no need to change the system parameters unless the default values are very much off, see [Starting with an estimated value on page 123](#).

Transfer tuning to system parameters

When using manual tuning, the tuning values are reset to default (100%) at power failure. System parameter settings are, however, permanent.

If a temporary tuning is made, that is only valid for a part of the program execution, it should not be transferred.

To transfer the friction level tuning value (`TUNE_FRIC_LEV`) to the parameter *Friction FFW Level* follow these steps:

	Action
1	In RobotStudio, open the Configuration Editor, Motion topic, and select the type Friction comp (except for the robots IRB 1400 and IRB 1410 where they belong to the type Control parameters).
2	Multiply <i>Friction FFW Level</i> with the tuning value. Set this value as the new <i>Friction FFW Level</i> and set the tuning value (<code>TUNE_FRIC_LEV</code>) to 100%. Example: The <i>Friction FFW Level</i> is 0.5 and the final tune value (<code>TUNE_FRIC_LEV</code>) is 120%. Set <i>Friction FFW Level</i> to 0.6 (1.20x0.5) and the tuning value to 100% (default value), which is equivalent.
3	Restart the controller for the changes to take effect.

Starting with an estimated value

The parameter *Friction FFW Level* will be the starting value for the tuning. If this value is very far from the correct value, tuning to the correct value might be impossible. This is unlikely to happen, since *Friction FFW Level* is by default set to a value approximately correct for most situations.

If the *Friction FFW Level* value, for some reason, is too far from the correct value, it can be changed to a new estimated value.

	Action
1	In RobotStudio, open the Configuration Editor, Motion topic, and select the type Friction comp (except for the robots IRB 1400 and IRB 1410 where they belong to the type Control parameters).
2	Set the parameter <i>Friction FFW Level</i> to an estimated value. Do not set the value 0 (zero), because that will make tuning impossible.
3	Restart the controller for the changes to take effect.

3 Motion performance

3.3.5 RAPID components

3.3.5 RAPID components

About the RAPID components

This is an overview of all instructions, functions, and data types in *Advanced Shape Tuning*.

For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instructions

Instructions	Description
FricIdInit	Initiate friction identification
FricIdEvaluate	Evaluate friction identification
FricIdSetFricLevels	Set friction levels after friction identification

Functions

Advanced Shape Tuning includes no functions.

Data types

Advanced Shape Tuning includes no data types.

3.4 Motion Process Mode [included in 687-1]

3.4.1 About Motion Process Mode

Purpose

The purpose of Motion Process Mode is to simplify application specific tuning, i.e. to optimize the performance of the robot for a specific application.

For most applications the default mode is the best choice.

Available motion process modes

A motion process mode consists of a specific set of tuning parameters for a robot. Each tuning parameter set, that is each mode, optimizes the robot tuning for a specific class of applications.

The following modes are predefined:

- *Optimal cycle time mode* – this is the default mode and gives the same tuning as the standard robot tuning in RobotWare releases prior to 6.0. This mode gives the shortest possible cycle time and is normally the default mode.
- *Low speed accuracy mode* – this mode is recommended for applications where path accuracy is important, and for process speeds up to approximately 500 mm/s. The cycle time will be increased compared to *Optimal cycle time mode*.
- *Low speed stiff mode* - this mode is recommended for contact applications where maximum servo stiffness is important. Could also be used in some low speed applications, where a minimum of path vibrations is desired. The cycle time will be increased compared to *Low speed accuracy mode*.

Selection of mode

The default mode is automatically selected and can be changed by changing the system parameter *Use Motion Process Mode* for type *Robot*.

Changing the *Motion Process Mode* from RAPID is only possible if the option *Advanced Robot Motion* is installed. The mode can only be changed when the robot is standing still, otherwise a fine point is enforced.

The following example shows a typical use of the RAPID instruction

`MotionProcessModeSet.`

```
MotionProcessModeSet OPTIMAL_CYCLE_TIME_MODE;  
! Do cycle-time critical movement  
MoveL *, vmax, ...;  
...
```

```
MotionProcessModeSet LOW_SPEED_ACCURACY_MODE;  
! Do cutting with high accuracy  
MoveL *, v150, ...;  
...
```

3 Motion performance

3.4.2 User-defined modes

3.4.2 User-defined modes

Available tune parameters

If a more specific tuning is needed, some tuning parameters can be modified in each motion process mode. It is not possible to create new modes, but the three predefined modes can be modified. In this way, the user can create a specific tuning for a specific application.

The following list contains a short description of the available tune parameters.

- *Accset Acc Factor* – changes acceleration
- *Accset Ramp Factor* – changes acceleration ramp
- *Accset Fine Point Ramp Factor* – changes deceleration ramp in fine points
- *Dh Factor* – changes path smoothness (effective system bandwidth)
- *Df Factor* – changes the predicted resonance frequency for a particular axis
- *Kp Factor* – changes the equivalent gain of the position controller for a particular axis
- *Kv Factor* – changes the equivalent gain of the speed controller for a particular axis
- *Ti Factor* – changes the integral time of the controller for a particular axis
- *Mounting Stiffness Factor X* – describes the stiffness of the robot foundation in x direction
- *Mounting Stiffness Factor Y* – describes the stiffness of the robot foundation in y direction
- *Mounting Stiffness Factor Z* – describes the stiffness of the robot foundation in z direction

For a detailed description, see *Motion Process Mode* in *Technical reference manual - System parameters*.

Tuning parameters from RAPID

All parameters except *Mounting Stiffness Factor* can also be changed using the `TuneServo` and `AccSet` instructions.



Note

All parameter settings are relative adjustments of the predefined parameter values. Although it is possible to combine the use of motion process modes and `TuneServo/Accset` instructions, it is recommended to choose either motion process modes or `TuneServo/AccSet`.

Example 1

Relative adjustment of acceleration = $[Predefined\ AccSet\ Acc\ Factor] * [AccSet\ Acc\ Factor] * [AccSet\ instruction\ acceleration\ factor / 100]$

Example 2

Relative adjustment of Kv = $[Predefined\ Kv\ Factor] * [Kv\ Factor] * [Tune\ value\ of\ TuneServo(TYPE_KV)\ instruction / 100]$

Continues on next page

Predefined parameter values

The predefined parameter values for each mode varies for different robot types.

Generally, all predefined parameters are set to 1.0 for *Optimal cycle time mode*.

For *Low speed accuracy mode* and *Low speed stiff mode*, the `AccSet` and `Dh` parameters are lowered for a smoother movement and a more accurate path, and the *Kv Factor*, *Kp Factor*, and *Ti Factor* are changed for higher servo stiffness.

For some robots, it might not be possible to increase the *Kv Factor* in *Low speed accuracy mode* and *Low speed stiff mode*. Always be careful and be observant for increased motor noise level when adjusting *Kv Factor* and do not use higher values than needed for fulfilling the application requirement. A *Kp Factor* which is too high, or a *Ti Factor* which is too low, can also increase vibrations due to mechanical resonances.

The *Df Factor* and the *Mounting Stiffness Factors* are always set to 1.0 in the predefined modes, since the optimal values of these parameters depends the specific installation, for example, the stiffness of the foundation on which the robot is mounted. These parameters can be optimized using *TuneMaster*. More information can be found in the *TuneMaster* application. Also note the limitations of *Mounting Stiffness Factor*.



WARNING

Incorrect setting of the *Motion Process Mode* parameters can cause oscillating movements or torques that can damage the robot.

3 Motion performance

3.4.3 General information about robot tuning

3.4.3 General information about robot tuning

Minimizing cycle time

For best possible cycle time, the motion process mode *Optimal cycle time mode* should be used. This mode is normally the default mode. The user only needs to define the tool load, payload, and arm loads if any. Once the robot path has been programmed, the *ABB QuickMove* motion technology automatically computes the optimal accelerations and speeds along the path. This results in a time-optimal path with the shortest possible cycle time. Hence, no tuning of acceleration is needed. The only way to improve the cycle time is to change the geometry of the path or to work in another region of the work space. This type of optimization, if needed, can be performed by simulation in RobotStudio.

Increasing path accuracy and reducing vibrations

For most applications, the *Optimal cycle time mode* will result in a satisfactory behavior in terms of path accuracy and vibrations. This is due to the *ABB TrueMove* motion technology. However, there are applications where the accuracy needs to be improved by modifying the tuning of the robot. This tuning has previously been performed by using the *TuneServo* and *AccSet* instructions in the RAPID program. The concept of motion process modes will simplify this application specific tuning and the three predefined modes should be useful in many cases with no further adjustments needed.

Here follows some general advice for solving accuracy problems, assuming that the default choice *Optimal cycle time mode* has been tested and that accuracy problems have been noticed:

- 1 Verify that tool load, payload, and arm loads are properly defined.
- 2 Inspect tool and process equipment attached to the robot arms. Make sure that everything is properly fastened and that rigidity of the tool is adequate.
- 3 Inspect the foundation on which the robot is mounted, see [Compensating for foundation flexibility on page 128](#).

Compensating for foundation flexibility

If the foundation does not fulfill the stiffness requirement of the robot product manual, then the foundation flexibility should be compensated for. See section *Requirements on foundation, Minimum resonance frequency* in the robot product manual.

This is performed by *Df Factor* for axis 1 and 2 or *Mounting Stiffness Factor* depending on robot type, see [Limitations on page 130](#).

Continues on next page

TuneMaster is used for finding the optimal value of *Df Factor / Mounting Stiffness Factor*. The obtained *Df Factor / Mounting Stiffness Factor* is then defined for the *Motion Process Modes* used.



Note

A foundation that does not fulfill the requirements always impairs the accuracy to some extent, even if the described compensation is used. If the foundation rigidity is very low, there might not be possible to solve the problem using *Df Factor / Mounting Stiffness Factor*.

In this case, the foundation must be improved or any of the solutions below used, for example, *Optimal cycle time mode* with a low *Dh Factor*, *Accset Acc Factor*, or *Accset Fine Point Ramp Factor* depending on the application.

If accuracy still needs to be improved

- For applications with high demands on path accuracy, for example cutting and laser welding, *Low speed accuracy mode* should be used.
- - If the path accuracy still needs improvement, the first tuning parameter to try is *Dh Factor*. Decrease *Dh Factor* from 1.0 (default) down to 0.4 in steps of 0.1. Note that a low value of *Dh Factor* can change the corner zones at high speeds. If this is not acceptable, use *Accset Acc Factor* instead of *Dh Factor*.
 - If path accuracy requirement is fulfilled but the cycle time is too high, then increase *Accset Acc Factor* to e.g. 2 or larger. This is since acceleration is reduced in *Low speed accuracy mode*.
- For cutting applications, the option *Advanced Shape Tuning* and *Low speed accuracy mode* should be used.
- For contact applications, for example milling and pre-machining, *Low speed stiff mode* is recommended. This mode can also be useful in some low speed applications (up to 100 mm/s) where a minimum of path vibrations is required, for example below 0.1 mm. Note that this mode has a very stiff servo tuning and that there may be cases where the *Kv Factor* needs to be reduced due to motor vibrations and noise.
- If overshoots and vibrations in fine points needs to be reduced. Use *Optimal cycle time mode* and decrease the value of *Accset Fine Point Ramp Factor* or *Dh Factor* until the problem is solved.
- If accuracy problems occur when starting or ending reorientation. Define a new zone with increased `pzone_ori` and `pzone_eax`. These should always have the same value, even if there are no external axes in the system. Also increase `zone_ori`. Always strive for smooth reorientations when programming.
- Finally, if the cycle time needs to be reduced after the tuning for accuracy is finished. Use different motion process modes in different sections of the RAPID program.

3 Motion performance

3.4.4 Additional information

3.4.4 Additional information

Motion Process Mode compared to TuneServo and AccSet

Motion process modes simplifies application specific tuning and makes it possible to define the tuning by system parameters instead of the RAPID program.

In general, motion process modes should be the first choice for solving accuracy problems. However, application specific tuning can still be performed using the `TuneServo` and `AccSet` instructions in the RAPID program.

There are a few situations where `TuneServo` and `AccSet` might be a better choice. One example of this is if an acceleration reduction in a section of the RAPID program solves the accuracy problem and the cycle time is to be optimized. In this case it might be better to use `AccSet` which can be changed without fine point whereas change of motion process mode requires a fine point.

Limitations

- The *Motion Process Mode* concept is currently available for all six-axes robots except paint robots.
- The *Mounting Stiffness Factor* parameters are only available for the following robots:
IRB 120, IRB 140, IRB 1200, IRB 1520, IRB 1600, IRB 2600, IRB 4600, IRB 6620 (not LX), IRB 6640, IRB 6700.
- For IRB 1410, only the three `Accset` parameters are available.

Related information

For information about	Further information
Configuration of <i>Motion Process Mode</i> parameters.	<i>Technical reference manual - System parameters</i>
RAPID instructions: <ul style="list-style-type: none">• <code>AccSet</code> - Reduces the acceleration• <code>MotionProcessModeSet</code> - Set motion process mode• <code>TuneServo</code> - Tuning servos	<i>Technical reference manual - RAPID Instructions, Functions and Data types</i>

3.5 Wrist Move [included in 687-1]

3.5.1 Introduction to Wrist Move

Purpose

The purpose of *Wrist Move* is to improve the path accuracy when cutting geometries with small dimensions. For geometrical shapes like small holes, friction effects from the main axes (1-3) of the robot often degrade the visual appearance of the shape. The key idea is that instead of controlling the robot's TCP, a wrist movement controls the point of intersection between the laser beam (or water jet or routing spindle, etc) and the cutting plane. For controlling the point of intersection, only two wrist axes are needed. Instead of using all axes of the robot, only two wrist axes are used, thereby minimizing the friction effects on the path. Which wrist axis pair to be used is decided by the programmer.

Using Wrist Move

Wrist Move is included in the RobotWare option *Advanced robot motion*.

Wrist Move is used together with the RAPID instruction `CirPathMode` and movement instructions for circular arcs, that is, `MoveC`, `TrigC`, `CapC` etc. The wrist movement mode is activated by the instruction `CirPathMode` together with one of the flags `Wrist45`, `Wrist46`, or `Wrist56`. With this mode activated, all subsequent `MoveC` instructions will result in a wrist movement. To go back to normal `MoveC` behavior, then `CirPathMode` has to be set with a flag other than `Wrist45`, `Wrist46`, and `Wrist56`, for example, `PathFrame`.



Note

During a wrist movement, the TCP height above the surface will vary. This is an unavoidable consequence of using only two axes. The height variation will depend on the robot position, the tool definition, and the radius of the circular arc. The larger the radius, the larger the height variation will be. Due to the height variation it is recommended that the movement is run at a very low speed the first time to verify that the height variation does not become too large. Otherwise it is possible that the cutting tool collides with the surface being cut.

Limitations

The Wrist Move option cannot be used if:

- The work object is moving
- The robot is mounted on a track or another manipulator that is moving

The Wrist Move option is only supported for robots running QuickMove, second generation.

The tool will not remain at right angle against the surface during the cutting. As a consequence, the holes cut with this method will be slightly conical. Usually this will not be a problem for thin plates, but for thick plates the conicity will become apparent.

Continues on next page

3 Motion performance

3.5.1 Introduction to Wrist Move

Continued

The height of the TCP above the surface will vary during the cut. The height variation will increase with the size of the shape being cut. What limits the possible size of the shape are therefore, beside risk of collision, process characteristics like focal length of the laser beam or the water jet.

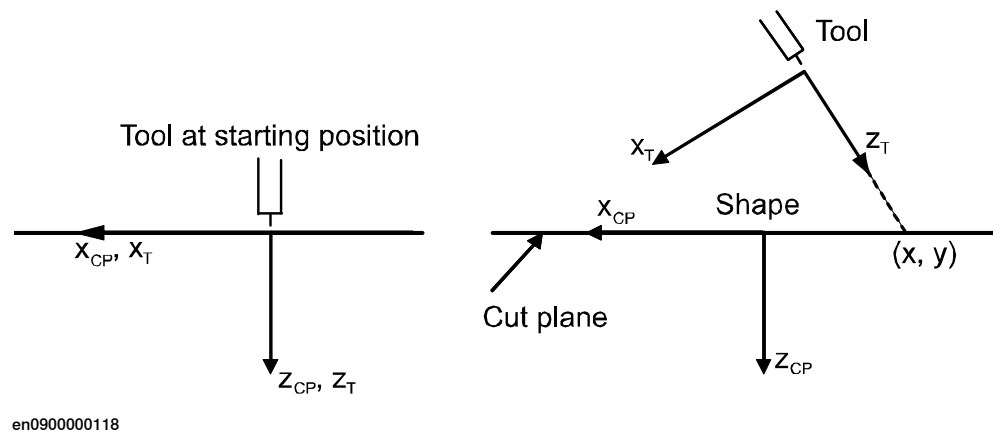
3.5.2 Cut plane frame

Defining the cut plane frame

Crucial to the wrist movement concept is the definition of the cut plane frame. This frame provides information about position and orientation of the object surface. The cut plane frame is defined by the robot's starting position when executing a `MoveC` instruction. The frame is defined to be equal to the tool frame at the starting position. Note that for a sequence of `MoveC` instructions, the cut plane frame stays the same during the whole sequence.

Illustration, cut plane

The left illustration shows how the cut plane is defined, and the right illustration shows the tool- and cut plane frames during cutting.



Prerequisites

Due to the way the cut plane frame is defined, the following must be fulfilled at the starting position:

- The tool must be at right angle to the surface
- The z-axis of the tool must coincide with the laser beam or water jet
- The TCP must be as close to the surface as possible

If the first two requirements are not fulfilled, then the shape of the cut contour will be affected. For example, a circular hole would look more like an ellipse. The third requirement is normally easy to fulfill as the TCP is often defined to be a few mm in front of, for example, the nozzle of a water jet. However, if the third requirement is not fulfilled, then it will only affect the radius of the resulting circle arc. That is, the radius of the cut arc will not agree with the programmed radius. For a linear segment, the length will be affected.



Tip

In the jog window of the FlexPendant there is a button for automatic alignment of the tool against a chosen coordinate frame. This functionality can be used to ensure that the tool is at a right angle against the surface when starting the wrist movement.

Continues on next page

3 Motion performance

3.5.2 Cut plane frame

Continued



Tip

Wrist movement is not limited to circular arcs only: If the targets of `MoveC` are collinear, then a straight line will be achieved.

3.5.3 RAPID components

Instruction

This is a brief description of the instruction used in Wrist Move. For more information, see the description of the instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Descriptions
CirPathMode	<p>CirPathMode makes it possible to select different modes to reorientate the tool during circular movements.</p> <p>The arguments Wrist45, Wrist46, and Wrist56 are used specifically for the Wrist Move option.</p>

3 Motion performance

3.5.4 RAPID code, examples

3.5.4 RAPID code, examples

Basic example

This example shows how to do two circular arcs, first using axes 4 and 5, and then using axes 5 and 6. After the two arcs, wrist movement is deactivated by CirPathMode.

```
! This position will define the cut plane frame
MoveJ p10, v100, fine, tWaterJet;

CirPathMode \Wrist45;
MoveC p20, p30, v50, z0, tWaterJet;

! The cut-plane frame remains the same in a sequence of MoveC
CirPathMode \Wrist56;
MoveC p40, p50, v50, fine, tWaterJet;

! Deactivate Wrist Movement, could use \ObjectFrame or \CirPointOri
as well
CirPathMode \PathFrame;
```

Advanced example

This example shows how to cut a slot with end radius R and length $L+2R$, using wrist movement. See [Illustration, pSlot and wSlot on page 137](#). The slot both begins and ends at the position pSlot, which is the center of the left semi-circle. To avoid introducing oscillations in the robot, the cut begins and ends with semi-circular lead-in and lead-out paths that connect smoothly to the slot contour. All coordinates are given relative the work object wSlot.

```
! Set the dimensions of the slot
R := 5;
L := 30;

! This position defines the cut plane frame, it must be normal to
the surface
MoveJ pSlot, v100, z1, tLaser, \wobj := wSlot;
CirPathMode \Wrist45;

! Lead-in curve
MoveC Offs(pSlot, R/2, R/2, 0), Offs(pSlot, 0, R, 0), v50, z0,
tLaser, \wobj := wSlot;

! Left semi-circle
MoveC Offs(pSlot, -R, 0, 0), Offs(pSlot, 0, -R, 0), v50, z0, tLaser,
\wobj := wSlot;

! Lower straight line, circle point passes through the mid-point
of the line
MoveC Offs(pSlot, L/2, -R, 0), Offs(pSlot, L, -R, 0), v50, z0,
tLaser, \wobj := wSlot;

! Right semi-circle
```

Continues on next page


```
MoveC Offs(pSlot, L+R, 0, 0), Offs(pSlot, L, R, 0), v50, z0, tLaser,  
      \wobj := wSlot;
```

```
! Upper straight line, circle point passes through the mid-point  
  of the line
```

```
MoveC Offs(pSlot, L/2, R, 0), Offs(pSlot, 0, R, 0), v50, z0, tLaser,  
      \wobj := wSlot;
```

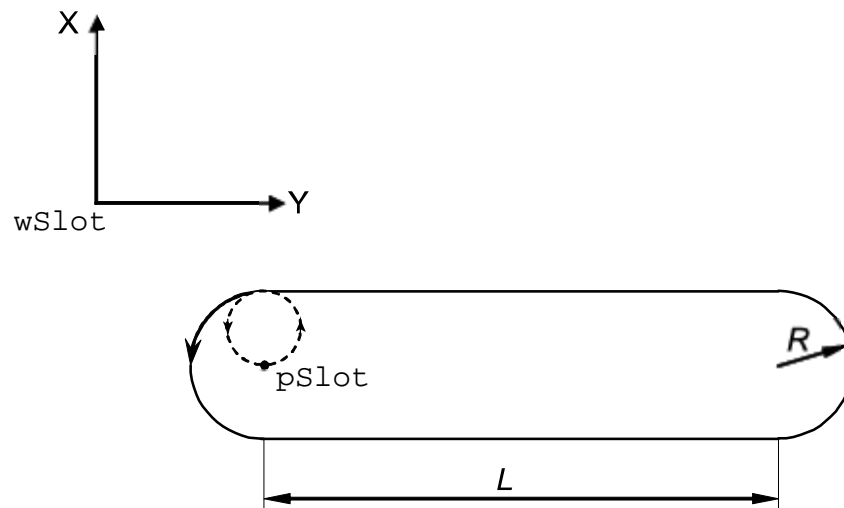
```
! Lead-out curve back to the starting point
```

```
MoveC Offs(pSlot, -R/2, R/2, 0), pSlot, v50, z1, tLaser, \wobj :=  
      wSlot;
```

```
Deactivate Wrist Movement
```

```
CirPathMode \ObjectFrame;
```

Illustration, pSlot and wSlot



xx0900000111

3 Motion performance

3.5.5 Trouble shooting

3.5.5 Trouble shooting

Unexpected cut shape

If the cut shape is not the expected, then check the following:

- The tool z-axis coincides with the laser beam or the water jet
- The tool z-axis is at right angle to the surface at the starting position of the first `MoveC`
- If you have the option Advanced Shape Tuning, then try tuning the friction for the involved wrist axes.

Mismatching radius

If the radius of the circular arc does not agree with the programmed radius, then check that the TCP is as close to the surface as possible at the starting position.

Impossible movement with chosen axis pair

If the movement is not possible with the selected axis pair, then try activating another pair by using one of the flags `Wrist45`, `Wrist46`, or `Wrist56`. As a last resort, try reaching the starting position with another robot configuration.

4 Motion coordination

4.1 Machine Synchronization [607-1], [607-2]

4.1.1 Overview

Two options

Machine Synchronization consists of two options, *Sensor Synchronization* and *Analog Synchronization*. The functionality is very similar for both these options, it is the hardware and configuration that differs.

The difference between the two options is that:

- Analog Synchronization is used together with a sensor that shows the position of the external mechanical unit as an analog signal.
- Sensor Synchronization requires an encoder that counts pulses as the external mechanical unit move, and an encoder interface unit which transforms the pulses into a sensor position.

All information in this chapter refers to both options, unless something else is specified. The term *synchronization option* refers to both options. Information that is only valid for one of the options is said to be specific for *Sensor Synchronization* or *Analog Synchronization*.

Purpose

The synchronization option adjusts the robot speed to an external moving device (for example a press or conveyor) with the help of a sensor. It can also be used to synchronize two robots with each other.

Description

For the synchronization, a sensor is used to detect the movements of a press door, conveyor, turn table or similar device. The speed of the robot TCP will be adjusted in correlation to the sensor output, so that the robot will reach its programmed target at the same time as the external device reaches its programmed position.

The synchronization with the external device does not affect the path of the robot TCP, but it affects the speed at which the robot moves along this path.

Functionality

The external device connected to the sensor cannot be controlled by the robot controller. However, in some ways it has similarities with a mechanical unit controlled by the robot controller:

- the sensor positions appears in the *Jogging Window* on the FlexPendant
- the sensor positions appears in the `robtarg` when a *MODPOS* operation is performed
- the mechanical unit may be activated, and deactivated

Continues on next page

4 Motion coordination

4.1.1 Overview

Continued

Basic approach

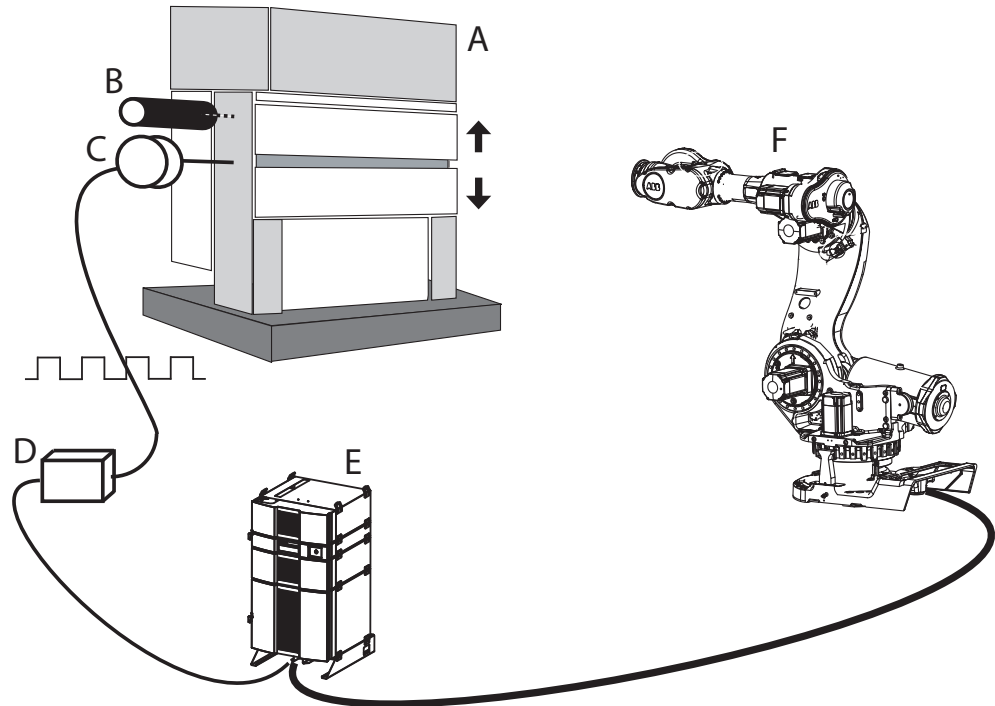
This is the general approach for setting up the synchronization option. For a more detailed description of how this is done, see the respective section.

- Install and connect hardware.
- Install the synchronization software.
- Configure the system parameters.
- Write a program that connects to the sensor and uses synchronization for robot movements (or a program for a master/slave robot application).

4.1.2 What is needed

Sensor Synchronisation

The Sensor Synchronization application consist of the following components:



en0400000655

A	External device that dictates the robot speed, e.g. a press door
B	Synchronization switch
C	Encoder
D	Encoder interface unit (DSQC 377)
E	Controller
F	Robot
B+C+D	Act as a sensor, giving input to the controller

Continues on next page

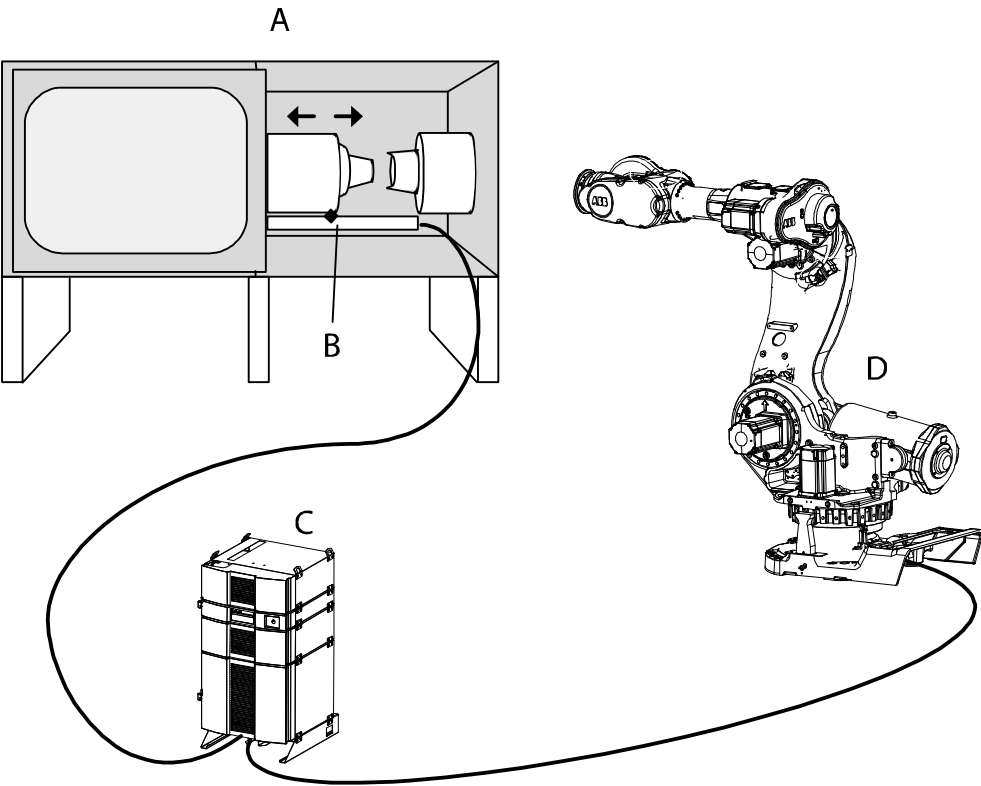
4 Motion coordination

4.1.2 What is needed

Continued

Analog Synchronization

The Analog Synchronization application consist of the following components:



xx0700000431

A	Mold press that dictates the robot speed
B	Analog sensor for press position
C	Controller
D	Robot

4.1.3 Synchronization features

Features

The synchronization option provides the following features:

Feature	Description
Accuracy	<p>In Auto operation at constant sensor speed, the Tool Center Point (TCP) of the robot will stay within the programmed position corresponding to the sensor, with an error margin of:</p> <ul style="list-style-type: none"> • +/- 50 ms for Sensor Synchronization • +/- 100 ms for Analog Synchronization <p>This is valid as long as the robot is within its dynamic limits with the added sensor motion. This figure depends on the calibration of the robot and sensor and is applicable for linear synchronization only.</p>
Object queue	<p>Only for Sensor Synchronization:</p> <p>Each time the external device trigger the synchronization switch, a sensor object is created in the object queue. The encoder interface unit will maintain the object queue, although for Sensor Synchronization the queue normally does not contain more than one object.</p>
RAPID access to sensor data	<p>A RAPID program has access to the current position and speed of the external device, via the sensor.</p>
Multiple sensors	<p>Up to 2 sensors are supported.</p> <p>For Sensor Synchronization, each sensor must have a DSQC 377.</p>

4 Motion coordination

4.1.4 General description of the synchronization process

4.1.4 General description of the synchronization process

Example with a press

This example shows the very basic steps when synchronization is used for material handling for a press.

When...	Then...
the press is closed and ready to start	a signal from the robot controller (or PLC) orders the press to start.
the press starts open	<p>For Sensor Synchronization, the synchronization switch is triggered and a sensor object is created in the object queue. The robot connects to the object.</p> <p>For both Sensor Synchronization and Analog Synchronization, the robot moves, synchronized with the press, towards the press and reaches it when the press is open enough.</p>
the press is open enough for the robot to enter	<p>the robot places (or removes) a work piece in the press. The synchronization is ended.</p> <p>For Sensor Synchronization, the sensor object is then dropped (removed from the object queue).</p>

4.1.5 Limitations

Limitations on additional axes

Each sensor is considered an additional axis. Thus the system limitation of 6 active additional axes must be reduced by the number of active and installed sensors.

The first installed sensor will use measurement node 6 and the second sensor will use measurement node 5. These measurement nodes are not available for additional axes and no resolvers should be connected to these nodes on any additional axes measurement boards.

Object queue lost on warm start or power failure

Only for Sensor Synchronization:

The object queue is kept on the encoder interface unit (DSQC 377). If the system is restarted or if the power supply to either the controller or the encoder interface unit fails, then the object queue will be lost.

Minimum speed

In order to maintain a smooth and accurate motion, there is a minimum speed of the external device that is detected. The device is considered to be still if its movement is slower than the minimum speed. This speed depends on the selection of encoder. It can vary from 4mm/s - 8mm/s.

Maximum speed

There is no determined maximum speed for the external device. Accuracy will decrease at speeds over those specified, and the robot will no longer be able to follow the sensor at very high sensor speeds (>1000mm/s) or with robot dynamic limitations.

Compatibility with the option Conveyor Tracking

If both Machine Synchronization and Conveyor Tracking options are installed, only one of the mechanical units SSYNC1 and CNV2 should be active at the same time.

For Machine Synchronization (Sensor Synchronization or Analog Synchronization), CNV2 must be deactivated.

For Conveyor Tracking, SSYNC1 must be deactivated.

4 Motion coordination

4.1.6.1 Encoder specification

4.1.6 Hardware installation for Sensor Synchronization

4.1.6.1 Encoder specification

Two phase type

The encoder must be of two phase type for quadrature pulses, to enable registration of reverse sensor motion, and to avoid false counts due to vibration etc. when the sensor is not moving.

Technical data

Output signal:	Open collector PNP output
Voltage:	10 - 30 V (normally supplied by 24 VDC from encoder interface unit)
Current:	50 - 100 mA
Phase:	2 phase with 90 degree phase shift
Duty cycle:	50%
Max. frequency:	20 kHz

Example encoder

An example of an encoder that fills these criteria, is the *Lenord & Bauer GEL 262*.

4.1.6.2 Encoder description

Overview

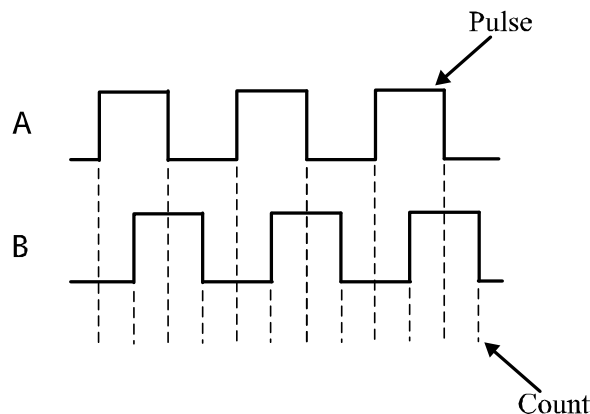
The encoder provides a series of pulses indicating the motion detected by the sensor. This is used to synchronize the motion between the robot and the external device.

Pulse channels

The encoder has two pulse channels, A and B which differ in phase by 90° . Each channel will send a fixed number of pulses per revolution depending on the construction of the encoder.

- The number of pulses per revolution for the encoder must be selected in relation to the gear reduction between the moving devices.
- The pulse ratio from the encoder should be in the range of 1250 - 2500 pulses per meter of sensor motion.
- The pulses from channel A and B are used in quadrature to multiply the pulse ratio by four to get counts.

This means that the control software will measure 5000 - 10000 counts per meter for an encoder with the pulse ratio 1250 - 2500.



en0300000556

Synchronization

To get an accurate synchronization, the movements of the external device must remain within some limits relative to robot movements. For every meter the robot moves, the external device movement must be between 0.2 and 5 meters (or radians).

4 Motion coordination

4.1.6.3 Installation recommendations

4.1.6.3 Installation recommendations

Overview

The encoder must be installed in such a way that it gives precise feedback of the sensor output (reflects the true motion of the external device). This means that the encoder should be installed as close to the robot as practically possible, no further away than 30 meters.

The encoder is normally installed on the drive unit of the external device. The encoder may be connected to an output shaft on the drive unit, directly or via a gear belt arrangement.



Note

The encoder is a sensitive measuring device and for that reason it is important that no other forces than the shaft rotation are transferred from the sensor to the encoder and that the encoder is mounted using shock absorbers etc. to prevent damage from vibration.

Placement

The following is to be considered before start-up

If...	Then...
the drive unit includes a clutch arrangement	the encoder must be connected on the sensor side of the clutch.
the encoder is connected directly to a drive unit shaft	it is important to install a specially designed flexible coupling to prevent applying mechanical forces to the encoder rotor..
the drive unit of the external device is located far away from the encoder	the moving device itself may be a source of inaccuracy as the moving device will stretch or flex over the distance from the drive unit to the encoder cell. In such a case it may be better to mount the encoder closer to the drive unit with a different coupling arrangement.

4.1.6.4 Connecting encoder and encoder interface unit

Overview

If the cable from the robot to the encoder is too long, the inductance in the cable will produce spike pulses on the encoder signal. This signal will over a period of time damage the opto couplers in the encoder interface unit.

See *Product manual - IRC5* for details on connecting to the encoder interface unit.

Reduce noise

To reduce noise, connect the encoder with a screened cable.

Reduce spike pulses

To reduce spike pulses, install a capacitor between the signal wire and ground for each of the two phases. The correct capacitance value can be determined by viewing the encoder signal on an oscilloscope.

The capacitor:

- should be connected on the terminal board where the encoder is connected.
- values are 100 nF - 1 μ F, depending on the length of the cable.

Encoder power supply

The encoder is normally supplied with 24 VDC from the encoder interface unit.

When connecting two encoder interface units to the same encoder, let only one of the encoder interface units supply power to the encoder. If both encoder interface units supply power, a diode must be installed on each of the 24 V DC connections to make sure the power supplies do not interfere with each other.

Connecting encoder and the synchronization switch

The following procedure describes how to install the encoder and the synchronization switch to the encoder interface unit.

- One encoder can be connected to several encoder interface units.
- each controller must have an encoder interface unit if more than one robot is to use the sensor.

	Action	Illustration
1	Connect the encoder to the encoder interface unit (DSQC 377) on the controller.	<p>en0300000611</p>

Continues on next page

4 Motion coordination

4.1.6.4 Connecting encoder and encoder interface unit

Continued

	Action	Illustration
2	Connect the synchronization switch to the encoder interface unit (DSQC 377) on the controller.	

Finding the Encoder rotating direction

The following procedure describes how to find the encoder rotating direction.

	Action	Illustration
1	On the FlexPendant, tap Inputs and Outputs .	
2	Tap View and select I/O Units	
3	Scroll down and selected Qtrack - d377	
4	Scroll down to c1position	
5	Run the encoder in forward direction while checking the value for C1Position. If the number counts up: <ul style="list-style-type: none"> No action is required. If the number counts down: <ul style="list-style-type: none"> the connection of the two encoder faces (0° and 90°) must be interchanged. 	<p>en0300000584</p>

4.1.7 Hardware installation for Analog Synchronization

4.1.7.1 Required hardware

Analog input board

An analog input board is required, for example DSQC355A. See *Application manual - DeviceNet Master/Slave*.

Analog linear sensor

An analog linear sensor is required, with analog signal input between 0 and 10 V.

4 Motion coordination

4.1.8.1 Sensor installation

4.1.8 Software installation

4.1.8.1 Sensor installation

Overview

Normally the synchronization option and the DeviceNet option are preloaded at ABB, and do not need to be re-installed. For more information on how to add options to the system, see *Operating manual - RobotStudio*.

The synchronization option automatically installs one sensor into the system parameters. To add more than one sensor, see [Installation of several sensors on page 155](#).

About the installation

The options will install three additional configurations:

- I/O for the encoder interface unit (only for Sensor Synchronization)
- Sensor process description
- Motion mechanical description

Configuration of the default installation for Sensor Synchronization

This procedure describes how to configure system parameters for Sensor Synchronization in the configuration editor in RobotStudio.

	Action
1	Change the parameter <i>Connected to Bus</i> for the unit from "Virtual1" to the correct bus, for example "DeviceNet1".
2	Specify the correct address for the unit, parameter <i>DeviceNet Address</i> .
3	If the parameter <i>DeviceNet Master Address</i> (in topic <i>I/O</i> , type <i>Bus</i>) is changed, then the parameter <i>Default Value</i> (in topic <i>I/O</i> , type <i>Fieldbus Command Type</i>) for the instance <i>TimeKeeperInit</i> must be changed to the same value.

Configuration of the default installation for Analog Synchronization

This procedure describes how to configure system parameters for Analog Synchronization in the configuration editor in RobotStudio.

	Action
1	Change the unit type, parameter <i>Type of Unit</i> , for the unit from "Virtual" to the correct unit type, for example "d355A".
2	Change the parameter <i>Connected to Bus</i> for the unit from "Virtual1" to the correct bus, for example "DeviceNet1".
3	Specify the correct address for the unit, parameter <i>DeviceNet Address</i> .
4	Change the communication interval for the unit type (e.g d355A) from 50 to 20 ms, parameter <i>Connection 1 Interval</i> . For more information about this parameter, see <i>Application manual - DeviceNet Master/Slave</i> .

Continues on next page

How to add a sensor manually for Sensor Synchronization

Use the following procedure to add a sensor manually.

	Action
1	Connect the encoder interface unit to the CAN bus. Note the address on the CAN bus.
2	In RobotStudio, click Load Parameters .
3	Select: <i>Load Parameters if no duplicates</i> and click Open .
4	Installation of a master sensor , connected to DeviceNet1 (first board). Load the following files one by one from the OPTIONS/CNV directory: <ul style="list-style-type: none"> • <i>syvm1_eio.cfg</i> • <i>syvm1_prc.cfg</i> • <i>syvm1_moc.cfg</i>
5	Installation of a slave sensor , connected to DeviceNet2 (second board). Load the following files one by one from the OPTIONS/CNV directory: <ul style="list-style-type: none"> • <i>syvs1_eio.cfg</i> • <i>syvs1_prc.cfg</i> • <i>syvs1_moc.cfg</i>
6	Restart the system.
7	If necessary, correct the address for the new encoder interface units. The default addresses in the file <i>syvxx_eio.cfg</i> should be replaced by the actual address of the board.

How to add a sensor manually for Analog Synchronization

There are no prepared files for adding a sensor for Analog Synchronization. It can be accomplished by copying the following files and edit them for the second sensor:

- *synvaileio.cfg*
- *synvailprc.cfg*
- *syim1.moc*

4 Motion coordination

4.1.8.2 Reloading saved Motion parameters

4.1.8.2 Reloading saved Motion parameters

Overview

During installation of the synchronization option, a specific sensor configuration for additional axes will be loaded into the Motion system parameters.



Note

If these parameters were loaded before the synchronization option, then the mechanical unit **SSYNC1** will not appear on the FlexPendant under the **Jogging** window.

Reloading the SSYNC1 parameter

Use RobotStudio and follow these steps (see *Operating manual - RobotStudio* for more information):

	Action
1	Open the Configuration Editor and select the topic <i>Motion</i> .
2	Select the type File .
3	Click Load parameters and select mode.
4	Click Open and select the file syn1_moc from the RobotWare installation.
5	Restart the controller for the changes to take effect.

Result

The mechanical unit **SSYNC1** should now be available on the FlexPendant under the **Jogging** window.

4.1.8.3 Installation of several sensors

About the installation

Normally the synchronization option and the DeviceNet option are preloaded at ABB, and do not need to be re-installed. For more information how to add options to the system, see *Operating manual - RobotStudio*.

The synchronization option automatically installs one sensor into the system parameters.

DeviceNet Dual option

When DeviceNet Dual is included, the following three sensors will be installed in the system:

- One sensor with "Robot to press syncro type": *SSYNC1*
- One virtual master sensor: *SSYNM1*
- One virtual slave sensor: *SSYNCS1*

Adding sensors manually

Up to four sensors can be used with the same controller, but the parameters for the three extra sensors must be loaded manually.

Use the following procedure to load the sensors manually.

	Action
1	For Sensor Synchronization, connect the encoder interface unit to the CAN bus. Note the address on the CAN bus.
2	Use RobotStudio to add new parameters.
3	Click Load Parameters .
4	Select: <i>Load Parameters if no duplicates</i> and click Open .
5	Installation of a master sensor , connected to DeviceNet1 (first board). Load the following files one by one from the OPTION/CNV directory: <ul style="list-style-type: none"> • for second sensor: <i>syvm2_eio.cfg</i>, <i>syvm2_prc</i> and <i>syvm2_moc.cfg</i> • for third sensor: <i>syvm3_eio.cfg</i>, <i>syvm3_prc.cfg</i> and <i>syvm3_moc.cfg</i> • for fourth sensor: <i>syvm4_eio.cfg</i>, <i>syvm4_prc.cfg</i> and <i>syvm4_moc.cfg</i>
6	Installation of a slave sensor , connected to DeviceNet2 (second board). Load the following files one by one from the OPTION/CNV directory: <ul style="list-style-type: none"> • for second sensor: <i>syvs2_eio.cfg</i>, <i>syvs2_prc.cfg</i> and <i>syvs2_moc.cfg</i> • for third sensor: <i>syvs3_eio.cfg</i>, <i>syvs3_prc.cfg</i> and <i>syvs3_moc.cfg</i> • for fourth sensor: <i>syvs4_eio.cfg</i>, <i>syvs4_prc.cfg</i> and <i>syvs4_moc.cfg</i>
7	Restart the system.
8	For Sensor Synchronization: If necessary, correct the address for the new encoder interface units. Find the respective encoder interface unit in the system parameters under the topic <i>I/O</i> . The default addresses in the file <i>syvxx_eio.cfg</i> should be replaced by the actual address of the board.

Available sensors

The second and third sensor (*SSYNC2*, *SSYNC3*) should now appear in *Motion/mechanical unit* and in the **Jogging window** on the FlexPendant.

4 Motion coordination

4.1.9.1 General issues when programming with the synchronization option

4.1.9 Programming the synchronization

4.1.9.1 General issues when programming with the synchronization option

Activate sensor

The sensor must be activated before it may be used for work object coordination, just like any other mechanical unit. The usual `ActUnit` instruction is used to activate the sensor and `DeactUnit` is used to deactivate the sensor.

By default, the sensor is installed inactive on start. If desired, the sensor may be configured to always be active upon start. See [Mechanical unit on page 191](#).

Automatic connection

Only for Sensor Synchronization:

When a sensor mechanical unit is activated, it first checks the state of the encoder interface unit to see whether the sensor was previously connected. If the encoder interface unit, via the I/O signal `c1Connected`, indicates connection, then the sensor will automatically be connected upon activation. The purpose of this feature is to automatically reconnect in case of a power failure with power backup on the encoder interface unit.

Connection via WaitSensor instruction

Motions that are to be synchronized with the external device cannot be programmed until an object has been connected to the sensor with a `WaitSensor` instruction.

If the object is already connected with a previous `WaitSensor` instruction, or if connection was established during activation, then execution of a second `WaitSensor` instruction will cause an error.

After connection to an object with a `WaitSensor` instruction the synchronized motion is started using `SyncToSensor\On` instruction.

For details about the instructions `WaitSensor` and `SyncToSensor\On`, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Programming Sensor Synchronization

In the following instructions, there are references to programming examples.

	Action	Information
1	Create a program with the following instructions: <code>ActUnit SSYNCl;</code> <code>MoveL waitp, v1000, fine, tool;</code> <code>WaitSensor SSYNCl;</code>	
2	Single-step the program past the <code>WaitSensor</code> instruction.	The instruction will return if there is an object in the object queue. If there is no object, the execution will stop while waiting for an object (i.e. a sync signal).

Continues on next page

4.1.9.1 General issues when programming with the synchronization option *Continued*

	Action	Information
3	Run the external device until a sync signal is generated by the synchronization switch.	The program should exit the <code>WaitSensor</code> and is now "connected" to the object.
4	Stop the external device in the position that should correspond to the robot target you are about to program.	
5	Start the synchronized motion with a <code>SyncToSensor SSYNC1\On</code> instruction. See Programming examples on page 158 .	
6	Program move instructions. For every time you modify a position, run the external device to the position that should correspond to the robot target.	Use corner zones for the move instructions, see Finepoint programming on page 162 .
7	End the synchronized motion with a <code>SyncToSensor SSYNC1\Off</code> instruction. See Programming examples on page 158 .	
8	Only for Sensor Synchronization: Program a <code>DropSensor SSYNC1;</code> instruction. See Programming examples on page 158 .	
9	Program a <code>DeactUnit SSYNC1;</code> instruction if this is the end of the program, or if the sensor is no longer needed. See Programming examples on page 158 .	

Synchronize the sensor

If it is not possible to move the external device to the desired position, modify the position first and then edit the sensor value in the robtarget (as for any additional axis).

4 Motion coordination

4.1.9.2 Programming examples

4.1.9.2 Programming examples

Sensor Synchronization program

```
MoveJ p0, vmax, fine, tool1;

!Activate sensor
ActUnit SSYNC1;

!Connect to the object
WaitSensor SSYNC1;

!Start the Synchronized motion
SyncToSensor SSYNC1\On;

!Instructions with coordinated robot targets
MoveL p10, v1000, z20, tool1;
MoveL p20, v1000, z20, tool1;
MoveL p30, v1000, z20, tool1;

!Stop the synchronized motion
SyncToSensor SSYNC1\Off;

!Exit coordinated motion
MoveL p40, v1000, fine, tool1;

!Disconnect from current object
DropSensor SSYNC1;

MoveL p0, v1000, fine;

!Deactivate sensor
DeactUnit SSYNC1;
```

Analog Synchronization program

```
VAR num startdist := 600;

MoveJ p0, vmax, fine, tool1;

!Activate sensor
ActUnit SSYNC1;

WaitSensor SSYNC1 \RelDist:=startdist;

!Start the Synchronized motion
SyncToSensor SSYNC1\On;

!Instructions with coordinated robot targets
MoveL p10, v1000, z20, tool1;
MoveL p20, v1000, z20, tool1;
MoveL p30, v1000, z20, tool1;
```

Continues on next page

```
!Exit coordinated motion
MoveL p40, v1000, fine, tool1;

!Stop the synchronized motion
SyncToSensor SSYNC1\Off;

MoveL p0, v1000, fine;

!Deactivate sensor
DeactUnit SSYNC1;
```

4 Motion coordination

4.1.9.3 Entering and exiting coordinated motion in corner zones

4.1.9.3 Entering and exiting coordinated motion in corner zones

Corner zones can be used

Once a `WaitSensor` instruction is connected to an object it is possible to enter and exit synchronized motion with the sensor via corner zones.

Dropping object after corner zone

If an instruction using a corner zone is used to exit coordinated motion, it cannot be followed directly by the `DropSensor` instruction. This would cause the object to be dropped before the robot has left the corner zone, when the motion still requires the conveyor coordinated work object.

If the work object is dropped when motion still requires its position, then a stop will occur.

To avoid this, either call a finepoint instruction or at least two corner zone instructions before dropping the work object.

Correct example

This is an example of how to enter and exit coordinated motion via corner zones.

```
MoveL p10, v1000, fine, tool1;  
WaitSensor SSYNC1;  
MoveL p20, v500, z50, tool1;  
!start synchronization after zone around p20  
SyncToSensor SSYNC1\On  
MoveL p30, v500, z20, tool1;  
MoveL p40, v500, z20, tool1;  
MoveL p50, v500, z20, tool1;  
MoveL p60, v500, z50, tool1;  
!Exit synchronization after zone around p60  
SyncToSensor SSYNC1\Off;  
MoveL p70, v500, fine, tool1;  
DropSensor SSYNC1;  
MoveL p10, v500, fine, tool1;
```

Incorrect example

This is an incorrect example of exiting coordination in corner zones. This will cause the program to stop with an error.

```
MoveL p50, v500, z20, tool1;  
MoveL p60, v500, z50, tool1;  
!Exit coordination in zone  
SyncToSensor SSYNC1\Off;  
DropSensor SSYNC1;
```

If coordinated motion is ended in a corner zone, another move instruction must be executed before the sensor is dropped.

4.1.9.4 Use several sensors

Overview

When several sensors are used the program must have at least one move instruction without any synchronization between parts of the path that are synchronized with two different sensors.

Program example

```
!Connect to the object
WaitSensor SSYNC1\RelDist:=Pickdist;

!Start the Synchronized motion
SyncToSensor SSYNC1\MaxSync:=1653\On;

!Instructions with coordinated robot targets
MoveL p30, v400, z20, currtool;

!Stop the synchronized motion
SyncToSensor SSYNC1\Off;

!Instructions with coordinated robot targets
MoveL p31, v400, z20, currtool;

!Connect to the object
WaitSensor SSYNC2\RelDist:=1720;

!Instructions with coordinated robot targets
MoveL p32, v400, z50, currtool;

!Start the Synchronized motion
SyncToSensor SSYNC2\MaxSync:=2090\On;

!Instructions with coordinated robot targets
MoveL p33, v400, z20, currtool;

!Stop the synchronized motion
SyncToSensor SSYNC2\Off;
```

4 Motion coordination

4.1.9.5 Finepoint programming

4.1.9.5 Finepoint programming

Overview

Avoid the use of fine points when using synchronized motion. The robot will stop and lose the synchronization with the sensor for 100 ms. Then the RAPID execution will continue.

Finepoint programming can be used on the last synchronized move instruction if the synchronization does not need to be accurate at the last target.

Program example

The following program example shows how synchronized motion may be stopped.

```
WaitSensor SSYNC1;  
SyncToSensor SSYNC1 \On;  
MoveL p1, v500, z20, tool1;  
MoveL p2, v500, fine, tool1;  
SyncToSensor SSYNC1 \Off;  
MoveL p3, v500, z20, tool1;  
MoveL p4, v500, fine, tool1;  
DropSensor SSYNC1;
```

At p4 the robot is no longer synchronized with the external device, and there are no restrictions for using fine points.

At p2 the synchronization will end and a fine point can be used, but the accuracy of the synchronization will be reduced.

4.1.9.6 Drop sensor object

Overview

For Sensor Synchronization, a connected object may be dropped, with a `DropSensor` instruction, once the synchronized motion has ended.

Example: `DropSensor SSYNC1;`

For Analog Synchronization, the instruction `DropSensor` must not be used.

Considerations

The following considerations must be considered when dropping an object:

- It is important to make sure that the robot motion is no longer using the sensor position when the object is dropped. If robot motion still requires the sensor position then a stop will occur when the object is dropped.
- As long as the `SyncToSensor \Off` instruction has not been issued, the robot motion will be synchronized with the sensor.
- It is not necessary to be connected in order to execute a `DropSensor` instruction. No error will be returned if there was no connected object.

4 Motion coordination

4.1.9.7 Information on the FlexPendant

4.1.9.7 Information on the FlexPendant

Overview

The user has access to the sensor position and speed via the FlexPendant

Jogging window

The position (in millimeters) of the sensor object is shown in the **Jogging** window. This value will be negative if a *Queue Tracking Distance* is defined. When the synchronization switch is triggered, the position will automatically be updated in the **Jogging** window.

I/O window

Sensor Synchronization

From the I/O window the user has access to all the signals that are defined on the encoder interface unit. From this window it is possible to view the sensor object position (in meters) and the sensor object speed (in m/s). The speed will be 0 m/s until the synchronization switch registers a sensor object.

Analog Synchronization

For Analog Synchronization, only the sensor position is shown in the I/O window.

4.1.9.8 Programming considerations

Performance limits

The synchronization will be lost if joint speed limits are reached, particularly in singularities. It is the responsibility of the programmer to ensure that the path during synchronized movement does not exceed the speed and motion capabilities of the robot.

Motion commands

All motion commands are allowed during synchronization.

Manual mode

The synchronization is not active in manual mode.

Speed reduction % button

The synchronization works only with 100% speed. As the robot speed is adjusted to sensor movements the defined robot speed percentage will be overridden.

Programmed speed

The best performance of the synchronization will be obtained if the programmed speed is near the real execution speed. The programmed speed should be chosen as the most probable execution speed. Large changes in speed between two move instructions should be avoided.

Finepoints

Finepoints are allowed during synchronization motion, but the robot will stop at the fine point and the synchronization will be lost if the external device is still moving. See [Finepoint programming on page 162](#).

Position warnings

If `robot_to_sensor` position ratio is higher than 10 or lower than 0.1 a warning will appear. The user should modify the `robtarg` position or the sensor value in the `robtarg` according to the warning text.

Speed warnings

If programmed `sensor_speed` is higher than:

- $(\text{max_sync_speed} * \text{sensor_nominal_speed}) / \text{robot_tcp_speed}$

then a speed warning will appear and the user should modify robot speed or `sensor_nominal_speed` or `max_sync_speed` according to the warning text.

If the programmed `sensor_speed` is lower than:

- $(\text{min_sync_speed} * \text{sensor_nominal_speed}) / \text{robot_tcp_speed}$

a similar warning will appear:

- `Programmed_sensor_speed` equals `sensor_distance/robot_interpolation_time`.

Continues on next page

4 Motion coordination

4.1.9.8 Programming considerations

Continued

Change of tools

Changing the tool is not allowed during synchronization if corvec is used.

Instructions that will deactivate the synchronization

The instructions `ActUnit`, `DeactUnit`, and `ClearPath` will deactivate any `SyncToSensor` or `SupSyncSensorOn` instruction. So the instructions `ActUnit`, `DeactUnit`, and `ClearPath` should not be used between `SyncToSensor` or `SupSyncSensorOn` instruction and the move instructions related to synchronized path or supervised path.

The correct order is:

```
ActUnit SSYNCl;  
WaitSensor SSYNCl;  
SyncToSensor SSYNCl\On;  
! move instructions  
...  
SyncToSensor SSYNCl\Off;
```

Other RAPID limitations

- The commands, `StorePath`, `RestoPath` do not work during synchronization.
- `EoffsSet`, `EoffsOn`, `EoffsOff` have an effect on the sensor taught position.
- Power fail restart is not possible with the synchronization option.

4.1.9.9 Modes of operation

Operation in manual reduced speed mode (< 250 mm/s)

The forward and backward hard buttons can be used to step through the program. New instructions may be added and MODPOS may be used to modify programmed positions.

The robot will recover as normal if the enabling device is released during motion.

The robot will not perform synchronized motions to the sensor while in Manual Reduced Speed mode.

Operation in automatic mode

Once a `SyncToSensor` instruction has been executed, then it is no longer possible to step through the program with the forward and backward buttons while the sensor is moving.

Start/Stop

The robot will stop and loose synchronization with the sensor if the STOP button is pressed or if RAPID instruction `Stop` or `StopMove` is executed between the `SyncToSensor` and `DropSensor` instructions.

The sensor object will not be lost but if the sensor is moving then the object will quickly move out of the max dist. Restart synchronization from the current instruction is not allowed if sensor is moving. The program must be restarted from `MAIN`. If a restart is forced the robot will stop with `max_dist` error where the sensor has stopped.

Emergency Stop/Restart

When the emergency stop is pressed the robot will stop immediately. If the program was stopped after a `SyncToSensor` then the sensor object will not be lost but if the sensor is moving then the object will quickly move out of the max distance. Restart synchronization from the current instruction is not possible and the program must be restarted from `MAIN`. If a restart is forced after the question "Do you want to regain", the robot will move unsynchronized to the sensor at programmed speed.

Operation under manual full speed mode (100%)

Operation in manual full speed mode is similar to operation in automatic mode. The program may be run by pressing and holding the start button, but once a `SyncToSensor` instruction has been executed then it is no longer possible to step through the program with the forward or backward buttons while the sensor is moving.

Hold to run button

Pressing and releasing the hold to run button will make the robot stop and restart. The synchronization is lost at robot stop. At restart the robot will try to regain synchronization at `max_adjustment_speed`.

Continues on next page

4 Motion coordination

4.1.9.9 Modes of operation

Continued

Stop/Restart

When the stop button is pressed, or emergency stop is pressed, the robot will stop immediately. If the program was stopped after a `SyncToSensor` then the synchronized object will not be lost but if the sensor is moving then the object will quickly move out of the max distance. Restart from the current instruction is not possible and the program must be restarted from `MAIN`.

4.1.10 Robot to robot synchronization

4.1.10.1 Introduction

Overview

It is possible to synchronize two robot systems in a synchronization application. This is done with a master and a slave robot setup.

Requirements

For cable connection and setup, see *Application manual - DeviceNet Master/Slave*.

4 Motion coordination

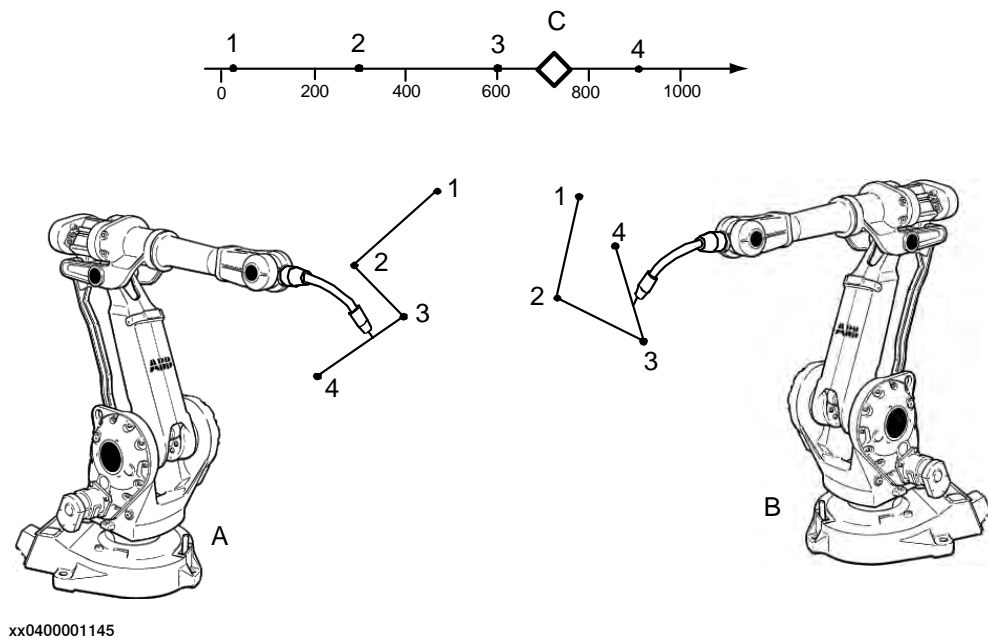
4.1.10.2 The concept of robot to robot synchronization

4.1.10.2 The concept of robot to robot synchronization

Description

The basic idea of robot to robot synchronization is that two robots should use a common virtual sensor. The master robot controls the virtual motion of this sensor. The slave robot uses the sensor's virtual position and speed to adjust its speed. The synchronization is achieved by defining positions where the two robots should be at the same time, and assigning a sensor value for each of these points.

Illustration



4.1.10.3 Master robot configuration parameters

Overview

Use the following parameters to set up the master robot.

Use RobotStudio to change the parameters.

Topic: Motion

SINGLE_TYPE/Parameter	Value
<i>Name</i>	SSYNC2
<i>mechanics</i>	SS_LIN
<i>process_name</i>	SSYNC2
<i>use_path</i>	PSSYNC

Topic: Process

SENSOR_SYSTEM/Parameter	Value
<i>Name</i>	SSYNC1
<i>sensor_type</i>	CAN
<i>use_sensor</i>	CAN1
<i>adjustment_speed</i>	1000
<i>min_dist</i>	600
<i>max_dist</i>	20000
<i>correction_vector_ramp_length</i>	10

Topic: I/O

EIO_UNIT

EIO_UNIT/Parameter	Value
<i>Name</i>	MASTER1
<i>UnitType</i>	DN_SLAVE
<i>Bus</i>	DeviceNet1
<i>DN_Address</i>	1

EIO_SIGNAL

EIO_SIGNAL/Parameter	Value
<i>Name</i>	ao1Position
<i>SignalType</i>	AO
<i>Unit</i>	MASTER1
<i>UnitMap</i>	0-15
<i>MaxLog</i>	10.0
<i>MaxPhys</i>	1
<i>MaxPhysLimit</i>	1

Continues on next page

4 Motion coordination

4.1.10.3 Master robot configuration parameters

Continued

EIO_SIGNAL/Parameter	Value
<i>MaxBitVal</i>	32767
<i>MinLog</i>	-10.0
<i>MinPhys</i>	-1
<i>MinPhysLimit</i>	-1
<i>MinBitVal</i>	-32767

EIO_SIGNAL/Parameters	Value
<i>Name</i>	ao1Speed
<i>SignalType</i>	AO
<i>Unit</i>	MASTER1
<i>UnitMap</i>	16-31
<i>MaxLog</i>	10.0
<i>MaxPhys</i>	1
<i>MaxPhysLimit</i>	1
<i>MaxBitVal</i>	32767
<i>MinLog</i>	-10.0
<i>MinPhys</i>	-1
<i>MinPhysLimit</i>	-1
<i>MinBitVal</i>	-32767

EIO_SIGNAL/Parameters	Value
<i>Name</i>	ao1PredTime
<i>SignalType</i>	AO
<i>Unit</i>	MASTER1
<i>UnitMap</i>	32-47
<i>MaxLog</i>	10.0
<i>MaxPhys</i>	1
<i>MaxPhysLimit</i>	1
<i>MaxBitVal</i>	32767
<i>MinLog</i>	-10.0
<i>MinPhys</i>	-1
<i>MinPhysLimit</i>	-1
<i>MinBitVal</i>	-32767

EIO_SIGNAL/Parameters	Value
<i>Name</i>	do1Dready
<i>SignalType</i>	DO
<i>Unit</i>	MASTER1
<i>UnitMap</i>	48

Continues on next page

EIO_SIGNAL/Parameters	Value
<i>Name</i>	do1Sync2
<i>SignalType</i>	DO
<i>Unit</i>	MASTER1
<i>UnitMap</i>	50

4 Motion coordination

4.1.10.4 Slave robot configuration parameters

4.1.10.4 Slave robot configuration parameters

Overview

For default configuration, see [System parameters on page 189](#).

Use RobotStudio to change the parameters and to set up the slave robot.

Description

To make the slave robot stop and restart synchronized with the master robot:

- Set the parameter value *min_sync_speed* to 0.0

The slave robot will also stop if a fine point is defined in the master robot path.

Topic: Process

SENSOR_SYSTEM

SENSOR_SYSTEM/Parameter	Value
<i>Name</i>	SSYNCS1
<i>sensor_type</i>	CAN
<i>use_sensor</i>	CAN1
<i>adjustment_speed</i>	1000
<i>min_dist</i>	600
<i>max_dist</i>	20000
<i>correction_vector_ramp_length</i>	10
<i>nominal_speed</i>	1000

CAN_INTERFACE

CAN_INTERFACE/Parameters	Value
<i>Name</i>	CAN1
<i>Signal delay</i>	34
<i>Connected signal</i>	c1Connected
<i>Position signal</i>	c1Position
<i>Velocity signal</i>	c1Speed
<i>Null speed signal</i>	c1NullSpeed
<i>Data ready signal</i>	
<i>Waitwobj signal</i>	c1WaitWObj
<i>Dropwobj signal</i>	c1DropWObj
<i>Data Time stamp</i>	c1DTimestamp
<i>RemAllPObj signal</i>	c1RemAllPObj
<i>Virtual sensor</i>	NO
<i>Sensor Speed filter</i>	0,33

Continues on next page

Topic: I/O

EIO_UNIT

EIO_UNIT/Parameters	Value
<i>Name</i>	SLAVE1
<i>UnitType</i>	DN_SLAVE
<i>Bus</i>	DeviceNet2
<i>DN_Address</i>	1

EIO_SIGNAL

EIO_SIGNAL/Parameters	Value
<i>Name</i>	ai1Position
<i>SignalType</i>	AI
<i>Unit</i>	SLAVE1
<i>UnitMap</i>	0-15
<i>MaxLog</i>	10.0
<i>MaxPhys</i>	1
<i>MaxPhysLimit</i>	1
<i>MaxBitVal</i>	32767
<i>MinLog</i>	-10.0
<i>MinPhys</i>	-1
<i>MinPhysLimit</i>	-1
<i>MinBitVal</i>	-32767

EIO_SIGNAL/Parameters	Value
<i>Name</i>	ai1Speed
<i>SignalType</i>	AI
<i>Unit</i>	SLAVE1
<i>UnitMap</i>	16-31
<i>MaxLog</i>	10.0
<i>MaxPhys</i>	1
<i>MaxPhysLimit</i>	1
<i>MaxBitVal</i>	32767
<i>MinLog</i>	-10.0
<i>MinPhys</i>	-1
<i>MinPhysLimit</i>	-1
<i>MinBitVal</i>	-32767

EIO_SIGNAL/Parameters	Value
<i>Name</i>	ai1PredTime
<i>SignalType</i>	AI

Continues on next page

4 Motion coordination

4.1.10.4 Slave robot configuration parameters

Continued

EIO_SIGNAL/Parameters	Value
<i>Unit</i>	SLAVE1
<i>UnitMap</i>	32-47
<i>MaxLog</i>	10.0
<i>MaxPhys</i>	1
<i>MaxPhysLimit</i>	1
<i>MaxBitVal</i>	32767
<i>MinLog</i>	-10.0
<i>MinPhys</i>	-1
<i>MinPhysLimit</i>	-1
<i>MinBitVal</i>	-32767

EIO_SIGNAL/Parameters	Value
<i>Name</i>	di1Dready
<i>SignalType</i>	DI
<i>Unit</i>	SLAVE1
<i>UnitMap</i>	48

EIO_SIGNAL/Parameters	Value
<i>Name</i>	di1Sync2
<i>SignalType</i>	DI
<i>Unit</i>	SLAVE1
<i>UnitMap</i>	50

4.1.10.5 Programming example for master robot

Overview

The following program is an example of how to program a master robot.

Master robot programming

```

syncstart:=20;
Syncpos1:=300;
Syncpos2:=600;
Syncpos3:=900;
Syncpos4:=1200;

!Synchronized motion between master and slave
robpos1.extax.eax_e:=syncpos1;
robpos2.extax.eax_e:=syncpos2;
robpos3.extax.eax_e:=syncpos3;
robpos4.extax.eax_e:=syncpos4;
robpos5.extax.eax_e:=syncstart;

!Init of external axis
pOutsideNext.extax.eax_e:=syncstart;

!Activate sensor
ActUnit SSYNCl;

!Instruction with coordinated robot targets
MoveJ pOutsideNext, v1000, fine, tool1;

!Init of external axis
robposstart.extax.eax_e:=syncstart;

!Set digital output
SetDO Dosync 1,0

!Instructions with coordinated robot targets
MoveJ robposstart, v2000, z50, tool1;

!Set digital output
PulseDO\PLength:= 0.1, doSync1;

!Instructions with coordinated robot targets
MoveJ robpos1, v2000, z10, tool1;
MoveJ robpos2, v2000, z10, tool1;
MoveJ robpos3, v2000, z10, tool1;
MoveJ robpos4, v2000, z10, tool1;
MoveJ robpos5, v2000, z10, tool1;

```

Continues on next page

4 Motion coordination

4.1.10.5 Programming example for master robot

Continued

Considerations

The following is to be considered

- The values of `extax.eax_e` should increase for every `robtarg` during synchronization. The first move instruction of the master robot, after the synchronization, should also have a higher `extax.eax_e` value than the previous instruction. Otherwise the value of `extax.eax_e` may decrease, and the synchronization end, before the slave robot has reached its target.
- The movement back to `syncstart` (move instruction to `robpos5` in the example) may be slower than the ordered speed (`v2000`). If this robot movement is short and the value of `extax.eax_e` is large, the maximum speed will be limited by the virtual sensor speed.
- Do not use `WaitSensor` or `DropSensor`.
- Verify that the virtual sensor max speed (`speed_out`) is less than 1m/s.

4.1.10.6 Programming example for slave robot

Overview

The following program is an example of how to program a slave robot.

Slave robot programming

```
syncstart:=20;
Syncpos1:=300;
Syncpos2:=600;
Syncpos3:=900;

!Synchronized motion between master and slave
robpos1.extax.eax_e:=syncpos1;
robpos2.extax.eax_e:=syncpos2;
robpos3.extax.eax_e:=syncpos3;

!Instructions with coordinated robot targets
MoveJ posstart, v500, z50, tool1;

!Wait for digital input
WaitDI diSync1; 1;

!Connect to the object
WaitSensor SSYNC1;\RelDist:=100;

!Start the Synchronized motion
SyncToSensor SSYNC1\On;

!Instructions with coordinated robot targets
MoveJ robpos1, v2000, z10, tool1;
MoveJ robpos2, v2000, z10, tool1;
MoveJ robpos3, v2000, z10, tool1;

!Stop the synchronized motion
SyncToSensor SSYNC1\Off;
```

Considerations

The following is to be considered:

- Do not use DropSensor.
- Do not use any corvecs.

4 Motion coordination

4.1.11.1 Introduction

4.1.11 Synchronize with hydraulic press using recorded profile

4.1.11.1 Introduction

Overview

This section describes how to use a recorded machine profile to improve the accuracy of robot's synchronization with a hydraulic press. This profile is used for modeling of press path. Not using a recorded profile will require a bigger distance between robot and press model when teaching the path.

Principles of hydraulic press synchronization

- 1 Record the movement of the hydraulic press.
- 2 Activate the record to be used in the next cycle.
- 3 Activate the sensor synchronization with the RAPID instruction `SyncToSensor`.

4.1.11.2 Configuration of system parameters

Introduction

This section describes how to configure the parameters to get the best result when using recorded sensor profiles with a hydraulic press. Start the tuning with the general settings. If the system is not using a DSQC377A encoder, see [Settings for analog input with no DSQC377A encoder on page 181](#). If the sensor is using group input, see [Settings for sensor using Group input on page 182](#). Descriptions of the system parameters are found in [System parameters on page 189](#).

General settings

This parameter belong to the configuration type *Fieldbus Command* in the topic *I/O*.

Parameter	Value
Parameter Value for the instance where Type of Fieldbus Command is IIRFFP.	10-15 Hz, Change this value to get good accuracy during start and stop.

This parameter belong to the configuration type *Path Sensor Synchronization* in the topic *Motion*.

Parameter	Value
Synchronization Type	ROBOT_TO_HPRES

The parameters belong to the configuration type *Sensor systems* in the topic *Process*.

Parameter	Value
Sensor start signal	Type the name of the I/O signal
Stop press signal	Type the name of the I/O signal
Sync Alarm signal	Type the name of the I/O signal

Settings for analog input with no DSQC377A encoder

The parameters belong to the configuration type *Can Interface* in the topic *Process*.

Parameter	Value
Virtual sensor	Yes
Position signal	Type the name of the analog input.



Note

All other signals except Position signal should be empty (i.e. "").



Tip

WaitSensor and DropSensor are not needed in the RAPID program.

Continues on next page

4 Motion coordination

4.1.11.2 Configuration of system parameters

Continued

Settings for sensor using Group input

The parameters belong to the configuration type *Sensor systems* in the topic *Process*.

Parameter	Value
Pos Group IO scale	Define the number of input data per meter, the default value is set to 10000.

The parameters belong to the configuration type *Can Interface* in the topic *Process*.

Parameter	Value
Virtual sensor	Yes
Position signal	Type the name of the used group input.



Note

All other signals except Position signal should be empty (i.e. "")



Tip

`WaitSensor` and `DropSensor` are not needed in the RAPID program.

4.1.11.3 Program example

Overview

This section describes the programming cycles that are typical for programming a hydraulic press.

Program example

First press cycle

A pulse on *sensor_start_signal* will start storing position in a record array.

During this cycle the robot is not synchronized with press.

```
ActUnit SSYNCl;
WaitSensor SSYNCl;
! Set up a recording for 2 seconds
PrxStartRecord SSYNCl, 2, PRX_HPRESS_PROF;
! Process waiting for sensor_start_signal
! then waiting for press movement and record it during 2 sec.
```

Second press cycle

A pulse on *sensor_start_signal* is needed to synchronize readings of record and actual positions for each cycle.

During press opening the robot moves synchronized with press.

```
PrxActivAndStoreRecord SSYNCl, 0, "profile.log";
WaitSensor Ssync1;
MoveL p10, v1000, z10, tool, \WObj:=wobj0;
SyncToSensor Ssync1\On;
MoveL p20, v1000, z20, tool, \WObj:=wobj0;
MoveL p30, v1000, z20, tool, \WObj:=wobj0;
SyncToSensor Ssync1\Off;
```

Third press cycle

No special instruction is needed, but a pulse on *sensor_start_signal* is needed to synchronize readings of record and actual positions for each cycle. A new record can also be started.

During press opening the robot moves synchronized with press.

```
WaitSensor Ssync1;
MoveL p10, v1000, z10, tool, \WObj:=wobj0;
SyncToSensor Ssync1\On;
MoveL p20, v1000, z20, tool, \WObj:=wobj0;
MoveL p30, v1000, z20, tool, \WObj:=wobj0;
SyncToSensor Ssync1\Off;
```

4 Motion coordination

4.1.12.1 Introduction

4.1.12 Synchronize with molding machine using recorded profile

4.1.12.1 Introduction

Overview

This section describes how to use a recorded machine profile to improve the accuracy of a robot's synchronization with a molding machine. This profile is used for modeling of mold path. Not using a recorded profile will require a bigger distance between robot and machine model when teaching the path.

Principles of mold synchronization

- 1 Record the movement of the Molding machine.
- 2 Activate the record to be used in the next cycle.
- 3 Activate the sensor synchronization with the RAPID instruction `SynctoSensor`.



Tip

When the molding machine is closing, supervision can be used instead of synchronization. For more information, see [Supervision on page 188](#).

4.1.12.2 Configuration of system parameters

Introduction

This section describes how to configure the parameters to get the best result when using recorded sensor profiles with a molding machine. Start the tuning with the general settings. If the system is not using a DSQC377A encoder, see [Settings for analog input with no DSQC377A encoder on page 185](#). If the sensor is using group input, see [Settings for sensor using Group input on page 186](#). Descriptions of the system parameters are found in [System parameters on page 189](#).

General settings

This parameter belong to the configuration type *Fieldbus Command* in the topic *I/O*.

Parameter	Value
Parameter Value for the instance where Type of Fieldbus Command is IIRFFP.	10-15 Hz, Change this value to get good accuracy during start and stop.

This parameter belong to the configuration type *Path Sensor Synchronization* in the topic *Motion*.

Parameter	Value
Synchronization Type	SYNC_TO_IMM

The parameters belong to the configuration type *Sensor systems* in the topic *Process*.

Parameter	Value
Sensor start signal	Type the name of the I/O signal
Stop press signal	Type the name of the I/O signal
Sync Alarm signal	Type the name of the I/O signal

Settings for analog input with no DSQC377A encoder

The parameters belong to the configuration type *Can Interface* in the topic *Process*.

Parameter	Value
Virtual sensor	Yes
Position signal	Type the name of the analog input.



Note

All other signals except Position signal should be empty (i.e. "").



Tip

WaitSensor and DropSensor are not needed in the RAPID program.

Continues on next page

4 Motion coordination

4.1.12.2 Configuration of system parameters

Continued

Settings for sensor using Group input

The parameters belong to the configuration type *Sensor systems* in the topic *Process*.

Parameter	Value
Pos Group IO scale	Define the number of increments per meter for the group input. The default value is set to 10000.

The parameters belong to the configuration type *Can Interface* in the topic *Process*.

Parameter	Value
Virtual sensor	Yes
Position signal	Type the name of the used group input.



Note

All other signals except Position signal should be empty (i.e. "")



Tip

`WaitSensor` and `DropSensor` are not needed in the RAPID program.

4.1.12.3 Program example

Overview

This section describes the programming cycles that are typical for programming a molding machine.

Program example

First press cycle

A pulse on *sensor_start_signal* will start storing position in a record array.

During this cycle the robot is not synchronized with press.

```
ActUnit SSYNCl;
WaitSensor SSYNCl;
! Set up a recording for 2 seconds
PrxStartRecord SSYNCl, 2, PRX_PROFILE_T1;
! Process waiting for sensor_start_signal
! then waiting for press movement and record it during 2 sec.
```

Second press cycle

A pulse on *sensor_start_signal* is needed to synchronize readings of record and actual positions for each cycle.

During press opening the robot moves synchronized with press.

```
PrxActivAndStoreRecord SSYNCl, 0, "profile.log";
WaitSensor Ssync1;
MoveL p10, v1000, z10, tool, \WObj:=wobj0;
SyncToSensor Ssync1\On;
MoveL p20, v1000, z20, tool, \WObj:=wobj0;
MoveL p30, v1000, z20, tool, \WObj:=wobj0;
SyncToSensor Ssync1\Off;
```

Third press cycle

No special instruction is needed, but a pulse on *sensor_start_signal* is needed to synchronize readings of record and actual positions for each cycle. A new record can also be started.

During press opening the robot moves synchronized with press.

```
WaitSensor Ssync1;
MoveL p10, v1000, z10, tool, \WObj:=wobj0;
SyncToSensor Ssync1\On;
MoveL p20, v1000, z20, tool, \WObj:=wobj0;
MoveL p30, v1000, z20, tool, \WObj:=wobj0;
SyncToSensor Ssync1\Off;
```

4 Motion coordination

4.1.13 Supervision

4.1.13 Supervision

Introduction

The supervision can be used to save cycle time when robot moves outside the mold or press. Instead of waiting to be outside the machine to enable close mold the robot enable close mold when it starts to move outside the mold after picking the part.

The supervision can stop the mold if it comes too near the robot by setting the output signal defined by the system parameter *Sync Alarm signal*.

`SupSyncSensorOn` is used to supervise the movement of the robot with the mold or press. Usually supervision is used until the robot is moved outside the mold or press. With supervision it is possible to turn off the synchronization and turn on supervision when a workpiece is dropped or collected in the molding machine.

`SupSyncSensorOn` protects the robot and machine from damaging.

Supervision does not deactivate the synchronization.

Example

For the case you cannot move the sensor to defined position you have to set the external axis value in your rapid program

```
p10.extax.eax_f:=sens10;
p20.extax.eax_f:=sens20;
p30.extax.eax_f:=sens30;
WaitSensor Ssync1;
MoveL p10, v1000, fine, tool, \WObj:=wobj0;
SupSyncSensorOn Ssync1, 150, -100, 650\SafetyDelay:=0;;
MoveL p20, v1000, z20, tool, \WObj:=wobj0;
MoveL p30, v1000, fine, tool, \WObj:=wobj0;
SupSyncSensorOff Ssync1;
```

`Sens10` is the expected position of the machine (model of the machine movement related to robot movement) when robot will be at `p10` and `sens20` is the expected position of the machine when robot will be at `p20`.

The supervision will be done between the sensor position 650 and 150 mm and triggers the output if the distance between the robot and the mould is smaller than 100 mm.

`Safetydist` (in this case -100) is the limit of the difference between expected machine position and the real machine position. It must be negative, i.e. the model should always be moving in advance of the real machine. In the case of decreasing machine positions the limit must be negative corresponding to maximum negative position difference (and minimum advance distance). In the case of increasing machine positions the limit must be positive corresponding to minimum positive position difference (and minimum advance distance).

4.1.14 System parameters

About system parameters

This section describes the system parameters in a general way. For more information about the parameters, see *Technical reference manual - System parameters*.

Fieldbus Command

Only for Sensor Synchronization.

These are different instances of the type *Fieldbus Command* in the topic *I/O*.

Type of Fieldbus Command	Description
Counts Per Meter	The number of counts per meter of the external device motion.
Sync Separation	Defines the minimum distance that the external device must move after a sync signal before a new sync signal is accepted as a valid object. For Sensor Synchronization, there is no need to change the default value.
Queue Tracking Distance	Defines the placement of the synchronization switch relative to the 0.0 meter point on the sensor. For Sensor Synchronization, there is no need to change the default value.
Start Window Width	Defines the size of the start window. It is possible to connect to objects within this window with the instruction <code>WaitSensor</code> . For Sensor Synchronization, there is no need to change the default value.
IIRFFP	Specifies the location of the real part of the poles in the left-half plane (in Hz).

Sensor systems

These parameters belong to the topic *Process* and the type *Sensor System*.

Parameter	Description
Adjustment speed	When entering sensor synchronization, the robot speed must be adjusted to the speed of the external device. The speed (in mm/s) at which the robot 'catches up' to this speed for the first motion is defined by <i>Adjustment Speed</i> .
Min dist	The minimum distance (in millimeters) that a connected object may have before being automatically dropped. For Sensor Synchronization, there is no need to change the default value. Not used for Analog Synchronization.
Max dist	The maximum distance (in millimeters) that a connected object may have before being automatically dropped. For Sensor Synchronization, there is no need to change the default value. Not used for Analog Synchronization.
Sensor nominal speed	The nominal work speed of the external device. If the speed of the device exceeds 200 mm/s this parameter must be increased.

Continues on next page

4 Motion coordination

4.1.14 System parameters

Continued

Parameter	Description
Stop press signal	Name of the digital input signal telling that press is stopping. This signal is needed for safe stop of robot.
Sensor start signal	Name of the digital input signal to synchronize recorded profile and new machine movement. The signal must be set before start of machine movement. The signal must be triggered 100 ms before the press moves.
Start ramp	Defines for how many calculation steps the position error may exceed <i>Max Advance Distance</i> . During this ramping period, the position error may be 5 times <i>Max Advance Distance</i> .
Sync Alarm signal	Name of the digital output signal to stop the synchronized machine. This signal may be set during supervision of sync sensor.

CAN Interface

These parameters belong to the topic *Process* and the type *CAN Interface*.

Parameter	Description
Connected signal	Name of the digital input signal for connection. Not used for Analog Synchronization.
Position signal	Name of the analog input signal for sensor position.
Velocity signal	Name of the analog input signal for sensor speed.
Null speed signal	Name of the digital input signal indicating zero speed on the sensor. Not used for Analog Synchronization.
Data ready signal	Name of the digital input signal indicating a poll of the encoder unit. Not used for Analog Synchronization.
Waitwobj signal	Name of the digital output signal to indicate that a connection is desired to an object in the queue. Not used for Analog Synchronization.
Dropwobj signal	Name of the digital output signal to drop a connected object on the encoder unit Not used for Analog Synchronization.
PassStartW signal	Name of the digital output signal to indicate that an object has gone past the start window without being connected. Not used for Analog Synchronization.
Pos Update time	Time (in ms) at which the synchronization process read the sensor position.

Motion Planner

These parameters belong to the topic *Motion* and the type *Motion planner*.

Parameter	Description
Path resolution	The period at which steps along the path are calculated.
Process update time	The time (in seconds) at which the sensor process updates the robot kinematics on the sensor position.
CPU load equalization	<i>CPU load equalization</i> needs to be lowered for the synchronization option. The default value is 2 but for the synchronization option it should be set equal to 1 to have a stable synchronization speed.

Continues on next page

Mechanical unit

These parameters belong to the topic *Motion* and the type *Mechanical unit*.

Parameter	Description
Name	The name of the unit (max. 7 characters).
Activate at start up	The sensor is to be activated automatically at start up.
Deactivate Forbidden	The sensor cannot be deactivated.

Single type

This parameter belongs to the topic *Motion* and the type *Single type*.

Parameter	Description
Mechanics	Specifies the mechanical structure of the sensor.

Transmission

This parameter belong to the topic *Motion* and the type *Transmission*.

Parameter	Description
Rotating move	Specifies if the sensor is rotating (Yes) or linear (No).

Path Sensor Synchronization

These parameters belong to the topic *Motion* and the type *Path Sensor Synchronization*. They are used to set allowed deviation between calculated and actual position of the external device, and minimum/maximum TCP speed for the robot.

Parameter	Description
Max Advance Distance	The max advance distance allowed from calculated position to actual position of the external device.
Max Delay Distance	The max delay distance allowed from calculated position to actual position of the external device.
Max Synchronization Speed	The max robot TCP speed allowed in m/s.
Min Synchronization Speed	The min robot TCP speed allowed in m/s.

4 Motion coordination

4.1.15 I/O signals

4.1.15 I/O signals

Overview

Sensor Synchronization provides several I/O signals which allow a user or RAPID program to monitor and control the object queue on the encoder interface unit. The object queue is designed for the option Conveyor Tracking and has more functionality than required by Sensor Synchronization. Since each closing of a press is considered an object in the object queue, signals for the object queue may occasionally be useful.

Object queue signals

The following table shows the I/O signals in the encoder unit DSQC 354 which impact the object queue.

Instruction	Description
c1ObjectsInQ	Group input showing the number of objects in the object queue. These objects are registered by the synchronization switch and have not been dropped.
c1Rem1PObj	Digital output that removes the first pending object from the object queue. Pending objects are objects that are in the queue but are not connected to a work object.
c1RemAllPObj	Digital output that removes all pending objects. If an object is connected, then it is not removed.
c1DropWObj	Digital output that will cause the encoder interface unit to drop the tracked object and disconnect it. The object is removed from the queue. Do not use <i>c1DropWObj</i> in RAPID code. Use the <code>DropWObj</code> instruction instead.

4.1.16 RAPID components

About the RAPID components

This is an overview of all instructions, functions, and data types in *Machine Synchronization*.

For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instructions

Instructions	Description
DropSensor	Drop object on sensor
PrxActivAndStoreRecord	Activate and store the recorded profile data
PrxActivRecord	Activate the recorded profile data
PrxDBgStoreRecord	Store and debug the recorded profile data
PrxDeactRecord	Deactivate a record
PrxResetPos	Reset the zero position of the sensor
PrxResetRecords	Reset and deactivate all records
PrxSetPosOffset	Set a reference position for the sensor
PrxSetRecordSampleTime	Set the sample time for recording a profile
PrxSetSynalarm	Set sync alarm behavior
PrxStartRecord	Record a new profile
PrxStopRecord	Stop recording a profile
PrxStoreRecord	Store the recorded profile data
PrxUseFileRecord	Use the recorded profile data
SupSyncSensorOff	Stop synchronized sensor supervision
SupSyncSensorOn	Start synchronized sensor supervision
SyncToSensor	Sync to sensor
WaitSensor	Wait for connection on sensor

Functions

Functions	Description
PrxGetMaxRecordpos	Get the maximum sensor position

Data types

Machine Synchronization includes no data types.

This page is intentionally left blank

5 Motion Events

5.1 World Zones [608-1]

5.1.1 Overview

Purpose

The purpose of World Zones is to stop the robot or set an output signal if the robot is inside a special user-defined zone. 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 World Zones supervision.
- When a permanent obstacle or some temporary external equipment is located inside the robot's work area. A forbidden zone can be created to prevent the robot from colliding with this equipment.
- Indication that the robot is at a position where it is permissible to start program execution from a Programmable Logic Controller (PLC).

A world zone is supervised during robot movements both during program execution and jogging. If the robot's TCP reaches the world zone or if the axes reaches the world zone in joints, the movement is stopped or a digital output signal is set.



WARNING

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

What is included

The RobotWare option World Zones gives you access to:

- instructions used to define volumes of various shapes
- instructions used to define joint zones in coordinates for axes
- instructions used to define and enable world zones

Basic approach

This is the general approach for setting up World Zones. For a more detailed example of how this is done, see [Code examples on page 199](#).

- 1 Declare the world zone as stationary or temporary.
- 2 Declare the shape variable.
- 3 Define the shape that the world zone shall have.
- 4 Define the world zone (that the robot shall stop or that an output signal shall be set when reaching the volume).

Continues on next page

5 Motion Events

5.1.1 Overview

Continued

Limitations

Supervision of a volume only works for the TCP. Any other part of the robot may pass through the volume undetected. To be certain to prevent this, you can supervise a joint world zone (defined by `WZLimJointDef` or `WZHomeJointDef`).

A variable of type `wzstationary` or `wztemporary` can not be redefined. They can only be defined once (with `WZLimSup` or `WZDOSet`).

5.1.2 RAPID components

Data types

This is a brief description of each data type in World Zones. For more information, see respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
wztemporary	<p>wztemporary is used to identify a temporary world zone and can be used anywhere in the RAPID program.</p> <p>Temporary world zones can be disabled, enabled again, or erased via RAPID instructions. Temporary world zones are automatically erased when a new program is loaded or when program execution start from the beginning in the MAIN routine.</p>
wzstationary	<p>wzstationary is used to identify a stationary world zone and can only be used in an event routine connected to the event POWER ON. For information on defining event routines, see <i>Operating manual - IRC5 with FlexPendant</i>.</p> <p>A stationary world zone is always active and is reactivated by a restart (switch power off then on, or change system parameters). It is not possible to disable, enable or erase a stationary world zone via RAPID instructions.</p> <p>Stationary world zones shall be used if security is involved.</p>
shapedata	<p>shapedata is used to describe the geometry of a world zone.</p> <p>World zones can be defined in 4 different geometrical shapes:</p> <ul style="list-style-type: none"> • a straight box, with all sides parallel to the world coordinate system • a cylinder, parallel to the z axis of the world coordinate system • a sphere • a joint angle area for the robot axes and/or external axes

Instructions

This is a brief description of each instruction in World Zones. For more information, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
WZBoxDef	<p>WZBoxDef is used to define a volume that has the shape of a straight box with all its sides parallel to the axes of the world coordinate system. The definition is stored in a variable of type shapedata.</p> <p>The volume can also be defined as the inverse of the box (all volume outside the box).</p>
WZCylDef	<p>WZCylDef is used to define a volume that has the shape of a cylinder with the cylinder axis parallel to the z-axis of the world coordinate system. The definition is stored in a variable of type shapedata.</p> <p>The volume can also be defined as the inverse of the cylinder (all volume outside the cylinder).</p>
WZSphDef	<p>WZSphDef is used to define a volume that has the shape of a sphere. The definition is stored in a variable of type shapedata.</p> <p>The volume can also be defined as the inverse of the sphere (all volume outside the sphere).</p>

Continues on next page

5 Motion Events

5.1.2 RAPID components

Continued

Instruction	Description
WZLimJointDef	<p>WZLimJointDef is used to define joint coordinate for axes, to be used for limitation of the working area. Coordinate limits can be set for both the robot axes and external axes.</p> <p>For each axis WZLimJointDef defines an upper and lower limit. For rotational axes the limits are given in degrees and for linear axes the limits are given in mm.</p> <p>The definition is stored in a variable of type <code>shapedata</code>.</p>
WZHomeJointDef	<p>WZHomeJointDef is used to define joint coordinates for axes, to be used to identify a position in the joint space. Coordinate limits can be set for both the robot axes and external axes.</p> <p>For each axis WZHomeJointDef defines a joint coordinate for the middle of the zone and the zones delta deviation from the middle. For rotational axes the coordinates are given in degrees and for linear axes the coordinates are given in mm.</p> <p>The definition is stored in a variable of type <code>shapedata</code>.</p>
WZLimSup	<p>WZLimSup is used to define, and enable, stopping the robot with an error message when the TCP reaches the world zone. This supervision is active both during program execution and when jogging.</p> <p>When calling WZLimSup you specify whether it is a stationary world zone, stored in a <code>wzstationary</code> variable, or a temporary world zone, stored in a <code>wztemporary</code> variable.</p>
WZDOSet	<p>WZDOSet is used to define, and enable, setting a digital output signal when the TCP reaches the world zone.</p> <p>When calling WZDOSet you specify whether it is a stationary world zone, stored in a <code>wzstationary</code> variable, or a temporary world zone, stored in a <code>wztemporary</code> variable.</p>
WZDisable	<p>WZDisable is used to disable the supervision of a temporary world zone.</p>
WZEnable	<p>WZEnable is used to re-enable the supervision of a temporary world zone.</p> <p>A world zone is automatically enabled on creation. Enabling is only necessary after it has been disabled with WZDisable.</p>
WZFree	<p>WZFree is used to disable and erase a temporary world zone.</p>

Functions

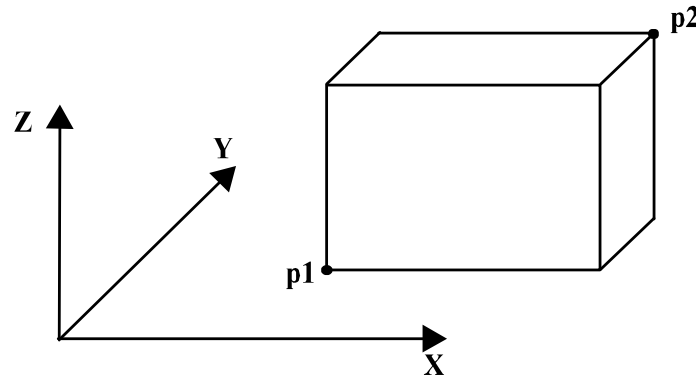
World Zones does not include any RAPID functions.

5.1.3 Code examples

Create protected box

To prevent the robot TCP from moving into stationary equipment, set up a stationary world zone around the equipment.

The routine `my_power_on` should then be connected to the event POWER ON. For information on how to do this, read about defining event routines in *Operating manual - IRC5 with FlexPendant*.



xx0300000178

```
VAR wzstationary obstacle;
PROC my_power_on()
  VAR shapedata volume;
  CONST pos p1 := [200, 100, 100];
  CONST pos p2 := [600, 400, 400];

  !Define a box between the corners p1 and p2
  WZBoxDef \Inside, volume, p1, p2;

  !Define and enable supervision of the box
  WZLimSup \Stat, obstacle, volume;
ENDPROC
```

Signal when robot is in position

When two robots share a work area it is important to know when a robot is out of the way, letting the other robot move freely.

This example defines a home position where the robot is in a safe position and sets an output signal when the robot is in its home position. The robot is standing on a travel track, handled as external axis 1. No other external axes are active.

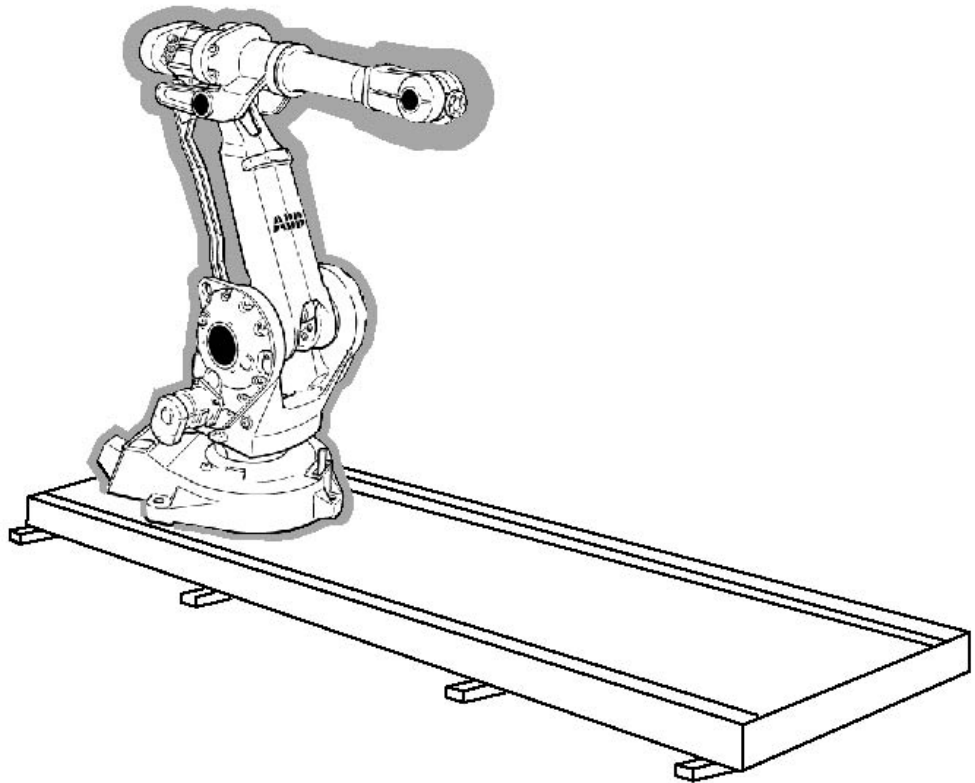
Continues on next page

5 Motion Events

5.1.3 Code examples

Continued

The shadowed area in the illustration shows the world zone.



xx0300000206

```
VAR wztemporary home;
PROC zone_output()
  VAR shapedata joint_space;

  !Define the home position
  CONST jointtarget home_pos := [[0, -20, 0, 0, 0, 0], [0, 9E9,
    9E9, 9E9, 9E9, 9E9]];

  !Define accepted deviation from the home position
  CONST jointtarget delta_pos := [[2, 2, 2, 2, 2, 2], [10, 9E9,
    9E9, 9E9, 9E9, 9E9]];

  !Define the shape of the world zone
  WZHomeJointDef \Inside, joint_space, home_pos, delta_pos;

  !Define the world zone, setting the
  !signal do_home to 1 when in zone
  WZDSet \Temp, home \Inside, joint_space, do_home, 1;
ENDPROC
```


6 Motion functions

6.1 Independent Axes [610-1]

6.1.1 Overview

Purpose

The purpose of Independent Axes is to move an axis independently of other axes in the robot system. Some examples of applications are:

- Move an external axis holding an object (for example rotating an object while the robot is spray painting it).
- Save cycle time by performing a robot task at the same time as an external axis performs another.
- Continuously rotate robot axis 6 (for polishing or similar tasks).
- Reset the measurement system after an axis has rotated multiple revolutions in the same direction. Saves cycle time compared to physically winding back.

An axis can move independently if it is set to independent mode. An axis can be changed to independent mode and later back to normal mode again.

What is included

The RobotWare option Independent Axes gives you access to:

- instructions used to set independent mode and specify the movement for an axis
- an instruction for changing back to normal mode and/or reset the measurement system
- functions used to verify the status of an independent axis
- system parameters for configuration.

Basic approach

This is the general approach for moving an axis independently. For detailed examples of how this is done, see [Code examples on page 205](#).

- 1 Call an independent move instruction to set the axis to independent mode and move it.
- 2 Let the robot execute another instruction at the same time as the independent axis moves.
- 3 When both robot and independent axis has stopped, reset the independent axis to normal mode.

Reset axis

Even without being in independent mode, an axis might rotate only in one direction and eventually lose precision. The measurement system can then be reset with the instruction `IndReset`.

The recommendation is to reset the measurement system for an axis before its motor has rotated 10000 revolutions in the same direction.

Continues on next page

6 Motion functions

6.1.1 Overview

Continued

Limitations

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

Axes in independent mode cannot be jogged.

The only robot axis that can be used as an independent axis is axis number 6. On IRB 1600, 2600 and 4600 models (except ID version), the instruction `IndReset` can also be used for axis 4.

6.1.2 System parameters

About the system parameters

This is a brief description of each parameter in Independent Axes. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Arm

These parameters belongs to the type *Arm* in the topic *Motion*.

Parameter	Description
Independent Joint	Flag that determines if independent mode is allowed for the axis.
Independent Upper Joint Bound	Defines the upper limit of the working area for the joint when operating in independent mode.
Independent Lower Joint Bound	Defines the lower limit of the working area for the joint when operating in independent mode.

Transmission

These parameters belong to the type *Transmission* in the topic *Motion*.

Parameter	Description
Transmission Gear High	Independent Axes requires high resolution in transmission gear ratio, which is therefore defined as <i>Transmission Gear High</i> divided by <i>Transmission Gear Low</i> . If no smaller number can be used, the transmission gear ratio will be correct if <i>Transmission Gear High</i> is set to the number of cogs on the robot axis side, and <i>Transmission Gear Low</i> is set to the number of cogs on the motor side.
Transmission Gear Low	See <i>Transmission Gear High</i> .

6 Motion functions

6.1.3 RAPID components

6.1.3 RAPID components

Data types

There are no data types for Independent Axes.

Instructions

This is a brief description of each instruction in Independent Axes. For more information, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

An independent move instruction is executed immediately, even if the axis is being moved at the time. If a new independent move instruction is executed before the last one is finished, the new instruction immediately overrides the old one.

Instruction	Description
IndAMove	IndAMove (Independent Absolute position Movement) change an axis to independent mode and move the axis to a specified position.
IndCMove	IndCMove (Independent Continuous Movement) change an axis to independent mode and start moving the axis continuously at a specified speed.
IndDMove	IndDMove (Independent Delta position Movement) change an axis to independent mode and move the axis a specified distance.
IndRMove	IndRMove (Independent Relative position Movement) change a rotational axis to independent mode and move the axis to a specific position within one revolution. Because the revolution information in the position is omitted, IndRMove never rotates more than one axis revolution.
IndReset	IndReset is used to change an independent axis back to normal mode. IndReset can move the measurement system for a rotational axis a number of axis revolutions. The resolution of positions is decreased when moving away from logical position 0, and winding the axis back would take time. By moving the measurement system the resolution is maintained without physically winding the axis back. Both the independent axis and the robot must stand still when calling IndReset.

Functions

This is a brief description of each function in Independent Axes. For more information, see respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
IndInpos	IndInpos indicates whether an axis has reached the selected position.
IndSpeed	IndSpeed indicates whether an axis has reached the selected speed.

6.1.4 Code examples

Save cycle time

An object in station A needs welding in two places. The external axis for station A can turn the object in position for the second welding while the robot is welding on another object. This saves cycle time compared to letting the robot wait while the external axis moves.

```
!Perform first welding in station A
!Call subroutine for welding
weld_stationA_1;

!Move the object in station A, axis 1, with
!independent movement to position 90 degrees
!at the speed 20 degrees/second
IndAMove Station_A,1\ToAbsNum:=90,20;

!Let the robot perform another task while waiting
!Call subroutine for welding
weld_stationB_1;

!Wait until the independent axis is in position
WaitUntil IndInpos(Station_A,1 ) = TRUE;
WaitTime 0.2;

!Perform second welding in station A
!Call subroutine for welding
weld_stationA_2;
```

Polish by rotating axis 6

To polish an object the robot axis 6 can be set to continuously rotate.

Set robot axis 6 to independent mode and continuously rotate it. Move the robot over the area you want to polish. Stop movement for both robot and independent axis before changing back to normal mode. After rotating the axis many revolutions, reset the measurement system to maintain the resolution.

Note that, for this example to work, the parameter *Independent Joint* for rob1_6 must be set to Yes.

```
PROC Polish()
!Change axis 6 of ROB_1 to independent mode and
!rotate it with 180 degrees/second
IndCMove ROB_1, 6, 180;

!Wait until axis 6 is up to speed
WaitUntil IndSpeed(ROB_1,6\InSpeed);
WaitTime 0.2;

!Move robot where you want to polish
MoveL p1,v10, z50, tool1;
MoveL p2,v10, fine, tool1;
```

Continues on next page

6 Motion functions

6.1.4 Code examples

Continued

```
!Stop axis 6 and wait until it's still
IndCMove ROB_1, 6, 0;
WaitUntil IndSpeed(ROB_1,6\ZeroSpeed);
WaitTime 0.2;

!Change axis 6 back to normal mode and
!reset measurement system (close to 0)
IndReset ROB_1, 6 \RefNum:=0 \Short;
ENDPROC
```

Reset an axis

This is an example of how to reset the measurement system for axis 1 in station A. The measurement system will change a whole number of revolutions, so it is close to zero ($\pm 180^\circ$).

```
IndReset Station_A, 1 \RefNum:=0 \Short;
```

6.2 Path Recovery [611-1]

6.2.1 Overview

Purpose

Path Recovery is used to store the current movement path, perform some robot movements and then restore the interrupted path. This is useful when an error or interrupt occurs during the path movement. An error handler or interrupt routine can perform a task and then recreate the path.

For applications like arc welding and gluing, it is important to continue the work from the point where the robot left off. If the robot started over from the beginning, then the work piece would have to be scrapped.

If a process error occurs when the robot is inside a work piece, moving the robot straight out might cause a collision. By using the path recorder, the robot can instead move out along the same path it came in.

What is included

The RobotWare option Path Recovery gives you access to:

- instructions to suspend and resume the coordinated synchronized movement mode on the error or interrupt level.
- a path recorder, with the ability to move the TCP out from a position along the same path it came.

Limitations

The instructions `StorePath` and `RestoPath` only handles movement path data. The stop position must also be stored.

Movements using the path recorder has to be performed on trap-level, i.e. `StorePath` has to be executed prior to `PathRecMoveBwd`.

6 Motion functions

6.2.2 RAPID components

6.2.2 RAPID components

Data types

This is a brief description of each data type in Path Recovery. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
pathrecid	pathrecid is used to identify a breakpoint for the path recorder.

Instructions

This is a brief description of each instruction in Path Recovery. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
StorePath	StorePath is used to store the movement path being executed when an error or interrupt occurs. StorePath is included in RobotWare base.
RestoPath	RestoPath is used to restore the path that was stored by StorePath. RestoPath is included in RobotWare base.
PathRecStart	PathRecStart is used to start recording the robot's path. The path recorder will store path information during execution of the robot program.
PathRecStop	PathRecStop is used to stop recording the robot's path.
PathRecMoveBwd	PathRecMoveBwd is used to move the robot backwards along a recorded path.
PathRecMoveFwd	PathRecMoveFwd is used to move the robot back to the position where PathRecMoveBwd was executed. It is also possible to move the robot partly forward by supplying an identifier that has been passed during the backward movement.
SyncMoveSuspend	SyncMoveSuspend is used to suspend synchronized movements mode and set the system to independent movement mode.
SyncMoveResume	SyncmoveResume is used to go back to synchronized movements from independent movement mode.

Functions

This is a brief description of each function in Path Recovery. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
PathRecValidBwd	PathRecValidBwd is used to check if the path recorder is active and if a recorded backward path is available.
PathRecValidFwd	PathRecValidFwd is used to check if the path recorder can be used to move forward. The ability to move forward with the path recorder implies that the path recorder must have been ordered to move backwards earlier.

6.2.3 Store current path

Why store the path?

The simplest way to use Path Recovery is to only store the current path to be able to restore it after resolving an error or similar action.

Let's say that an error occurs during arc welding. To resolve the error the robot might have to be moved away from the part. When the error is resolved, the welding should be continued from the point it left off. This is solved by storing the path information and the position of the robot before moving away from the path. The path can then be restored and the welding resumed after the error has been handled.

Basic approach

This is the general approach for storing the current path:

- 1 At the start of an error handler or interrupt routine:
 - stop the movement
 - store the movement path
 - store the stop position
- 2 At the end of the error handler or interrupt routine:
 - move to the stored stop position
 - restore the movement path
 - start the movement

Example

This is an example of how to use Path Recovery in error handling. First the path and position is stored, the error is corrected and then the robot is moved back in position and the path is restored.

```
MoveL p100, v100, z10, gun1;
...
ERROR
  IF ERRNO=MY_GUN_ERR THEN
    gun_cleaning();
  ENDIF
...
PROC gun_cleaning()
  VAR robtarg p1;

  !Stop the robot movement, if not already stopped.
  StopMove;

  !Store the movement path and current position
  StorePath;
  p1 := CRobT(\Tool:=gun1\WObj:=wobj0);

  !Correct the error
  MoveL pclean, v100, fine, gun1;
```

Continues on next page

6 Motion functions

6.2.3 Store current path

Continued

```
...
!Move the robot back to the stored position
MoveL p1, v100, fine, gun1;

!Restore the path and start the movement
RestoPath;
StartMove;
RETRY;
ENDPROC
```

Store path in a MultiMove system

In a MultiMove system the robots can keep the synchronized movement mode after `StorePath` with the argument `KeepSync`. However the robots can't switch from independent mode to synchronized mode, only the other way around.

After a Multimove system is set with the argument `KeepSync`, the system can change between synchronized, semi coordinated and independent mode on the `StorePath` level. The changes are made with the instructions `SyncMoveResume` and `SyncMoveSuspend`.

“SyncArc” example with coordinated synchronized movement

This is an example on how to use Path Recovery and keep synchronized mode in the error handler for a MultiMove system. Two robots perform arc welding on the same work piece. To make the example simple and general, we use move instructions instead of weld instructions. The work object is rotated by a positioner. For more information on the SyncArc example, see *Application manual - MultiMove*.

T_ROB1 task program

```
MODULE module1
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;
PERS tasks all_tasks{3} := [{"T_ROB1"}, {"T_ROB2"}, {"T_STN1"}];
PERS wobjdata wobj_stn1 := [ FALSE, FALSE, "STN_1", [ [0, 0, 0],
[1, 0, 0, 0] ], [ [0, 0, 250], [1, 0, 0, 0] ] ];
TASK PERS tooldata tool1 := ...
CONST robtarget p100 := ...
CONST robtarget p199 := ...
PROC main()
...
SyncMove;
ENDPROC

PROC SyncMove()
MoveJ p100, v1000, z50, tool1;
WaitSyncTask sync1, all_tasks;
MoveL p101, v500, fine, tool1;
SyncMoveOn sync2, all_tasks;
MoveL p102\ID:=10, v300, z10, tool1 \WObj:=wobj_stn1;
MoveC p103, p104\ID:=20, v300, z10, tool1 \WObj:=wobj_stn1;
MoveL p105\ID:=30, v300, z10, tool1 \WObj:=wobj_stn1;
```

Continues on next page

```
MoveC p106, p101\ID:=40, v300, fine, tool1 \WObj:=wobj_stn1;
SyncMoveOff sync3;
MoveL p199, v1000, fine, tool1;
ERROR
  IF ERRNO = ERR_PATH_STOP THEN
    gun_cleaning();
  ENDIF
UNDO
  SyncMoveUndo;
ENDPROC

PROC gun_cleaning()
  VAR robtarget p1;
  !Store the movement path and current position
  ! and keep synchronized mode.
  StorePath \KeepSync;
  p1 := CRobT(\Tool:=tool1 \WObj:=wobj_stn1);
  !Correct the error
  MoveL pclean1 \ID:=50, v100, fine, tool1 \WObj:=wobj_stn1;
  ...
  !Move the robot back to the stored position
  MoveL p1 \ID:=60, v100, fine, tool1 \WObj:=wobj_stn1;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
ENDMODULE
```

T_ROB2 task program

```
MODULE module2
VAR syncident sync1;
VAR syncident sync2;
VAR syncident sync3;
PERS tasks all_tasks{3};
PERS wobjdata wobj_stn1;
TASK PERS tooldata tool2 := ...
CONST robtarget p200 := ...
CONST robtarget p299 := ...
PROC main()
  ...
  SyncMove;
ENDPROC
PROC SyncMove()
  MoveJ p200, v1000, z50, tool2;
  WaitSyncTask sync1, all_tasks;
  MoveL p201, v500, fine, tool2;
  SyncMoveOn sync2, all_tasks;
  MoveL p202\ID:=10, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveC p203, p204\ID:=20, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveL p205\ID:=30, v300, z10, tool2 \WObj:=wobj_stn1;
```

Continues on next page

6 Motion functions

6.2.3 Store current path

Continued

```
MoveC p206, p201\ID:=40, v300, fine, tool2 \WObj:=wobj_stn1;
SyncMoveOff sync3;
MoveL p299, v1000, fine, tool2;
ERROR
  IF ERRNO = ERR_PATH_STOP THEN
    gun_cleaning();
  ENDIF
UNDO
  SyncMoveUndo;
ENDPROC
PROC gun_cleaning()
  VAR robtargt p2;
  !Store the movement path and current position.
  StorePath \KeepSync;
  p2 := CRobT(\Tool:=tool2 \WObj:=wobj_stn1);
  !Correct the error
  MoveL pclean2 \ID:=50, v100, fine, tool2 \WObj:=wobj_stn1;
  ...
  !Move the robot back to the stored position.
  MoveL p2 \ID:=60, v100, fine, tool2 \WObj:=wobj_stn1;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
ENDMODULE
```

T_STN1 task program

```
MODULE module3
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  PERS tasks all_tasks{3};
  CONST jointtarget angle_neg20 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9,
    9E9], [ -20, 9E9, 9E9, 9E9, 9E9, 9E9] ];
  ...
  CONST jointtarget angle_340 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9],[
    340, 9E9, 9E9, 9E9, 9E9, 9E9] ];
  PROC main()
    ...
    SyncMove;
    ...
  ENDPROC
  PROC SyncMove()
    MoveExtJ angle_neg20, vrot50, fine;
    WaitSyncTask sync1, all_tasks;
    ! Wait for the robots
    SyncMoveOn sync2, all_tasks;
    MoveExtJ angle_20\ID:=10, vrot100, z10;
    MoveExtJ angle_160\ID:=20, vrot100, z10;
    MoveExtJ angle_200\ID:=30, vrot100, z10;
```

Continues on next page

```

MoveExtJ angle_340\ID:=40, vrot100, fine;
SyncMoveOff sync3;
ERROR
  IF ERRNO = ERR_PATH_STOP THEN
    gun_cleaning();
  ENDIF
UNDO
  SyncMoveUndo;
ENDPROC
PROC gun_cleaning()
  VAR jointtarget resume_angle;
  !Store the movement path and current angle.
  StorePath \KeepSync;
  resume_angle := CJointT();
  !Correct the error
  MoveExtJ clean_angle \ID:=50, vrot100, fine;
  ...
  !Move the robot back to the stored position.
  MoveExtJ resume_angle \ID:=60, vrot100, fine;
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
ENDMODULE

```

Suspend and resume synchronized movements in the “SyncArc” example

SyncMoveSuspend is used to suspend synchronized movements mode and set the system to independent or semi coordinated movement mode.

SyncMoveResume is used to go back once more to synchronized movements.

These instructions can only be used after StorePath\KeepSync has been executed.

T_ROB1

```

PROC gun_cleaning()
  VAR robtarget p1;
  !Store the movement path and current position
  ! and keep synchronized mode.
  StorePath \KeepSync;
  p1 := CRobT(\Tool:=tool1 \WObj:=wobj_stn1);
  !Move in synchronized motion mode
  MoveL p104 \ID:=50, v100, fine, tool1 \WObj:=wobj_stn1;
  SyncMoveSuspend;
  !Move in independent mode
  MoveL pclean1, v100, fine, tool1;
  ...
  !Move the robot back to the stored position
  SyncMoveResume;
  MoveL p1 \ID:=60, v100, fine, tool1 \WObj:=wobj_stn1;
  !Restore the path and start the movement

```

Continues on next page

6 Motion functions

6.2.3 Store current path

Continued

```
RestoPath;  
StartMove;  
RETRY;  
ENDPROC
```

T_ROB2

```
PROC gun_cleaning()  
  VAR robtarget p2;  
  !Store the movement path and current position.  
  StorePath \KeepSync;  
  p2 := CRobT(\Tool:=tool2 \WObj:=wobj_stn1);  
  !Move in synchronized motion mode  
  MoveL p104 \ID:=50, v100, fine, tool2 \WObj:=wobj_stn1;  
  SyncMoveSuspend;  
  !Move in independent mode  
  MoveL pclean2 v100, fine, tool2;  
  ...  
  !Move the robot back to the stored position.  
  SyncMoveResume;  
  !Move in synchronized motion mode  
  MoveL p2 \ID:=60, v100, fine, tool2 \WObj:=wobj_stn1;  
  !Restore the path and start the movement  
  RestoPath;  
  StartMove;  
  RETRY;  
ENDPROC
```

T_STN1

```
PROC gun_cleaning()  
  VAR jointtarget resume_angle;  
  !Store the movement path and current angle.  
  StorePath \KeepSync;  
  resume_angle := CJointT();  
  !Move in synchronized motion mode  
  MoveExtJ plclean_angle \ID:=50, vrot100, fine;  
  SyncMoveSuspend;  
  ! Move in independent mode  
  MoveExtJ p2clean_angle,vrot, fine;  
  ...  
  !Move the robot back to the stored position.  
  SyncMoveResume;  
  ! Move in synchronized motion mode  
  MoveExtJ resume_angle \ID:=60, vrot100, fine;  
  !Restore the path and start the movement  
  RestoPath;  
  StartMove;  
  RETRY;  
ENDPROC
```

6.2.4 Path recorder

What is the path recorder

The path recorder can memorize a number of move instructions. This memory can then be used to move the robot backwards along that same path.

How to use the path recorder

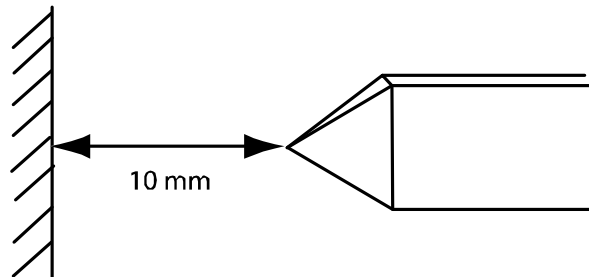
This is the general approach for using the path recorder:

- 1 Start the path recorder
- 2 Move the robot with regular move, or process, instructions
- 3 Store the current path
- 4 Move backwards along the recorded path
- 5 Resolve the error
- 6 Move forward along the recorded path
- 7 Restore the interrupted path

Lift the tool

When the robot moves backward in its own track, you may want to avoid scraping the tool against the work piece. For a process like arc welding, you want to stay clear of the welding seam.

By using the argument `ToolOffs` in the instructions `PathRecMoveBwd` and `PathRecMoveFwd`, you can set an offset for the TCP. This offset is set in tool coordinates, which means that if it is set to `[0,0,10]` the tool will be 10mm from the work object when it moves back along the recorded path.



xx0400000828



Note

When a MultiMove system is in synchronized mode all tasks must use `ToolOffs` if a tool is going to be lifted.

However if you only want to lift one tool, set `ToolOffs=[0,0,0]` in the other tasks.

Simple example

If an error occurs between p1 and p4, the robot will return to p1 where the error can be resolved. When the error has been resolved, the robot continues from where the error occurred.

Continues on next page

6 Motion functions

6.2.4 Path recorder

Continued

When p4 is reached without any error, the path recorder is switched off. The robot then moves from p4 to p5 without the path recorder.

```
...
VAR pathrecid start_id;
...
MoveL p1, vmax, fine, tool1;
PathRecStart start_id;
MoveL p2, vmax, z50, tool1;
MoveL p3, vmax, z50, tool1;
MoveL p4, vmax, fine, tool1;
PathRecStop \Clear;
MoveL p5, vmax, fine, tool1;

ERROR
  StorePath;
  PathRecMoveBwd;
  ! Fix the problem
  PathRecMoveFwd;
  RestoPath;
  StartMove;
  RETRY;
ENDIF
...
```

Complex example

In this example, the path recorder is used for two purposes:

- If an error occurs, the operator can choose to back up to p1 or to p2. When the error has been resolved, the interrupted movement is resumed.
- Even if no error occurs, the path recorder is used to move the robot from p4 to p1. This technique is useful when the robot is in a narrow position that is difficult to move out of.

Note that if an error occurs during the first move instruction, between p1 and p2, it is not possible to go backwards to p2. If the operator choose to go back to p2, PathRecValidBwd is used to see if it is possible. Before the robot is moved forward to the position where it was interrupted, PathRecValidFwd is used to see if it is possible (if the robot never backed up it is already in position).

```
...
VAR pathrecid origin_id;
VAR pathrecid corner_id;
VAR num choice;
...
MoveJ p1, vmax, z50, tool1;
PathRecStart origin_id;
MoveJ p2, vmax, z50, tool1;
PathRecStart corner_id;
MoveL p3, vmax, z50, tool1;
MoveL p4, vmax, fine, tool1;

! Use path record to move safely to p1
```

Continues on next page


```

StorePath;
PathRecMoveBwd \ID:=origin_id
    \ToolOffs:=[0,0,10];
RestoPath;
PathRecStop \Clear;
Clear Path;
Start Move;

ERROR
StorePath;

! Ask operator how far to back up
TPReadFK choice,"Extract to:", stEmpty, stEmpty,
    stEmpty, "Origin", "Corner";

IF choice=4 THEN
    ! Back up to p1
    PathRecMoveBwd \ID:=origin_id
        \ToolOffs:=[0,0,10];
ELSEIF choice=5 THEN
    ! Verify that it is possible to back to p2,
    IF PathRecValidBwd(\ID:=corner_id) THEN
        ! Back up to p2
        PathRecMoveBwd \ID:=corner_id
            \ToolOffs:=[0,0,10];
    ENDIF
ENDIF

! Fix the problem

! Verify that there is a path record forward
IF PathRecValidFwd() THEN
    ! Return to where the path was interrupted
    PathRecMoveFwd \ToolOffs:=[0,0,10];
ENDIF

! Restore the path and resume movement
RestoPath;
StartMove;
RETRY;
...

```

Resume path recorder

If the path recorder is stopped, it can be started again from the same position without losing its history.

In the example below, the `PathRecMoveBwd` instruction will back the robot to p1. If the robot had been in any other position than p2 when the path recorder was restarted, this would not have been possible.

Continues on next page

6 Motion functions

6.2.4 Path recorder

Continued

For more information, see the section about `PathRecStop` in *Technical reference manual - RAPID Instructions, Functions and Data types*.

```
...
MoveL p1, vmax, z50, tool1;
PathRecStart id1;
MoveL p2, vmax, z50, tool1;
PathRecStop;
MoveL p3, vmax, z50, tool1;
MoveL p4, vmax, z50, tool1;
MoveL p2, vmax, z50, tool1;
PathRecStart id2;
MoveL p5, vmax, z50, tool1;
StorePath;
PathRecMoveBwd \ID:=id1;
RestoPath;
...
```

"SyncArc" example with coordinated synchronized movement

This is an example on how to use Path Recorder in error handling for a MultiMove system.

In this example two robots perform arc welding on the same work piece. To make the example simple and general, we use move instructions instead of weld instructions. The work object is rotated by a positioner.

For more information on the SyncArc example, see *Application manual - MultiMove*.

T_ROB1 task program

```
MODULE module1
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  PERS tasks all_tasks{3} := [{"T_ROB1"}, {"T_ROB2"}, {"T_STN1"}];
  PERS wobjdata wobj_stn1 := [ FALSE, FALSE, "STN_1", [ [0, 0, 0],
    [1, 0, 0, 0] ], [ [0, 0, 250], [1, 0, 0, 0] ] ];
  TASK PERS tooldata tool1 := ...
  CONST robtarg p100 := ...
  CONST robtarg p199 := ...
  PROC main()
    ...
    SyncMove;
  ENDPROC

  PROC SyncMove()
    WaitSyncTask sync1, all_tasks;
    MoveJ p100, v1000, z50, tool1;
    ! Start recording
    PathRecStart HomeROB1;
    MoveL p101, v500, fine, tool1;
    SyncMoveOn sync2, all_tasks;
    MoveL p102\ID:=10, v300, z10, tool1 \WObj:=wobj_stn1;
    MoveC p103, p104\ID:=20, v300, z10, tool1 \WObj:=wobj_stn1;
```

Continues on next page

```

MoveL p105\ID:=30, v300, z10, tool1 \WObj:=wobj_stn1;
MoveC p106, p101\ID:=40, v300, fine, tool1 \WObj:=wobj_stn1;
!Stop recording
PathRecStop \Clear;
SyncMoveOff sync3;
MoveL p199, v1000, fine, tool1;
ERROR
! Weld error in this program task
IF ERRNO = AW_WELD_ERR THEN
    gun_cleaning();
ENDIF
UNDO
    SyncMoveUndo;
ENDPROC
PROC gun_cleaning()
    VAR robtarget p1;
    !Store the movement path
    IF IsSyncMoveOn() THEN
        StorePath \KeepSync;
    ELSE
        StorePath;
    ENDIF
    !Move this robot backward to p100.
    PathRecMoveBwd \ID:=HomeROB1 \ToolOffs:=[0,0,10];
    !Correct the error
    MoveJ pclean1 ,v100, fine, tool1;
    ...
    !Move the robot back to p100
    MoveJ p100, v100, fine, tool1;
    PathRecMoveFwd \ToolOffs:=[0,0,10];
    !Restore the path and start the movement
    RestoPath;
    StartMove;
    RETRY;
ENDPROC
ENDMODULE

```

T_ROB2 task program

```

MODULE module2
    VAR syncident sync1;
    VAR syncident sync2;
    VAR syncident sync3;
    PERS tasks all_tasks{3};
    PERS wobjdata wobj_stn1;
    TASK PERS tooldata tool2 := ...
    CONST robtarget p200 := ...
    CONST robtarget p299 := ...
    PROC main()
        ...
        SyncMove;
    ENDPROC

```

Continues on next page

6 Motion functions

6.2.4 Path recorder

Continued

```
PROC SyncMove()
  WaitSyncTask sync1, all_tasks;
  MoveJ p200, v1000, z50, tool2;
  PathRecStart HomeROB2;
  MoveL p201, v500, fine, tool2;
  SyncMoveOn sync2, all_tasks;
  MoveL p202\ID:=10, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveC p203, p204\ID:=20, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveL p205\ID:=30, v300, z10, tool2 \WObj:=wobj_stn1;
  MoveC p206, p201\ID:=40, v300, fine, tool2 \WObj:=wobj_stn1;
  PathRecStop \Clear;
  SyncMoveOff sync3;
  MoveL p299, v1000, fine, tool2;
ERROR
  IF ERRNO = ERR_PATH_STOP THEN
    gun_move_out();
  ENDIF
UNDO
  SyncMoveUndo;
ENDPROC
PROC gun_move_out()
  IF IsSyncMoveOn() THEN
    StorePath \KeepSync;
  ELSE
    StorePath;
  ENDIF
  ! Move this robot backward to p201
  PathRecMoveBwd \ToolOffs:=[0,0,10];
  ! Wait for the other gun to get clean
  PathRecMoveFwd \ToolOffs:=[0,0,10];
  !Restore the path and start the movement
  RestoPath;
  StartMove;
  RETRY;
ENDPROC
ENDMODULE
```

T_STN1 task program

```
MODULE module3
  VAR syncident sync1;
  VAR syncident sync2;
  VAR syncident sync3;
  PERS tasks all_tasks{3};
  CONST jointtarget angle_neg20 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9,
    9E9], [ -20, 9E9, 9E9, 9E9, 9E9, 9E9] ];
  ...
  CONST jointtarget angle_340 :=[ [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9],[
    340, 9E9, 9E9, 9E9,9E9, 9E9] ];
  PROC main()
    ...
    SyncMove;
```

Continues on next page

```

...
ENDPROC
PROC SyncMove()
    WaitSyncTask sync1, all_tasks;
    MoveExtJ angle_neg20, vrot50, fine;
    PathRecStart HomeSTN1;
    SyncMoveOn sync2, all_tasks;
    MoveExtJ angle_20\ID:=10, vrot100, z10;
    MoveExtJ angle_160\ID:=20, vrot100, z10;
    MoveExtJ angle_200\ID:=30, vrot100, z10;
    MoveExtJ angle_340\ID:=40, vrot100, fine;
    PathRecStop \Clear;
    SyncMoveOff sync3;
ERROR
    IF ERRNO = ERR_PATH_STOP THEN
        gun_move_out();
    ENDIF
UNDO
    SyncMoveUndo;
ENDPROC
PROC gun_move_out()
    !Store the movement
    IF IsSyncMoveOn() THEN
        StorePath \KeepSync;
    ELSE
        StorePath;
    ENDIF
    !Move the manipulator backward to angle_neg 20
    PathRecMoveBwd \ToolOffs:=[0,0,0];
    ...
    !Wait for the gun to get clean
    PathRecMoveFwd \ToolOffs:=[0,0,0];
    RestoPath;
    StartMove;
    RETRY;
ENDPROC

```

6 Motion functions

6.3.1 Overview

6.3 Path Offset [612-1]

6.3.1 Overview

Purpose

The purpose of Path Offset is to be able to make online adjustments of the robot path according to input from sensors. With the set of instructions that Path Offset offers, the robot path can be compared and adjusted with the input from sensors.

What is included

The RobotWare option Path Offset gives you access to:

- the data type `corrdescr`
- the instructions `CorrCon`, `CorrDiscon`, `CorrClear` and `CorrWrite`
- the function `CorrRead`

Basic approach

This is the general approach for setting up Path Offset. For a detailed example of how this is done, see [Code example on page 225](#).

- 1 Declare the correction generator.
- 2 Connect the correction generator.
- 3 Define a trap routine that determines the offset and writes it to the correction generator.
- 4 Define an interrupt to frequently call the trap routine.
- 5 Call a move instruction using the correction. The path will be repeatedly corrected.



Note

If two or more move instructions are called after each other with the `\Corr` switch it is important to know that all `\Corr` offsets are reset each time the robot starts from a finepoint. So, when using finepoints, on the second move instruction the controller does not know that the path already has an offset. To avoid any strange behavior it is recommended only to use zones together with the `\Corr` switch and avoid finepoints.

Limitations

It is possible to connect several correction generators at the same time (for instance one for corrections along the Z axis and one for corrections along the Y axis). However, it is not possible to connect more than 5 correction generators at the same time.

After a controller restart, the correction generators have to be defined once again. The definitions and connections do not survive a controller restart.

The instructions can only be used in motion tasks.

6.3.2 RAPID components

Data types

This is a brief description of each data type in Path Offset. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
corrdescr	<code>corrdescr</code> is a correction generator descriptor that is used as the reference to the correction generator.

Instructions

This is a brief description of each instruction in Path Offset. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
CorrCon	<code>CorrCon</code> activates path correction. Calling <code>CorrCon</code> will connect a correction generator. Once this connection is made, the path can be continuously corrected with new offset inputs (for instance from a sensor).
CorrDiscon	<code>CorrDiscon</code> deactivates path correction. Calling <code>CorrDiscon</code> will disconnect a correction generator.
CorrClear	<code>CorrClear</code> deactivate path correction. Calling <code>CorrClear</code> will disconnect all correction generators.
CorrWrite	<code>CorrWrite</code> sets the path correction values. Calling <code>CorrWrite</code> will set the offset values to a correction generator.

Functions

This is a brief description of each function in Path Offset. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
CorrRead	<code>CorrRead</code> reads the total correction made by a correction generator.

6.3.3 Related RAPID functionality

The argument \Corr

The optional argument \Corr can be set for some move instructions. This will enable path corrections while the move instruction is executed.

The following instructions have the optional argument \Corr:

- MoveL
- MoveC
- SearchL
- SearchC
- TriggL (only if the controller is equipped with the base functionality Fixed Position Events)
- TriggC (only if the controller is equipped with the base functionality Fixed Position Events)
- CapL (only if the controller is equipped with the option Continuous Application Platform)
- CapC (only if the controller is equipped with the option Continuous Application Platform)
- ArcL (only if the controller is equipped with the option RobotWare Arc)
- ArcC (only if the controller is equipped with the option RobotWare Arc)

For more information on these instructions, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Interrupts

To create programs using Path Offset, you need to be able to handle interrupts. For more information on interrupts, see *Technical reference manual - RAPID overview*.

6.3.4 Code example

Linear movement with correction

This is a simple example of how to program a linear path with online path correction. This is done by having an interrupt 5 times per second, calling a trap routine which makes the offset correction.

Program code

```
VAR intnum int_nol;
VAR corrdescr id;
VAR pos sens_val;
PROC PathRoutine()
  !Connect to the correction generator
  CorrCon id;

  !Setup a 5 Hz timer interrupt.
  CONNECT int_nol WITH UpdateCorr;
  ITimer\Single, 0.2, int_nol

  !Position for start of contour tracking
  MoveJ p10,v100,z10,tool1;

  !Run MoveL with correction.
  MoveL p20,v100,z10,tool1\Corr;

  !Remove the correction generator.
  CorrDiscon id;

  !Remove the timer interrupt.
  IDelete int_nol;
ENDPROC
TRAP UpdateCorr
  !Call a routine that read the sensor
  ReadSensor sens_val.x, sens_val.y, sens_val.z;

  !Execute correction
  CorrWrite id, sens_val;

  !Setup interrupt again
  IDelete int_nol;
  CONNECT int_nol WITH UpdateCorr;
  ITimer\Single, 0.2, int_nol;
ENDTRAP
```

This page is intentionally left blank

7 Motion Supervision

7.1 Collision Detection [613-1]

7.1.1 Overview

Purpose

Collision Detection is a software option that reduces collision impact forces on the robot. This helps protecting the robot and external equipment from severe damage.



WARNING

Collision Detection cannot protect equipment from damage at a full speed collision.

Description

The software option Collision Detection identifies a collision by high sensitivity, model based supervision of the robot. Depending on what forces you deliberately apply on the robot, the sensitivity can be tuned as well as turned on and off. Because the forces on the robot can vary during program execution, the sensitivity can be set on-line in the program code.

Collision detection is more sensitive than the ordinary supervision and has extra features. When a collision is detected, the robot will immediately stop and relieve the residual forces by moving in reversed direction a short distance along its path. After a collision error message has been acknowledged, the movement can continue without having to press **Motors on** on the controller.

What is included

The RobotWare option Collision Detection gives you access to:

- system parameters for defining if Collision Detection should be active and how sensitive it should be (without the option you can only turn detection on and off for Auto mode)
- instruction for on-line changes of the sensitivity: `MotionSup`

Basic approach

Collision Detection is by default always active when the robot is moving. In many cases this means that you can use Collision Detection without having to take any active measures.

If necessary, you can turn Collision Detection on and off or change its sensitivity in two ways:

- temporary changes can be made on-line with the RAPID instruction `MotionSup`
- permanent changes are made through the system parameters.

7 Motion Supervision

7.1.2 Limitations

7.1.2 Limitations

Load definition

In order to detect collisions properly, the payload of the robot must be correctly defined.



Tip

Use Load Identification to define the payload. For more information, see *Operating manual - IRC5 with FlexPendant*.

Robot axes only

Collision Detection is only available for the robot axes. It is not available for track motions, orbit stations, or any other external axes.

Independent joint

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

Soft servo

The collision detection may trigger without a collision when the robot is used in soft servo mode. Therefore, it is recommended to turn the collision detection off when the robot is in soft servo mode.

No change until the robot moves

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 have an unexpected value at program start before the robot starts to move.

Reversed movement distance

The distance the robot is reversed after a collision is proportional to the speed of the motion before the collision. If repeated low speed collisions occur, the robot may not be reversed 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, turn Collision Detection off temporarily and jog the robot away from the obstacle.

Delay before reversed movement

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

Robot on track motion

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

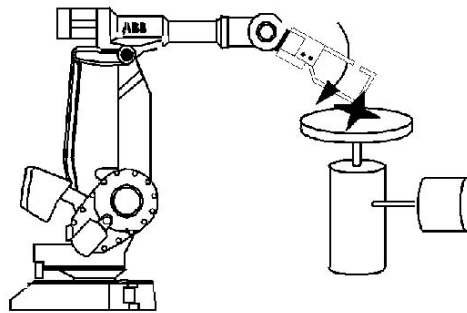
7.1.3 What happens at a collision

Overview

When the collision detection is triggered, the robot will stop as quickly as possible. Then it will move in the reverse direction to remove residual forces. The program execution will stop with an error message. The robot remains in the state *motors on* so that program execution can be resumed after the collision error message has been acknowledged.

A typical collision is illustrated below.

Collision illustration



xx0300000361

Robot behavior after a collision

This list shows the order of events after a collision. For an illustration of the sequence, see the diagram below.

When ...	then ...
the collision is detected	the motor torques are reversed and the mechanical brakes applied in order to stop the robot
the robot has stopped	the robot moves in reversed direction a short distance along the path in order to remove any residual forces which may be present if a collision or jam occurred
the residual forces are removed	the robot stops again and remains in the <i>motors on</i> state

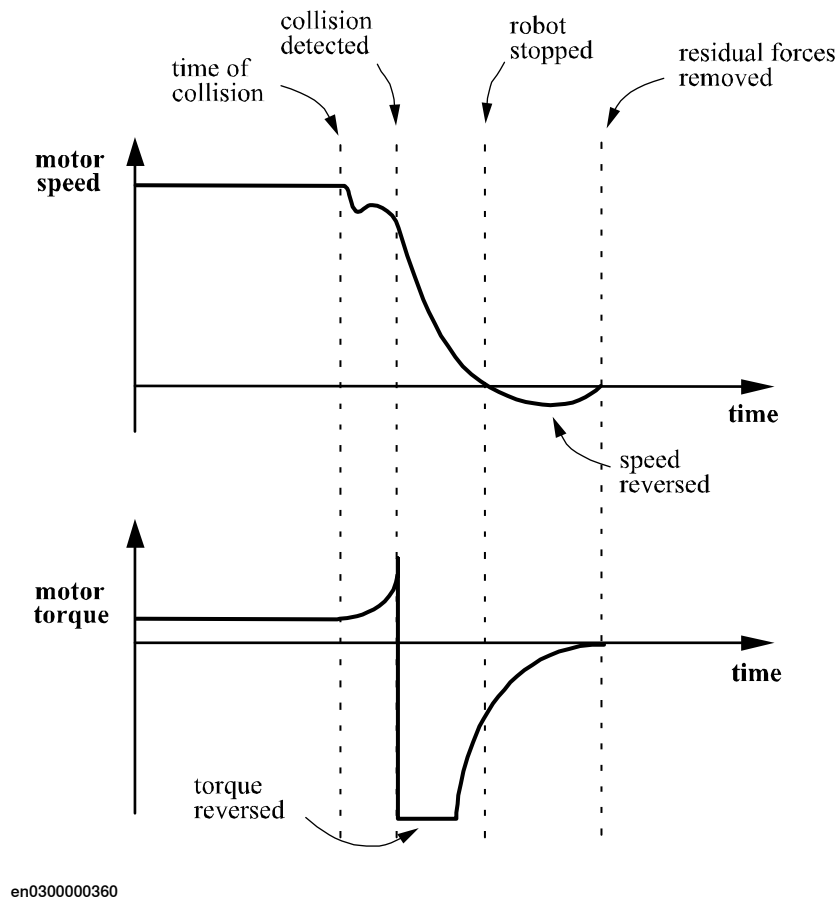
Continues on next page

7 Motion Supervision

7.1.3 What happens at a collision

Continued

Speed and torque diagram



7.1.4 Additional information

Motion error handling

For more information regarding error handling for a collision, see *Technical reference manual - RAPID kernel*.

7 Motion Supervision

7.1.5.1 System parameters

7.1.5 Configuration and programming facilities

7.1.5.1 System parameters

About system parameters

The parameters for Collision Detection do **not** require a restart to take effect.

For more information about the parameters, see *Technical reference manual - System parameters*.

Motion Supervision

These parameters belong to the type *Motion Supervision* in the topic *Motion*.

Parameter	Description
Path Collision Detection	Turn the collision detection On or Off for program execution. <i>Path Collision Detection</i> is by default set to On.
Jog Collision Detection	Turn the collision detection On or Off for jogging. <i>Jog Collision Detection</i> is by default set to On.
Path Collision Detection Level	Modifies the Collision Detection supervision level for program execution by the specified percentage value. A large percentage value makes the function less sensitive. <i>Path Collision Detection Level</i> is by default set to 100%.
Jog Collision Detection Level	Modifies the Collision Detection supervision level for jogging by the specified percentage value. A large percentage value makes the function less sensitive. <i>Jog Collision Detection Level</i> is by default set to 100%.
Collision Detection Memory	Defines how much the robot moves in reversed direction on the path after a collision, specified in seconds. If the robot moved fast before the collision it will move away a larger distance than if the speed was slow. <i>Collision Detection Memory</i> is by default set to 75 ms.
Manipulator Supervision	Turns the supervision for the loose arm detection on or off for IRB 340 and IRB 360. A loose arm will stop the robot and cause an error message. <i>Manipulator Supervision</i> is by default set to On.
Manipulator Supervision Level	Modifies the supervision level for the loose arm detection for the manipulators IRB 340 and IRB 360. A large value makes the function less sensitive. <i>Manipulator Supervision Level</i> is by default value set to 100%.

Motion Planner

These parameters belong to the type *Motion Planner* in the topic *Motion*.

Parameter	Description
Motion Supervision Max Level	Set the maximum level to which the total collision detection tune level can be changed. It is by default set to 300%.

Continues on next page

General RAPID

These parameters belong to the type *General RAPID* in the topic *Controller*.

Parameter	Description
Collision Error Handler	Enables RAPID error handling for collision. <i>Collision Error Handler</i> is default set to Off. For more information regarding error handling for a collision, see <i>Technical reference manual - RAPID kernel</i>

7 Motion Supervision

7.1.5.2 RAPID components

7.1.5.2 RAPID components

Instructions

This is a brief description of the instructions in Collision Detection. For more information, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
MotionSup	<p>MotionSup is used to:</p> <ul style="list-style-type: none">• activate or deactivate Collision Detection. This can only be done if the parameter <i>Path Collision Detection</i> is set to On.• modify the supervision level with a specified percentage value (1-300%). A large percentage value makes the function less sensitive.

7.1.5.3 Signals

Digital outputs

This is a brief description of the digital outputs in Collision Detection. For more information, see respective digital output in *Technical reference manual - System parameters*.

Digital output	Description
MotSupOn	<p><i>MotSupOn</i> is high when Collision Detection is active and low when it is not active.</p> <p>Note that a change in the state takes effect when a motion starts. Thus, if Collision Detection is active and the robot is moving, <i>MotSupOn</i> is high. If the robot is stopped and Collision Detection turned off, <i>MotSupOn</i> is still high. When the robot starts to move, <i>MotSupOn</i> switches to low.</p>
MotSupTrigg	<p><i>MotSupTrigg</i> goes high when the collision detection triggers. It stays high until the error code is acknowledged from the FlexPendant.</p>

7 Motion Supervision

7.1.6.1 Set up system parameters

7.1.6 How to use Collision Detection

7.1.6.1 Set up system parameters

Activate supervision

To be able to use Collision Detection during program execution, the parameter *Path Collision Detection* must be set to *On*.

To be able to use Collision Detection during jogging, the parameter *Jog Collision Detection* must be set to *On*.

Define supervision levels

Set the parameter *Path Collision Detection Level* to the percentage value you want as default during program execution.

Set the parameter *Jog Collision Detection Level* to the percentage value you want as default during jogging.

7.1.6.2 Adjust supervision from FlexPendant

Speed adjusted supervision level

Collision Detection uses a variable supervision level. At low speeds it is more sensitive than at 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. These parameters are described in [System parameters on page 232](#).

Set jog supervision on FlexPendant

On the FlexPendant, select **Control Panel** from the **ABB** menu and then tap **Supervision**.

Supervision can be turned on or off and the sensitivity can be adjusted for both programmed paths and jogging. The sensitivity level is set in percentage. A large value makes the function less sensitive.

If the motion supervision for jogging is turned off in the dialog box and a program is executed, Collision Detection can still be active during execution of the program.



Note

The supervision settings correspond to system parameters of the type *Motion Supervision*. These can be set using the supervision settings on the FlexPendant, as described above. They can also be changed using RobotStudio or FlexPendant configuration editor or Quickset Mechanical unit menu.

7 Motion Supervision

7.1.6.3 Adjust supervision from RAPID program

7.1.6.3 Adjust supervision from RAPID program

Default values

If Collision Detection is activated with the system parameters, it is by default active during program execution with the tune value 100%. These values are set automatically:

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



Note

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%.

Temporarily deactivate supervision

If external forces will affect the robot during a part of the program execution, temporarily deactivate the supervision with the following instruction:

```
MotionSup \Off;
```

Reactivate supervision

If the supervision has been temporarily deactivated, it can be activated with the following instruction:

```
MotionSup \On;
```



Note

If the supervision is deactivated with the system parameters, it cannot be activated with RAPID instructions.

Tuning

The supervision level can be tuned during program execution with the instruction *MotionSup*. The tune values are set in percent of the basic tuning where 100% corresponds to the basic values. A higher percentage gives a less sensitive system.

This is an example of an instruction that increase the supervision level to 200%:

```
MotionSup \On \TuneValue:=200;
```

7.1.6.4 How to avoid false triggering

About false triggering

Because the supervision is designed to be very sensitive, it may trigger if the load data is incorrect or if there are large process forces acting on the robot.

Actions to take

If ...	then ...
the payload is incorrectly defined	use Load Identification to define it. For more information, see <i>Operating manual - IRC5 with FlexPendant</i> .
the payload has large mass or inertia	increase supervision level
the arm load (cables or similar) cause trigger	manually define the arm load or increase supervision level
the application involves many external process forces	increase the supervision level for jogging and program execution in steps of 30 percent until you no longer receive the error code.
the external process forces are only temporary	use the instruction <code>MotionSup</code> to raise the supervision level or turn the function off temporarily.
everything else fails	turn off Collision Detection.

This page is intentionally left blank

8 Communication

8.1 FTP Client [614-1]

8.1.1 Introduction to FTP Client

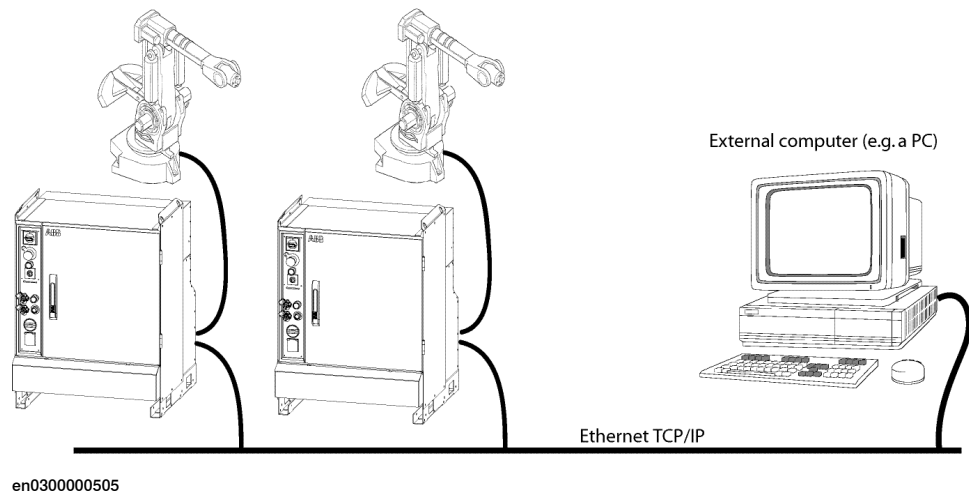
Purpose

The purpose of FTP Client is to enable the robot to access remote mounted disks, for example a hard disk drive on a PC.

Here are some examples of applications:

- Backup to a remote computer.
- Load programs from a remote computer.

Network illustration



Description

Several robots can access the same computer over an Ethernet network.

Once the FTP application protocol is configured, the remote computer can be accessed in the same way as the controller's internal hard disk.

What is included

The RobotWare option FTP Client gives you access to the system parameter type *Application protocol* and its parameters: *Name*, *Type*, *Transmission protocol*, *Server address*, *Trusted*, *Local path*, *Server path*, *Username*, and *Password*

Basic approach

This is the general approach for using FTP Client. For more detailed examples of how this is done, see [Examples on page 244](#).

- 1 Configure an *Application protocol* to point out a disk or directory on a remote computer that will be accessible from the robot.

Continues on next page

8 Communication

8.1.1 Introduction to FTP Client

Continued

- 2 Read and write to the remote computer in the same way as with the controller's internal hard disk.

Requirements

The external computer must have:

- TCP/IP stack
- FTP Server

Limitations

When using the FTP Client the maximum length for a file name is 99 characters.

When using the FTP Client the maximum length for a file path including the file name is 200 characters. The whole path is included in the 200 characters, not only the server path. When ordering a backup towards a mounted disk all the directories created by the backup has to be included in the max path.

Example

Parameter	Value
Local path	pc:
Server path	C:\robot_1

- A backup is saved to pc:/Backups/Backup_20130109
(27 characters)
- The path on the PC will be C:\robot_1\Backups\Backup_20130109
(34 characters)
- The longest file path inside this backup is
C:\robot_1\Backups\Backup_20130109\RAPID\TASK1\PROGMOD\myprogram.mod
(54+13 characters)

The maximum path length for this example first looks like 27 characters but is actually 67 characters.

8.1.2 System parameters

Application protocol

This is a brief description of the parameters used to configure an application protocol. For more information, see the respective parameter in [System parameters on page 243](#).

These parameters belongs to the type *Application protocol* in the topic *Communication*.

Parameter	Description
Name	Name of the application protocol.
Type	Type of application protocol. Set this to "FTP".
Transmission protocol	Name of the transmission protocol the protocol should use. For FTP, this is always "TCP/IP".
Server address	The IP address of the computer with the FTP server.
Trusted	This flag decides if this computer should be trusted, i.e. if losing the connection should make the program stop.
Local path	Defines what the shared unit will be called on the robot. The parameter value must end with a colon (:). If, for example the unit is named "pc:", the name of the test.prg on this unit would be pc:test.prg
Server path	The name of the disk or folder to connect to, on the remote computer. If not specified, the application protocol will reference the directory that is shared by the FTP server. Note: The exported path should not be specified if communicating with an FTP server of type Distinct FTP, FileZilla or MS IIS.
Username	The user name used by the robot when it logs on to the remote computer. The user account must be set up on the FTP server.
Password	The password used by the robot when it logs on to the remote computer. Note that the password written here will be visible to all who has access to the system parameters.

Transmission protocol

There is a configured transmission protocol called TCP/IP, but no changes can be made to it. This is used by the FTP application protocol.

8 Communication

8.1.3 Examples

8.1.3 Examples

Example configuration

This is an example of how an application protocol can be configured for FTP.

Parameter	Value
Name	my FTP protocol
Type	FTP
Transmission protocol	TCPIP1
Server address	100.100.100.100
Trusted	No
Local path	pc:
Server path	C:\robot_1
Username	Robot1
Password	robot1

Note: The value of *Server path* should exclude the exported path if communicating with an FTP server of type Distinct FTP, FileZilla or MS IIS.

Example with FlexPendant

This example shows how to use the FlexPendant to make a backup to the remote PC. We assume that the configuration is done according to the example configuration shown above.

- 1 Tap **ABB** and select **Backup and Restore**.
- 2 Tap on **Backup Current System**.
- 3 Save the backup to pc:/Backup/Backup_20031008 (the path on the PC will be C:\robot_1\Backup\Backup_20031008).

Example with RAPID code

This example shows how to open the file C:\robot_1\files\file1.doc on the remote PC from a RAPID program on the controller. We assume that the configuration is done according to the example configuration shown above.

```
Open "HOME:" \File:= "pc:/files/file1.doc", file;
```

8.2 NFS Client [614-1]

8.2.1 Introduction to NFS Client

Purpose

The purpose of NFS Client is to enable the robot to access remote mounted disks, for example a hard disk drive on a PC.

Here are some examples of applications:

- Backup to a remote computer.
- Load programs from a remote computer.

Description

Several robots can access the same computer over an Ethernet network.

Once the NFS application protocol is configured, the remote computer can be accessed in the same way as the controller's internal hard disk.

What is included

The RobotWare option NFS Client gives you access to the system parameter type *Application protocol* and its parameters: *Name*, *Type*, *Transmission protocol*, *Server address*, *Trusted*, *Local path*, *Server path*, *User ID*, and *Group ID*.

Basic approach

This is the general approach for using NFS Client. For more detailed examples of how this is done, see [Examples on page 244](#).

- 1 Configure an *Application protocol* to point out a disk or directory on a remote computer that will be accessible from the robot.
- 2 Read and write to the remote computer in the same way as with the controller's internal hard disk.

Prerequisites

The external computer must have:

- TCP/IP stack
- NFS Server

Limitations

When using the NFS Client the maximum length for a file name is 99 characters.

When using the NFS Client the maximum length for a file path including the file name is also 99 characters. The whole path is included in the 99 characters, not only the server path. When ordering a backup towards a mounted disk all the directories created by the backup has to be included in the max path.

Example

Parameter	Value
Local path	pc:

Continues on next page

8 Communication

8.2.1 Introduction to NFS Client

Continued

Parameter	Value
Server path	C:\robot_1

- A backup is saved to pc:/Backups/Backup_20130109
(27 characters)
- The path on the PC will be C:\robot_1\Backups\Backup_20130109
(34 characters)
- The longest file path inside this backup is
C:\robot_1\Backups\Backup_20130109\RAPID\TASK1\PROGMOD\myprogram.mod
(54+13 characters)

The maximum path length for this example first looks like 27 characters but is actually 67 characters.

8.2.2 System parameters

Application protocol

This is a brief description of the parameters used to configure an application protocol. For more information, see the respective parameter in [System parameters on page 247](#)

These parameters belongs to the type *Application protocol* in the topic *Communication*.

Parameter	Description
Name	Name of the application protocol.
Type	Type of application protocol. Set this to "NFS".
Transmission protocol	Name of the transmission protocol the protocol should use. For NFS, this is always "TCP/IP".
Server address	The IP address of the computer with the NFS server.
Trusted	This flag decides if this computer should be trusted, i.e. if losing the connection should make the program stop.
Local path	Defines what the shared unit will be called on the robot. The parameter value must end with a colon (:). If, for example the unit is named "pc:", the name of the test.prg on this unit would be pc:test.prg
Server path	The name of the exported disk or folder on the remote computer. For NFS, Server Path must be specified.
User ID	Used by the NFS protocol as a way of authorizing the user to access a specific server. If this parameter is not used, which is usually the case on a PC, set it to the default value 0. Note that <i>User ID</i> must be the same for all mountings on one robot controller.
Group ID	Used by the NFS protocol as a way of authorizing the user to access a specific server. If this parameter is not used, which is usually the case on a PC, set it to the default value 0. Note that <i>Group ID</i> must be the same for all mountings on one robot controller.

Transmission protocol

There is a configured transmission protocol called TCP/IP, but no changes can be made to it. This is used by the NFS application protocol.

8 Communication

8.2.3 Examples

8.2.3 Examples

Example configuration

This is an example of how an application protocol can be configured for NFS.

Parameter	Value
Name	my NFS protocol
Type	NFS
Transmission protocol	TCP/IP
Server address	100.100.100.100
Trusted	No
Local path	pc:
Server path	C:\robot_1
User ID	Robot1
Group ID	robot1

Example with FlexPendant

This example shows how to use the FlexPendant to make a backup to the remote PC. We assume that the configuration is done according to the example configuration shown above.

- 1 Tap **ABB** and select **Backup and Restore**.
- 2 Tap on **Backup Current System**.
- 3 Save the backup to pc:/Backup/Backup_20031008 (the path on the PC will be C:\robot_1\Backup\Backup_20031008).

Example with RAPID code

This example shows how to open the file C:\robot_1\files\file1.doc on the remote PC from a RAPID program on the controller. We assume that the configuration is done according to the example configuration shown above.

```
Open "HOME:" \File:= "pc:/files/file1.doc", file;
```


8.3 PC Interface [616-1]

8.3.1 Introduction to PC Interface

Purpose

PC Interface is used for communication between the controller and a PC.

The option PC Interface is required when connecting to a controller over LAN with RobotStudio.

With PC Interface, data can be sent to and from a PC. This is, for example, used for:

- Backup.
- Production statistics logging.
- Operator information presented on a PC.
- Send command to the robot from a PC operator interface.
- RobotStudio add-in that performs operations on the controller.

**Note**

If connecting over the service port, then the functionality is available without the option PC Interface.

What is included

The RobotWare option PC Interface gives you access to:

- An Ethernet communication interface, which is used by some ABB software products.

Basic approach

The general approach for using PC Interface is the same as setting up a PC SDK client application on a PC. For more information, see <http://developercenter.robotstudio.com>.

8.3.2 Send variable from RAPID

SCWrite instruction

The instruction `SCWrite` (*Superior Computer Write*) can be used to send persistent variables to a client application on a PC. For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

The PC must have a client application that can subscribe to the information that is sent to or from the controller.

Code example

In this example the robot moves objects to a position where they can be treated by a process that is controlled by the PC. When the object is ready the robot moves it to its next station.

The program uses `SCWrite` to inform the PC when the object is in position and when it has been moved to the next station. It also sends a message to the PC about how many objects that have been handled.

RAPID module for the sender

```
VAR rmqslot destination_slot;
VAR user_def

RMQFindSlot destination_slot, "RMQ_Task2";

WHILE TRUE DO
    ! Wait for next object
    WaitDI di1,1;

    ! Call first routine
    move_obj_to_pos();

    ! Send message to PC that object is in position
    user_def = 0;
    in_position:=TRUE;
    RMQSendMessage destination_slot, in_position \UserDef:=user_def;

    ! Wait for object to be ready
    WaitDI di2,1;

    ! Call second routine
    move_obj_to_next();

    ! Send message to PC that object is gone
    in_position:=FALSE;
    RMQSendMessage destination_slot, in_position \UserDef:=user_def;

    ! Inform PC how many object has been handled
    nbr_objects:= nbr_objects+1;
    user_def = 1;
```

Continues on next page

```
RMQSendMessage destination_slot, nbr_objects \UserDef:=user_def;
```

```
ENDWHILE
```

PC SDK for the receiver

```
public void ReceiveObjectPosition()
{
    const string destination_slot = "RMQ_Task2";
    IpcQueue queue = Controller.Ipc.CreateQueue(destination_slot,
        16, Ipc.MaxMessageSize);

    // Until application is closed
    while (uiclose)
    {
        IpcMessage message = new IpcMessage();
        IpcReturnType retValue = IpcReturnType.Timeout;

        retValue = queue.Receive(1000, message);
        if (IpcReturnType.OK == retValue)
        {
            string receivemessage = message.Data.ToString().ToLower();
            // if message.UserDef is 0 means Object position data else
            // number of objects
            if (message.UserDef == 0)
            {
                if (receivemessage == "true")
                {
                    // Object is in position
                }
                else
                {
                    // Object is not in position
                }
            }
            else
            {
                // number of objects in receivemessage
            }
        }
    }
}
```

8.3.3 ABB software using PC Interface

Overview

PC Interface provides a communication interface between the controller and a PC connected to an Ethernet network.

This functionality can be used by different software applications from ABB. Note that the products mentioned below are examples of applications using PC Interface, not a complete list.

RobotStudio

RobotStudio is a software product delivered with the robot. Some of the functionality requires PC Interface when connecting over the LAN port.

The following table shows some examples of RobotStudio functionality that is only available if you have PC Interface:

Functionality	Description
Event recorder	Error messages and similar events can be shown or logged on the PC.
RAPID editor	Allows on-line editing against the controller from the PC.

For more information, see *Operating manual - RobotStudio*.

8.4 Socket Messaging [616-1]

8.4.1 Introduction to Socket Messaging

Purpose

The purpose of Socket Messaging is to allow a RAPID programmer to transmit application data between computers, using the TCP/IP network protocol. A socket represents a general communication channel, independent of the network protocol being used.

Socket communication is a standard that has its origin in Berkeley Software Distribution Unix. Besides Unix, it is supported by, for example, Microsoft Windows. With Socket Messaging, a RAPID program on a robot controller can, for example, communicate with a C/C++ program on another computer.

What is included

The RobotWare option Socket Messaging gives you access to RAPID data types, instructions and functions for socket communication between computers.

Basic approach

This is the general approach for using Socket Messaging. For a more detailed example of how this is done, see [Code examples on page 258](#).

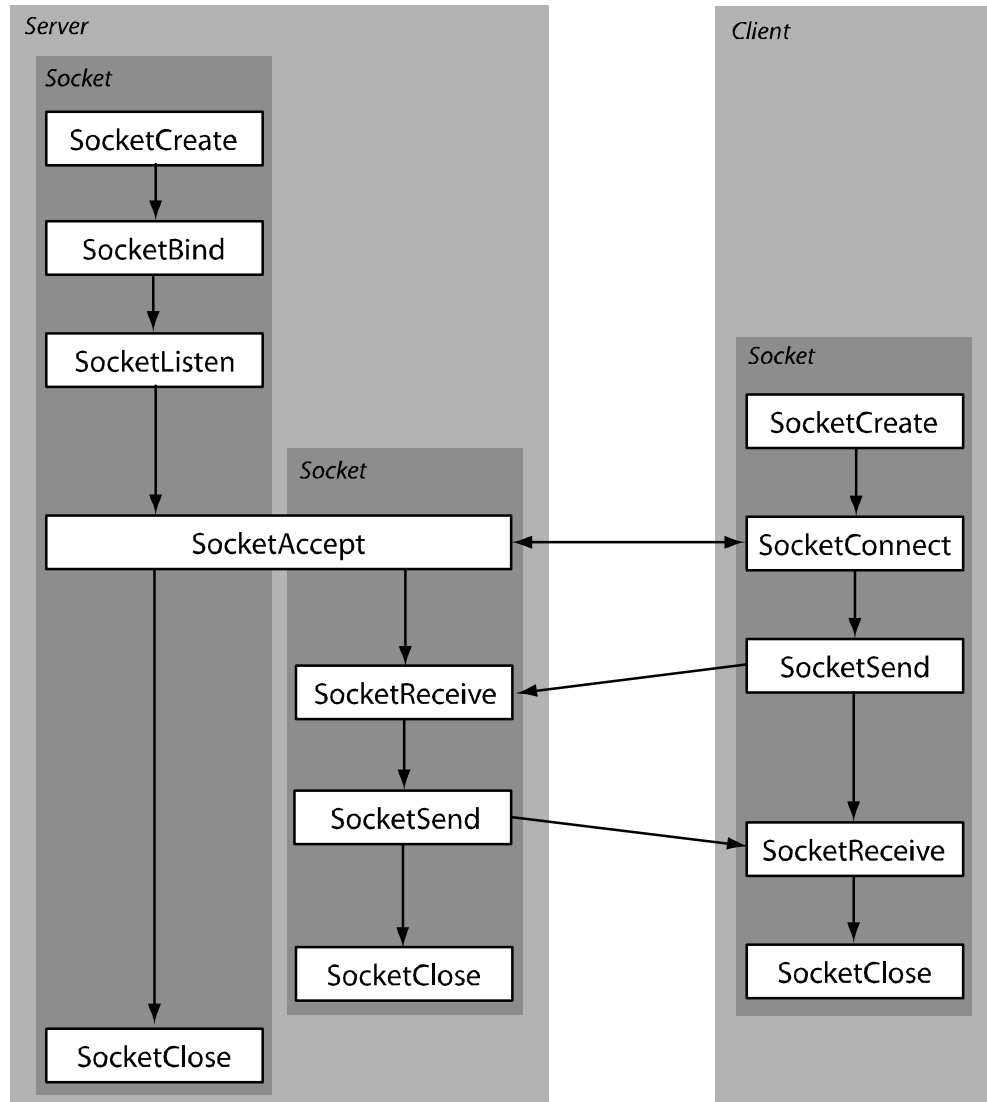
- 1 Create a socket, both on client and server. A robot controller can be either client or server.
- 2 Use `SocketBind` and `SocketListen` on the server, to prepare it for a connection request.
- 3 Order the server to accept incoming socket connection requests.
- 4 Request socket connection from the client.
- 5 Send and receive data between client and server.

8 Communication

8.4.2 Schematic picture of socket communication

8.4.2 Schematic picture of socket communication

Illustration of socket communication



en0600003224



Tip

Do not create and close sockets more than necessary. Keep the socket open until the communication is completed. The socket is not really closed until a certain time after `SocketClose` (due to TCP/IP functionality).

8.4.3 Technical facts about Socket Messaging

Overview

When using RAPID functionality Socket Messaging to communicate with a client or server that is not a RAPID task, it can be useful to know how some of the implementation is done.

No string termination

When sending a data message, no string termination sign is sent in the message. The number of bytes sent is equal to the return value of the function `strlen(str)` in the programming language C.

Unintended merge of messages

If sending two messages with no delay between the sendings, the result can be that the second message is appended to the first. The result is one big message instead of two messages. To avoid this, use acknowledge messages from the receiver of the data, if the client/server is just receiving messages.

Non printable characters

If a client that is not a RAPID task needs to receive non printable characters (binary data) in a string from a RAPID task, this can be done by RAPID as shown in the example below.

```
SocketSend socket1 \Str:="\0D\0A";
```

For more information, see *Technical reference manual - RAPID kernel*, section *String literals*.

8 Communication

8.4.4 RAPID components

8.4.4 RAPID components

Data types

This is a brief description of each data type in Socket Messaging. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
socketdev	A socket device used to communicate with other computers on a network.
socketstatus	Can contain status information from a <code>socketdev</code> variable.

Instructions for client

This is a brief description of each instruction used by the a Socket Messaging client. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
SocketCreate	Creates a new socket and assigns it to a <code>socketdev</code> variable.
SocketConnect	Makes a connection request to a remote computer. Used by the client to connect to the server.
SocketSend	Sends data via a socket connection to a remote computer. The data can be a <code>string</code> or <code>rawbytes</code> variable, or a <code>byte</code> array.
SocketReceive	Receives data and stores it in a <code>string</code> or <code>rawbytes</code> variable, or in a <code>byte</code> array.
SocketClose	Closes a socket and release all resources.



Tip

Do not use `SocketClose` directly after `SocketSend`. Wait for acknowledgement before closing the socket.

Instructions for server

A Socket Messaging server use the same instructions as the client, except for `SocketConnect`. In addition, the server use the following instructions:

Instruction	Description
SocketBind	Binds the socket to a specified port number on the server. Used by the server to define on which port (on the server) to listen for a connection. The IP address defines a physical computer and the port defines a logical channel to a program on that computer.
SocketListen	Makes the computer act as a server and accept incoming connections. It will listen for a connection on the port specified by <code>SocketBind</code> .
SocketAccept	Accepts an incoming connection request. Used by the server to accept the client's request.

Continues on next page



Note

The server application must be started before the client application, so that the instruction `SocketAccept` is executed before any client execute `SocketConnect`.

Functions

This is a brief description of each function in Socket Messaging. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
SocketGetStatus	Returns information about the last instruction performed on the socket (created, connected, bound, listening, closed). <code>SocketGetStatus</code> does not detect changes from outside RAPID (such as a broken connection).

8.4.5 Code examples

Example of client/server communication

This example shows program code for a client and a server, communicating with each other.

The server will write on the FlexPendant:

```
Client wrote - Hello server  
Client wrote - Shutdown connection
```

The client will write on its FlexPendant:

```
Server wrote - Message acknowledged  
Server wrote - Shutdown acknowledged
```

In this example, both the client and the server use RAPID programs. In reality, one of the programs would often be running on a PC (or similar computer) and be written in another program language.

Code example for client, contacting server with IP address 192.168.0.2:

```
! WaitTime to delay start of client.  
! Server application should start first.  
WaitTime 5;  
VAR socketdev socket1;  
VAR string received_string;  
PROC main()  
  SocketCreate socket1;  
  SocketConnect socket1, "192.168.0.2", 1025;  
  ! Communication  
  SocketSend socket1 \Str:="Hello server";  
  SocketReceive socket1 \Str:=received_string;  
  TPWrite "Server wrote - " + received_string;  
  received_string := "";  
  ! Continue sending and receiving  
  ...  
  ! Shutdown the connection  
  SocketSend socket1 \Str:="Shutdown connection";  
  SocketReceive socket1 \Str:=received_string;  
  TPWrite "Server wrote - " + received_string;  
  SocketClose socket1;  
ENDPROC
```

Code example for server (with IP address 192.168.0.2):

```
VAR socketdev temp_socket;  
VAR socketdev client_socket;  
VAR string received_string;  
VAR bool keep_listening := TRUE;  
PROC main()  
  SocketCreate temp_socket;  
  SocketBind temp_socket, "192.168.0.2", 1025;  
  SocketListen temp_socket;  
  WHILE keep_listening DO  
    ! Waiting for a connection request  
    SocketAccept temp_socket, client_socket;
```

Continues on next page

```

! Communication
SocketReceive client_socket \Str:=received_string;
TPWrite "Client wrote - " + received_string;
received_string := "";
SocketSend client_socket \Str:="Message acknowledged";
! Shutdown the connection
SocketReceive client_socket \Str:=received_string;
TPWrite "Client wrote - " + received_string;
SocketSend client_socket \Str:="Shutdown acknowledged";
SocketClose client_socket;
ENDWHILE
SocketClose temp_socket;
ENDPROC

```

Example of error handler

The following error handlers will take care of power failure or broken connection.

Error handler for client in previous example:

```

! Error handler to make it possible to handle power fail
ERROR
IF ERRNO=ERR_SOCKET_TIMEOUT THEN
    RETRY;
ELSEIF ERRNO=ERR_SOCKET_CLOSED THEN
    SocketClose socket1;
    ! WaitTime to delay start of client.
    ! Server application should start first.
    WaitTime 10;
    SocketCreate socket1;
    SocketConnect socket1, "192.168.0.2", 1025;
    RETRY;
ELSE
    TPWrite "ERRNO = "\Num:=ERRNO;
    Stop;
ENDIF

```

Error handler for server in previous example:

```

! Error handler for power fail and connection lost
ERROR
IF ERRNO=ERR_SOCKET_TIMEOUT THEN
    RETRY;
ELSEIF ERRNO=ERR_SOCKET_CLOSED THEN
    SocketClose temp_socket;
    SocketClose client_socket;
    SocketCreate temp_socket;
    SocketBind temp_socket, "192.168.0.2", 1025;
    SocketListen temp_socket;
    SocketAccept temp_socket, client_socket;
    RETRY;
ELSE
    TPWrite "ERRNO = "\Num:=ERRNO;
    Stop;
ENDIF

```

8 Communication

8.5.1 Introduction to RAPID Message Queue

8.5 RAPID Message Queue [included in 616-1, 623-1]

8.5.1 Introduction to RAPID Message Queue

Purpose

The purpose of RAPID Message Queue is to communicate with another RAPID task or PC application using PC SDK.

Here are some examples of applications:

- Sending data between two RAPID tasks.
- Sending data between a RAPID task and a PC application.

RAPID Message Queue can be defined for interrupt or synchronous mode. Default setting is interrupt mode.

What is included

The RAPID Message Queue functionality is included in the RobotWare options:

- PC Interface
- Multitasking

RAPID Message Queue gives you access to RAPID instructions, functions, and data types for sending and receiving data.

Basic approach

This is the general approach for using RAPID Message Queue. For a more detailed example of how this is done, see [Code examples on page 267](#).

- 1 For *interrupt* mode: The receiver sets up a trap routine that reads a message and connects an interrupt so the trap routine is called when a new message appears.

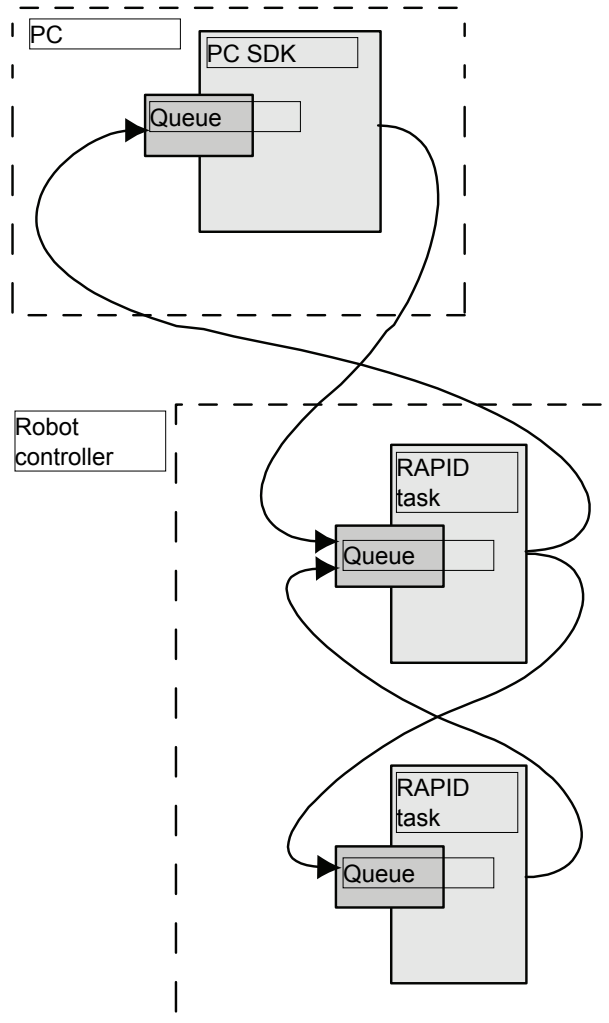
For *synchronous* mode: The message is handled by a waiting or the next executed `RMQReadWait` instruction.

- 2 The sender looks up the slot identity of the queue in the receiver task.
- 3 The sender sends the message.

8.5.2 RAPID Message Queue behavior

Illustration of communication

The picture below shows various possible senders, receivers, and queues in the system. Each arrow is an example of a way to post a message to a queue.



en0700000430

Creating a PC SDK client

This manual only describes how to use RAPID Message Queue to make a RAPID task communicate with other RAPID tasks and PC SDK clients. For information about how to set up the communication on a PC SDK client, see <http://developer-center.robotstudio.com>.

What can be sent in a message

The data in a message can be any data type in RAPID, except:

- non-value
- semi-value

Continues on next page

8 Communication

8.5.2 RAPID Message Queue behavior

Continued

- `motsetdata`

The data in a message can also be an array of a data type.

User defined records are allowed, but both sender and receiver must have identical declarations of the record.



Tip

To keep backward compatibility, do not change a user defined record once it is used in a released product. It is better to create a new record. This way, it is possible to receive messages from both old and new applications.

Queue name

The name of the queue configured for a RAPID task is the same as the name of the task with the prefix `RMQ_`, for example `RMQ_T_ROB1`. This name is used by the instruction `RMQFindSlot`.

Queue handling

Messages in queues are handled in the order that they are received. This is known as FIFO, first in first out. If a message is received while a previous message is being handled, the new message is placed in the queue. As soon as the first message handling is completed, the next message in the queue is handled.

Queue modes

The queue mode is defined with the system parameter *RMQ Mode*. Default behavior is interrupt mode.

Interrupt mode

In interrupt mode the messages are handled depending on data type. Messages are only handled for connected data types.

A cyclic interrupt must be set up for each data type that the receiver should handle. The same trap routine can be called from more than one interrupt, that is for more than one data type.

Messages of a data type with no connected interrupt will be discarded with only a warning message in the event log.

Receiving an answer to the instruction `RMQSendWait` does not result in an interrupt. No interrupt needs to be set up to receive this answer.

Synchronous mode

In synchronous mode, the task executes an `RMQReadWait` instruction to receive a message of any data type. All messages are queued and handled in order they arrive.

If there is a waiting `RMQReadWait` instruction, the message is handled immediately.

If there is no waiting `RMQReadWait` instruction, the next executed `RMQReadWait` instruction will handle the message.

Continues on next page

Message content

A RAPID Message Queue message consists of a header, containing receiver identity, and a RAPID message. The RAPID message is a pretty-printed string with data type name (and array dimensions) followed by the actual data value.

RAPID message examples:

```
"robtargt;[[930,0,1455],[1,0,0,0],[0,0,0,0],
          [9E9,9E9,9E9,9E9,9E9,9E9]]"
"string;"A message string"
"msgrec:[100,200]"
"bool{2,2};[[TRUE,TRUE],[FALSE,FALSE]]"
```

RAPID task not executing

It is possible to post messages to a RAPID task queue even though the RAPID task containing the queue is not currently executing. The interrupt will not be executed until the RAPID task is executing again.

Message size limitations

Before a message is sent, the maximum size (for the specific data type and dimension) is calculated. If the size is greater than 5000 bytes, the message will be discarded and an error will be raised. The sender can get same error if the receiver is a PC SDK client with a maximum message size smaller than 400 bytes. Sending a message of a specific data type with specific dimensions will either always be possible or never possible.

When a message is received (when calling the instruction `RMQGetMsgData`), the maximum size (for the specific data type and dimension) is calculated. If the size is greater than the maximum message size configured for the queue of this task, the message will be discarded and an error will be logged. Receiving a message of a specific data type with specific dimensions will either always be possible or never possible.

Message lost

In interrupt mode, any messages that cannot be received by a RAPID task will be discarded. The message will be lost and a warning will be placed in the event log.

Example of reasons for discarding a message:

- The data type that is sent is not supported by the receiving task.
- The receiving task has not set up an interrupt for the data type that is sent, and no `RMQSendWait` instruction is waiting for this data type.
- The interrupt queue of the receiving task is full

Queue lost

The queue is cleared at power fail.

When the execution context in a RAPID task is lost, for example when the program pointer is moved to main, the corresponding queue is emptied.

Continues on next page

8 Communication

8.5.2 RAPID Message Queue behavior

Continued

Related information

For more information on queues and messages, see *Technical reference manual - RAPID kernel*.

8.5.3 System parameters

About the system parameters

This is a brief description of each parameter in RAPID Message Queue. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Type Task

These parameters belong to the type *Task* in the topic *Controller*.

Parameter	Description
RMQ Type	<p>Can have one of the following values:</p> <ul style="list-style-type: none"> • <i>None</i> - Disable all communication with RAPID Message Queue for this RAPID task. • <i>Internal</i> - Enable the receiving of RAPID Message Queue messages from other tasks on the controller, but not from external clients (FlexPendant and PC applications). The task is still able to send messages to external clients. • <i>Remote</i> - Enable communication with RAPID Message Queue for this task, both with other tasks on the controller and external clients (FlexPendant and PC applications). <p>The default value is <i>None</i>.</p>
RMQ Mode	<p>Defines the mode of the queue.</p> <p>Can have one of the following values:</p> <ul style="list-style-type: none"> • <i>Interrupt</i> - A message can only be received by connecting a trap routine to a specified message type. • <i>Synchronous</i> - A message can only be received by executing an <code>RMQReadWait</code> instruction. <p>Default value is <i>Interrupt</i>.</p>
RMQ Max Message Size	<p>The maximum data size, in bytes, for a message.</p> <p>The default value is 400.</p> <p>This value cannot be changed in RobotStudio or on the FlexPendant. The value can only be changed by editing the <code>sys.cfg</code> file. The maximum value is 3000.</p>
RMQ Max No Of Messages	<p>Maximum number of messages in queue.</p> <p>Default is 5.</p> <p>This value cannot be changed in RobotStudio or on the FlexPendant. The value can only be changed by editing the <code>sys.cfg</code> file. The maximum value is 10.</p>

8 Communication

8.5.4 RAPID components

8.5.4 RAPID components

About the RAPID components

This is a brief description of each instruction, function, and data type in RAPID Message Queue. For more information, see the respective parameter in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instructions

Instruction	Description
RMQFindSlot	Find the slot identity number of the queue configured for a RAPID task or Robot Application Builder client.
RMQSendMessage	Send data to the queue configured for a RAPID task or Robot Application Builder client.
IRMQMessage	Order and enable cyclic interrupts for a specific data type.
RMQGetMessage	Get the first message from the queue of this task. Can only be used if <i>RMQ Mode</i> is defined as <i>Interrupt</i> .
RMQGetMsgHeader	Get the header part from a message.
RMQGetMsgData	Get the data part from a message.
RMQSendWait	Send a message and wait for the answer. Can only be used if <i>RMQ Mode</i> is defined as <i>Interrupt</i> .
RMQReadWait	Wait for a message. Can only be used if <i>RMQ Mode</i> is defined as <i>Synchronous</i> .
RMQEmptyQueue	Empty the queue.

Functions

Function	Description
RMQGetSlotName	Get the name of the queue configured for a RAPID task or Robot Application Builder client, given a slot identity number, i.e. given a <code>rmqslot</code> .

Data types

Data type	Description
<code>rmqslot</code>	Slot identity of a RAPID task or Robot Application Builder client.
<code>rmqmessage</code>	<p>A message used to store data in when communicating with RAPID Message Queue. It contains information about what type of data is sent, the slot identity of the sender, and the actual data.</p> <p>Note: <code>rmqmessage</code> is a large data type. Declaring too many variables of this data type can lead to memory problems. Reuse the same <code>rmqmessage</code> variables as much as possible.</p>
<code>rmqheader</code>	The <code>rmqheader</code> describes the message and can be read by the RAPID program.

8.5.5 Code examples

Example with RMQSendMessage and RMQGetMessage

This is an example where the sender creates data (x and y value) and sends it to another task. The receiving task gets the message and extract the data to the variable named data.

Sender

```

MODULE SenderMod
  RECORD msgrec
    num x;
    num y;
  ENDRECORD

  PROC main()
    VAR rmqslot destinationSlot;
    VAR msgrec data;
    VAR robtarget p_current;

    ! Connect to queue in other task
    RMQFindSlot destinationSlot "RMQ_OtherTask";

    ! Perform cycle
    WHILE TRUE DO
      ...
      p_current := CRobT(\Tool:=tool1 \WObj:=wobj0);
      data.x := p_current.trans.x;
      data.y := p_current.trans.y;
      ! Send message
      RMQSendMessage destinationSlot, data;
      ...
    ENDWHILE
  ERROR
    IF ERRNO = ERR_RMQ_INVALID THEN
      WaitTime 1;
      ! Reconnect to queue in other task
      RMQFindSlot destinationSlot "RMQ_OtherTask";
      ! Avoid execution stop due to retry count exceed
      ResetRetryCount;
      RETRY;
    ELSIF ERRNO = ERR_RMQ_FULL THEN
      WaitTime 1;
      ! Avoid execution stop due to retry count exceed
      ResetRetryCount;
      RETRY;
    ENDIF
  ENDPROC
ENDMODULE

```

PC SDK client

```
public void RMQReceiveRecord()
```

Continues on next page

8 Communication

8.5.5 Code examples

Continued

```
{
    const string destination_slot = "RMQ_OtherTask";
    IpcQueue queue = Controller.Ipc.CreateQueue(destination_slot,
        16, Ipc.MaxMessageSize);

    // Till application is closed
    while (uiclose)
    {
        IpcMessage message = new IpcMessage();
        IpcReturnType retValue = IpcReturnType.Timeout;

        retValue = queue.Receive(1000, message);
        if (IpcReturnType.OK == retValue)
        {
            // PCSDK App will receive following record
            // RECORD msgrec
            // num x;
            // num y;
            // ENDRECORD

            // num data type in RAPID is 3 bytes long, hence will receive
            // 6 bytes for x and y
            // first byte do left shift by 16,
            // second byte do left shift by 8 and OR all three byte to
            // get x
            // do similar for y
            Int32 x = (message.Data[0] << 16) | (message.Data[1] << 8)
                | message.Data[2];
            Int32 y = (message.Data[3] << 16) | (message.Data[4] << 8)
                | message.Data[5];

            // Display x and y
        }
    }

    if (Controller.Ipc.Exists(destination_slot))
        Controller.Ipc.DeleteQueue(Controller.Ipc.GetQueueId(destination_slot));
}
```

Example with RMQSendWait

This is an example of a RAPID program that sends a message and wait for an answer before execution continues by getting the answer message.

```
MODULE SendAndReceiveMod
    VAR rmqslot destinationSlot;
    VAR rmqmessage recmsg;
    VAR string send_data := "How many units should be produced?";
    VAR num receive_data;

    PROC main()
        ! Connect to queue in other task
        RMQFindSlot destinationSlot "RMQ_OtherTask";
```

Continues on next page

```
! Send message and wait for the answer
RMQSendWait destinationSlot, send_data, recmsg, receive_data
    \Timeout:=30;

! Handle the received data
RMQGetMsgData recmsg, receive_data;
TPWrite "Units to produce: " \Num:=receive_data;

ERROR
IF ERRNO = ERR_RMQ_INVALID THEN
    WaitTime 1;
    ! Reconnect to queue in other task
    RMQFindSlot destinationSlot "RMQ_OtherTask";
    ! Avoid execution stop due to retry count exceed
    ResetRetryCount;
    RETRY;
ELSIF ERRNO = ERR_RMQ_FULL THEN
    WaitTime 1;
    ! Avoid execution stop due to retry count exceed
    ResetRetryCount;
    RETRY;
ELSEIF ERRNO = ERR_RMQ_TIMEOUT THEN
    ! Avoid execution stop due to retry count exceed
    ResetRetyCount;
    RETRY;
ENDIF
ENDPROC
ENDMODULE
```

Example with RMQReceiveSend

```
public void RMQReceiveSend()
{
    const string destination_slot = "RMQ_OtherTask";
    IpcQueue queue = Controller.Ipc.CreateQueue(destination_slot,
        16, Ipc.MaxMessageSize);

    // Till application is closed
    while (uiclose)
    {
        IpcMessage message = new IpcMessage();
        IpcReturnType retValue = IpcReturnType.Timeout;

        retValue = queue.Receive(1000, message);
        if (IpcReturnType.OK == retValue)
        {
            // Received message "How many units should be produced?"
            if (message.ToString() == "How many units should be
                produced?")
            {
                Int32 UnitsToProduce = 100;
            }
        }
    }
}
```

Continues on next page

8 Communication

8.5.5 Code examples

Continued

```
        // num data type in Rapid is 3 bytes long, hence will
        // send 3 bytes to Rapid Module
        byte[] @bytes = new byte[3];
        bytes[0] = (byte)(UnitsToProduce >> 16);
        bytes[1] = (byte)(UnitsToProduce >> 8);
        bytes[2] = (byte)UnitsToProduce;

        // Send UnitsToProduce to Rapid Module
        message.SetData(@bytes);
        queue.Send(message);
    }
}

if (Controller.Ipc.Exists(destination_slot))
    Controller.Ipc.DeleteQueue(Controller.Ipc.GetQueueId(destination_slot));
}
```

9 Engineering tools

9.1 Multitasking [623-1]

9.1.1 Introduction to Multitasking

Purpose

The purpose of the option *Multitasking* is to be able to execute more than one program at a time.

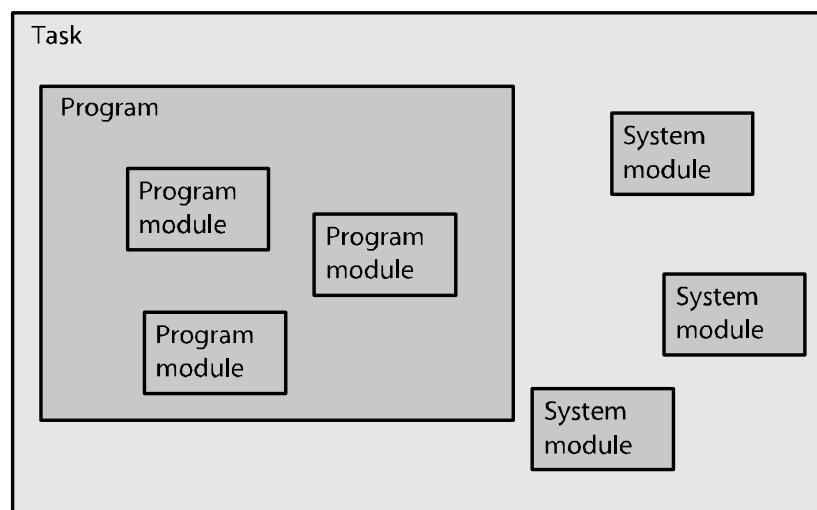
Examples of applications to run in parallel with the main program:

- Continuous supervision of signals, even if the main program has stopped. This can in some cases take over the job of a PLC. However, the response time will not match that of a PLC.
- Operator input from the FlexPendant while the robot is working.
- Control and activation/deactivation of external equipment.

Basic description

Up to 20 tasks can be run at the same time.

Each task consists of one program (with several program modules) and several system modules. The modules are local in the respective task.



en0300000517

Variables and constants are local in the respective task, but persistents are not. Every task has its own trap handling and event routines are triggered only on its own task system states.

What is included

The RobotWare option Multitasking gives you access to:

- The possibility to run up to 20 programs in parallel (one per task).
- The system parameters: The type *Task* and all its parameters.
- The data types: `taskid`, `syncident`, and `tasks`.

Continues on next page

9 Engineering tools

9.1.1 Introduction to Multitasking

Continued

- The instruction: `WaitSyncTask`.
- The functions: `TestAndSet`, `TaskRunMec`, and `TaskRunRob`.



Note

`TestAndSet`, `TaskRunMec`, and `TaskRunRob` can be used without the option `Multitasking`, but they are much more useful together with `Multitasking`.

Basic approach

This is the basic approach for setting up Multitasking. For more information, see [Debug strategies for setting up tasks on page 276](#), and [RAPID components on page 275](#).

- 1 Define the tasks you need.
- 2 Write RAPID code for each task.
- 3 Specify which modules to load in each task.

9.1.2 System parameters

About the system parameters

This is a brief description of each parameter in Multitasking. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Task

These parameters belongs to the type *Task* in the topic *Controller*.

Parameter	Description
Task	<p>The name of the task.</p> <p>Note that the name of the task must be unique. This means that it cannot have the same name as the mechanical unit, and no variable in the RAPID program can have the same name.</p> <p>Note that editing the task entry in the configuration editor and changing the task name will remove the old task and add a new one. This means that any program or module in the task will disappear after a restart with these kind of changes.</p>
Task in foreground	<p>Used to set priorities between tasks.</p> <p><i>Task in foreground</i> contains the name of the task that should run in the foreground of this task. This means that the program of the task, for which the parameter is set, will only execute if the foreground task program is idle.</p> <p>If <i>Task in foreground</i> is set to empty string for a task, it runs at the highest level.</p>
Type	<p>Controls the start/stop and system restart behavior:</p> <ul style="list-style-type: none"> NORMAL - The task program is manually started and stopped (e.g. from the FlexPendant). The task stops at emergency stop. STATIC - At a restart the task program continues from where the it was. The task program is normally not stopped by the FlexPendant or by emergency stop. SEMISTATIC - The task program restarts from the beginning at restart. The task program is normally not stopped by the FlexPendant or by emergency stop. <p>A task that controls a mechanical unit must be of the type NORMAL.</p>
Main entry	The name of the start routine for the task program.
Check unresolved references	This parameter should be set to NO if the system is to accept unsolved references in the program while linking a module, otherwise set to YES.
TrustLevel	<p><i>TrustLevel</i> defines the system behavior when a STATIC or SEMISTATIC task program is stopped (e.g. due to error):</p> <ul style="list-style-type: none"> SysFail - If the program of this task stops, the system will be set to SYS_FAIL. This will cause the programs of all NORMAL tasks to stop (STATIC and SEMISTATIC tasks will continue execution if possible). No jogging or program start can be made. A restart is required. SysHalt - If the program of this task stops, the programs of all NORMAL tasks will be stopped. If "motors on" is set, jogging is possible, but not program start. A restart is required. SysStop - If the program of this task stops, the programs of all NORMAL tasks will be stopped but are restartable. Jogging is also possible. NoSafety - Only the program of this task will stop.

Continues on next page

9 Engineering tools

9.1.2 System parameters

Continued

Parameter	Description
MotionTask	Indicates whether the task program can control robot movement with RAPID move instructions. Only one task can have <i>MotionTask</i> set to YES unless the option MultiMove is used.

9.1.3 RAPID components

Data types

This is a brief description of each data type in Multitasking. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
taskid	taskid identify available tasks in the system. This identity is defined by the system parameter <i>Task</i> , and cannot be defined in the RAPID program. However, the data type <i>taskid</i> can be used as a parameter when declaring a routine. For code example, see taskid on page 293 .
syncident	syncident is used to identify the waiting point in the program, when using the instruction <i>WaitSyncTask</i> . The name of the <i>syncident</i> variable must be the same in all task programs. For code example, see WaitSyncTask example on page 287 .
tasks	A variable of the data type <i>tasks</i> contains names of the tasks that will be synchronized by the instruction <i>WaitSyncTask</i> . For code example, see WaitSyncTask example on page 287 .

Instructions

This is a brief description of each instruction in Multitasking. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
WaitSyncTask	<i>WaitSyncTask</i> is used to synchronize several task programs at a special point in the program. A <i>WaitSyncTask</i> instruction will delay program execution and wait for the other task programs. When all task programs have reached the point, the respective program will continue its execution. For code example, see WaitSyncTask example on page 287 .

Functions

This is a brief description of each function in Multitasking. For more information, see the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
TestAndSet	<i>TestAndSet</i> is used, together with a boolean flag, to ensure that only one task program at the time use a specific RAPID code area or system resource. For code example, see Example with flag and TestAndSet on page 291 .
TaskRunMec	Check if the task program controls any mechanical unit (robot or other unit). For code example, see Test if task controls mechanical unit on page 292 .
TaskRunRob	Check if the task program controls any robot with TCP. For code example, see Test if task controls mechanical unit on page 292 .

9 Engineering tools

9.1.4.1 Debug strategies for setting up tasks

9.1.4 Task configuration

9.1.4.1 Debug strategies for setting up tasks



Tip

The instructions below show the safe way to make updates. By setting the parameter *Type* to NORMAL and *TrustLevel* to NoSafety the task program will be easier to test and any error that may occur will be easier to correct.

If you are certain that the code you introduce is correct, you can skip changing values for *Type* and *TrustLevel*. If you do not change any system parameters you may not have to do any restart mode.

Setting up tasks

Follow this instruction when adding a new task to your system.

	Action
1	Define the new task by adding an instance of the system parameter type <i>Task</i> , in the topic <i>Controller</i> .
2	Set the parameter <i>Type</i> to NORMAL. This will make it easier to create and test the modules in the task.
3	Create the modules that should be in the task, either from the FlexPendant or offline, and save them.
4	In the system parameters for topic <i>Controller</i> and type <i>Automatic loading of Modules</i> , specify all modules that should be preloaded to the new task. For NORMAL tasks the modules can be loaded later, but STATIC or SEMISTATIC tasks the modules must be preloaded.
5	Stop the controller.
6	In Motors on state, test and debug the modules until the functionality is satisfactory.
7	Change the parameters <i>Type</i> and <i>TrustLevel</i> to desired values (e.g. SEMISTATIC and SysFail).
8	Restart the system.

Make changes to task program

Follow this instruction when editing a program in an existing task with *Type* set to STATIC or SEMISTATIC.

	Action
1	Change the system parameter <i>TrustLevel</i> to NoSafety. This will make it possible to change and test the modules in the task.
2	If the system parameter needed to be changed, restart the controller.
3	On the FlexPendant, start the Control Panel from the ABB menu. Then tap FlexPendant and Task Panel Settings . Select All tasks and tap OK .
4	In the Quickset menu, select which tasks to start and stop manually. See Select which tasks to start with START button on page 281 .
5	Press the STOP button to stop the selected STATIC and SEMISTATIC tasks.

Continues on next page

	Action
6	Start the Program Editor . The STATIC and SEMISTATIC tasks are now also editable.
7	Change, test, and save the modules.
8	Start the Control Panel again and open the Task Panel Settings . Select Only Normal tasks and tap OK .
9	Change the parameter <i>TrustLevel</i> back to desired value (e.g. SysFail).
10	Restart the system.

9 Engineering tools

9.1.4.2 Priorities

9.1.4.2 Priorities

How priorities work

The default behavior is that all task programs run at the same priority, in a Round Robin way.

It is possible to change the priority of one task by setting it in the background of another task. Then the program of the background task will only execute when the foreground task program is idle, waiting for an event, for example. Another situation when the background task program will execute is when the foreground task program has executed a move instruction, as the foreground task will then have to wait until the robot has moved.

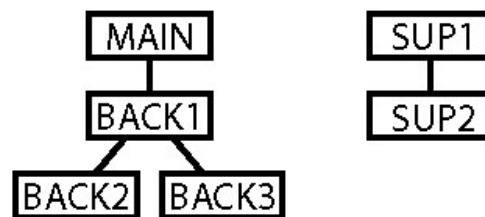
To set a task in the background of another task, use the parameter *Task in foreground*.

Example of priorities

6 tasks are used, with *Task in foreground* set as shown in the table below.

Task name	Task in foreground
MAIN	
BACK1	MAIN
BACK2	BACK1
BACK3	BACK1
SUP1	
SUP2	SUP1

The priority structure will then look like this:



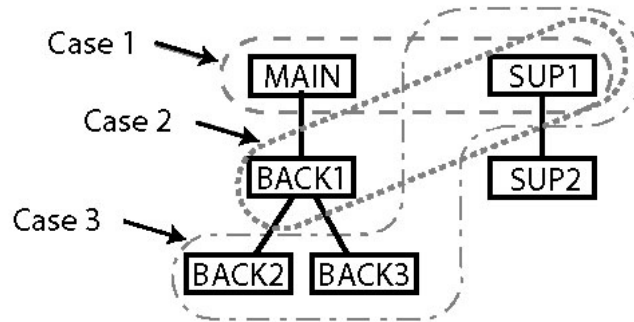
en0300000451

The programs of the tasks MAIN and SUP1 will take turns in executing an instruction each (Case 1 in figure below).

If the MAIN task program is idle, the programs of BACK1 and SUP1 will take turns in executing an instruction each (Case 2 in figure below).

Continues on next page

If both MAIN and BACK1 task programs are idle, the programs of BACK2, BACK3, and SUP1 will take turns in executing an instruction each (Case 3 in figure below).



en0300000479

9.1.4.3 Task Panel Settings

Purpose of Task Panel Settings

The default behavior is that only NORMAL tasks are started and stopped with the START and STOP buttons. In the Task Selection Panel you can select which NORMAL tasks to start and stop, see [Select which tasks to start with START button on page 281](#).

In the Task Panel Settings the default behavior can be altered so that STATIC and SEMISTATIC tasks also can be stepped, started and stopped with the START and STOP buttons. However, these tasks can only be started and stopped if they have *TrustLevel* set to NoSafety and they can only be started and stopped in manual mode.

Allow selection of STATIC and SEMISTATIC tasks in tasks panel

The following procedure details how to make STATIC and SEMISTATIC tasks selectable in the tasks panel.

	Action
1	On the ABB menu, tap Control Panel , then FlexPendant and then Task Panel Settings .
2	Select All tasks (Normal/Static/Semistatic) with trustlevel nosafety and tap OK .

9.1.4.4 Select which tasks to start with START button

Background

The default behavior is that the programs of all NORMAL tasks are started simultaneously when pressing the START button. However, not all NORMAL task programs need to run at the same time. It is possible to select which of the NORMAL task programs will start when pressing the START button.

If **All Tasks** is selected in the **Task Panel Settings**, the programs of all STATIC and SEMISTATIC tasks with *TrustLevel* set to NoSafety can be selected to be started with the START button, forward stepped with the FWD button, backward stepped with the BWD button, and stopped with the STOP button.

If **Task Panel Settings** is set to **Only Normal tasks**, all STATIC and SEMISTATIC tasks are greyed out and cannot be selected in the task panel, Quickset menu (see *Operating manual - IRC5 with FlexPendant*, section *Quickset menu*). All STATIC and SEMISTATIC tasks will be started if the start button is pressed.

If **Task Panel Settings** is set to **All tasks**, STATIC and SEMISTATIC tasks with *TrustLevel/NoSafety* can be selected in the task panel. All selected STATIC and SEMISTATIC tasks can be stopped, stepped, and started. .

A STATIC or SEMISTATIC task, not selected in the task panel, can still be executing. This is not possible for a NORMAL task.

Run Mode is always continuous for STATIC and SEMISTATIC tasks. The Run Mode setting in the Quickset menu is only applicable for NORMAL tasks (see *Operating manual - IRC5 with FlexPendant*, section *Quickset menu*).

This will only work in manual mode, no STATIC or SEMISTATIC task can be started, stepped, or stopped in auto mode.

Task Panel Settings

To start the **Task Panel Settings**, tap the ABB menu, and then **Control Panel**, **FlexPendant** and **Task Panel Settings**.

Selecting tasks

Use this procedure to select which of the tasks are to be started with the START button.

	Action
1	Set the controller to manual mode.
2	On the FlexPendant, tap the QuickSet button and then the tasks panel button to show all tasks. If Task Panel Settings is set to Only Normal tasks , all STATIC and SEMISTATIC tasks are greyed out and cannot be selected. If Task Panel Settings is set to All tasks , STATIC and SEMISTATIC tasks with <i>TrustLevel/NoSafety</i> can be selected, while STATIC and SEMISTATIC tasks with <i>TrustLevel</i> set to other values are grayed out and cannot be selected.
3	Select the check boxes for the tasks whose program should be started by the START button.

Continues on next page

9 Engineering tools

9.1.4.4 Select which tasks to start with START button

Continued

Resetting debug settings in manual mode

Use this procedure to resume normal execution manual mode.

	Action
1	Select Only Normal tasks in the Task Panel Settings .
2	Press START button. All STATIC and SEMISTATIC will run continuously and not be stopped by the STOP button or emergency stop.

Switching to auto mode

When switching to auto mode, all STATIC and SEMISTATIC tasks will be deselected from the tasks panel. The stopped STATIC and SEMISTATIC tasks will start next time any of the START, FWD or BWD button are pressed. These tasks will then run continuously forward and not be stopped by the STOP button or emergency stop.

What happens with NORMAL tasks that has been deselected in the tasks panel depends on the system parameter *Reset* in type *Auto Condition Reset* in topic *Controller*. If *Reset* is set to Yes, all NORMAL tasks will be selected in the tasks panel and be started with the START button. If *Reset* is set to No, only those NORMAL tasks selected in tasks panel will be started by the START button.



Note

Note that changing the value of the system parameter *Reset* will affect all the debug resettings (for example speed override and simulated I/O). For more information, see *Technical reference manual - System parameters*, section *Auto Condition Reset*.

Restarting the controller

If the controller is restarted, all NORMAL tasks will keep their status while all STATIC and SEMISTATIC tasks will be deselected from the tasks panel. As the controller starts up all STATIC and SEMISTATIC tasks will be started and then run continuously.

Deselect task in synchronized mode

If a task is in a synchronized mode, that is program pointer between *SyncMoveOn* and *SyncMoveOff*, the task can be deselected but not reselected. The task cannot be selected until the synchronization is terminated. If the execution continues, the synchronization will eventually be terminated for the other tasks, but not for the deselected task. The synchronization can be terminated for this task by moving the program pointer to main or to a routine.

If the system parameter *Reset* is set to Yes, any attempt to change to Auto mode will fail while a deselected task is in synchronized mode. Changing to Auto mode should make all NORMAL tasks selected, and when this is not possible it is not possible to change to Auto mode.

9.1.5 Communication between tasks

9.1.5.1 Persistent variables

About persistent variables

To share data between tasks, use persistent variables.

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

It is sufficient to specify an initial value for the persistent variable in one task. If initial values are specified in several tasks, only the initial value of the first module to load will be used.



Tip

When a program is saved, the current value of a persistent variable will be used as initial value in the future. If this is not desired, reset the persistent variable directly after the communication.

Example with persistent variable

In this example the persistent variables `startsync` and `stringtosend` are accessed by both tasks, and can therefore be used for communication between the task programs.

Main task program:

```
MODULE module1
  PERS bool startsync:=FALSE;
  PERS string stringtosend:=" ";
  PROC main()
    stringtosend:="this is a test";
    startsync:= TRUE
  ENDPROC
ENDMODULE
```

Background task program:

```
MODULE module2
  PERS bool startsync;
  PERS string stringtosend;
  PROC main()
    WaitUntil startsync;
    IF stringtosend = "this is a test" THEN
      ...
    ENDIF
    !reset persistent variables
    startsync:=FALSE;
    stringtosend:=" ";
  ENDPROC
ENDMODULE
```

Continues on next page

9 Engineering tools

9.1.5.1 Persistent variables

Continued

Module for common data

When using persistent variables in several tasks, there should be declarations in all the tasks. The best way to do this, to avoid type errors or forgetting a declaration somewhere, is to declare all common variables in a system module. The system module can then be loaded into all tasks that require the variables.

9.1.5.2 Waiting for other tasks

Two techniques

Some applications have task programs that execute independently of other tasks, but often task programs need to know what other tasks are doing.

A task program can be made to wait for another task program. This is accomplished either by setting a persistent variable that the other task program can poll, or by setting a signal that the other task program can connect to an interrupt.

Polling

This is the easiest way to make a task program wait for another, but the performance will be the slowest. Persistent variables are used together with the instructions `WaitUntil` or `WHILE`.

If the instruction `WaitUntil` is used, it will poll internally every 100 ms.



CAUTION

Do not poll more frequently than every 100 ms. A loop that polls without a wait instruction can cause overload, resulting in lost contact with the FlexPendant.

Polling example

Main task program:

```
MODULE module1
  PERS bool startsync:=FALSE;
  PROC main()
    startsync:= TRUE;
    ...
  ENDPROC
ENDMODULE
```

Background task program:

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

    WaitUntil startsync;
    ! This is the point where the execution
    ! continues after startsync is set to TRUE
    ...
  ENDPROC
ENDMODULE
```

Interrupt

By setting a signal in one task program and using an interrupt in another task program, quick response is obtained without the work load caused by polling.

The drawback is that the code executed after the interrupt must be placed in a trap routine.

Continues on next page

9 Engineering tools

9.1.5.2 Waiting for other tasks

Continued

Interrupt example

Main task program:

```
MODULE module1
  PROC main()
    SetDO d01,1;
    ...
  ENDPROC
ENDMODULE
```

Background task program:

```
MODULE module2
  VAR intnum intn01;

  PROC main()
    CONNECT intn01 WITH wait_trap;
    ISignalDO d01, 1, intn01;
    WHILE TRUE DO
      WaitTime 10;
    ENDWHILE
  ENDPROC

  TRAP wait_trap
    ! This is the point where the execution
    ! continues after d01 is set in main task
    ...
    IDelete intn01;
  ENDTRAP
ENDMODULE
```

9.1.5.3 Synchronizing between tasks

Synchronizing using WaitSyncTask

Synchronization is useful when task programs are depending on each other. No task program will continue beyond a synchronization point in the program code until all task programs have reached that point in the respective program code. The instruction `WaitSyncTask` is used to synchronize task programs. No task program will continue its execution until all task programs have reached the same `WaitSyncTask` instruction.

WaitSyncTask example

In this example, the background task program calculates the next object's position while the main task program handles the robots work with the current object.

The background task program may have to wait for operator input or I/O signals, but the main task program will not continue with the next object until the new position is calculated. Likewise, the background task program must not start the next calculation until the main task program is done with one object and ready to receive the new value.

Main task program:

```
MODULE module1
  PERS pos object_position:= [0,0,0];
  PERS tasks task_list{2} := [ ["MAIN"], ["BACK1"] ];
  VAR syncident sync1;

  PROC main()
    VAR pos position;
    WHILE TRUE DO
      !Wait for calculation of next object_position
      WaitSyncTask sync1, task_list;
      position:=object_position;
      !Call routine to handle object
      handle_object(position);
    ENDWHILE
  ENDPROC

  PROC handle_object(pos position)
    ...
  ENDPROC
ENDMODULE
```

Background task program:

```
MODULE module2
  PERS pos object_position:= [0,0,0];
  PERS tasks task_list{2} := [ ["MAIN"], ["BACK1"] ];
  VAR syncident sync1;
```

Continues on next page

9 Engineering tools

9.1.5.3 Synchronizing between tasks

Continued

```
PROC main()
  WHILE TRUE DO
    !Call routine to calculate object_position
    calculate_position;

    !Wait for handling of current object
    WaitSyncTask sync1, task_list;
  ENDWHILE
ENDPROC

PROC calculate_position()
  ...
  object_position:= ...
ENDPROC
ENDMODULE
```


9.1.5.4 Using a dispatcher

What is a dispatcher?

A digital signal can be used to indicate when another task should do something. However, it cannot contain information about what to do.

Instead of using one signal for each routine, a dispatcher can be used to determine which routine to call. A dispatcher can be a persistent string variable containing the name of the routine to execute in another task.

Dispatcher example

In this example, the main task program calls routines in the background task by setting `routine_string` to the routine name and then setting `do5` to 1. In this way, the main task program initialize that the background task program should execute the routine `clean_gun` first and then `routine1`.

Main task program:

```
MODULE module1
  PERS string routine_string:="";

  PROC main()
    !Call clean_gun in background task
    routine_string:="clean_gun";
    SetDO do5,1;
    WaitDO do5,0;

    !Call routine1 in background task
    routine_string:="routine1";
    SetDO do5,1;
    WaitDO do5,0;

    ...
  ENDPROC
ENDMODULE
```

Background task program:

```
MODULE module2
  PERS string routine_string:="";

  PROC main()
    WaitDO do5,1;
    %routine_string%;
    SetDO do5,0;
  ENDPROC

  PROC clean_gun()
    ...
  ENDPROC

  PROC routine1()
    ...
  ENDPROC
```

Continues on next page

9 Engineering tools

9.1.5.4 Using a dispatcher

Continued

```
ENDPROC  
ENDMODULE
```

9.1.6 Other programming issues

9.1.6.1 Share resource between tasks

Flag indicating occupied resource

System resources, such as FlexPendant, file system and I/O signals, are available from all tasks. However, if several task programs use the same resource, make sure that they take turns using the resource, rather than using it at the same time.

To avoid having two task programs using the same resource at the same time, use a flag to indicate that the resource is already in use. A boolean variable can be set to true while the task program uses the resource.

To facilitate this handling, the instruction `TestAndSet` is used. It will first test the flag. If the flag is false, it will set the flag to true and return true. Otherwise, it will return false.

Example with flag and TestAndSet

In this example, two task programs try to write three lines each to the FlexPendant. If no flag is used, there is a risk that these lines are mixed with each other. By using a flag, the task program that first execute the `TestAndSet` instruction will write all three lines first. The other task program will wait until the flag is set to false and then write all its lines.

Main task program:

```
PERS bool tproutine_inuse := FALSE;
...
WaitUntil TestAndSet(tproutine_inuse);
TPWrite "First line from MAIN";
TPWrite "Second line from MAIN";
TPWrite "Third line from MAIN";
tproutine_inuse := FALSE;
```

Background task program:

```
PERS bool tproutine_inuse := FALSE;
...
WaitUntil TestAndSet(tproutine_inuse);
TPWrite "First line from BACK1";
TPWrite "Second line from BACK1";
TPWrite "Third line from BACK1";
tproutine_inuse := FALSE;
```

9.1.6.2 Test if task controls mechanical unit

Two functions for inquiring

There are functions for checking if the task program has control of any mechanical unit, `TaskRunMec`, or of a robot, `TaskRunRob`.

`TaskRunMec` will return true if the task program controls a robot or other mechanical unit. `TaskRunRob` will only return true if the task program controls a robot with TCP.

`TaskRunMec` and `TaskRunRob` are useful when using `MultiMove`. With `MultiMove` you can have several tasks controlling mechanical units, see *Application manual - MultiMove*.



Note

For a task to have control of a robot, the parameter *Type* must be set to `NORMAL` and *MotionTask* must be set to `YES`. See [System parameters on page 273](#).

Example with `TaskRunMec` and `TaskRunRob`

In this example, the maximum speed for external equipment is set. If the task program controls a robot, the maximum speed for external equipment is set to the same value as the maximum speed for the robot. If the task program controls external equipment but no robot, the maximum speed is set to 5000 mm/s.

```
IF TaskRunMec() THEN
  IF TaskRunRob() THEN
    !If task controls a robot
    MaxExtSpeed := MaxRobSpeed();
  ELSE
    !If task controls other mech unit than robot
    MaxExtSpeed := 5000;
  ENDIF
ENDIF
```

9.1.6.3 taskid

taskid syntax

A task always has a predefined variable of type taskid that consists of the name of the task and the postfix "Id". For example, the variable name of the MAIN task is MAINId.

Code example

In this example, the module PART_A is saved in the task BACK1, even though the Save instruction is executed in another task.

BACK1Id is a variable of type taskid that is automatically declared by the system.

```
Save \TaskRef:=BACK1Id, "PART_A"  
  \FilePath:="HOME:/DOORDIR/PART_A.MOD" ;
```

9 Engineering tools

9.1.6.4 Avoid heavy loops

9.1.6.4 Avoid heavy loops

Background tasks loop continuously

A task program is normally executed continuously. This means that a background task program is in effect an eternal loop. If this program does not have any waiting instruction, the background task may use too much computer power and make the controller unable to handle the other tasks.

Example

```
MODULE background_module
  PROC main()
    WaitTime 1;
    IF di1=1 THEN
      ...
    ENDIF
  ENDPROC
ENDMODULE
```

If there was no wait instruction in this example and `di1` was 0, then this background task would use up the computer power with a loop doing nothing.

9.2 Sensor Interface [628-1]

9.2.1 Introduction to Sensor Interface

Purpose

The option Sensor Interface is used for communication with external sensors via a serial channel.

The serial channel may be accessed using a package of RAPID instructions that provide the ability to read and write sensor data.

An interrupt feature allows subscriptions on changes in sensor data.



Tip

The communication provided by Sensor Interface is integrated in arc welding instructions for seam tracking and adaptive control of process parameters. These instructions handle communication and corrections for you, whereas with Sensor Interface you handle this yourself. For more information, see *Application manual - Arc and Arc Sensor* and *Application manual - Continuous Application Platform*.

What is included

The RobotWare option Sensor Interface gives you access to:

- Instruction used to connect to a sensor device: `SenDevice`.
- Instruction used to set up interrupt, based on input from the serial sensor interface: `IVarValue`.
- Instructions used to read and write to and from a device connected to the serial sensor interface: `ReadBlock`, `WriteBlock` and `WriteVar`.
- Function for reading from a device connected to the serial sensor interface: `ReadVar`.

Basic approach

This is the basic approach for using Sensor Interface.

- 1 Configure the sensor. See [Configuring sensors over serial channels on page 297](#).
- 2 Use interrupts in the RAPID code to make adjustments according to the input from the sensor. For an example, see [Interrupt welding to adjust settings on page 302](#).

Limitations

Interrupts with `IVarValue` is only possible to use with the instructions `ArcL`, `ArcC`, `CapL`, and `CapC`. The switch `Track` must be used. That is, the controller must be equipped with either *RobotWare Arc* or *Continuous Application Platform* together with *Optical Tracking*, or with the option *Weldguide*.

9 Engineering tools

9.2.2.1 About the sensors

9.2.2 Configuring sensors

9.2.2.1 About the sensors

Supported sensors

Sensor Interface supports:

- Sensors connected via serial channels using the RTP1 protocol. For configuration, see [Configuring sensors over serial channels on page 297](#).
- Sensors connected to Ethernet using the RoboCom Light protocol from Servo-Robot Inc or LTAPP protocol from ABB. For configuration, see [Configuring sensors over Ethernet channel on page 298](#).

9.2.2.2 Configuring sensors over serial channels

Overview

Sensor Interface communicates with a maximum of two sensors over serial channels using the RTP1 protocol.

System parameters

This is a brief description of the parameters used when configuring a sensor. For more information about the parameters, see *Technical reference manual - System parameters*.

These parameters belong to the type *Transmission Protocol* in the topic *Communication*.

Parameter	Description
Name	The name of the transmission protocol. For a sensor the name must end with ":". For example "laser1:" or "swg:".
Type	The type of transmission protocol. For a sensor using serial channel, it has to be "RTP1".
Serial Port	The name of the serial port that will be used for the sensor. This refers to the parameter <i>Name</i> in the type <i>Serial Port</i> . For information on how to configure a serial port, see <i>Technical reference manual - System parameters</i> .

Configuration example

This is an example of how a transmission protocol can be configured for a sensor. We assume that there already is a serial port configured with the name "COM1".

Name	Type	Serial Port
laser1:	RTP1	COM1

9.2.2.3 Configuring sensors over Ethernet channel

Overview

Sensor Interface communicates with a maximum of six sensors over Ethernet channel using the RoboCom Light protocol version E04 (from Servo-Robot Inc) or the LTAPP protocol (from ABB). RoboCom Light is an XML based protocol using TCP/IP.

The sensor acts as a server, the robot controller acts as a client. I.e. the robot controller initiates the connection to the sensor.

RoboCom Light expects TCP port 6344 on the external sensor side, and LTAPPTCP expects TCP port 5020.

System parameters

This is a brief description of the parameters used when configuring a sensor. For more information about the parameters, see *Technical reference manual - System parameters*.

These parameters belong to the type *Transmission Protocol* in the topic *Communication*.

Parameter	Description
Name	The name of the transmission protocol. For a sensor the name must end with ":". For example "laser1:" or "swg:".
Type	The type of transmission protocol. For RoboCom Light the protocol type SOCKDEV has to be configured, and for LTAPPTCP it is LTAPPTCP.
Serial Port	The name of the serial port that will be used for the sensor. This refers to the parameter <i>Name</i> in the type <i>Serial Port</i> . For information on how to configure a serial port, see <i>Technical reference manual - System parameters</i> . For IP based transmission protocols (i.e. <i>Type</i> has value TCP/IP, SOCKDEV, LTAPPTCP or UDPUC), <i>Serial Port</i> is not used and has the value N/A.
Remote Address	The IP address of the sensor. This refers to the type <i>Remote Address</i> . For information on how to configure Remote Address, see <i>Technical reference manual - System parameters</i> .

Configuration examples

These are examples of how a transmission protocol can be configured for a sensor.

Name	Type	Serial Port	Remote Address
laser2:	SOCKDEV	N/A	192.168.125.101
laser3:	LTAPPTCP	N/A	192.168.125.102

9.2.3 RAPID

9.2.3.1 RAPID components

Data types

There are no data types for *Sensor Interface*.

Instructions

This is a brief description of each instruction in *Sensor Interface*. For more information, see respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
SenDevice	SenDevice is used, to connect to a physical sensor device.
IVarValue	IVarVal (Interrupt Variable Value) is used to order and enable an interrupt when the value of a variable accessed via the serial sensor interface is changed.
ReadBlock	ReadBlock is used to read a block of data from a device connected to the serial sensor interface. The data is stored in a file. ReadBlock can only be used with a serial channel connected sensor (not Ethernet connected sensor.)
WriteBlock	WriteBlock is used to write a block of data to a device connected to the serial sensor interface. The data is fetched from a file. WriteBlock can only be used with a serial channel connected sensor (not Ethernet connected sensor.)
WriteVar	WriteVar is used to write a variable to a device connected to the serial sensor interface.

Functions

This is a brief description of each function in *Sensor Interface*. For more information, see respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
ReadVar	ReadVar is used to read a variable from a device connected to the serial sensor interface.

Modules

The option *Sensor Interface* includes one system module, *LTAPP__Variables*. This module contains the variable numbers defined in the protocol LTAPP. It is automatically loaded as SHARED and makes the variables (CONST num) available in all RAPID tasks.

Note! A copy of the module is placed in the robot system directory HOME/LTC, but the copy is NOT the loaded module.

Continues on next page

9 Engineering tools

9.2.3.1 RAPID components

Continued

Constants

Name	Number	Read/write	Description
LTAPP__VERSION	1	R	A value that identifies the sensor software version.
LTAPP__RESET	3	W	Reset the sensor to the initial state, regardless of what state it is currently in.
LTAPP__PING	4	W	Sensor returns a response indicating its status.
LTAPP__CAMCHECK	5	W	Start camera check of the sensor. If this cannot be done within the time limit specified in the link protocol a <i>Not ready yet</i> status will be returned.
LTAPP__POWER_UP	6	RW	Turn power on (1) or off (0) for the sensor and initialize the filters. (Power on can take several seconds!)
LTAPP__LASER_OFF	7	RW	Switch the laser beam off (1) or on (0) and measure.
LTAPP__X	8	R	Measured X value, unsigned word. The units are determined by the variable <i>Unit</i> .
LTAPP__Y	9	R	Measured Y value, unsigned word. The units are determined by the variable <i>Unit</i> .
LTAPP__Z	10	R	Measured Z value, unsigned word. The units are determined by the variable <i>Unit</i> .
LTAPP__GAP	11	R	The gap between two sheets of metal. The units are determined by the variable <i>Unit</i> , -32768 if not valid.
LTAPP__MISMATCH	12	R	Mismatch, unsigned word. The units are determined by the variable <i>Unit</i> . -32768 if not valid.
LTAPP__AREA	13	R	Seam area, units in mm ² , -32768 if not valid.
LTAPP__THICKNESS	14	RW	Plate thickness of sheet that the sensor should look for, LSB=0.1mm.
LTAPP__STEPIR	15	RW	Step direction of the joint: Step on left (1) or right (0) side of path direction.
LTAPP__JOINT_NO	16	RW	Set or get active joint number.
LTAPP__AGE	17	R	Time since profile acquisition (ms), unsigned word.
LTAPP__ANGLE	18	R	Angle of the normal to the joint relative sensor coordinate system Z direction - in 0.1 degrees.
LTAPP__UNIT	19	RW	Units of X, Y, Z, gap, and mismatch. 0= 0.1mm, 1= 0.01mm.
-	20	-	Reserved for internal use.
LTAPP__APM_P1	31	R	Servo robot only! Adaptive parameter 1

Continues on next page

Name	Number	Read/write	Description
LTAPP__APM_P2	32	R	Servo robot only! Adaptive parameter 2
LTAPP__APM_P3	33	R	Servo robot only! Adaptive parameter 3
LTAPP__APM_P4	34	R	Servo robot only! Adaptive parameter 4
LTAPP__APM_P5	35	R	Servo robot only! Adaptive parameter 5
LTAPP__APM_P6	36	R	Servo robot only! Adaptive parameter 6
LTAPP__ROT_Y	51	R	Measured angle around sensor Y axis
LTAPP__ROT_Z	52	R	Measured angle around sensor Z axis A

9.2.4 Examples

9.2.4.1 Code examples

Interrupt welding to adjust settings

This is an example of a welding program where a sensor is used. The sensor reads the gap (in mm) and an interrupt occurs every time the value from the sensor changes. The new value from the sensor is then used to determine correct settings for voltage, wire feed and speed.

```
LOCAL PERS num adptVlt{8}:=
    [1,1.2,1.4,1.6,1.8,2,2.2,2.5];
LOCAL PERS num adptWfd{8}:=
    [2,2.2,2.4,2.6,2.8,3,3.2,3.5];
LOCAL PERS num adptSpd{8}:=
    [10,12,14,16,18,20,22,25];
LOCAL CONST num GAP_VARIABLE_NO:=11;
PERS num gap_value:=0;
PERS trackdata track:=[0,FALSE,150,[0,0,0,0,0,0,0,0],
    [3,1,5,200,0,0,0]];
VAR intnum IntAdap;

PROC main()
    ! Setup the interrupt. The trap routine AdapTrap will be called
    when the gap variable with number GAP_VARIABLE_NO in the
    sensor interface has been changed. The new value will be
    available in the gap_value variable.
    CONNECT IntAdap WITH AdapTrap;
    IVarValue "laser1:", GAP_VARIABLE_NO, gap_value, IntAdap;

    ! Start welding
    ArcLStart p1,v100,adaptSm,adaptWd,fine, tool\j\Track:=track;
    ArcLEnd p2,v100,adaptSm,adaptWd,fine, tool\j\Track:=track;
ENDPROC

TRAP AdapTrap
    VAR num ArrInd;
    ! Scale the raw gap value received
    ArrInd:=ArrIdx(gap_value);

    ! Update active weld data variable adaptWd with new data from
    the predefined parameter arrays.
    ! The scaled gap value is used as index in the voltage, wirefeed
    and speed arrays.
    adaptWd.weld_voltage:=adptVlt{ArrInd};
    adaptWd.weld_wirefeed:=adptWfd{ArrInd};
    adaptWd.weld_speed:=adptSpd{ArrInd};

    ! Request a refresh of welding parameters using the new data in
    adaptWd
    ArcRefresh;
```

Continues on next page

```
ENDTRAP

FUNC ArrIndx(num value)
  IF value < 0.5 THEN RETURN 1;
  ELSEIF value < 1.0 THEN RETURN 2;
  ELSEIF value < 1.5 THEN RETURN 3;
  ELSEIF value < 2.0 THEN RETURN 4;
  ELSEIF value < 2.5 THEN RETURN 5;
  ELSEIF value < 3.0 THEN RETURN 6;
  ELSEIF value < 3.5 THEN RETURN 7;
  ELSE RETURN 8;
ENDIF
ENDFUNC
```

Reading positions from sensor

In this example, the sensor is turned on and the coordinates are read from the sensor.

```
! Define variable numbers
CONST num SensorOn := 6;
CONST num YCoord := 9;
CONST num ZCoord := 10;

! Define the transformation matrix
CONST pose SensorMatrix := [[100,0,0],[1,0,0,0]];

VAR pos SensorPos;
VAR pos RobotPos;

! Request start of sensor measurements
WriteVar SensorOn, 1;

! Read a Cartesian position from the sensor
SensorPos.x := 0;
SensorPos.y := ReadVar (YCoord);
SensorPos.z := ReadVar (ZCoord);

! Stop sensor
WriteVar SensorOn, 0;

! Convert to robot coordinates
RobotPos := PoseVect(SensorMatrix, SensorPos);
```

9.3 Externally Guided Motion [689-1]

9.3.1 Introduction to EGM

9.3.1.1 Overview

Purpose

Externally Guided Motion (EGM) offers two different features:

- *EGM Position Guidance:*
The robot does not follow a programmed path in RAPID but a path generated by an external device.
- *EGM Path Correction:*
The programmed robot path is modified/corrected using measurements provided by an external device.

EGM Position Guidance

The purpose of *EGM Position Guidance* is to use an external device to generate position data for one or several robots. The robots will be moved to that given position.

Some examples of applications are:

- Place an object (for example a car door or a window) at a location (for example a car body) that was given by an external sensor.
- Bin picking. Pick objects from a bin using an external sensor to identify the object and its position.

EGM Path Correction

The purpose of *EGM Path Correction* is to use external robot mounted devices to generate path correction data for one or several robots. The robots will be moved along the corrected path, which is the programmed path with added measured corrections.

Some examples of applications are:

- Seam tracking.
- Tracking of objects moving near a known path.

What is included

The RobotWare option *Externally Guided Motion* gives you access to:

- Instructions to set up, activate, and reset EGM Position Guidance.
- Instructions to set up, activate, and reset EGM Path Correction.
- Instructions to initiate EGM Position Guidance movements and to stop them.
- Instructions to perform EGM Path Correction movements.
- A function to retrieve the current EGM state.
- System parameters to configure EGM and set default values.

Continues on next page

Limitations

Limitations for EGM Position Guidance

- It is not possible to perform linear movements using EGM Position Guidance, since EGM Position Guidance does not contain interpolator functionality. The actual path of the robot will depend on the robot configuration, the start position, and the generated position data.
- EGM Position Guidance does not support MultiMove.
- It is not possible to use EGM Position Guidance to guide a mechanical unit in a moving work object.
- If the robot ends up near a singularity, i.e. when two robot axis are nearly parallel, the robot movement will be stopped with an error message. In that situation the only way is to jog the robot out of the singularity.
-

Limitations for EGM Path Correction

- The external device has to be robot mounted.
- Corrections can only be applied in the path coordinate system.
- Only position correction in y and z can be performed. It is not possible to perform orientation corrections, nor corrections in x (which is the path direction/tangent).

Common limitations for EGM

- EGM can only be used on 6-axis robots.
- EGM can only be used in RAPID tasks with a robot, i.e. it is not possible to use it in a task that contains only additional axis, i.e. in robtargets there are values in the `pose` portion of the data.
- An EGM movement has to start in a fine point.
- Only one external device can be used for each robot to provide correction data.

9.3.1.2 Introduction to EGM Position Guidance

What is EGM Position Guidance

EGM Position Guidance is designed for advanced users and provides a low level interface to the robot controller, by by-passing the path planning that can be used when highly responsive robot movements are needed. EGM Position Guidance can be used to read positions from and write positions to the motion system at a high rate. This can be done every 4 ms with a control lag of 10–20 ms depending on the robot type. The references can either be specified using joint values or a pose. The pose can be defined in any work object that is not moved during the EGM Position Guidance movement.

All necessary filtering, supervision of references, and state handling is handled by EGM Position Guidance. Examples of state handling are program start/stop, emergency stop, etc.

The main advantage of EGM Position Guidance is the high rate and low delay/latency compared to other means of external motion control. The time between writing a new position until that given position starts to affect the actual robot position, is usually around 20 ms.

EGM handles *Absolute Accuracy*.

What EGM Position Guidance does not do

EGM goes directly into the motor reference generation, i.e. it does not provide any path planning. This means that you cannot order a movement to a pose target and expect a linear movement. It is not possible either to order a movement with a specified speed or order a movement that is supposed to take a specified time.

For ordering such movements path planning is needed and we refer you to the standard movement instructions in RAPID, i.e. `MoveL`, `MoveJ`, etc.



WARNING

Since the path planning is by-passed by EGM in the robot controller, the robot path is created directly from user input. It is therefore important to make sure that the stream of position references sent to the controller is as smooth as possible. The robot will react quickly to all position references sent to the controller, also faulty ones.

9.3.1.3 Introduction to EGM Path Correction

What is EGM Path Correction

EGM Path Correction gives the user the possibility to correct a programmed robot path. The device or sensor that is used to measure the actual path has to be mounted on the tool flange of the robot and it must be possible to calibrate the sensor frame.

The corrections are performed in the path coordinate system, which gets its x-axis from the tangent of the path, the y-axis is the cross product of the path tangent, and the z-direction of the active tool frame and the z-axis is the cross product of x-axis and y-axis.

EGM Path correction has to start and end in a fine point. The sensor measurements can be provided at a maximum rate of 20 Hz.

9 Engineering tools

9.3.2.1 Basic approach

9.3.2 Using EGM

9.3.2.1 Basic approach

Basic approach for EGM Position Guidance

This is the general approach to move/guide a robot using an external device (sensor) to give the target for the movement.

	Action
1	Move the robot to a fine point.
2	Register an EGM client and get an EGM identity. This identity is then used to link setup, activation, movement, deactivation etc. to a certain EGM usage. The EGM state is still <code>EGM_STATE_DISCONNECTED</code> .
3	Call an EGM setup instruction to set up the position data source using signals or UdpUc protocol connection. The EGM state changes to <code>EGM_STATE_CONNECTED</code> .
4	Choose if the position is given as joint values or as a pose and give the position convergence criteria, i.e. when the position is considered to be reached.
5	If pose was chosen, define which frames are used to define the target position and in which frame the movement is to be applied.
6	Give the stop mode, an optional time-out and perform the movement itself. Now the EGM state is <code>EGM_STATE_RUNNING</code> . This is when the robot is moving.
7	The EGM movement will stop when the position is considered to be reached, i.e. the convergence criteria is fulfilled. Now the EGM state has changed back to <code>EGM_STATE_CONNECTED</code> .

Basic approach for EGM Path Correction

This is the general approach to correct a programmed path with EGM Path Correction.

	Action
1	Move the robot to a fine point.
2	Register an EGM client and get an EGM identity. This identity is then used to link setup, activation, movement, deactivation etc. to a certain EGM usage. The EGM state is still <code>EGM_STATE_DISCONNECTED</code> .
3	Call an EGM setup instruction to set up the position data source using signals or UdpUc protocol connection. The EGM state changes to <code>EGM_STATE_CONNECTED</code> .
4	Define the sensor correction frame, which always is a tool frame.
5	Perform the movement itself. Now the EGM state is <code>EGM_STATE_RUNNING</code> .
	At the next fine point EGM will return to the state <code>EGM_STATE_CONNECTED</code> .
6	To free an EGM identity for use with another sensor you have to reset EGM, which returns EGM to the state <code>EGM_STATE_DISCONNECTED</code> .

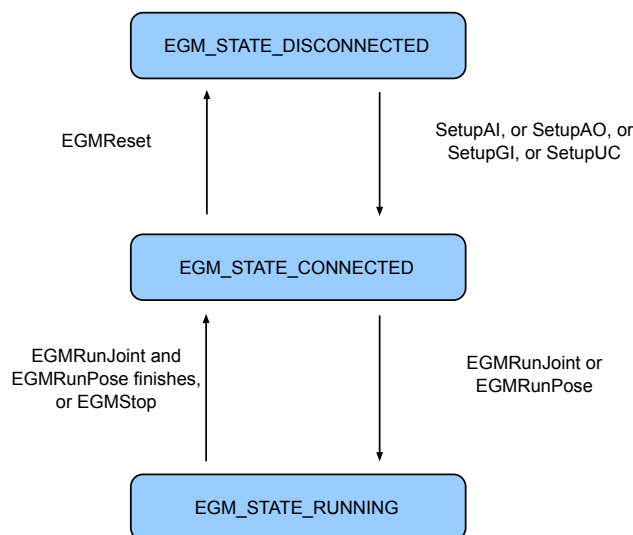
9.3.2.2 Execution states

Description

The EGM process has different states:

Value	Description
EGM_STATE_DISCONNECTED	The EGM state of the specific process is undefined. No setup is active.
EGM_STATE_CONNECTED	The specified EGM process is not activated. Setup has been made, but no EGM movement is active.
EGM_STATE_RUNNING	The specified EGM process is running. The EGM movement is active, i.e. the robot is moved.

Transitions between the different states are according to the figure below.



xx1400001082

The RAPID instructions `EGMRunJoint` and `EGMRunPose` start from `EGM_STATE_CONNECTED` and change the state to `EGM_STATE_RUNNING` as long as the convergence criteria for the target position have not been met or the timeout time has not expired. When one of these conditions is met, the EGM state is changed to `EGM_STATE_CONNECTED` again and the instruction ends, i.e. RAPID execution continues to the next instruction.

If EGM has the state `EGM_STATE_RUNNING` and RAPID execution is stopped, EGM enters the state `EGM_STATE_CONNECTED`. At program restart, EGM returns to the state `EGM_STATE_RUNNING`.

If the program pointer is moved using `PP to Main` or `PP to cursor`, the EGM state is changed to `EGM_STATE_CONNECTED`, if the state was `EGM_STATE_RUNNING`.

9.3.2.3 Input data

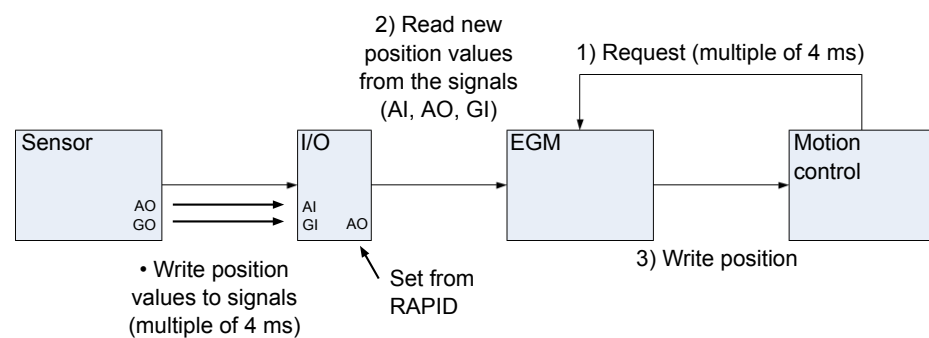
Input data for EGM Position Guidance

The source for input data is selected using the EGM setup instructions. The three first instructions select a signal interface and the last instruction a UdpUc interface (*User Datagram Protocol Unicast Communication*).

Instructions	Description
EGMSetupAI	Setup analog input signals for EGM
EGMSetupAO	Setup analog output signals for EGM
EGMSetupGI	Setup group input signals for EGM
EGMSetupUC	Setup the UdpUc protocol for EGM

Input data for EGM contain mainly position data either as joints or as a pose, i.e. Cartesian position plus orientation.

The data flow for the signal interface is illustrated below:



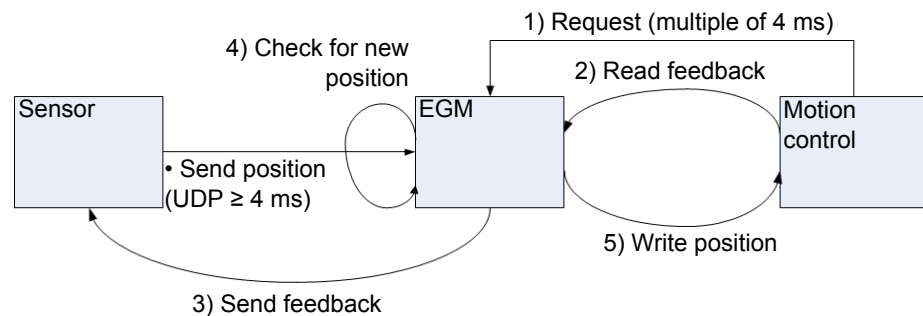
xx1400002016

- 1 Motion control calls EGM.
 - 2 EGM reads the position values from the signals.
 - 3 EGM writes the position data to motion control.
- The sensor writes position data to the signals.

If signals are used as data source, the input is limited to 6 for the robot, i.e. 6 joint values or 3 Cartesian position values (x, y, z) plus 3 Euler angle values (rx, ry, rz), and up to 6 values for additional axes.

Continues on next page

The data flow for the UdpUc interface is illustrated below:



xx1400002017

- 1 Motion control calls EGM.
 - 2 EGM reads feedback data from motion control.
 - 3 EGM sends feedback data to the sensor.
 - 4 EGM checks the UDP queue for messages from the sensor.
 - 5 If there is a message, EGM reads the next message and step 5 writes the position data to motion control. If no position data had been sent, motion control continues to use the latest position data previously written by EGM.
- The sensor sends position data to the controller (EGM). Our recommendation is to couple this to step 3. Then the sensor will be in phase with the controller.

The control loop is based on the following relation between speed and position:

$$speed = k * (pos_ref - pos) + speed_ref$$

k - factor
 pos_ref - reference position
 pos - desired position
 $speed_ref$ - reference speed

For instructions on how to implement the UdpUc protocol for an external device, see [The EGM sensor protocol on page 318](#). There you will also find a description of input data.

Input data for EGM Path Correction

The source for input data is selected using the EGM setup instructions. The three first instructions select a signal interface and the last instruction a UdpUc interface (*User Datagram Protocol Unicast Communication*).

Instructions	Description
EGMSetupAI	Setup analog input signals for EGM
EGMSetupAO	Setup analog output signals for EGM
EGMSetupGI	Setup group input signals for EGM
EGMSetupUC	Setup the UdpUc protocol for EGM

Continues on next page

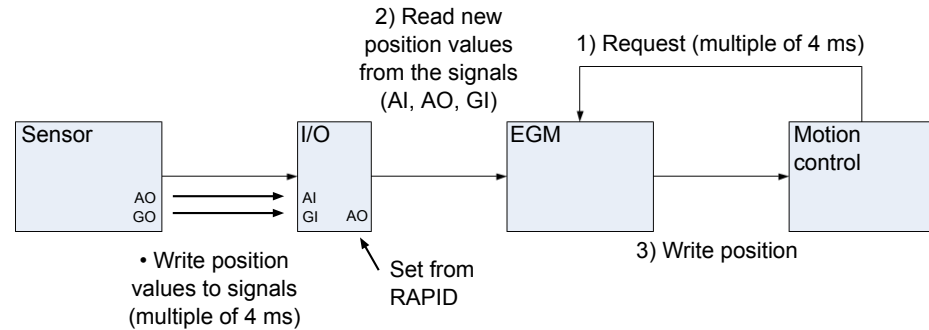
9 Engineering tools

9.3.2.3 Input data

Continued

Input data for EGM contain mainly position data.

The data flow for the signal interface is illustrated below:



xx1400002016

- 1 Motion control calls EGM.
- 2 The measurement data (y- and z-values) are read from the signals or fetched from the sensor at multiples of about 24 ms.
- 3 EGM calculates the position correction and writes it to motion control. If the UdpUc protocol is used, feedback is sent to the sensor.

9.3.2.4 Output data

Description

Output data is only available for the UdpUc interface.

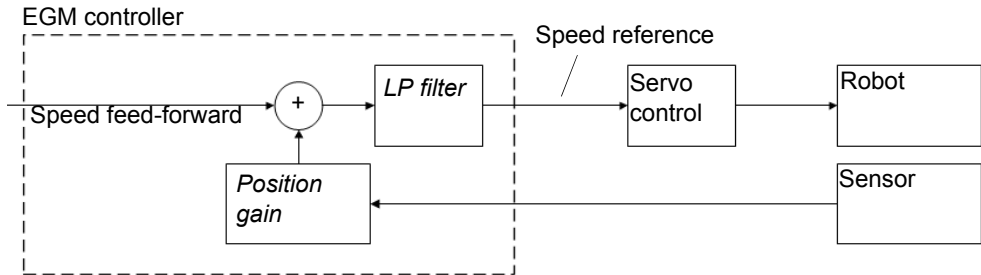
For instructions on how to implement the UdpUc protocol for an external device, see [The EGM sensor protocol on page 318](#). There you will also find a description of output data.

9.3.2.5 Configuration

Configuration for EGM Position Guidance

EGM behavior can be influenced using the system parameters of type *External Motion Interface Data* topic *Motion*. For a description of all available EGM parameters, see [System parameters on page 322](#).

Here follows a closer description of the two parameters that influence the EGM control loop. The figure shows a simplified view of the EGM control system.



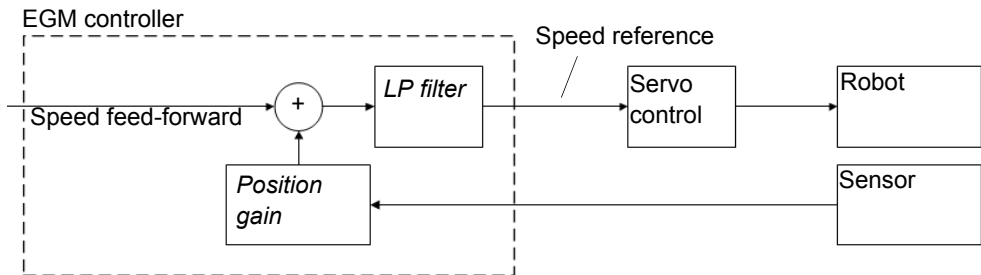
xx1400001083

<i>Default proportional Position Gain</i>	The parameter <i>Position gain</i> in the figure influences the responsiveness moving to the target position, given by the sensor, in relation to the current robot position. The higher the value, the faster the response.
<i>Default Low Pass Filter Bandwidth Time</i>	The parameter <i>LP Filter</i> in the figure is the default value used to filter the speed contribution from EGM.

Configuration for EGM Path Correction

EGM behavior can be influenced using the system parameters of type *External Motion Interface Data* topic *Motion*. For a description of all available EGM parameters, see [System parameters on page 322](#).

Here follows a closer description of the two parameters that influence the EGM control loop. The figure shows a simplified view of the EGM control system.



xx1400001083

Continues on next page

<i>Default proportional Position Gain</i>	The parameter <i>Position gain</i> in the figure influences the responsiveness moving to the target position, given by the sensor, in relation to the current robot position. The higher the value, the faster the response.
<i>Default Low Pass Filter Bandwidth Time</i>	The parameter <i>LP Filter</i> in the figure is the default value used to filter the speed contribution from EGM.

9.3.2.6 Frames

Frames for EGM Position Guidance

EGM can be run in two different modes, joint mode and pose mode. The following section applies to the EGM pose mode only.

For the joint mode there is no need for reference frames, because both sensor values and position values are axis angles given in degrees relative to the calibration position of each axis. But for the pose mode reference frames are necessary. Measurements from the sensor and directions for position change can only be given relative to reference frames.

The RAPID instruction `EGMActPose` defines all frames that are available in EGM:

Frame	Description
Tool	The tool data to be used for the EGM process is defined with the optional <code>\Tool</code> argument.
Work object	The work object data used for the EGM process is defined with the optional <code>\Wobj</code> argument.
Correction	The frame to be used to give the final movement direction is defined by the mandatory <code>CorrFrame</code> argument.
Sensor	The frame to be used to interpret the sensor data is defined by the mandatory <code>SensorFrame</code> argument.

Tools and work objects

The tool and the work object may be defined in two combinations only:

- 1 If the tool is attached to the robot, the work object has to be fixed.
- 2 If the tool is fixed, the work object has to be attached to the robot.



Note

It is not possible to use a work object or tool that is attached to any other mechanical unit than the EGM robot.

Predefined frame types

For the frames `CorrFrame` and `SensorFrame` it is also necessary to know what they are related to. This information is specified using the predefined frame types in the data type `egmframetype`:

Value	Description
<code>EGM_FRAME_BASE</code>	The frame is defined relative to the base frame (pose mode).
<code>EGM_FRAME_TOOL</code>	The frame is defined relative to the used tool (pose mode).
<code>EGM_FRAME_WOBJ</code>	The frame is defined relative to the used work object (pose mode).
<code>EGM_FRAME_WORLD</code>	The frame is defined relative to the world frame (pose mode).
<code>EGM_FRAME_JOINT</code>	The values are joint values (joint mode).

Frames for EGM Path Correction

EGM can be run in two different modes, joint mode and pose mode. The following section applies to the EGM pose mode only.

Continues on next page

For the joint mode there is no need for reference frames, because both sensor values and position values are axis angles given in degrees relative to the calibration position of each axis. But for the pose mode reference frames are necessary. Measurements from the sensor and directions for position change can only be given relative to reference frames.

The RAPID instruction `EGMActPose` defines all frames that are available in EGM:

Frame	Description
Tool	The tool data to be used for the EGM process is defined with the optional <code>\Tool</code> argument.
Work object	The work object data used for the EGM process is defined with the optional <code>\Wobj</code> argument.
Correction	The frame to be used to give the final movement direction is defined by the mandatory <code>CorrFrame</code> argument.
Sensor	The frame to be used to interpret the sensor data is defined by the mandatory <code>SensorFrame</code> argument.

Tools and work objects

The tool and the work object may be defined in two combinations only:

- 1 If the tool is attached to the robot, the work object has to be fixed.
- 2 If the tool is fixed, the work object has to be attached to the robot.



Note

It is not possible to use a work object or tool that is attached to any other mechanical unit than the EGM robot.

Predefined frame types

For the frames `CorrFrame` and `SensorFrame` it is also necessary to know what they are related to. This information is specified using the predefined frame types in the data type `egmframetype`:

Value	Description
<code>EGM_FRAME_BASE</code>	The frame is defined relative to the base frame (pose mode).
<code>EGM_FRAME_TOOL</code>	The frame is defined relative to the used tool (pose mode).
<code>EGM_FRAME_WOBJ</code>	The frame is defined relative to the used work object (pose mode).
<code>EGM_FRAME_WORLD</code>	The frame is defined relative to the world frame (pose mode).
<code>EGM_FRAME_JOINT</code>	The values are joint values (joint mode).

9.3.3 The EGM sensor protocol

Description

The EGM sensor protocol is designed for high speed communication between a robot controller and a communication endpoint with minimum overhead.

The communication endpoint is typically a sensor, so *sensor* will be used from now on instead of communication endpoint. Sometimes the sensor is connected to a PC, and the PC then transfers the sensor data to the robot. The purpose of the sensor protocol is to communicate sensor data frequently between the robot controller and sensors. The EGM sensor protocol is using Google Protocol Buffers for encoding and UDP as a transport protocol. Google Protocol Buffers has been selected due to its speed and its language-neutrality. UDP has been chosen as a transport protocol since the data sent is *real-time* data sent with high frequency and if packets get lost it is useless to re-send the data.

The EGM sensor protocol data structures are defined by the EGM proto file. Sensor name, IP-address and port number of sensors are configured in the system parameters. A maximum of eight sensors can be configured.

The sensor is acting as a server and it cannot send anything to the robot before it has received a first message from the robot controller. Messages can be sent independently of each other in both directions after that first message. Applications using the protocol may put restrictions on its usage but the protocol itself has no built-in synchronization of request responses or supervision of lost messages.

There are no special connect or disconnect messages, only data which can flow in both directions independently of each other. The first message from the robot is a data message. One has also to keep in mind, that a sender of an UDP message continues to send even though the receiver's queue may be full. The receiver has to make sure, that its queue is emptied.

By default, the robot will send and read data from the sensor every 4 milliseconds, independently of when data is sent from the sensor. This cycle time can be changed to a multiple of 4 ms using the optional argument `\SampleRate` of the RAPID instructions `EGMActJoint` or `EGMActPose`.

Google Protocol Buffers

Google Protocol Buffers or *Protobuf*, are a way to serialize/de-serialize data in a very efficient way. Protobuf is in general 10-100 times faster than XML. There is plenty of information on the Internet about Protobuf and the *Google overview* is a good start.

In short, message structures are described in a *.proto* file. The *.proto* file is then compiled. The compiler generates serialized/de-serialized code which is then used by the application. The application reads a message from the network, runs the de-serialization, creates a message, calls serialization method, and then sends the message.

It is possible to use Protobuf in most programming languages since Protobuf is language neutral. There are many different implementations depending on the language.

Continues on next page

The main disadvantage with Protobuf is that Protobuf messages are serialized into a binary format which makes it more difficult to debug packages using a network analyzer.

Third party tools

Except for the *Google C++* tool, we have also verified the following third party tools and code:

- *Nanopb*, generates C-code and it does not require any dynamic memory allocations.
- *Protobuf-net*, a Google Protobuf .NET library.
- *Protobuf-csharp*, a Google Protobuf .NET library, the C# API is similar to the Google C++ API.



Note

Note that the code mentioned above is open source, which means that you have to check the license that the code is allowed to be used in your product.

EGM sensor protocol description

The EGM sensor protocol is not a request/response protocol, the sensor can send data at any frequency after the sensor gets the first message from the robot.

The EGM sensor protocol has two main data structures, *EgmRobot* and *EgmSensor*. *EgmRobot* is sent from the robot and *EgmSensor* is sent from the sensor. All message fields in both the data structures are defined as optional which means that a field may or may not be present in a message. Applications using *Google Protocol Buffers* must check if optional fields are present or not.

The *EgmHeader* is common for both *EgmRobot* and *EgmSensor*.

```
message EgmHeader
{
  optional uint32 seqno = 1; // sequence number (to be able to find
    lost messages)
  optional uint32 tm = 2; // time stamp in milliseconds

  enum MessageType {
    MSGTYPE_UNDEFINED = 0;
    MSGTYPE_COMMAND = 1; // for future use
    MSGTYPE_DATA = 2; // sent by robot controller
    MSGTYPE_CORRECTION = 3; // sent by sensor
  }

  optional MessageType mtype = 3 [default = MSGTYPE_UNDEFINED];
}
```

Variable	Description
seqno	Sequence number. Applications shall increase the sequence number by one for each message they send. It makes it possible to check for lost messages in a series of messages.

Continues on next page

9 Engineering tools

9.3.3 The EGM sensor protocol

Continued

Variable	Description
tm	Timestamp in milliseconds. (Can be used for monitoring of delays).
mtype	Message type. Shall be set to MSGTYPE_CORRECTION by the sensor, and is set to MSGTYPE_DATA by the robot controller.

The Google protobuf data structure can include the *repeated* element, i.e. a list of elements of the same type. The *repeated* element count is a maximum of six elements in the EGM sensor protocol.

See the *egm.proto* file for a description of `EgmRobot` and `EgmSensor`, [UdpUc code examples on page 334](#).

How to build an EGM sensor communication endpoint using .Net

This guide assumes that you build and compile using Visual Studio and are familiar with its operation.

Here is a short description on how to install and create a simple test application using *protobuf-csharp-port*.

	Action
1	Download protobuf-csharp binaries from: https://code.google.com/p/protobuf-csharp-port/ .
2	Unpack the zip-file.
3	Copy the <i>egm.proto</i> file to a sub catalogue where protobuf-csharp was un-zipped, e.g. <code>~\protobuf-csharp\tools\egm</code> .
4	Start a Windows console in the tools directory, e.g. <code>~\protobuf-csharp\tools</code> .
5	Generate an EGM C# file (<i>egm.cs</i>) from the <i>egm.proto</i> file by typing in the Windows console: <code>protogen .\egm\egm.proto --proto_path=.\egm</code>
6	Create a C# console application in Visual Studio. Create a C# Windows console application in Visual Studio, e.g. <i>EgmSensorApp</i> .
7	Install NuGet, in Visual Studio, click Tools and then Extension Manager . Go to Online , find the <i>NuGet Package Manager extension</i> and click Download .
8	Install protobuf-csharp in the solution for the C# Windows Console application using NuGet. The solution has to be open in Visual Studio.
9	In Visual Studio select, Tools , Nuget Package Manager , and Package Manager Console . Type <code>PM>Install-Package Google.ProtocolBuffers</code>
10	Add the generated file <i>egm.cs</i> to the Visual Studio project (add existing item).
11	Copy the example code into the Visual Studio Windows Console application file (<i>EgmSensorApp.cpp</i>) and then compile, link and run.

How to build an EGM sensor communication endpoint using C++

When building using C++ there are no other third party libraries needed.

C++ is supported by Google. It can be a bit tricky to build the Google tools in Windows but here is a guide on how to build protobuf for Windows.

Continues on next page

Use the following procedure when you have built *libprotobuf.lib* and *protoc.exe*:

	Action
1	Run Google protoc to generate access classes, <code>protoc --cpp_out=. egm.proto</code>
2	Create a win32 console application
3	Add Protobuf source as include directory.
4	Add the generated <i>egm.pb.cc</i> file to the project, exclude the file from precompile headers.
5	Copy the code from the <i>egm-sensor.cpp</i> file, see UdpUc code examples on page 334 .
6	Compile and run.

Configuring UdpUc devices

UdpUc communicates with a maximum of eight devices over Udp. The devices act as servers, and the robot controller acts as a client. It is the robot controller that initiates the connection to the sensor.

System parameters

This is a brief description of the parameters used when configuring a device. For more information about the parameters, see *Technical reference manual - System parameters*.

These parameters belong to the type *Transmission Protocol* in topic *Communication*.

Parameter	Description
<i>Name</i>	The name of the transmission protocol. For example <i>EGMsensor</i> .
<i>Type</i>	The type of transmission protocol. It has to be <i>UDPUC</i> .
<i>Serial Port</i>	The name of the serial port that will be used for the sensor. This refers to the parameter <i>Name</i> in the type <i>Serial Port</i> . For IP based transmission protocols (i.e. <i>Type</i> has value TCP/IP, SOCKDEV, LTAPPTCP or UDPUC), <i>Serial Port</i> is not used and has the value N/A.
<i>Remote Address</i>	The IP address of the remote device.
<i>Remote Port Number</i>	The IP port number that the remote device has opened.

Configuration example

The device which provides the input data for EGM, has to be configured as an UdpUc device in the following way:

Name	Type	Serial Port	Remote Address	Remote Port Number
UCdevice	UDPUC	N/A	192.168.10.20	6510

After this configuration change, the controller has to be restarted. Now the device can be used by EGM to guide a robot. For more information, see [Using EGM Position Guidance with an UdpUc device on page 325](#).

9.3.4 System parameters

About the system parameters

This is a brief description of the system parameters used by *Externally Guided Motion*. For more information about the parameters, see *Technical reference manual - System parameters*.

Type External Motion Interface Data

The system parameters used by *Externally Guided Motion* belong to the type *External Motion Interface Data* in topic *Motion*.

Parameter	Description
<i>Name</i>	The name of the external motion interface data. This name is referenced by the parameter <i>ExtConfigName</i> in the RAPID instructions <i>EGMSetupAI</i> , <i>EGMSetupAO</i> , <i>EGMSetupGI</i> , and <i>EGMSetupUC</i> .
<i>Level</i>	External motion interface level determines the system level at which the corrections are applied. Level 0 corresponds to raw corrections, added just before the servo controllers. Level 1 applies extra filtering on the correction, but also introduces some extra delays and latency. Level 2 has to be used for path correction.
<i>Do Not Restart After Motors Off</i>	Determines if the external motion interface execution should automatically restart after the controller has been in the motors off state, for instance after emergency stop.
<i>Return to Programmed Position when Stopped</i>	Determines if axes currently running external motion interface should return to the programmed position, when program execution is stopped. If <i>False</i> , axes will stop in their current position. If <i>True</i> , axes will move to the programmed finepoint.
<i>Default Ramp Time</i>	Defines the default total time for stopping external motion interface movements when external motion interface execution is stopped. The value will be used to determine how fast the speed contribution from external motion should be ramped to zero when program execution is stopped, and how fast axes return to the programmed position if <i>Return to Programmed Position when Stopped</i> is <i>True</i> .
<i>Default Proportional Position Gain</i>	Defines the default proportional gain of the external motion interface position feedback control. For more information, see Configuration on page 314 .
<i>Default Low Pass Filter Bandwidth</i>	Defines the default bandwidth of the low-pass filter used to filter the speed contribution from the external motion interface execution. For more information, see Configuration on page 314 .

9.3.5 RAPID components

About the RAPID components

This is an overview of all instructions, functions, and data types in *Externally Guided Motion*.

For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instructions

Instructions	Description
EGMActJoint	EGMActJoint activates a specific EGM process and defines static data for the sensor guided joint movement, i.e. data that is not changed frequently between different EGM movements.
EGMActMove	EGMActMove is used to activate a specific EGM process and defines static data for the movement with path correction, i.e. data that is not changed frequently between different EGM path correction movements.
EGMActPose	EGMActPose activates a specific EGM process and defines static data for the sensor guided pose movement, i.e. data that is not changed frequently between different EGM movements.
EGMGetId	EGMGetId is used to reserve an EGM identity (EGMId). That identity is then used in all other EGM RAPID instructions and functions to identify a certain EGM process connected to the RAPID motion task from which it is used. An egmid is identified by its name, i.e. a second or third call of EGMGetId with the same egmid will neither reserve a new EGM process nor change its content.
EGMMoveC	EGMMoveC is used to move the tool center point (TCP) circularly to a given destination with path correction. During the movement the orientation normally remains unchanged relative to the circle.
EGMMoveL	EGMMoveL is used to move the tool center point (TCP) linearly to a given destination with path correction. When the TCP is to remain stationary then this instruction can also be used to reorient the tool.
EGMReset	EGMReset resets a specific EGM process (EGMId), i.e. the reservation is canceled.
EGMRunJoint	EGMRunJoint performs a sensor guided joint movement from a fine point for a specific EGM process (EGMId) and defines which joints will be moved.
EGMRunPose	EGMRunPose performs a sensor guided pose movement from a fine point for a specific EGM process (EGMId) and defines which directions and orientations will be changed.
EGMSetupAI	EGMSetupAI is used to set up analog input signals for a specific EGM process (EGMId) as the source for position destination values to which the robot (plus up to 6 additional axis) is to be guided.
EGMSetupAO	EGMSetupAO is used to set up analog output signals for a specific EGM process (EGMId) as the source for position destination values to which the robot, and up to 6 additional axis, is to be guided.
EGMSetupGI	EGMSetupGI is used to set up group input signals for a specific EGM process (EGMId) as the source for position destination values to which the robot, and up to 6 additional axis, is to be guided.

Continues on next page

9 Engineering tools

9.3.5 RAPID components

Continued

Instructions	Description
EGMSetupLTAPP	EGMSetupLTAPP is used to set up an <i>LTAPP</i> protocol for a specific EGM process (EGMid) as the source for path corrections.
EGMSetupUC	EGMSetupUC is used to set up a UdpUc device for a specific EGM process (EGMid) as the source for position destination values to which the robot, and up to 6 additional axis, are to be guided. The position may be given in joints, for EGMRunJoint, or in cartesian format for EGMRunPose.
EGMStop	EGMStop stops a specific EGM process (EGMid).

Functions

Functions	Description
EGMGetState	EGMGetState retrieves the state of an EGM process (EGMid).

Data types

Data types	Description
egmframetype	egmframetype is used to define the frame types for corrections and sensor measurements in EGM.
egmident	egmident identifies a specific EGM process.
egm_minmax	egm_minmax is used to define the convergence criteria for EGM to finish.
egmstate	egmstate is used to define the state for corrections and sensor measurements in EGM.
egmstopmode	egmstopmode is used to define the stop modes for corrections and sensor measurements in EGM.

9.3.6 RAPID code examples

9.3.6.1 Using EGM Position Guidance with an UdpUc device

Description

The device which provides the input data for EGM, first has to be configured as an UdpUc device. See [Configuring UdpUc devices on page 321](#).

Now the device can be used by EGM to guide a robot. A simple example is the following:

Example

```

MODULE EGM_test
VAR egmident egmID1;
VAR egmstate egmSt1;

! limits for cartesian convergence: +-1 mm
CONST egm_minmax egm_minmax_lin1:=[-1,1];
! limits for orientation convergence: +-2 degrees
CONST egm_minmax egm_minmax_rot1:=[-2,2];

! Start position
CONST jointtarget
  jpos10:=[[0,0,0,0,40,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
! Used tool
TASK PERS tooldata tFroniusCMT:=[TRUE,[[12.3313,-0.108707,416.142],
  [0.903899,-0.00320735,0.427666,0.00765917]],
  [2.6,[-111.1,24.6,386.6],[1,0,0,0],0,0,0.072]];
! corr-frame: wobj, sens-frame: wobj
TASK PERS wobjdata wobj_EGM1:=[FALSE,TRUE,"",
  [[150,1320,1140],[1,0,0,0]], [[0,0,0],[1,0,0,0]]];
! Correction frame offset: none
VAR pose corr_frame_offs:=[[0,0,0],[1,0,0,0]];

PROC main()
! Move to start position. Fine point is demanded.
MoveAbsJ jpos10\NoEOffs, v1000, fine, tFroniusCMT;
testuc;
ENDPROC

PROC testuc()
EGMReset egmID1;
EGMGetId egmID1;

egmSt1:=EGMGetState(egmID1);
TPWrite "EGM state: "\Num:=egmSt1;

IF egmSt1 <= EGM_STATE_CONNECTED THEN
! Set up the EGM data source: UdpUc server using device "EGMsensor:"
and

```

Continues on next page

Continued

```
! configuration "default"
EGMSetupUC ROB_1, egmID1, "default", "EGMsensor:"\pose;
ENDIF

! Correction frame is the World coordinate system and the sensor
  measurements are relative
! to the tool frame of the used tool (tFroniusCMT)
EGMActPose egmID1\Tool:=tFroniusCMT, corr_frame_offs,
  EGM_FRAME_WORLD, tFroniusCMT.tframe, EGM_FRAME_TOOL
  \x:=egm_minmax_lin1 \y:=egm_minmax_lin1 \z:=egm_minmax_lin1
  \rx:=egm_minmax_rot1 \ry:=egm_minmax_rot1 \rz:=egm_minmax_rot1
  \LpFilter:=20;

! Run: the convergence condition has to be fulfilled during 2
  seconds before RAPID

! executeion continues to the next instruction
EGMRunPose egmID1, EGM_STOP_HOLD \x \y \z \CondTime:=2
  \RampInTime:=0.05;

egmSt1:=EGMGetState(egmID1);
IF egmSt1 = EGM_STATE_CONNECTED THEN
TPWrite "Reset EGM instance egmID1";
EGMReset egmID1;
ENDIF
ENDPROC
ENDMODULE
```

9.3.6.2 Using EGM Position Guidance with signals as input

Description

All signals that are used together with EGM has to be defined in the I/O configuration of the system. I.e. the signals that are set up with `EGMSetupAI`, `EGMSetupAO`, or `EGMSetupGI`. After that, the signals can be used by EGM to guide a robot.

The following RAPID program example uses analog output signals as input. The main reason for analog output signals is, that they are easier to simulate than analog input signals. In a real application group input signals and analog input signals might be more common.

In the example we also set the analog output signals to a constant value before the `EGMRun` instruction just for simplicity. Normally an external device will update the signal values to give the desired robot positions.

Example

```
MODULE EGM_test
VAR egmident egmID1;
VAR egmident egmID2;

CONST egm_minmax egm_minmax_lin1:=[-1,1];
CONST egm_minmax egm_minmax_rot1:=[-2,2];
CONST egm_minmax egm_minmax_joint1:=[-0.1,0.1];

CONST robtarget p20:=[[150,1320,1140],
[0.000494947,0.662278,-0.749217,-0.00783173], [0,0,-1,0],
[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST robtarget p30:=[[114.50,1005.42,1410.38],
[0.322151,-0.601023,0.672381,0.287914], [0,0,-1,0],
[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
CONST jointtarget
jpos10:=[[0,0,0,0,35,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];

CONST pose posecor:=[[1200,400,900],[1,0,0,0]];
CONST pose posesens:=[[12.3313,-0.108707,416.142],
[0.903899,-0.00320735,0.427666,0.00765917]];

! corr-frame: world, sens-frame: world
VAR pose posecor0:=[[0,0,0],[1,0,0,0]];
VAR pose posesen0:=[[0,0,0],[1,0,0,0]];

TASK PERS tooldata tFroniusCMT:=[TRUE,[[12.3313,-0.108707,416.142],
[0.903899,-0.00320735,0.427666,0.00765917]],
[2.6,[-111.1,24.6,386.6],[1,0,0,0],0,0,0.072]];
TASK PERS loaddata load1:=[5,[0,1,0],[1,0,0,0],0,0,0];
! corr-frame: wobj, sens-frame: wobj
TASK PERS wobjdata
wobj_EGM1:=[FALSE,TRUE,"",[[150,1320,1140],[1,0,0,0]],
[[0,0,0],[1,0,0,0]]];
VAR pose posecor1:=[[0,0,0],[1,0,0,0]];
VAR pose posesen1:=[[0,0,0],[1,0,0,0]];
```

Continues on next page

9 Engineering tools

9.3.6.2 Using EGM Position Guidance with signals as input

Continued

```
TASK PERS wobjdata
    wobj_EGM2:=[FALSE,TRUE,"",[0,1000,1000],[1,0,0,0]],
    [[0,0,0],[1,0,0,0]];
VAR pose posecor2:=[[150,320,0],[1,0,0,0]];
VAR pose posesen2:=[[150,320,0],[1,0,0,0]];

PROC main()
MoveAbsJ jpos10\NoEOffs, v1000, fine, tFroniusCMT;
testAO;
ENDPROC

PROC testAO()
! Get two different EGM identities. They will be used for two
different eGM setups.
EGMGetId egmID1;
EGMGetId egmID2;

! Set up the EGM data source: Analog output signals and
configuration "default"
! One guidance using Pose mode and one using Joint mode
EGMSetupAO ROB_1, egmID1, "default" \Pose \aoR1x:=ao_MoveX
\aoR2y:=ao_MoveY \aoR3z:=ao_MoveZ \aoR5ry:=ao_RotY
\aoR6rz:=ao_RotZ;
EGMSetupAO ROB_1, egmID2, "default" \Joint \aoR1x:=ao_MoveX
\aoR2y:=ao_MoveY \aoR3z:=ao_MoveZ \aoR4rx:=ao_RotX
\aoR5ry:=ao_RotY \aoR6rz:=ao_RotZ;

! Move to the starting point - fine point is needed.
MoveJ p30, v1000, fine, tool0;
! Set the signals
SetAO ao_MoveX, 150;
SetAO ao_MoveY, 1320;
SetAO ao_MoveZ, 900;
! Correction frame is the World coordinate system and the sensor
measurements are also relative to the world frame
! No offset is defined (posecor0 and posesen0)
EGMActPose egmID1 \Tool:=tFroniusCMT \WObj:=wobj0 \TLoad:=load1,
posecor0, EGM_FRAME_WORLD, posesen0, EGM_FRAME_WORLD
\X:=egm_minmax_lin1 \Y:=egm_minmax_lin1 \Z:=egm_minmax_lin1
\RX:=egm_minmax_rot1 \RY:=egm_minmax_rot1 \RZ:=egm_minmax_rot1
\LpFilter:=20 \SampleRate:=16 \MaxPosDeviation:=1000;
! Run: keep the end position without returning to the start position
EGMRunPose egmID1,
EGM_STOP_HOLD\X\Y\Z\RampInTime:=0.05\PosCorrGain:=16;

! Move to the starting point - fine point is needed.
MoveJ p20, v1000, fine, tFroniusCMT;
! Set the signals
SetAO ao_MoveX, 150;
SetAO ao_MoveY, 1320;
SetAO ao_MoveZ, 1100;
```

Continues on next page

9.3.6.2 Using EGM Position Guidance with signals as input

Continued

```

! Run with the same frame definitions: ramp down to the start
  position after having reached
! the EGM end position
EGMRunPose egmID1,
  EGM_STOP_RAMP_DOWN\X\Y\Z\RampInTime:=0.05\PosCorrGain:=16;

! Move to the starting point - fine point is needed.
MoveJ p30, v1000, fine, tool0;
! Set the signals
SetAO ao_MoveX, 50;
SetAO ao_MoveY, -20;
SetAO ao_MoveZ, -20;
! Correction frame is the Work object wobj_EGM1 and the sensor
  measurements are also
! relative to the same work object. No offset is defined (posecor1
  and posesen1)
EGMActPose egmID1 \Tool:=tFroniusCMT \WObj:=wobj_EGM1 \TLoad:=load1,
  posecor1, EGM_FRAME_WOBJ, posesen1, EGM_FRAME_WOBJ
  \X:=egm_minmax_lin1 \Y:=egm_minmax_lin1 \Z:=egm_minmax_lin1
  \RX:=egm_minmax_rot1 \RY:=egm_minmax_rot1 \RZ:=egm_minmax_rot1
  \LpFilter:=20;
! Run: keep the end position without returning to the start position
EGMRunPose egmID1,
  EGM_STOP_HOLD\X\Y\Z\RampInTime:=0.05\PosCorrGain:=16;

! Move to the starting point - fine point is needed.
MoveJ p20, v1000, fine, tFroniusCMT;
! Set the signals
SetAO ao_MoveX, 0;
SetAO ao_MoveY, 0;
SetAO ao_MoveZ, 0;
! Correction frame is the Work object wobj_EGM2 and the sensor
  measurements are also
! relative to the same work object. This time an offset is defined
  for the correction frame
! (posecor2), and for the sensor frame (posesen2)
EGMActPose egmID1 \Tool:=tFroniusCMT \WObj:=wobj_EGM2 \TLoad:=load1,
  posecor2, EGM_FRAME_WOBJ, posesen2, EGM_FRAME_WOBJ
  \X:=egm_minmax_lin1 \Y:=egm_minmax_lin1 \Z:=egm_minmax_lin1
  \RX:=egm_minmax_rot1 \RY:=egm_minmax_rot1 \RZ:=egm_minmax_rot1
  \LpFilter:=20;
! Run: keep the end position without returning to the start position
EGMRunPose egmID1,
  EGM_STOP_HOLD\X\Y\Z\RampInTime:=0.05\PosCorrGain:=16;

! Move to the starting point - fine point is needed.
MoveJ p20, v1000, fine, tFroniusCMT;
! Set the signals
SetAO ao_MoveX, 0;
SetAO ao_MoveY, 0;
SetAO ao_MoveZ, 0;

```

Continues on next page

9 Engineering tools

9.3.6.2 Using EGM Position Guidance with signals as input

Continued

```
! Correction frame is of tool type and the sensor measurements are
  relative to the work
! object wobj_EGM2. This time an offset is defined for the
  correction frame (posecor2), and
! for the sensor frame (posesen2)
EGMActPose egmID1 \Tool:=tFroniusCMT \WObj:=wobj_EGM2, posecor2,
  EGM_FRAME_TOOL, posesen2, EGM_FRAME_WOBJ \x:=egm_minmax_lin1
  \y:=egm_minmax_lin1 \z:=egm_minmax_lin1 \rx:=egm_minmax_rot1
  \ry:=egm_minmax_rot1 \rz:=egm_minmax_rot1 \LpFilter:=20;
EGMRunPose egmID1,
  EGM_STOP_HOLD\x\y\z\RampInTime:=0.05\PosCorrGain:=16;

! Move to the starting point - fine point is needed.
MoveJ p20, v1000, fine, tFroniusCMT\TLoad:=load1;
! Set the signals
SetAO ao_MoveX, 150;
SetAO ao_MoveY, 1320;
SetAO ao_MoveZ, 1100;
! Same as last, but with tool0 and wobj0
EGMActPose egmID1, posecor2, EGM_FRAME_TOOL, posesen2,
  EGM_FRAME_WOBJ \x:=egm_minmax_lin1 \y:=egm_minmax_lin1
  \z:=egm_minmax_lin1 \rx:=egm_minmax_rot1 \ry:=egm_minmax_rot1
  \rz:=egm_minmax_rot1 \LpFilter:=20;
! Run: keep the end position without returning to the start position
EGMRunPose egmID1,
  EGM_STOP_HOLD\x\y\z\RampInTime:=0.05\PosCorrGain:=16;

! Move to the starting point - fine point is needed.
MoveJ p20, v1000, fine, tFroniusCMT\TLoad:=load1;
! Set the signals
SetAO ao_MoveX, 70;
SetAO ao_MoveY, -5;
SetAO ao_MoveZ, 0;
SetAO ao_RotX, 0;
SetAO ao_RotY, 0;
SetAO ao_RotZ, 0;
! Joint guidance for joints 2-6
EGMActJoint egmID2 \J2:=egm_minmax_joint1 \J3:=egm_minmax_joint1
  \J4:=egm_minmax_joint1 \J5:=egm_minmax_joint1
  \J6:=egm_minmax_joint1 \LpFilter:=20;
! Run: keep the end position without returning to the start position
EGMRunJoint egmID2, EGM_STOP_HOLD \J2 \J3 \J4 \J5 \J6 \CondTime:=0.1
  \RampInTime:=0.05 \PosCorrGain:=16;

EGMReset egmID1;
EGMReset egmID2;
ENDPROC
ENDMODULE
```

9.3.6.3 Using EGM Path Correction with different protocol types

Description

This example contains examples for different sensor and protocol types. The basic RAPID program structure is the same for all of them and they use the same external motion data configuration.

Example

```

MODULE EGM_PATHCORR
! Used tool
PERS tooldata tEGM:=[TRUE,[[148.62,0.25,326.31],
    [0.833900724,0,0.551914471,0]], [1,[0,0,100],
    [1,0,0,0],0,0,0]];
! Sensor tool, has to be calibrated
PERS tooldata
    tLaser:=[TRUE,[[148.619609537,50.250017146,326.310337954],
    [0.390261856,-0.58965743,-0.58965629,0.390263064]],
    [1,[-0.920483747,-0.000000536,-0.390780849],
    [1,0,0,0],0,0,0]];
! Displacement used
VAR pose PP:=[[0,-3,2],[1,0,0,0]];
VAR egmident egmId1;

! Protocol: LTAPP
! Example for a look ahead sensor, e.g. Laser Tracker
PROC Part_2_EGM_OT_Pth_1()
    EGMGetId egmId1;
    ! Set up the EGM data source: LTAPP server using device "Optsim",
    ! configuration "pathCorr", joint type 1 and look ahead sensor.
    EGMSetupLTAPP ROB_1, egmId1, "pathCorr", "OptSim", 1\LATR;
    ! Activate EGM and define the sensor frame.
    ! Correction frame is always the path frame.
    EGMActMove egmId1, tLaser.tframe\SampleRate:=50;
    ! Move to a suitable approach position.
    MoveJ p100,v1000,z10,tEGM\WObj:=wobj0;
    MoveL p110,v1000,z100,tEGM\WObj:=wobj0;
    MoveL p120,v1000,z100,tEGM\WObj:=wobj0;
    ! Activate displacement (not necessary but possible)
    PDispSet PP;
    ! Move to the start point. Fine point is demanded.
    MoveL p130, v10, fine, tEGM\WObj:=wobj0;
    ! movements with path corrections.
    EGMMoveL egmId1, p140, v10, z5, tEGM\WObj:=wobj0;
    EGMMoveL egmId1, p150, v10, z5, tEGM\WObj:=wobj0;
    EGMMoveC egmId1, p160, p165, v10, z5, tEGM\WObj:=wobj0;
    ! Last path correction movement has to end with a fine point.
    EGMMoveL egmId1, p170, v10, fine, tEGM\WObj:=wobj0;
    ! Move to a safe position after path correction.
    MoveL p180,v1000,z10,tEGM\WObj:=wobj0;
    ! Release the EGM identity for reuse.

```

Continues on next page

9 Engineering tools

9.3.6.3 Using EGM Path Correction with different protocol types

Continued

```
    EGMReset egmId1;
ENDPROC

! Protocol: LTAPP
! Example for an at point sensor, e.g. Weldguide
PROC Part_2_EGM_WG_Pth_1()
    EGMGetId egmId1;
    ! Set up the EGM data source: LTAPP server using device "wglsim",
    ! configuration "pathCorr", joint type 1 and at point sensor.
    EGMSetupLTAPP ROB_1, egmId1, "pathCorr", "wglsim", 1\APTR;
    ! Activate EGM and define the sensor frame,
    ! which is the tool frame for at point trackers.
    ! Correction frame is always the path frame.
    EGMActMove egmId1, tEGM.tframe\SampleRate:=50;
    ! Move to a suitable approach position.
    MoveJ p100,v1000,z10,tEGM\WObj:=wobj0;
    MoveL p110,v1000,z100,tEGM\WObj:=wobj0;
    MoveL p120,v1000,fine,tEGM\WObj:=wobj0;
    ! Activate displacement (not necessary but possible)
    PDispSet PP;
    ! Move to the start point. Fine point is demanded.
    MoveL p130, v10, fine, tEGM\WObj:=wobj0;
    ! movements with path corrections.
    EGMMoveL egmId1, p140, v10, z5, tEGM\WObj:=wobj0;
    EGMMoveL egmId1, p150, v10, z5, tEGM\WObj:=wobj0;
    EGMMoveC egmId1, p160, p165, v10, z5, tEGM\WObj:=wobj0;
    ! Last path correction movement has to end with a fine point.
    EGMMoveL egmId1, p170, v10, fine, tEGM\WObj:=wobj0;
    ! Move to a safe position after path correction.
    MoveL p180,v1000,z10,tEGM\WObj:=wobj0;
    ! Release the EGM identity for reuse.
    EGMReset egmId1;
ENDPROC

! Protocol: UdpUc
! Example for an at point sensor, e.g. Weldguide
PROC Part_2_EGM_UDPUC_Pth_1()
    EGMGetId egmId1;
    EGMSetupUC ROB_1, egmId1, "pathCorr", "UCdevice"\PathCorr\APTR;
    EGMActMove egmId1, tEGM.tframe\SampleRate:=50;
    ! Move to a suitable approach position.
    MoveJ p100,v1000,z10,tEGM\WObj:=wobj0;
    MoveL p110,v1000,z100,tEGM\WObj:=wobj0;
    MoveL p120,v1000,fine,tEGM\WObj:=wobj0;
    ! Activate displacement (not necessary but possible)
    PDispSet PP;
    ! Move to the start point. Fine point is demanded.
    MoveL p130, v10, fine, tEGM\WObj:=wobj0;
    ! movements with path corrections.
    EGMMoveL egmId1, p140, v10, z5, tEGM\WObj:=wobj0;
    EGMMoveL egmId1, p150, v10, z5, tEGM\WObj:=wobj0;
```

Continues on next page

9.3.6.3 Using EGM Path Correction with different protocol types

Continued

```
EGMMoveC egmId1, p160, p165, v10, z5, tEGM\WObj:=wobj0;  
! Last path correction movement has to end with a fine point.  
EGMMoveL egmId1, p170, v10, fine, tEGM\WObj:=wobj0;  
! Move to a safe position after path correction.  
MoveL p180,v1000,z10,tEGM\WObj:=wobj0;  
! Release the EGM identity for reuse.  
EGMReset egmId1;  
ENDPROC  
ENDMODULE
```

9.3.7 UdpUc code examples

File locations

The following code examples are available in the RobotWare distribution.

File	Description
<i>egm-sensor.cs</i>	Example using protobuf-csharp-port
<i>egm-sensor.cpp</i>	Example using Google protocol buffers C++
<i>egm.proto</i>	The <i>egm.proto</i> file defines the data contract between the robot and the sensor.

The files can be obtained from the PC or the IRC5 controller.

- In the RobotWare installation folder in RobotStudio: ...\\RobotPackages\\RobotWare_RPK_<version>\\utility\\Template\\EGM\\
- On the IRC5 Controller:
<SystemName>\\PRODUCTS\\<RobotWare_xx.xx.xxxx>\\utility\\Template\\EGM\\



Note

Navigate to the RobotWare installation folder from the RobotStudio **Add-Ins** tab, by right-clicking on the installed RobotWare version in the **Add-Ins** browser and selecting **Open Package Folder**.

9.4 Robot Reference Interface [included in 689-1]

9.4.1 Introduction to Robot Reference Interface

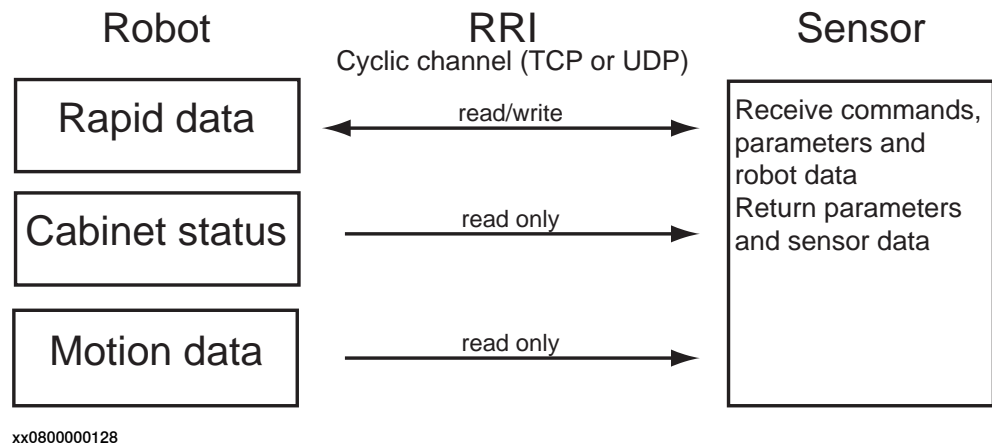
Introduction

Robot Reference Interface is included in the RobotWare option *Externally Guided Motion*.

Robot Reference Interface supports data exchange on the cyclic channel. It provides the possibility to periodically send planned and actual robot position data from the robot controller, as well as the exchange of other RAPID variables from and to the robot controller. The message contents are represented in XML format and are configured using appropriate sensor configuration files.

Robot Reference Interface

The cyclic communication channel (TCP or UDP) can be executed in the high-priority network environment of the IRC5 Controller which ensures a stable data exchange up to 250Hz.



9 Engineering tools

9.4.2.1 Connecting the communication cable

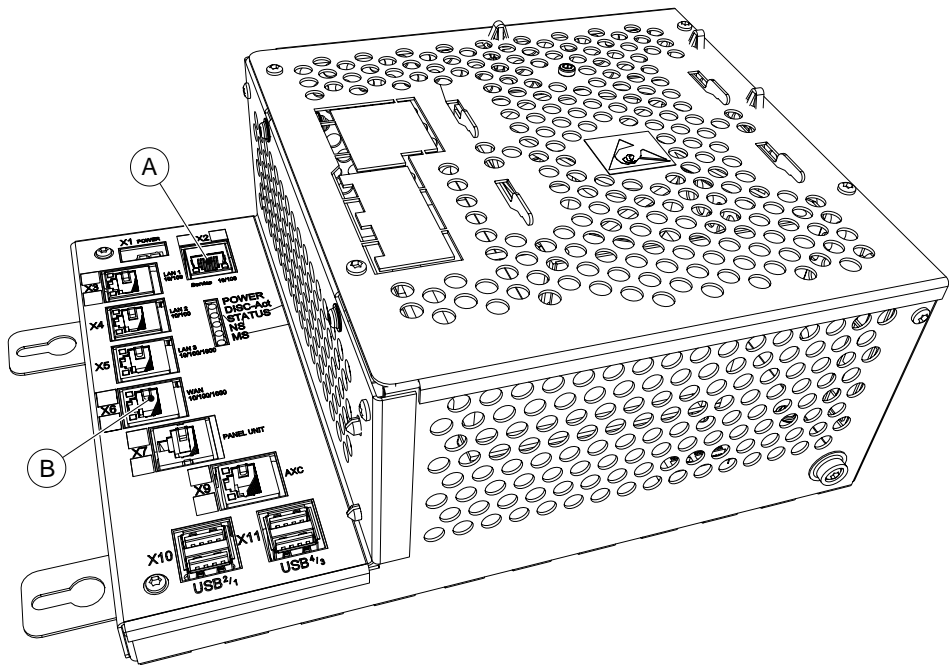
9.4.2 Installation

9.4.2.1 Connecting the communication cable

Overview


This section describes where to connect the communication cable on the controller. For further instructions, see the corresponding product manual for your robot system.

Location



xx1300000609

A	Service port on the computer unit (connected to the service port on the controller)
B	WAN port on the computer unit

	Action	Note
1	Use one of these two connections (A or B).	<div> Note</div> <div>The service connection can only be used if it is free.</div>

9.4.2.2 Prerequisites

Overview

This section describes the prerequisites for using *Robot Reference Interface*.

UDP/IP or TCP IP

Robot Reference Interface supports the communication over the standard IP protocols UDP or TCP.

Recommendations

The delay in the overall communication mostly depends on the topology of the employed network. In a switched network the transmission will be delayed due to buffering of the messages in the switches. In a parallel network collisions with multiple communication partners will lead to messages being resent.

Therefore we recommended using a dedicated Ethernet link between the external system and the robot controller to provide the required performance for real-time applications. *Robot Reference Interface* can be used to communicate with any processor-based devices, that support IP via Ethernet and can serialize data into XML format.

9 Engineering tools

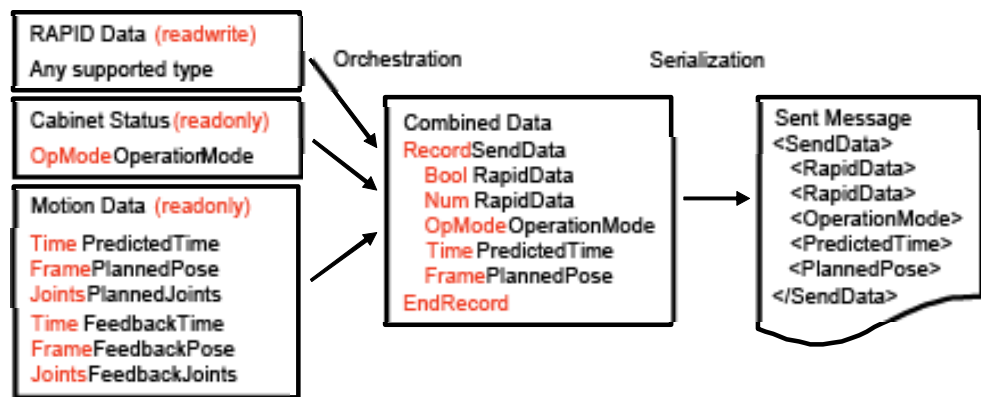
9.4.2.3 Data orchestration

9.4.2.3 Data orchestration

Overview

The outgoing message can be combined from any data from the RAPID level and internal data from the cabinet and motion topic. The orchestration of the data is defined in the device configuration by setting the Link attribute of internally linked data to *Intern*.

Illustration



xx0800000178

Data from the Controller topic

Name	Type	Description	Comment
OperationMode	OpMode	Operation mode of the robot.	The mapping of the members for the Op-Mode type can be defined in the configuration file.

Data from the Motion topic

Name	Type	Description	Comment
FeedbackTime	Time	Time stamp for the robot position from drive feedback.	There is a delay of approximately 8ms.
FeedbackPose	Frame	Robot TCP calculated from drive feedback.	Current tool and workobject are used for calculation.
FeedbackJoints	Joints	Robot joint values gathered from drive feedback.	
PredictedTime	Time	Timestamp for planned robot TCP position and joint values.	Prediction time from approximately 24ms to 60ms depending on robot type.
PlannedPose	Frame	Planned robot TCP.	Current tool and workobject are used for calculation.
PlannedJoints	Joints	Planned robot joint values.	

9.4.2.4 Supported data types

Overview

This section contains a short description of the *Robot Reference Interface* supported data types, for more detailed information about the supported data types see [References on page 11](#).

Data types

Robot Reference Interface supports the following simple data types:

Data type	Description	RAPID type mapping
bool	Boolean value.	bool
real	Single precision, floating point value.	num
time	Time in seconds expressed as floating point value.	num
string	String with max length of 80 characters.	string
frame	Cartesian position and orientation in Euler Angles (Roll-Pitch-Yaw).	pose
joint	Robot joint values.	robjoint

In addition, user-defined records can also be transferred from the external system to the robot controller, which are composed from the supported simple data types. User defined record types must be specified in the configuration file of the external device. See [Device configuration on page 345](#) for a description on how to create user-defined record types.

9 Engineering tools

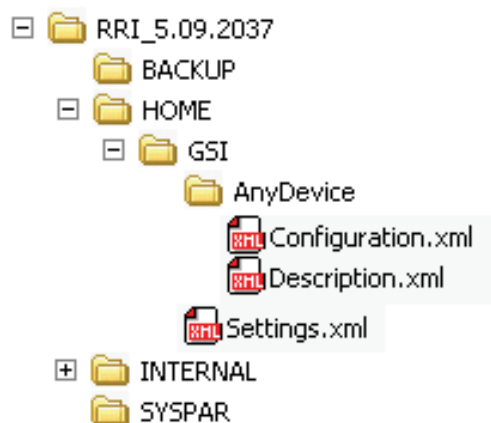
9.4.3.1 Interface configuration

9.4.3 Configuration

9.4.3.1 Interface configuration

Configuration files

The configuration and settings files for the interface must be located in the folder HOME/GSI. This ensures that the configuration files are included in system backups.



xx0800000177

Related information

For more detailed information of the *Settings.xml* file see [Interface settings on page 341](#).

For more detailed information of the *Description.xml* file see [Device description on page 342](#).

For more detailed information of the *Configuration.xml* file see [Device configuration on page 345](#).

9.4.3.2 Interface settings

Overview

This section describes the use of the xml file *Settings.xml*.

Settings.xml

The settings file *Settings.xml* contains the general settings for the GSI interface. It is located in the folder HOME/GSI. For the option *Robot Reference Interface* this file refers to a list of all communication clients for external systems installed in the controller. The *Settings.xml* file can be defined according to the XML schema *Settings.xsd*.

Example

For each communication client installed on the controller, the file *Settings.xml* must contain a *Client* entry in the *Clients* section. The *Convention* attribute identifies the protocol convention used by the client, for the *Robot Reference Interface* option only CDP is supported. The *Name* attribute identifies the name of the client and also specifies the folder with the device related configuration files.

```
<?xml version="1.0" encoding="UTF-8"?>
<Settings>
  <Clients>
    <Client Convention="CDP" Name="MySensor" />
  </Clients>
</Settings>
```

CDP stands for *cyclic data protocol* and is the internal name of the protocol, on which *Robot Reference Interface* messages are transferred.

An internal client node of the interface module will be created, which is able to connect to the external system *MySensor* that runs a data server application and can communicate via *Robot Reference Interface* with the robot.

For each sensor system, a subdirectory named with the sensor system identifier, for example *MySensor*, contains further settings.

9.4.3.3 Device description

Overview

This section describes the use of the xml file *Description.xml*.

Description.xml

The device description file Description.xml is located in the corresponding subdirectory of the device. It specifies the general device parameters, network connection and CDP specific communication settings for an installed device. A device description can be defined according to the XML schema Description.xsd.

Example

This is an example of a device description:

```
<?xml version="1.0" encoding="utf-8"?>
<Description>
  <Name>AnyDevice</Name>
  <Convention>CDP</Convention>
  <Type>IntelligentCamera</Type>
  <Class>MachineVision</Class>
  <Network Address="10.49.65.74" Port="Service">
    <Channel Type="Cyclic" Protocol="Udp" Port="3002" />
  </Network>
  <Settings>
    <TimeOut>2000</TimeOut>
    <MaxLost>30</MaxLost>
    <DryRun>false</DryRun>
  </Settings>
</Description>
```

Name

The first section defines the general device parameters. The Name element identifies the name of the device and should correspond to the device name specified in the settings file. It must correspond to the identifier specified for the device descriptor on the RAPID level, because the descriptor name will be used initially to refer to the device in the RAPID instructions.

Element	Attribute	Description	Value	Comment
Name		Device identifier	Any string	Maximum 16 characters

Convention

The Convention element identifies the protocol that should be used by the device, for the *Robot Reference Interface* option only the Cyclic Data Protocol (CDP) is supported.

Element	Attribute	Description	Value	Comment
Convention		Protocol type	CDP	

Continues on next page

Type and Class

The Type and Class elements identifies the device type and class and are currently not validated, therefore they can also contain undefined device types or classes.

Element	Attribute	Description	Value	Comment
Type		Sensor type	Any string	Not validated
Class		Sensor class	Any string	Not validated

Network

The Network section defines the network connection settings for the device. The Address attribute specifies the IP address or host name of the device on the network. The optional Port attribute is used to specify the physical Ethernet port on the controller side that the cable is plugged into. Valid values are *WAN* and *Service*. The attribute can be omitted if the WAN port is used for communication.

Element	Attribute	Description	Value	Comment
Network		Network settings		
	Address	IP address or host name of the device	Any valid IP address or host name	10.49.65.249 DE-L-0328122
	Port	Physical Ethernet port on the controller	WAN Service	Optional. Can be omitted if WAN port is used.

Channel

The Channel element defines the settings for the communication channel between the robot controller and the external device. The Type attribute identifies the channel type, only *Cyclic* is supported by *Robot Reference Interface*.

The Protocol attribute identifies the IP protocol used on the channel, for *Robot Reference Interface* you can specify to use *Tcp* or *Udp*. The Port attribute specifies the logical port number for the channel on the device side.

Element	Attribute	Description	Value	Comment
Channel		Channel settings		
	Type	Channel type	Cyclic	
	Protocol	The IP protocol type	Tcp Udp	
	Port	The logical port number of the channel	uShort	Any available port number on the device, maximum 65535.

Continues on next page

Settings

The Settings section contains communication parameters specific to the CDP protocol. The TimeOut element defines the timeout for not received messages. This element identifies the time until the connection is considered broken and is only needed for bidirectional communication. The MaxLost attribute defines the maximum number of not acknowledged or lost messages allowed. The DryRun element identifies, if the acknowledgement of messages is supervised and can be used to setup an unidirectional communication.

Element	Description	Value	Comment
TimeOut	Time out for communication		Time in milliseconds, a multiple of 4 ms.
MaxLost	Maximum loss of packages allowed	Integer	
DryRun	Interface run mode	Bool	If TRUE, TimeOut and MaxLost will not be checked.

If the element DryRun in the Description.xml is set to FALSE, communication supervision is established on the protocol level of the *Robot Reference Interface*, using the settings for *TimeOut* and *MaxLost*. This supervision requires that each message that is sent out from the robot controller is answered by the connected device. The supervision generates a communication error, if the maximum response time or the maximum number of lost packages is exceeded. Each sent out message has an ID, which needs to be used for the ID in the reply too, to identify the reply message and to detect which packages have been lost. See also the example in section [Transmitted XML messages on page 352](#).

9.4.3.4 Device configuration

Overview

The device configuration file *Configuration.xml* is located in the corresponding subdirectory of the device. It defines the enumerated and complex types used by the device and identifies the available parameters, which can be subscribed for cyclic transmission. The configuration file can be defined according to the XML schema Configuration.xsd. The following document shows a simplified device configuration.

Example

```
<?xml version="1.0" encoding="utf-8"?>
<Configuration>
  <Enums>
    <Enum Name="opmode" Link="Intern">
      <Member Name="ReducedSpeed" Alias="Alias"/>
    </Enum>
  </Enums>
  <Records>
    <Record Name="senddata">
      <Field Name="PlannedPose" Type="Pose" Link="Intern" />
    </Record>
  </Records>
  <Properties>
    <Property Name="DataToSend" Type="senddata" Flag="WriteOnly" />
  </Properties>
</Configuration>
```

Enums

In the Enums section each Enum element defines an enumerated type. The Name attribute of the Enum element specifies the name of the enumerated type, the optional Link attribute identifies if the members of the enumerated type have internal linkage.

Element	Attribute	Descriptions	Value	Comment
Enum	Name	Name of enumerated type	A valid RAPID symbol name	Maximum 16 characters.
	Link	Linkage of members of enumerated type	<i>Intern</i>	Optional. Can be omitted if members only have RAPID linkage.

Continues on next page

9 Engineering tools

9.4.3.4 Device configuration

Continued

Member

Each Member element defines a member element of the enumerated type. The Name attribute specifies the name of the member on the controller side (on RAPID level). The Alias attribute identifies the name of the member on the device side (and in the transmitted message).

Element	Attribute	Descriptions	Value	Comment
Member	Name	Name of enumerated type member	A valid RAPID symbol name	Maximum 16 characters. Valid internal RAPID symbol names. See Data orchestration on page 338 .
	Alias	Alias name of enumerated type member	String	Optional. The alias name is used on the device side and in message

Record

In the Records section each Record element defines a declaration of a complex type. In RAPID this complex type will be represented as a RECORD declaration. The Name attribute identifies the name of the complex type on the controller side. The Alias attribute defines the alias name of the type on the device side and in the message.

Element	Attribute	Descriptions	Value	Comment
Record	Name	Name of the complex type.	A valid RAPID symbol name	Maximum 16 characters.
	Alias	Alias name of complex type.	String	Optional. The alias name is used on the device side and in message.

Field

Each Field element defines a field element of a complex type. The Name attribute identifies the name of the field. The Type attribute identifies the enumerated, complex or simple type associated with the field. The Size attribute defines the size of a multi-dimensional field. The Link attribute identifies if the field has internal linkage.

Element	Attribute	Descriptions	Value	Comment
Field	Name	Name of the complex type field	A valid RAPID symbol name	Maximum 16 characters. Valid internal RAPID symbol names. See Data orchestration on page 338 .
	Type	Data type of the field	All supported data types	Described in section Supported data types on page 339 .
	Size	Dimensions of the field (size of array)	Integer	Optional. Only basic types can be defined as array.
	Link	Linkage of complex type field	<i>Intern</i>	Optional. Can be omitted if field has RAPID linkage.
	Alias	Alias name of complex type field	String	Optional. The alias name is used on device side and in message.

Continues on next page

Properties

In the Properties section each Property element defines a RAPID variable that can be used in the `SiGetCyclic` and `SiSetCyclic` instructions.

Element	Attribute	Descriptions	Value	Comment
Property	Name	Name of the property	An valid RAPID symbol name	Maximum 16 characters.
	Type	Data type of the property	All supported data types	Described in section Supported data types on page 339 .
	Size	Dimension (Size of array)	Integer	Optional. Only basic types can be defined as array.
	Flag	Access Flag	None <i>ReadOnly</i> <i>WriteOnly</i> <i>ReadWrite</i>	Optional. Can be omitted if property is read and write enabled.
	Link	Linkage of property	<i>Intern</i>	Mandatory if field has RAPID linkage.
	Alias	Alias name of the property	String	Optional. The alias name is used on device side and in message.

9 Engineering tools

9.4.4.1 RAPID programming

9.4.4 Configuration examples

9.4.4.1 RAPID programming

RAPID module

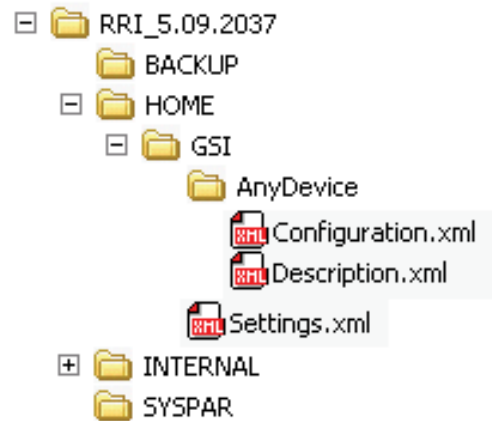
A RAPID module containing the corresponding RAPID record declarations and variable declarations must be created and loaded.

The FlexPendant user interface is not included in RobotWare.

9.4.4.2 Example configuration

Overview

The files Settings.xml, Description.xml, and Configuration.xml are located in the folder HOME\GSI\



xx0800000177



Note

The name of the folder must correspond to the name of the device. See [Device description on page 342](#). In this example we have used the name *AnyDevice*. The network address used in Description.xml is to the PC running the server, not the robot controller. See [Device description on page 342](#).

Settings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Settings>
  <Servers>
  <Servers/>
  <Clients>
    <Client Convention="CDP" Name="AnyDevice" />
  </Clients>
</Settings>
```

Description.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Description>
  <Name>AnyDevice</Name>
  <Convention>CDP</Convention>
  <Type>IntelligentCamera</Type>
  <Class>MachineVision</Class>
  <Network Address="10.49.65.74" Port="Service">
    <Channel Type="Cyclic" Protocol="Udp" Port="3002" />
  </Network>
  <Settings>
    <TimeOut>2000</TimeOut>
```

Continues on next page

9 Engineering tools

9.4.4.2 Example configuration

Continued

```
<MaxLost>30</MaxLost>
<DryRun>>false</DryRun>
</Settings>
</Description>
```

Configuration.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <Enums>
    <Enum Name="OperationMode" Link="Intern">
      <Member Name="Automatic" Alias="Auto" />
      <Member Name="ReducedSpeed" Alias="ManRS" />
      <Member Name="FullSpeed" Alias="ManFS" />
    </Enum>
  </Enums>
  <Records>
    <Record Name="RobotData">
      <Field Name="OperationMode" Type="OperationMode" Link="Intern"
        Alias="RobMode" />
      <Field Name="FeedbackTime" Type="Time" Link="Intern"
        Alias="Ts_act" />
      <Field Name="FeedbackPose" Type="Frame" Link="Intern"
        Alias="P_act" />
      <Field Name="FeedbackJoints" Type="Joints" Link="Intern"
        Alias="J_act" />
      <Field Name="PredictedTime" Type="Time" Link="Intern"
        Alias="Ts_des" />
      <Field Name="PlannedPose" Type="Frame" Link="Intern"
        Alias="P_des" />
      <Field Name="PlannedJoints" Type="Joints" Link="Intern"
        Alias="J_des" />
      <Field Name="ApplicationData" Type="Num" Size="18"
        Alias="AppData" />
    </Record>
    <Record Name="SensorData">
      <Field Name="ErrorString" Type="String" Alias="EStr" />
      <Field Name="ApplicationData" Type="Num" Size="18"
        Alias="AppData" />
    </Record>
  </Records>
  <Properties>
    <Property Name="RobData" Type="RobotData" Flag="WriteOnly"/>
    <Property Name="SensData" Type="SensorData" Flag="ReadOnly"/>
  </Properties>
</Configuration>
```

Continues on next page

RAPID configuration

This is an example for an RRI implementation. The out data uses an array of 18 num (robdata). The in data receives a string and an array of 18 num (sensdata). This needs to be defined according to the file configuration.xml.

```

RECORD applicationdata
  num Item1;
  num Item2;
  num Item3;
  num Item4;
  num Item5;
  num Item6;
  num Item7;
  num Item8;
  num Item9;
  num Item10;
  num Item11;
  num Item12;
  num Item13;
  num Item14;
  num Item15;
  num Item16;
  num Item17;
  num Item18;
ENDRECORD
RECORD robdata
  applicationdata AppData;
ENDRECORD
RECORD sensdata
  string ErrString;applicationdata AppData;
ENDRECORD
! Sensor Declarations
PERS sensor AnyDevice := [1,4,0];
PERS robdata DataOut := [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]];
PERS sensdata DataIn :=
  ["No",[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]];
! Setup Interface Procedure
PROC RRI_Open()
  SiConnect AnyDevice;
  ! Send and receive data cyclic with 64 ms rate
  SiGetCyclic AnyDevice, DataIn, 64;
  SiSetCyclic AnyDevice, DataOut, 64;
ENDPROC
! Close Interface Procedure
PROC RRI_Close()
  ! Close the connection
  SiClose RsMaster;
ENDPROC
ENDMODULE

```

Continues on next page

9 Engineering tools

9.4.4.2 Example configuration

Continued

Transmitted XML messages

Each XML message has the data variable name as root element with the attributes **Id** (the message ID) and **Ts** (the time stamp of the message). The subelements are then the record fields. The values of a multiple value field (array or record) are expressed as attributes.

Message sent out from robot controller

The time unit is second (float) with a resolution of 1 ms. The position (length) unit is millimeter (float). The position (angle) unit is radians.

Name	Data type	Description
Id	Integer	Last received robot data message ID
Ts	Float	Time stamp (message)
RobMode	Operationmode	Operation mode
TS_act	Float	Time stamp (actual position)
P_act	Pose	Actual cartesian position
J_act	Joint	Actual joint position
TS_des	Float	Time stamp (desired position)
P_des	Pose	Desired cartesian position
J_des	Joint	Desired joint position
AppData	Array of 18 Floats	Free defined application data

```
<RobData Id="111" Ts="1.202" >
  <RobMode>Auto</RobMode>
  <Ts_act>1.200</Ts_act>
  <P_act X="1620.0" Y="1620.0" Z="1620.0" Rx="100.0" Ry="100.0"
    Rz="100.0" />
  <J_act J1="1.0" J2="1.0" J3="1.0" J4="1.0" J5="1.0" J6="1.0" />
  <Ts_des>1.200</Ts_des>
  <P_des X="1620.0" Y="1620.0" Z="1620.0" Rx="100.0" Ry="100.0"
    Rz="100.0" />
  <J_des J1="1.0" J2="1.0" J3="1.0" J4="1.0" J5="1.0" J6="1.0" />
  <AppData X1="1" X2="1620.000" X3="1620.000" X4="1620.000"
    X5="1620.000" X6="1620.000" X7="1620.000" X8="1620.000"
    X9="1620.000" X10="1620.000" X11="1620.000" X12="1620.000"
    X13="1620.000" X14="1620.000" X15="1620.000" X16="1620.000"
    X17="1620.000" X18="1620.000" />
</RobData>
```

Message received from robot controller

The time unit is seconds (float).

Name	Data type	Description
Id	Integer	Last received data message ID. This ID must correspond to the ID sent from the robot controller.
Ts	Float	Time stamp
EStr	String	Error message

Continues on next page

Name	Data type	Description
AppData	Array of 18 floats	Free defined application data

The corresponding XML message on the network would look like this:

```
<SensData Id="111" Ts="1.234">
  <EStr>xxxx</Estr>
  <AppData X1="232.661" X2="1620.293" X3="463.932"
    X4="1231.053" X5="735.874" X6="948.263" X7="2103.584"
    X8="574.228" X9="65.406" X10="2372.633" X11="20.475"
    X12="96.729" X13="884.382" X14="927.954" X15="748.294"
    X16="3285.574" X17="583.293" X18="684.338" />
</SensData>
```

9 Engineering tools

9.4.5 RAPID components

9.4.5 RAPID components

About the RAPID components

This is an overview of all instructions, functions, and data types in *Robot Reference Interface*.

For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instructions

Instructions	Description
SiConnect	Sensor Interface Connect
SiClose	Sensor Interface Close
SiGetCyclic	Sensor Interface Get Cyclic
SiSetCyclic	Sensor Interface Set Cyclic

Functions

Robot Reference Interface includes no functions.

Data types

Data types	Description
sensor	External device descriptor
sensorstate	Communication state of the device

10 Servo motor control

10.1 Servo Tool Change [630-1]

10.1.1 Overview

Purpose

The purpose of Servo Tool Change is to be able to change tools on-line.

With the option Servo Tool Change it is possible to disconnect the cables to the motor of an additional axis and connect them to the motor of another additional axis. This can be done on the run, in production.

This option is designed with servo tools in mind, but can be used for any type of additional axes.

Examples of advantages are:

- One robot can handle several tools.
- Less equipment is needed since one drive-measurement system is shared by several tools.

What is included

The RobotWare option Servo Tool Change enables you to:

- change tool on-line
- have up to 8 different servo tools to change between.

Note that the option Servo Tool Change only provides the software functionality. Hardware, such as a tool changer is not included.

Basic approach

This is the general approach for using Servo Tool Change. For a more detailed description of how this is done, see [Tool change procedure on page 361](#).

- 1 Deactivate the first tool.
- 2 Disconnect the first tool from the cables.
- 3 Connect the second tool to the cables.
- 4 Activate the second tool.

10 Servo motor control

10.1.2 Requirements and limitations

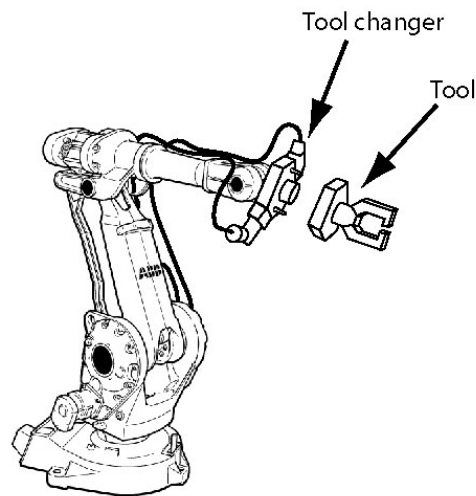
10.1.2 Requirements and limitations

Additional Axes

To use Servo Motor Control, you must have the option Additional Axes. All additional axes used by servo motor control must be configured according to the instructions in [Requirements and limitations on page 356](#).

Tool changer

To be able to change tools in production with a plug-in mechanism, a mechanical tool changer interface is required.



en0300000549

All cables are connected to the tool changer. The tool changer interface includes connections for signals, power, air, water or whatever needs to be transmitted to and from the tool.

Up to 8 tools

Up to 8 additional axes (servo tools or other axes) can be installed simultaneously in one robot controller. Some of them (or all) may be servo tools sharing a tool changer.

Moving deactivated tool

The controller remembers the position of a deactivated tool. When the tool is reconnected and activated this position is used.

If the servo tool axis is moved during deactivation, the position of the axis might be wrong after activation, and this will not be detected by the controller.

Continues on next page

The position after activation will be correct if the axis has not been moved, or if the movement is less than 0.5 motor revolutions.



Tip

If you have the Spot Servo option you can use tool change calibration.

After a tool is activated, call the instruction `STCalib` to calibrate the tool. This will adjust any positional error caused by tool movements during deactivation.

Activating wrong tool

It is important not to activate a mechanical unit that is not connected.

An activation of the wrong mechanical unit may cause unexpected movements or errors. The same errors occur if a tool is activated when no tool at all is connected.



Tip

A connection relay can be configured so that activation of a mechanical unit is only allowed when it is connected. See [Connection relay on page 359](#).

10.1.3 Configuration

Configuration overview

The option Servo Tool Change allows configuration of several tools for the same additional axis.

One individual set of parameters is installed for each gun tool.

How to configure each tool

Each tool is configured the same way as if it was the only tool. For information on how to do this, see [Configuration on page 358](#).

The parameter *Deactivate PTC superv. at disconnect*, in the type *Mechanical Unit*, must be set to Yes.

The parameter *Disconnect deactivate*, in the type *Measurement Channel*, must be set to Yes.

The parameter *Logical Axis*, in the type *Joint*, can be set to the same number for several tools. Since the tools are never used at the same time, the tools are allowed to use the same logical axis.

The parameter *allow_activation_from_any_motion_task*, in the type *Mechanical Unit*, must be set for the specific servo gun. The servo gun .cfg files are created by the servo gun manufacture.

For a detailed description of the respective parameter, see [Configuration on page 358](#).

10.1.4 Connection relay

Overview

To make sure a disconnected mechanical unit is not activated, a connection relay can be used. A connection relay can prevent a mechanical unit from being activated unless a specified digital signal is set.

Some tool changers support I/O signals that specify which gun is currently connected. Then a digital input signal from the tool changer is used by the connection relay.

If the tool changer does not support I/O signals, a similar behavior can be created with RAPID instructions. Set a digital output signal to 1 with the instruction `SetDO` each time the tool is connected, and set the signal to 0 when the tool is disconnected.

System parameters

This is a brief description of each parameter used to configure a connection relay. For more information, see the respective parameter in [Connection relay on page 359](#)

The following parameters have to be set for the type *Mechanical Unit* in the topic *Motion*:

Parameter	Description
Use Connection Relay	The name of the relay to use. Corresponds to the name specified in the parameter <i>Name</i> in the type <i>Relay</i> .

The following parameters must be set for the type *Relay* in the topic *Motion*:

Parameter	Description
Name	Name of the relay. Used by the parameter <i>Use Connection Relay</i> in the type <i>Mechanical Unit</i> .
Input Signal	The name of the digital signal used to indicate if it should be possible to activate the mechanical unit.

Example of connection relay configuration

This is an example of how to configure connection relays for two gun tools. gun1 can only be activated when signal di1 is 1, and gun2 can only be activated when di2 is 1.

If the tool changer sets di1 to 1 only when gun1 is connected, and di2 to 1 only when gun2 is connected, there is no risk of activating the wrong gun.

The following parameter values are set for gun1 and gun2 in the type *Mechanical Unit*:

Name	Use Connection Relay
gun1	gun1_relay
gun2	gun2_relay

Continues on next page

10 Servo motor control

10.1.4 Connection relay

Continued

The following parameter values are set for gun1 and gun2 in the type *Relay*:

Name	Input Signal
gun1_relay	di1
gun2_relay	di2

10.1.5 Tool change procedure

How to change tool

This is a description of how to change from gun1 to gun2.

Step	Action
1	Deactivate gun1 with the instruction: <code>DeactUnit gun1;</code>
2	Disconnect gun1 from the tool changer.
3	Connect gun2 to the tool changer.
4	Activate gun2 with the instruction: <code>ActUnit gun2;</code>
5	Optional but recommended: Calibrate gun2 with the instruction: <code>STCalib gun1 \ToolChg;</code> Note that this calibration requires option Servo Tool Control or Spot Servo .

10 Servo motor control

10.1.6 Jogging servo tools with activation disabled

10.1.6 Jogging servo tools with activation disabled

Overview

Only one of the servo tools used by the tool changer may be activated at a time, the others are set to activation disabled. This is to make sure that the user is jogging the servo tool presently connected with right configuration.

What to do when Activation disabled appears

Follow these steps when you need to jog a servo tool but cannot activate the unit because activation is disabled.

Step	Action
1.	Make sure that the right servo tool is mounted on the tool changer. If the wrong tool is mounted, see Tool change procedure on page 361 .
2.	If no tool is activated, open the RAPID execution and activate the right tool.
3.	If the right tool is mounted on the tool changer, deactivate the wrong tool and activate the right tool from RAPID execution.

10.2 Servo Tool Control [included in 635-6]

10.2.1 Overview

Purpose

Servo Tool Control can be used to control a servo tool, for example in a spot weld application. *Servo Tool Control* makes it possible to close the tool to a specific plate thickness and force, and maintain the force during the process until the tool is requested to be opened.

What is included

The RobotWare option *Servo Tool Control* is included in the RobotWare option *Spot*.

Servo Tool Control gives you access to:

- RAPID instructions to open, close and calibrate servo tools
- RAPID instructions for tuning system parameter values
- RAPID functions for checking status of servo tools
- system parameters to configure servo tools
- service routines for different types of servo tool calibrations

Basic approach

This is the general approach for using *Servo Tool Control*.

- 1 Configure and calibrate the servo tool.
- 2 Perform a force calibration.
- 3 Create the RAPID program.

Prerequisites

A servo tool is an additional axis. The option *Additional Axes* must be present on the robot system using a servo tool. Required hardware, such as drive module and measurement board, is specified in *Application manual - Additional axes and stand alone controller*.

10.2.2 Servo tool movements

Closing and opening of a servo tool

The servo tool can be closed to a predefined plate thickness and tip force. When the tips reach the programmed contact position, the movement is stopped and there is an immediate switch from position control mode to force control mode. In the force control mode a motor torque will be applied to achieve the desired tip force.

The force remains constant until an opening is ordered. Opening of the tool will reduce the tip force to zero and move the tool arm back to the pre-close position.

Synchronous and asynchronous movements

Normally a servo tool axis is moved synchronous with the robot movements in such a way that both movements will be completed exactly at the same time. However the servo tool may be closed asynchronously (independent of current robot movement). The closing will immediately start to run the tool arm to the expected contact position (thickness). The closing movement will interrupt an on-going synchronous movement of the tool arm.

The tool opening may also take place while the robot is moving. But it is not possible if the robot movement includes a synchronized movement of the servo tool axis. A motion error, "tool opening could not synchronize with robot movement", will occur.

10.2.3 Tip management

About tip management

The tip management functionality will find and calibrate the contact position of the tool tips automatically. It will also update and monitor the total tip wear of the tool tips.

The tips can be calibrated with a service routine (see [Service routines on page 378](#)) or the RAPID instruction `STCalib` (see [Instructions on page 368](#)). Typically, two tool closings will be performed during a calibration.

Three different types of calibrations are supported: tip wear, tip change and tool change. All three will calibrate the contact position of the tips. The total tip wear will, however, be updated differently by these methods.

Tip wear calibration

As the tips are worn down, they need to be dressed. After the tip dressing, a tip wear calibration is required. The tool contact position is calibrated and the total tip wear of the tool is updated. The calibration movements are fast and the switch to force control mode will take place at the zero position.

This method must only be used to make small position adjustments (< 3 mm) caused by tip wear / tip dressing.



Tip

A variable in your RAPID program can keep track of the tip wear and inform you when the tips need to be replaced.

Tip change calibration

The tip change calibration is to be used after mounting a new pair of tips. The tool contact position is calibrated and the total tip wear of the tool is reset. The first calibration movement is slow in order to find the unknown contact position and switch to force control. The second calibration movement is fast. This calibration method will handle big position adjustments of the servo tool.

This calibration may be followed by a tool closing in order to squeeze the tips in place. A new tip change calibration is then done to update possible position differences after the tip squeeze.

Tool change calibration

The tool change calibration is to be used after reconnecting and activating a servo tool. The tool contact position is calibrated and the total tip wear of the tool remains unchanged. The first calibration movement is slow in order to find the unknown tip collision position and switch to force control. The second calibration movement is fast. This calibration method will handle big position adjustments of the tool.

The method should always be used after reconnecting a tool since the activation will restore the latest known position of the tool, and that position may be different from the actual tool position; the tool arm may have been moved when

Continues on next page

10 Servo motor control

10.2.3 Tip management

Continued

disconnected. This calibration method will handle big position adjustments of the tool.



Tip

Tool change calibration is most commonly used together with the RobotWare option Servo Tool Change.

10.2.4 Supervision

Max and min stroke

An out of range supervision will stop the movement if the tool is reaching max stroke or if it is closed to contact with the tips (reaching min stroke). See *Upper Joint Bound* and *Lower Joint Bound* in [Arm on page 370](#).

Motion supervision

During the position control phase of the closing/opening, motion supervision is active for the servo tool to detect if the arm collides or gets stuck. A collision will cause a motion error and the motion will be stopped.

During the force control phase, the motion supervision will supervise the tool arm position not to exceed a certain distance from the expected contact position. See parameter *Max Force Control Position Error* in [Supervision Type on page 372](#).

Maximum torque

There is a maximum motor torque for the servo tool that never will be exceeded in order to protect the tool from damage. If the force is programmed out of range according to the tools force-torque table, the output force will be limited to this maximum allowed motor torque and a motion warning will be logged. See parameter *Max Force Control Motor Torque* in [SG Process on page 369](#).

Speed limit

During the force control phase there is a speed limitation. The speed limitation will give a controlled behavior of the tool even if the force control starts before the tool is completely closed. See *Speed limit 1- 6* in [Force Master Control on page 370](#).

10 Servo motor control

10.2.5 RAPID components

10.2.5 RAPID components

About the RAPID components

This is an overview of all instructions, functions, and data types in *Servo Tool Control*.

For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instructions

Instruction	Description
STClose	Close the servo tool with a predefined force and thickness.
STOpen	Open the servo tool.
STCalib	Calibrate the servo tool. An argument determines which type of calibration will be performed: <ul style="list-style-type: none">• \ToolChg for tool change calibration• \TipChg for tip change calibration• \TipWear for tip wear calibration
STTune	Tune motion parameters for the servo tool. A temporary value can be set for a parameter specified in the instruction.
STTuneReset	Reset tuned motion parameters for the servo tool. Cancel the effect of all STTune instructions.
STRecalib	Activates the force calibration values without a controller restart.

Functions

Function	Description
STIsClosed	Test if the servo tool is closed.
STIsOpen	Test if the servo tool is open.
STIsCalib	Tests if a servo tool is calibrated.
STCalcTorque	Calculate the motor torque for a servo tool.
STCalcForce	Calculate the force for a servo tool.
STIsServoTool	Tests if a mechanical unit is a servo tool.
STIsIndGun	Tests if servo tool is in independent mode.

Data types

Servo Tool Control includes no RAPID data types.

10.2.6 System parameters

About the system parameters

When using a servo tool, a motion parameter file for the tool is normally installed on the controller. A servo tool is a specific variant of an additional axis and the description of how to configure the servo tool is found in *Application manual - Additional axes and stand alone controller*.

In this section, the parameters used in combination with *Servo Tool Control* is briefly described. For more information, see the respective parameter in *Technical reference manual - System parameters*.

SG Process

These parameters belong to the type *SG Process* in the topic *Motion*.

SG Process is used to configure the behavior of a servo gun (or other servo tool).

Parameter	Description
<i>Close Time Adjust</i>	Adjustment of the ordered minimum close time of the gun.
<i>Close Position Adjust</i>	Adjustment of the ordered position (plate thickness) where force control should start, when closing the gun.
<i>Force Ready Delay</i>	Delays the close ready event after achieving the ordered force.
<i>Max Force Control Motor Torque</i>	Max allowed motor torque for force control. Commanded force will be reduced, if the required motor torque is higher than this value.
<i>Post-synchronization Time</i>	Anticipation of the open ready event. This can be used to synchronize the gun opening with the next robot movement.
<i>Calibration Mode</i>	Defines the number of times the servo gun closes during a tip wear calibration.
<i>Calibration Force Low</i>	The minimum tip force used during a tip wear calibration.
<i>Calibration Force High</i>	The maximum tip force used during a tip wear calibration.
<i>Calibration Time</i>	The time that the servo gun waits in closed position during calibration.
<i>Number of Stored Forces</i>	Defines the number of points in the force-torque relation specified in <i>Tip Force 1 - 10</i> and <i>Motor Torque 1 - 10</i> .
<i>Tip Force 1 - 10</i>	<i>Tip Force 1</i> defines the tip force that corresponds to the motor torque in <i>Motor Torque 1</i> . <i>Tip Force 2</i> corresponds to <i>Motor Torque 2</i> , etc.
<i>Motor Torque 1- 10</i>	<i>Motor Torque 1</i> defines the motor torque that corresponds to the tip force in <i>Tip Force 1</i> . <i>Motor Torque 2</i> corresponds to <i>Tip Force 2</i> , etc.
<i>Squeeze Position 1 - 10</i>	Defines the joint position at each force level in the force calibration table.
<i>Soft Stop Timeout</i>	Defines how long the force will be maintained if a soft stop occurs during constant force.

Continues on next page

10 Servo motor control

10.2.6 System parameters

Continued

Force Master

These parameters belong to the type *Force Master* in the topic *Motion*.

Force Master is used to define how a servo gun behaves during force control. The parameters only affect the servo gun when it is in force control mode.

Parameter	Description
<i>References Bandwidth</i>	The frequency limit for the low pass filter for reference values.
<i>Use ramp time</i>	Determines if the ramping of the tip force should use a constant time or a constant gradient.
<i>Ramp when Increase Force</i>	Determines how fast force is built up while closing the tool when <i>Use ramp time</i> is set to No.
<i>Ramp time</i>	Determines how fast force is built up while closing the tool when <i>Use ramp time</i> is set to Yes.
<i>Collision LP Bandwidth</i>	Frequency limit for the low pass filter used for tip wear calibration.
<i>Collision Alarm Torque</i>	Determines how hard the tool tips will be pressed together during the first gun closing of new tips calibrations and tool change calibrations.
<i>Collision Speed</i>	Determines the servo gun speed during the first gun closing of new tips calibrations and tool change calibrations.
<i>Collision Delta Position</i>	Defines the distance the servo tool has gone beyond the contact position when the motor torque has reached the value specified in <i>Collision Alarm Torque</i> .
<i>Max pos err. closing</i>	Determines how close to the ordered plate thickness the tool tips must be before the force control starts.
<i>Delay ramp</i>	Delays the starting of torque ramp when force control is started.
<i>Ramp to real contact</i>	Determines if the feedback position should be used instead of reference position when deciding the contact position.

Force Master Control

These parameters belong to the type *Force Master Control* in the topic *Motion*.

Force Master Control is used to set the speed limit and speed loop gain as functions of the torque.

Parameter	Description
<i>No. of speed limits</i>	The number of points used to define speed limit and speed loop gain as functions of the torque. Up to 6 points can be defined.
<i>torque 1 - torque 6</i>	The torque levels, corresponding to the ordered tip force, for which the speed limit and speed loop gain values are defined.
<i>Speed Limit 1 - 6</i>	<i>Speed Limit 1</i> to <i>Speed Limit 6</i> are used to define the maximum speed depending on the ordered tip force.
<i>Kv 1 - 6</i>	<i>Kv 1</i> to <i>Kv 6</i> are used to define the speed loop gain for reducing the speed when the speed limit is exceeded.

Arm

These parameters belong to the type *Arm* in the topic *Motion*.

The type *Arm* defines the characteristics of an arm.

Parameter	Description
<i>Upper Joint Bound</i>	Defines the upper limit of the working area for the joint.

Continues on next page

Parameter	Description
<i>Lower Joint Bound</i>	Defines the lower limit of the working area for the joint.

Acceleration Data

These parameters belong to the type *Acceleration Data* in the topic *Motion*.
Acceleration Data is used to specify some acceleration characteristics for axes without any dynamic model.

Parameter	Description
<i>Nominal Acceleration</i>	Worst case motor acceleration.
<i>Nominal Deceleration</i>	Worst case motor deceleration.
<i>Acceleration Derivate Ratio</i>	Indicates how fast the acceleration can be increased.
<i>Deceleration Derivate Ratio</i>	Indicates how fast the deceleration can be increased.

Motor Type

These parameters belong to the type *Motor Type* in the topic *Motion*.
Motor Type is used to describe characteristics for a motor.

Parameter	Description
<i>Pole Pairs</i>	Defines the number of pole pairs for the motor.
<i>Inertia</i>	The inertia of the motor, including the resolver but excluding the brake.
<i>Stall Torque</i>	The continuous stall torque, i.e. the torque the motor can produce at no speed and during an infinite time.
<i>ke Phase to Phase</i>	Nominal voltage constant. The induced voltage (phase to phase) that corresponds to the speed 1 rad/s.
<i>Max Current</i>	Max current without irreversible magnetization.
<i>Phase Resistance</i>	Nominal winding resistance per phase at 20 degrees Celsius.
<i>Phase Inductance</i>	Nominal winding inductance per phase at zero current.

Motor Calibration

These parameters belong to the type *Motor Calibration* in the topic *Motion*.
Motor Calibration is used to calibrate a motor.

Parameter	Description
<i>Commutator Offset</i>	Defines the position of the motor (resolver) when the rotor is in the electrical zero position relative to the stator.
<i>Calibration Offset</i>	Defines the position of the motor (resolver) when it is in the calibration position.

Stress Duty Cycle

These parameters belong to the type *Stress Duty Cycle* in the topic *Motion*.
Stress Duty Cycle is used for protecting axes, gearboxes, etc.

Parameter	Description
<i>Speed Absolute Max</i>	The absolute highest motor speed to be used.

Continues on next page

10 Servo motor control

10.2.6 System parameters

Continued

Parameter	Description
<i>Torque Absolute Max</i>	The absolute highest motor torque to be used.

Supervision Type

These parameters belong to the type *Supervision Type* in the topic *Motion*.

Supervision Type is used for continuous supervision of position, speed and torque.

Parameter	Description
<i>Max Force Control Position Error</i>	When a servo gun is in force control mode it is not allowed to move more than the distance specified in <i>Max Force Control Position Error</i> . This supervision will protect the tool if, for instance, one tip is lost.
<i>Max Force Control Speed Limit</i>	Speed error factor during force control. If the speed limits, defined in the type <i>Force Master Control</i> , multiplied with <i>Max Force Control Speed Limit</i> is exceeded, all movement is stopped.

Transmission

These parameters belong to the type *Transmission* in the topic *Motion*.

Transmission is used to define the transmission gear ratio between a motor and its axis.

Parameter	Description
<i>Rotating Move</i>	Defines if the axis is rotating or linear.
<i>Transmission Gear Ratio</i>	Defines the transmission gear ratio between motor and joint.

Lag Control Master 0

These parameters belong to the type *Lag Control Master 0* in the topic *Motion*.

Lag Control Master 0 is used for regulation of axes without any dynamic model.

Parameter	Description
<i>FFW Mode</i>	Defines if the position regulation should use feed forward of speed and torque values.
<i>Kp, Gain Position Loop</i>	Proportional gain in the position regulation loop.
<i>Kv, Gain Speed Loop</i>	Proportional gain in the speed regulation loop.
<i>Ti Integration Time Speed Loop</i>	Integration time in the speed regulation loop.

Uncalibrated Control Master 0

These parameters belong to the type *Uncalibrated Control Master 0* in the topic *Motion*.

Uncalibrated Control Master 0 is used to regulate uncalibrated axes.

Parameter	Description
<i>Kp, Gain Position Loop</i>	Proportional gain in the position regulation loop.
<i>Kv, Gain Speed Loop</i>	Proportional gain in the speed regulation loop.
<i>Ti Integration Time Speed Loop</i>	Integration time in the speed regulation loop.
<i>Speed Max Uncalibrated</i>	The maximum allowed speed for an uncalibrated axis.

Continues on next page

Parameter	Description
<i>Acceleration Max Uncalibrated</i>	The maximum allowed acceleration for an uncalibrated axis.
<i>Deceleration Max Uncalibrated</i>	The maximum allowed deceleration for an uncalibrated axis.

10 Servo motor control

10.2.7 Commissioning and service

10.2.7 Commissioning and service

Commissioning the servo tool

For a new servo tool, follow these steps for installing and commissioning:

Step	Action
1	Install the servo tool according to the description in <i>Application manual - Additional axes and stand alone controller</i> .
2	Load a .cfg file with the servo tool configuration. For detailed description on how to do this, see <i>Operating manual - RobotStudio</i> . If you do not have any .cfg file for the servo tool, you can load a template file and configure the system parameters with the values of your servo tool. Template files are found in the RobotWare distribution, see Template file locations on page 374 .
3	Use the RAPID instruction <code>STTune</code> and iterate to find the optimal parameter values. Once found, these optimal values should be written to the system parameters to be permanent.
4	Fine calibrate the servo tool, see Fine calibration on page 376 .
5	Unless force calibration was included in a loaded .cfg file, perform a force calibration, see Service routines on page 378 .

Template file locations

The template files can be obtained from the PC or the IRC5 controller.

- In the RobotWare installation folder in RobotStudio: ...\\RobotPackages\\RobotWare_RPK_<version>\\utility\\AdditionalAxis\\
- On the IRC5 Controller:
<SystemName>\\PRODUCTS\\<RobotWare_xx.xx.xxxx>\\utility\\AdditionalAxis\\



Note

Navigate to the RobotWare installation folder from the RobotStudio **Add-Ins** tab, by right-clicking on the installed RobotWare version in the **Add-Ins** browser and selecting **Open Package Folder**.

Disconnect/reconnect a servo tool

If the servo tool is deactivated, using the `DeactUnit` instruction, it may be disconnected and removed. The tool position at deactivation will be restored when the tool is connected and reactivated. Make a tool change calibration to make sure the tip position is OK.

The whole process of changing a tool can be performed by a RAPID program if you use the RobotWare option Servo Tool Change and the instruction `STCalib`.

Recover from accidental disconnection

If the motor cables are disconnected by accident when the servo tool is active, the system will go into system failure state. After restart of the system the servo tool must be deactivated in order to jog the robot to a service position.

Deactivation may be performed from the **Jogging** window. Tap on **Activate...**, select the servo tool and tap on **Deactivate**.

Continues on next page

After service / repair the revolution counter must be updated since the position has been lost, see [Update revolution counter on page 376](#).

10 Servo motor control

10.2.8 Mechanical unit calibrations

10.2.8 Mechanical unit calibrations

Fine calibration

Fine calibration must be performed when installing a new servo tool or if the servo tool axis is in state 'Not Calibrated'. Fine calibration of servo tools requires, unlike other kinds of additional axes, the running of a service routine, `ManServiceCalib`.

Step	Action
1	From the ABB menu, select Calibration .
2	Tap on the name of the servo tool axis.
3	Tap the button Calib. Parameters .
4	Tap on Fine Calibration....
5	Confirm by tapping Yes .
6	Tick the box in front of the servo tool axis and tap Calibrate .
7	Confirm calibration by tapping Calibrate .
8	Initialize the servo tool position by running the service routine <code>ManualServiceCalib</code> , see Service routines on page 378 .

Update revolution counter

An update of the revolution counter must be performed if the position of the axis is lost. If this happens, this is indicated by the calibration state 'Rev. Counter not updated'. Update of revolution counters for servo tools requires, unlike other kinds of additional axes, the running of a service routine, `ManualServiceCalib`.

Step	Action
1	From the ABB menu, select Calibration .
2	Tap on the name of the servo tool axis.
3	Tap the button Rev. Counters .
4	Tap on Update Revolution Counters....
5	Confirm by tapping Yes .
6	Tick the box in front of the servo tool axis and tap Update .
7	Confirm calibration by tapping Update .
8	Synchronize the tip position by running the service routine <code>ManualServiceCalib</code> , see Service routines on page 378 .

10.2.9 RAPID code example

How to use the code package

The normal programming technique for *Servo Tool Control* is to customize shell routines based on the example code below. These shell routines are then called from your program.

Using shell routines

This example shows a main routine in combination with a customized routine (rMoveSpot) that uses the standard servo tool instructions. The external process (for example a weld timer) is indicated with the routine rWeld.

```
PROC main()
  MoveJ p1, v500, z50, weldtool;
  MoveL p2, v1000, z50, weldtool;
  ! Perform weld process
  rMoveSpot weldpos1, v2000, curr_gun_name, 1000, 2, 1,
    weldtool\WObj:=weldwobj;
  rMoveSpot weldpos2, v2000, curr_gun_name, 1000, 2, 1,
    weldtool\WObj:=weldwobj;
  rMoveSpot weldpos3, v2000, curr_gun_name, 1500, 3, 1,
    weldtool\WObj:=weldwobj;
  MoveL p3, v1000, z50, weldtool;
ENDPROC

PROC rMoveSpot (robtarget ToPoint,
  speeddata Speed,
  gunname Gun,
  num Force,
  num Thickness,
  PERS tooldata Tool
  \PERS wobjdata WObj)
  ! Move the gun to weld position.
  ! Always use FINE point to prevent too early closing.
  MoveL ToPoint, Speed, FINE, weldtool \WObj=WObj;
  STCloseGun Gun, Thickness;
  rWeld;
  STOpenGun Gun;
ENDPROC

PROC rWeld()
  ! Request weld start from weld timer
  SetDO doWeldstart,1;
  ! Wait until weld is performed
  WaitDI diWeldready,1;
  SetDO doWeldstart,0;
ENDPROC
```

10 Servo motor control

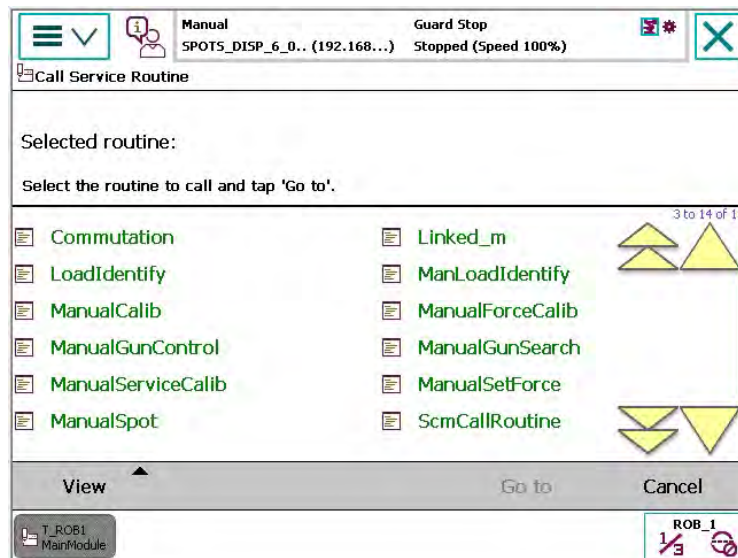
10.2.10 Service routines

10.2.10 Service routines

About the service routines

Some useful service routines are predefined to be used for manual actions during programming and test.

- From the Spot GUI application, tap **ABB**, **RobotWare Spot** and **Manual Actions**.
- From the Program Editor, tap **Debug**, and then tap **Call Service Routine**



en1200000242



Tip

It is also possible to access and run the service routines from the FlexPendant interface, see *Application manual - Spot options*.

Available service routines

The following service routines can be used together with *Servo Tool Control* For information about the other routines, see *Application manual - Spot options*.

Service routine	Description
ManualSetForce	Perform a <code>SetForce</code> action according to data in <code>curr_forcedata</code> . The gun equalize signal is also activated/deactivated.
ManualCalib	Perform a calibration of the servo gun, 1 Tool Change, 2 Tip Change or 3 Tip Wear calibration.
ManualForceCalib	Perform a force calibration of the servo gun. 2 - 10 force and positions can be stored.
ManualServiceCalib	<ol style="list-style-type: none">1 Synchronize the servo gun without jogging after the revolution counter has been updated, the servo gun will close slowly until it reaches the contact position.2 Synchronize the servo gun without jogging after the gun has been fine calibrated, the servo gun will move to zero position and then close slowly until it reaches the contact position.

Continues on next page

If several guns are used then a dialog will appear asking for the gun number of the gun to be handled.

This page is intentionally left blank

Index

A

Absolute Accuracy, 93
 Absolute Accuracy calibration, 105
 Absolute Accuracy compensation, 103
 Absolute Accuracy verification, 106
 Acceleration Data, 371
 Acceleration Derivate Ratio, 371
 Acceleration Max Uncalibrated, 373
 accidental disconnection, 374
 acknowledge messages, 255
 activate Absolute Accuracy, 97
 Activate at start up, 191
 activate supervision, 238
 activation disabled, 362
 actor signals, 88–89
 additional axes, 363
 additional axis, 49
 Add or replace parameters, 154
 Adjustment Speed, 189
 Advanced RAPID, 15
 Advanced Shape Tuning, 117
 AliasIO, 22–23
 alignment, 111
 analog signal, 45
 Analog Signal Interrupt, 45
 Analog Synchronization, 139
 AND, 89
 Application protocol, 243, 247
 ArgName, 43
 argument name, 43
 Arm, 370
 arm replacement, 99
 asynchronous movements, 364
 Auto acknowledge input, 11, 48
 automatic friction tuning, 118
 Auto mode, 282
 axis, 201
 axis reset, 201

B

binary communication, 72
 binary data, 255
 birth certificate, Absolute Accuracy, 107
 BitAnd, 17
 BitCheck, 17
 BitClear, 17
 bit functionality, 16
 BitLSh, 17
 BitNeg, 17
 BitOr, 17
 BitRSh, 17
 BitSet, 17
 BitXOr, 17
 BookErrNo, 39
 bool, 339
 byte, 17
 ByteToStr, 17

C

C# API, 319
 calibrate follower axis, 55
 calibrate tool, 115
 calibration data, 98
 Calibration Force High, 369

Calibration Force Low, 369
 Calibration Mode, 369
 Calibration Offset, 371
 Calibration Pendulum, 96
 calibration process, 105
 Calibration Time, 369
 calibration tools, 96
 CalibWare, 96
 cell alignment, 111
 certificate, Absolute Accuracy, 107
 change calibration data, 98
 change of tool, Machine Synchronization, 166
 channel, 343
 character based communication, 72
 Check unresolved references, Task type, 273
 CirPathMode, 135
 class, 343
 ClearIOBuff, 73
 ClearRawBytes, 77
 Close, 73
 CloseDir, 81
 Close position adjust, 369
 Close time adjust, 369
 code example, 377
 collision, 229
 Collision Alarm Torque, 370
 Collision Delta Position, 370
 Collision Detection Memory, 232
 Collision Error Handler, 233
 Collision LP Bandwidth, 370
 Collision Speed, 370
 commissioning, 374
 common data, 284
 communication channel, 335
 communication client, 341
 Commutator Offset, 371
 compensation, 103
 compensation parameters, 93, 108
 compliance errors, 102
 communication cable
 connecting, 336
 configuration
 Absolute Accuracy, 97
 configuration.xml, 345
 configuration example, 349
 configuration files, 340
 configuration functionality, 25
 configure Collision Detection, 236
 configuring
 sensors, 296
 tasks, 276
 Connected signal, 190
 connection relay, 359
 constants
 Sensor Interface, 300
 convention, 342
 coordinate systems, 111
 CopyFile, 81
 CopyRawBytes, 77
 Corr argument, 224
 CorrClear, 223
 CorrCon, 223
 corrdescr, 223
 CorrDiscon, 223
 correction generator, 222
 CorrRead, 223

CorrWrite, 223
Counts Per Meter, 189
CPU_load_equalization, 190
creating tasks, 276
cross connections, 88
cut plane, 133
cut shape, 138

D

data, 261
data exchange, 335
datapoints, 20
Data ready signal, 190
data search functionality, 19
data types
 Multitasking, 275
 supported, 339
data variable example
 Electronically Linked Motors, 63
data variables
 Electronically Linked Motors, 61
Deactivate PTC superv. at disconnect, 358
deactivate supervision, 238
deactivate tasks, 281
debugging
 strategies, 276
Deceleration Derivate Ratio, 371
Deceleration Max Uncalibrated, 373
declarations, 284
deflection, 103
Delay ramp, 370
description.xml, 342
digital I/O signals, 88
dir, 81
directory management, 80
discarded message, 263
Disconnect deactivate, 358
disconnection, 374
dispatcher, 289
displacement, 62
Do not allow deact, 191

E

EGM, 304
EGM .proto file, 334
EGM execution states, 309
EGM Path Correction, 304
EGM Position Guidance, 304
EGM RAPID components, 323
EGM sensor protocol, 318
EGM system parameters, 322
Electronically Linked Motors, 49
elements
 channel, 343
 class, 343
 convention, 342
 enum, 345
 field, 346
 member, 346
 network, 343
 property, 347
 record, 346
 settings, 344
 type, 343
enums element, 345
errdomain, 36
error interrupts, 35

error sources in accuracy, 102
ErrRaise, 36
errtype, 36
Ethernet, 241, 245
Ethernet link, 337
event messages, 38
event number, 38
Event Preset Time, 68
Event recorder, 252
external axes, 228
external axis, 201
Externally Guided Motion, 304
External Motion Interface Data, 322

F

fake target, 103
false triggering, 239
FeedbackJoints, 338
FeedbackPose, 338
FeedbackTime, 338
FFW Mode, 372
Fieldbus Command Interface, 84
field element, 346
FIFO, 262
file communication, 71
Filedbus Command, 189
file management, 80
FileSize, 81
file structures, 80
fine calibration, 376
finepoints, Machine Synchronization, 165
fixed position events, 65
fixture alignment, 112
FlexPendant, 291
follower, 49
Follower to Joint, 51
Force Master, 370
Force Master Control, 370
Force Ready Delay, 369
frame, 339
frame relationships, 114
frames, 111
FricIdEvaluate, 124
FricIdInit, 124
FricIdSetFricLevels, 124
friction compensation, 117
Friction FFW Level, 122
Friction FFW On, 122
Friction FFW Ramp, 122
friction level tuning, 118
FSSize, 81
functions
 Advanced RAPID, 43
 Multitasking, 275
 Sensor Interface, 299

G

General RAPID, 233
GetDataVal, 20
GetNextSym, 20
GetTrapData, 36
Google C++, 319
Google C++ API, 319
Google overview, 318
Google Protocol Buffers, 318
group I/O signals, 88
Group ID, 247

H

hydraulic press, 180

I

IError, 36
 IIRFFP, 189
 IndAMove, 204
 IndCMove, 204
 IndDMove, 204
 Independent Axes, 201
 independent joint, 228
 Independent Joint, 203
 Independent Lower Joint Bound, 203
 independent movement, 201
 Independent Upper Joint Bound, 203
 IndInpos, 204
 IndReset, 204
 IndRMove, 204
 IndSpeed, 204
 Inertia, 371
 Input Signal, 359
 installation, 374
 instructions
 Advanced RAPID, 43
 Multitasking, 275
 Sensor Interface, 299
 interrupt, 45, 262, 285, 299, 302
 interrupt functionality, 35
 iodev, 73
 IPers, 36
 IP protocols, 337
 IRMQMessage, 266
 IsFile, 81
 ISignalAI, 46
 ISignalAO, 46
 IsStopStateEvent, 43
 IVarValue, 299

J

Jog Collision Detection, 232, 236
 Jog Collision Detection Level, 232
 Jog Collision Detection Level, 236
 joint, 339
 Joint, 51
 joint zones, 195

K

ke Phase to Phase, 371
 kinematic errors, 102
 Kp, Gain Position Loop, 372
 Kv 1 - 6, 370
 Kv, Gain Speed Loop, 372
 Kv, Gain Speed Loop, 372

L

l_f_axis_name, 61
 l_f_axis_no, 61
 l_f_mecunt_n, 61
 l_m_axis_no, 61
 l_m_mecunt_n, 61
 Lag Control Master 0, 372
 Linked M Process, 51
 load calibration data, 98
 Load Identification, 96
 Local path, 243, 247
 Lock Joint in Ipol, 51
 logical AND, 90

Logical Axis, 358
 Logical Cross Connections, 88
 logical operations, 88
 logical OR, 90
 loss of accuracy, 101
 lost message, 263
 lost queue, 263
 Lower Joint Bound, 371
 LTAPP, 298

M

Main entry, Task type, 273
 maintenance, 99
 MakeDir, 81
 ManCalib, 378
 ManForceCalib, 378
 manipulator replacement, 100
 Manipulator Supervision, 232
 Manipulator Supervision Level, 232
 ManSetForce, 378
 Manual actions, 378
 manual friction tuning, 120
 manual mode, Machine Synchronization, 165, 167
 ManualServiceCalib, 378
 master, 49
 Master Follower kp, 52
 Max Advance Distance, 190–191
 Max Current, 371
 Max Delay Distance, 191
 Max Follower Offset, 51
 Max Force Control Motor Torque, 369
 Max Force Control Position Error, 372
 Max Force Control Speed Limit, 372
 Max Offset Speed, 51
 Max pos err. closing, 370
 Max Synchronization Speed, 191
 measurement system, 204
 mechanical unit, 292
 Mechanics, 191
 member element, 346
 merge of messages, 255
 messages
 outgoing, 338
 received, 352
 sent, 352
 Min Synchronization Speed, 191
 modes of operation, Machine Synchronization, 167
 modules
 Sensor Interface, 299
 molding machine, 184
 motion commands, Machine Synchronization, 165
 Motion Planner, 232
 Motion Process Mode, 125
 MotionSup, 234, 238
 Motion Supervision, 232
 Motion Supervision Max Level, 232
 MotionTask, Task type, 274
 Motor Calibration, 371
 motor replacement, 99
 Motor Torque 1- 10, 369
 Motor Type, 371
 MotSupOn, 235
 MotSupTrigg, 235
 MoveCSync, 67
 MoveJSync, 67
 MoveLSync, 67
 Multitasking, 271

N

- Name, 191, 243, 247
- Name, Transmission Protocol type, 297–298
- Nanopb, 319
- network, 343
- NFS Client, 245
- No. of speed limits, 370
- Nominal Acceleration, 371
- Nominal Deceleration, 371
- Nominal Speed, 189
- non printable characters, 255
- NORMAL, 273
- NoSafety, 273
- NOT, 90
- Not Calibrated, 376
- Null speed signal, 190
- num, 339
- Number of Stored Forces, 369

O

- object queue, 144
- offset_ratio, 61
- Offset Adjust Delay Time, 51
- Offset Speed Ratio, 51
- Open, 73
- OpenDir, 81
- OperationMode, 338
- OR, 89
- outgoing message, 338

P

- PackDNHeader, 85
- PackRawBytes, 77
- parameters
 - accuracy compensation, 108
- Password, 243
- path, 29
- Path Collision Detection, 232, 236
- Path Collision Detection Level, 232, 236
- path correction, 222
- path offset, 222
- pathrecid, 208
- PathRecMoveBwd, 208
- PathRecMoveFwd, 208
- path recorder, 215
- Path Recovery, 207
- PathRecStart, 208
- PathRecStop, 208
- PathRecValidBwd, 208
- PathRecValidFwd, 208
- Path resolution, 190
- PC Interface, 249
- PC SDK client, 261
- performance limits, Machine Synchronization, 165
- persistent variables, 283
- PFRestart, 29
- Phase Inductance, 371
- Phase Resistance, 371
- pitch, 102
- PlannedJoints, 338
- PlannedPose, 338
- Pole Pairs, 371
- polling, 285
- pose, 339
- position accuracy reduction, 58
- position event, 65
- Position signal, 190

- position warnings, Machine Synchronization, 165
- Post-synchronization Time, 369
- power failure functionality, 29
- PredictedTime, 338
- prerequisites, 337
- priorities, 278
- Process, 51
- process support functionality, 31
- Process update time, 190
- programmed speed, Machine Synchronization, 165
- program pointer, 43
- programs
 - editing, 276
- property element, 347
- proportional signal, 32
- Protobuf, 318
- Protobuf-csharp, 319
- Protobuf-net, 319
- protocols
 - Ethernet, 298
 - serial channels, 297

Q

- queue handling, 262
- queue name, 262

R

- r1_calib, 97
- Ramp time, 370
- Ramp Time, 52
- Ramp to real contact, 370
- Ramp when Increase Force, 370
- RAPID components
 - Advanced RAPID, 43
 - Multitasking, 275
 - Sensor Interface, 299
- RAPID editor, 252
- RAPID limitations, Machine Synchronization, 166
- RAPID Message Queue, 260
- RAPID support functionality, 42
- RAPID variables, 335
- rawbytes, 77
- RawBytesLen, 77
- raw data, 76
- ReadAnyBin, 73
- ReadBin, 73
- ReadBlock, 299
- ReadCfgData, 26
- ReadDir, 81
- ReadErrData, 36
- ReadNum, 73
- ReadRawBytes, 77
- ReadStr, 73
- ReadStrBin, 73
- ReadVar, 299
- real, 339
- received message, 352
- reconnect a servo tool, 374
- record, 261
- recorded path, 215
- recorded profile, 180, 184
- record element, 346
- recover path, 207
- References Bandwidth, 370
- relay, 359
- Remote Address, 298
- RemoveDir, 81

- RemoveFile, 81
- RenameFile, 81
- replacements, 99
- reset, 204
- reset axis, 201
- reset follower axis, 57
- resolver offset calibration, 105
- restartdata, 32
- RestoPath, 208
- resultant signal, 88–89
- resume signals, 33
- Rev. Counter not updated, 376
- reversed movement, 229
- Rewind, 73
- RMQEmptyQueue, 266
- RMQFindSlot, 266
- RMQGetMessage, 266
- RMQGetMsgData, 266
- RMQGetMsgHeader, 266
- RMQGetSlotName, 266
- rmqheader, 266
- RMQ Max Message Size, 265
- RMQ Max No Of Messages, 265
- rmqmessage, 266
- RMQ Mode, 265
- RMQReadWait, 266
- RMQSendMessage, 266
- RMQSendWait, 266
- rmqslot, 266
- RMQ Type, 265
- robjoint, 339
- RoboCom Light, 298
- robot alignment, 113
- RobotStudio, 252
- roll, 102
- Rotating move, 191
- Rotating Move, 372
- routine call, 289
- RTP1 protocol, 297
- S**
- SCWrite, 250
- select tasks, 281
- SEMISTATIC, 273
- SenDevice, 299
- send message, 352
- sensor, 222, 295
- sensor_speed, 165
- Sensor Interface, 295
- sensor object, 144
- sensors
 - configuring, 296
- Sensor Synchronization, 139
- Sensor systems, 189
- serial channel communication, 71
- Serial Port, Transmission Protocol type, 297–298
- Server address, 243, 247
- Server path, 243, 247
- service, 374
- service connection, 336
- service program, 54
- service routines, 378
- Service routines, 378
- Servo Tool Change, 355
- SetAllDataVal, 20
- SetDataSearch, 20
- SetDataVal, 20
- SetSysData, 43
- settings.xml, 341
- settings element, 344
- setting up tasks, 276
- set up Collision Detection, 236
- SG Process, 369
- shapedata, 197
- shared resources, 291
- signal, 285, 289
- SiTool, 348
- SiWobj, 348
- SocketAccept, 256
- SocketBind, 256
- SocketClose, 256
- SocketConnect, 256
- SocketCreate, 256
- socketdev, 256
- SocketGetStatus, 257
- SocketListen, 256
- Socket Messaging, 253
- SocketReceive, 256
- SocketSend, 256
- socketstatus, 256
- soft servo, 228
- Soft Stop Timeout, 369
- speed, 230
- speed_ratio, 61
- Speed Absolute Max, 371
- Speed Limit 1 - 6, 370
- Speed Max Uncalibrated, 372
- speed reduction % button, Machine
- Synchronization, 165
- speed warnings, Machine Synchronization, 165
- Squeeze Position 1 -10, 369
- Stall Torque, 371
- STATIC, 273
- stationary world zone, 197
- STCalcForce, 368
- STCalcTorque, 368
- STCalib, 368
- STClose, 368
- StepBwdPath, 32
- STIsCalib, 368
- STIsClosed, 368
- STIsIndGun, 368
- STIsOpen, 368
- STIsServoTool, 368
- STOpen, 368
- StorePath, 208
- STRecalib, 368
- Stress Duty Cycle, 371
- string, 339
- string termination, 255
- StrToByte, 17
- STTune, 368
- STTuneReset, 368
- supervision level, 232, 234, 238
- Supervision Type, 372
- synchronizing tasks, 287
- synchronous movements, 364
- syncident, 287
- syncident, data type, 275
- SyncMoveResume, 208
- SyncMoveSuspend, 208
- SysFail, 273
- SysHalt, 273

SysStop, 273
system parameters
 configuration functionality, 25
 Controller topic, 338
 Motion topic, 338
 Multitasking, 273
 Sensor Interface, 297–298
system resources, 291

T
Task, Task type, 273
Task, type, 273
taskid, 293
taskid, data type, 275
Task in foreground, 278
Task in foreground, Task type, 273
Task Panel Settings, 280
task priorities, 278
TaskRunMec, 292
TaskRunMec, function, 275
TaskRunRob, 292
TaskRunRob, function, 275
tasks, 271, 281, 287
 adding, 276
 data type, 275
 editing programs, 276
 setting up, 276
tasks, data type, 275
temporary world zone, 197
TestAndSet, 291
TestAndSet, function, 275
TextGet, 39
TextTabFreeToUse, 39
TextTabGet, 39
TextTabInstall, 39
text table file, 38
Ti Integration Time Speed Loop, 372
time, 339
tip change calibration, 365
Tip Force 1 - 10, 369
tip wear calibration, 365
tool, 356
tool calibration, 115
tool change calibration, 365
tool changer, 356
tools, 96
torque, 230
torque 1 - torque 6, 370
Torque Absolute Max, 372
torque distribution, 58
torque follower, 58
track motion, 228
Transmission, 372
Transmission Gear High, 203
Transmission Gear Low, 203
Transmission Gear Ratio, 372
Transmission protocol, 243, 247
Transmission protocol, 243, 247
Transmission Protocol, type, 297–298
trapdata, 36
trap routine, 262
TriggC, 67
TriggCheckIO, 67
triggdata, 66
TriggEquip, 66
triggering, 239
TriggInt, 66

TriggIO, 66
triggios, 66
triggiosdnum, 66
TriggJ, 67
TriggL, 67
TriggLIOs, 67
TriggRampAO, 67
TriggSpeed, 32
TriggStopProc, 32
triggstrgo, 66
Trusted, 243, 247
TrustLevel, Task type, 273
TUNE_FRIC_LEV, 120
TUNE_FRIC_RAMP, 120
TuneServo, 120
tuning, 238
tuning, automatic, 118
tuning, manual, 120
type, 343
Type, 243, 247
Type, Task type, 273
Type, Transmission Protocol type, 297–298

U
UDP, 318
UdpUc, 310–311
Udp Unicast Communication, 310–311
uncalib, 97
Uncalibrated Control Master 0, 372
UnpackRawBytes, 77
unsynchronize, 55
Update revolution counter, 376
Upper Joint Bound, 370
Use Connection Relay, 359
Use Linked Motor Process, 51
Use Process, 51
Use ramp time, 370
User ID, 247
user message functionality, 38
Username, 243
Use Robot Calibration, 97

V
Velocity signal, 190
verification, 106

W
waiting for tasks, 287
WaitSyncTask, 287
WaitSyncTask, instruction, 275
WaitUntil, 285
WAN port, 336
WarmStart, 26
world zones, 195
Wrist Move, 131
wrist replacement, 99
Write, 73
WriteAnyBin, 73
WriteBin, 73
WriteBlock, 299
WriteCfgData, 26
WriteRawBytes, 77
WriteStrBin, 73
WriteVar, 299
WZBoxDef, 197
WZCylDef, 197
WZDisable, 198

WZDOSet, 198
WZEnable, 198
WZFree, 198
WZHomeJointDef, 198
WZLimJointDef, 198
WZLimSup, 198
WZSphDef, 197

wzstationary, 197
wztemporary, 197

Y

yaw, 102

Z

zones, 195

Contact us

ABB AB

**Discrete Automation and Motion
Robotics**

S-721 68 VÄSTERÅS, Sweden

Telephone +46 (0) 21 344 400

ABB AS, Robotics

Discrete Automation and Motion

Nordlysvegen 7, N-4340 BRYNE, Norway

Box 265, N-4349 BRYNE, Norway

Telephone: +47 51489000

ABB Engineering (Shanghai) Ltd.

No. 4528 Kangxin Hingway

PuDong District

SHANGHAI 201319, China

Telephone: +86 21 6105 6666

www.abb.com/robotics