ABB

ROBOTICS

Application manual

Controller software IRC5

Trace back information:
Workspace Main version a644
Checked in 2025-02-06
Skribenta version 5.6.018

Application manual

Controller software IRC5

RobotWare 6.16

Document ID: 3HAC050798-001
Revision: V

Original instructions.

Table of contents

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
Product specification - Controller software IRC5 IRC5 with main computer DSQC1000 (or later) and RobotWare 6.	*3HAC050945-001*
Product specification - Controller IRC5 IRC5 with main computer DSQC1000 or later.	*3HAC047400-001*
Operating manual - RobotStudio	*3HAC032104-001*
Operating manual - IRC5 with FlexPendant	*3HAC050941-001*
Technical reference manual - RAPID Instructions, Functions and Data types	*3HAC050917-001*
Technical reference manual - RAPID Overview	*3HAC050947-001*
Technical reference manual - System parameters	*3HAC050948-001*

Revisions

Revision	Description
-	Released with RobotWare 6.0. First release.
A	Released with RobotWare 6.01. • Added Auto Acknowledge Input, see *Auto Acknowledge Input on page 377*. • The functionality of RAPID Message Queue is corrected, see *RAPID Message Queue [included in 616-1, 623-1] on page 312*. • Minor corrections.

Continues on next page

Revision	Description
B	Released with RobotWare 6.02. • Updated the path to the template files, for UdpUc code examples and *Commissioning and service on page 398*. • The TCP ports and protocols are updated for the option Sensor Interface [628-1], see *Configuring sensors on Ethernet channels on page 350*. • Added the functionality EGM Path Correction with corresponding RAPID instructions. • 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 352*.
C	Released with RobotWare 6.03. • Added the functionality *Cyclic bool on page 57*. • Added the functionality *Remote Service Embedded*. • Functionality is added and updated for option *Motion Process Mode [included in 687-1] on page 164*. • The option *Servo Tool Control [included in 635-6]* is replaced by the option *Tool Control [1180-1] on page 387*. • Added the option *I/O Controlled Axes [included in 1180-1] on page 404*. • Minor corrections.
D	Released with RobotWare 6.04. • Added the possibility to configure Cyclic bool, see *Configuration on page 58*. • Updated the section *Common limitations for EGM*. • Added information on how a 7-axis robot can be used with EGM joint mode. • Added new constants for the option *Sensor Interface*, see *Constants on page 352*. • Updated the option *I/O Controlled Axes*, see *I/O Controlled Axes [included in 1180-1] on page 404*. • Remote Service Embedded is updated and renamed to Connected Services. See *Connected Services on page 110*. • Added procedure for Configuring Connected Services. See *Configuring Connected Services on page 116*.
E	Released with RobotWare 6.05. • Added the functionality *User logs on page 132*. • Added new section, *Connected Services on LAN 3 on page 123*. • Added the functionality *Remote control of operating mode on page 377*. • Minor corrections.
F	Released with RobotWare 6.06. • Minor corrections.
G	Released with RobotWare 6.07. • Corrections in the code examples for Robot Reference Interface. • Added info about copying service program file and loading cfg files for the function *Electronically Linked Motors on page 65*. • Added protocol LTPROTOBUF to *Sensor Interface [628-1] on page 347*. • Added section *SFTP Client [614-1] on page 293*. • Added information about EGM Position Stream.
H	Released with RobotWare 6.08. • Added *Press tending mode* to Motion Process Mode. • Added information about directory listing style to *FTP Client*.

Continues on next page

Revision	Description
	• Added information about multiple mechanical units and motion tasks to *Externally Guided Motion [689-1]*. • Maximum length for file paths in *NFS Client* increased to 248 characters. • Added information about SFTP setting to *SFTP Client [614-1] on page 293*. • Added *Collision Avoidance on page 283*. • EGM RAPID instructions `EGMStreamStart` and `EGMStreamStop` corrected.
J	Released with RobotWare 6.09. • Section *Independent Axis [610-1] on page 243* updated with detailed information about limitations for option *Independent Axes*. • Section EGM updated with information about sampling time. • Section *ABB software using PC Interface on page 304* corrected. • Updated information about *Absolute Accuracy*.
K	Released with RobotWare 6.10. • Updated information in *Advanced RAPID* about encoding of text table files. • The information regarding *Externally Guided Motion* is moved to a separate manual, *3HAC073319-001*. • List of limitations of supported robots updated in section *Collision Avoidance on page 283*.
L	Released with RobotWare 6.10.01. • Information regarding disabling of Collision Avoidance updated in section *Collision Avoidance on page 283*.
M	Released with RobotWare 6.11. • Added information regarding servo tool in the following sections: *Overview on page 387*, *Tip management on page 389*, *RAPID components on page 392* • Added accuracy limitation for Absolute Accuracy and MultiMove, see *About Absolute Accuracy on page 135*.
N	Released with RobotWare 6.12. • NOTE added in section *Data orchestration on page 359* that work object data needs to refer to a fixed work object. • Minor change in section *Text table files on page 50*. • Information about the digital output *MotSupOn* updated in section *Signals on page 278*. • Section *System parameters on page 317* updated with information about how to adjust the values of the attributes RMQ Max Message Size and RMQ Max No Of Messages. • Limitation for MultiMove removed in section *About Absolute Accuracy on page 135*.
P	Released with RobotWare 6.13. • Minor corrections in sections *FTP Client [614-1] on page 289*, *SFTP Client [614-1] on page 293* and *NFS Client [614-1] on page 297*. • Updated limitation for *Collision Avoidance on page 283*.
Q	Released with RobotWare 6.13.02. • Updated the section *Connected Services on LAN 3 on page 123*. • Updated limitation regarding lead-through, see *Overview of World Zones on page 237*. • Added the section *SafeMove Assistant on page 286*.
R	Released with RobotWare 6.14. • An incorrect prerequisite regarding a software option is removed for *Tool Control [1180-1] on page 387*.

Continues on next page

Revision	Description
S	Released with RobotWare 6.15. • Added information about deactivation/deactivation and trigger signals, see *Collision Avoidance on page 283*. • Corrected graphic in section *Connected Services registration on page 125*.
T	Released with RobotWare 6.15.07. • Updated the server error details in the section *Advanced page on page 130*. • Added clarification regarding the option *PC Interface*.
U	Released with RobotWare 6.15.08. • Added limitation in *Independent Axis* regarding tool control.
V	Released with RobotWare 6.16. • Added the section *RAPID language and programming environment on page 19*. • Minor corrections.

Open source and 3rd party components

Open source and 3rd party components

ABB products use software provided by third parties, including open source software. The following copyright statements and licenses apply to various components that are distributed inside the ABB software. Each ABB product does not necessarily use all of the listed third party software components. Licensee must fully agree and comply with these license terms or the user is not entitled to use the product. Start using the ABB software means accepting also referred license terms. The third party license terms apply only to the respective software to which the license pertains, and the third party license terms do not apply to ABB products. With regard to programs provided under the GNU general public license and the GNU lesser general public license licensor will provide licensee on demand, a machine-readable copy of the corresponding source code. This offer is valid for a period of three years after delivery of the product.

ABB software is licensed under the ABB end user license agreement, which is provided separately.

RobotWare

For RobotWare, there is license information in the folder **licenses** in the RobotWare distribution package.

OpenSSL

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

CTM

For OleOS, the Linux based operating system used on the conveyor tracking module (CTM), a list of copyright statements and licenses is available in the file **/etc/licenses.txt** located on the CTM board and accessible via the console port or by downloading the file over SFTP.

For the CTM application, a list of copyright statements and licenses is available in the file **/opt/ABB.com/ctm/licenses.txt** located on the CTM board and accessible via the console port or by downloading the file over SFTP.

This page is intentionally left blank

1 Introduction to RobotWare

1.1 Products, classes, and options

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	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. For a description of RobotWare-OS, see the product specification for the robot controller.
RobotWare options	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. **Note** Not all RobotWare options are described in this manual. Some options are more comprehensive and are therefore described in separate manuals.
Process application options	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. The process application options are all described in separate manuals.
RobotWare Add-ins	A RobotWare Add-in is a self-contained package that extends the functionality of the robot system. Some software products from ABB Robotics are delivered as Add-ins. For example track motion (IRT), positioner (IRP), and standalone controller. For more information see the product specification for the robot controller. The purpose of RobotWare Add-ins is also that a robot program developer outside of ABB can create options for the ABB robots, 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*.

Continues on next page

Option groups

For OmniCore, 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 control 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.

1.2 RAPID language and programming environment

General

RAPID is the primary programming language used for ABB Robotics, designed to facilitate the control and automation of industrial robots. It is a high-level language that is both powerful and user-friendly, making it accessible for both novice and experienced programmers. Its syntax and structure are designed to be intuitive, reducing the learning curve for new users.

RAPID is suitable for a wide range of applications, from simple pick-and-place tasks to complex assembly operations. The language is designed to be reliable and robust, ensuring consistent performance in industrial environments.

Key features of RAPID

RAPID uses a structured text format similar to other programming languages like Python or C, which includes loops, conditionals, and variable handling. It excels in handling complex motion commands, allowing precise control over robot movements.

RAPID supports various data types and operations, enabling efficient data handling and processing. Users can create custom functions and procedures, enhancing the flexibility and adaptability of the programming environment.

It allows seamless communication with external devices and systems, making it ideal for integrated automation solutions.

Overall, RAPID is a versatile and powerful tool that enhances the capabilities of ABB robots, making automation more efficient and accessible.

Summary of the RAPID concept

- Hierarchical and modular program structure to support structured programming and reuse
- Routines can be *Functions* or *Procedures*
- Local or global data and routines
- Data typing, including structured and array data types
- User defined names on variables, routines, and I/O
- Extensive program flow control
- Arithmetic and logical expressions
- Interrupt handling
- Error handling
- User defined instructions (appear as an inherent part of the system)
- Backward handler (user definition of how a procedure should behave when stepping backwards)
- Many powerful built-in functions, for example mathematics and robot specific
- Unlimited language (no maximum number of variables etc., only memory limited). Built-in RAPID support in user interfaces, for example user defined pick lists, facilitate working with RAPID.
- Support for Unicode symbols in strings and comments

Continues on next page

Ease of use

Creating and editing RAPID programs is done using the integrated code editors in RobotStudio or on the FlexPendant. Additionally, there is an app for the FlexPendant called Wizard, where RAPID programming is further simplified to block programming.

RAPID programs can range from simple movement procedures to complex structures including sending and receiving data from sensors, cameras, I/O devices, other machines, and more. This to enable a highly flexible automation, utilizing the robot's capability.

Simple RAPID program examples

Hello world

```
MODULE HelloWorld
PROC main()
TPWrite "Hello, World!";
ENDPROC
ENDMODULE
```

In this example:

The module `HelloWorld` defines a module named HelloWorld.

`PROC main()` defines a procedure named main.

`TPWrite "Hello, World!";` is the command that outputs "Hello, World!" to the FlexPendant.

Displaying messages on the FlexPendant

```
MODULE MainModule
  VAR num length;
  VAR num width;
  VAR num area;

  PROC main()
    length := 10;
    width := 5;
    area := length * width;
    TPWrite "The area of the rectangle is " \Num:=area;
  ENDPROC
ENDMODULE
```

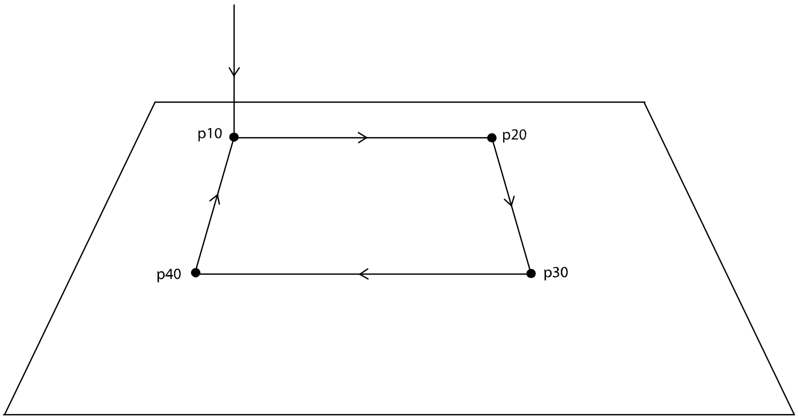
This program will calculate the area of a rectangle and show the answer on the FlexPendant.

```
The area of the rectangle is 50
```

Continues on next page

Draw a square

The robot is holding a pen above a piece of paper on a table. This program will make the robot move the tip of the pen down to the paper and then draw a square.



xx0700000362

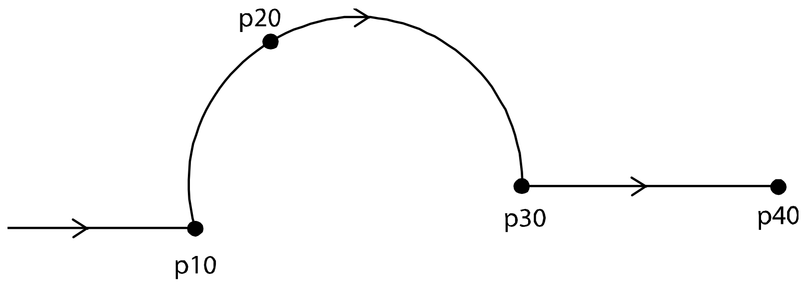
```
PERS tooldata tPen := [ TRUE, [[200, 0, 30], [1, 0, 0 ,0]], [0.8,
     [62, 0, 17], [1, 0, 0, 0], 0, 0, 0]];
CONST robtarget p10 := [ [600, -100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST robtarget p20 := [ [600, 100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST robtarget p30 := [ [800, 100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];
CONST robtarget p40 := [ [800, -100, 800], [0.707170, 0, 0.707170,
     0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];

PROC main()
  MoveL p10, v200, fine, tPen;
  MoveL p20, v200, fine, tPen;
  MoveL p30, v200, fine, tPen;
  MoveL p40, v200, fine, tPen;
  MoveL p10, v200, fine, tPen;
ENDPROC
```

Draw an arc

To add a curve or arc to the previous program, the instruction `MoveC` is added.

```
MoveL p10, v500, fine, tPen;
MoveC p20, p30, v500, fine, tPen;
MoveL p40, v500, fine, tPen;
```

Continues on next page



xx0700000364

References

The RAPID programming language is described in detail in the following documents.

What do you want to know	Where to read about it
• More detailed information about the functionality • What instructions are there for a specific category, for example, move instructions • Descriptions of specific functionality, for example, interrupts or error handling	*Technical reference manual - RAPID Overview*
• Information about a specific instruction, function, or data type	*Technical reference manual - RAPID Instructions, Functions and Data types*
• Details about how the robot controller handles different parts of RAPID	*Technical reference manual - RAPID kernel*

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.

2.1.2 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 `byte` 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 byte.
BitOr	`BitOr` is used to execute a logical bitwise OR operation on data types byte.
BitXOr	`BitXOr` (Bit eXclusive Or) is used to execute a logical bitwise XOR operation on data types byte.
BitNeg	`BitNeg` is used to execute a logical bitwise negation operation (one's complement) on data types byte.
BitLSh	`BitLSh` (Bit Left Shift) is used to execute a logical bitwise left shift operation on data types byte.
BitRSh	`BitRSh` (Bit Right Shift) is used to execute a logical bitwise right shift operation on data types byte.
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 `GetNextSym`.

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
`SetDataSearch`	`SetDataSearch` is used together with `GetNextSym` to retrieve data objects from the system.
`GetDataVal`	`GetDataVal` 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 `GetNextSym`.
`SetDataVal`	`SetDataVal` 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 `GetNextSym`.
`SetAllDataVal`	`SetAllDataVal` 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
`GetNextSym`	`GetNextSym` (Get Next Symbol) is used together with `SetDataSearch` 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 zerovar:=0;
SetAllDataVal "num"\Object:=my_string,zerovar;
```

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 zerovar:=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,zerovar;
ENDWHILE
```

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

```
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 iroutine1;
  IPers counter, int1;
  ...
  counter := counter + 1;
  ...
  Idelete int1;
ENDPROC

TRAP iroutine1
  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 sig1int;

!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 di1=1
  IF di1=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
```

Continues on next page

```
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 with encoding ISO-8859-1.

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 number of the text string 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;
proc1 my_nbr;

PROC proc1 (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 56*.

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).

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 `ai1`.

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 ai1_warning;
VAR intnum ai1_exeeded;

PROC main()
  CONNECT ai1_warning WITH temp_warning;
  CONNECT ai1_exeeded WITH temp_exeeded;
  ISignalAI ai1, AIO_BETWEEN, 130, 120, 0.5, \DPos, ai1_warning;
  ISignalAI \Single, ai1, AIO_ABOVE_HIGH, 130, 120, 0, ai1_exeeded;
  ...
  IDelete ai1_warning;
  IDelete ai1_exeeded;
ENDPROC

TRAP temp_warning
  TPWrite "Warning: Temperature is "\Num:=ai1;
ENDTRAP

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

2.3 Cyclic bool

2.3.1 Cyclically evaluated logical conditions

Purpose

The purpose of cyclically evaluated logical conditions, *Cyclic bool*, is to allow a RAPID programmer to connect a logical condition to a persistent boolean variable. The logical condition will be evaluated every 12 ms and the result will be written to the connected variable.

What is included

The RobotWare base functionality *Cyclic bool* includes:

- instructions for setting up *Cyclic bool*: `SetupCyclicBool`, `RemoveCyclicBool`, `RemoveAllCyclicBool`
- functions for retrieving the status of *Cyclic bool*: `GetMaxNumberOfCyclicBool`, `GetNextCyclicBool`, `GetNumberOfCyclicBool`.

Basic approach

This is the general approach for using *Cyclic bool*. For more detailed examples of how this is done, see *Cyclic bool examples on page 60*.

1. Declare a persistent boolean variable, for example:
   ```
   PERS bool cyclicbool1;
   ```
2. Connect a logical condition to the variable, for example:
   ```
   SetupCyclicBool cyclicbool1, doSafetyIsOk = 1;
   ```
3. Use the variable when programming, for example:
   ```
   WHILE cyclicbool1 = 1 DO
     ! Do what’s only allowed when all safety is ok
     ...
   ENDWHILE
   ```
4. Remove connection when no longer useful, for example:
   ```
   RemoveCyclicBool cyclicbool1;
   ```

Restart and reset behavior

The table below describes the functionality of Cyclic bool when the program pointer is moved or when the controller is restarted.

Action	Description
Program pointer to `main`	The behavior when the program pointer is set to `main` is configurable, see *Configuration on page 58*.
Restart or power fail	This will have no effect. All connected *Cyclic bool* conditions will remain and the evaluation will be restarted immediately.
Reset RAPID	This will remove all connected *Cyclic bool* conditions.
Reset system	

Continues on next page

Configuration

The following behavior of the Cyclic bool functionality can be configured:

Parameter	Description
RemoveAtPpToMain	It is possible to configure if the cyclically evaluated logical conditions shall be removed or not when setting the program pointer to `main`. • *On* - remove. • *Off* - do not remove (default behavior).
ErrorMode	It is possible to configure which error mode to use when the evaluation of a Cyclic bool fails. • *SysStopError*[i] - stop RAPID execution and produce an error log (default behavior). • *Warning* - produce a warning log. • *None* - do nothing.
RecoveryMode	It is possible to configure if a failing Cyclic bool shall be recovered or not. • *On* - try to recover the evaluation of a failing Cyclic bool (default behavior). • *Off* - do not try to recover the evaluation of a Cyclic bool.

[i] Error mode *SysStopError* can only be combined with *RecoveryMode* - *"On"*.

For more information, see *System parameters on page 63*.

Syntax

`SetupCyclicBool Flag Cond [\Signal]`

`Flag` shall be of:

- Data type: `bool`
  - Object type: `PERS` or `TASK PERS`

`Cond` shall be a `bool` expression that may consist of:

- Data types: `num`, `dnum` and `bool`
  - Object type: `PERS`, `TASK PERS`, or `CONST`
- Data types: `signaldi`, `signaldo` or physical di and do
  - Object type: `VAR`
- Operands: '`NOT`', '`AND`', '`OR`', '`XOR`', '`=`', '`(`', '`)`'

`\Signal` shall be of:

- Object type: `signaldo`

`RemoveCyclicBool Flag`

`Flag` shall be of:

- Data type: `bool`
  - Object type: `PERS` or `TASK PERS`

Limitations

- Records and arrays are not allowed in the logical condition.
- A maximum of 60 conditions can be connected at the same time.

Continues on next page

- Any `PERS` `num` or `dnum`, `CONST` `num` or `dnum` or literal `num` or `dnum` used in a condition must be of integer type. If using any decimal value this will cause a fatal error.

2.3.2 Cyclic bool examples

Using digital input and output signals

```
! Wait until all signals are set
PERS bool cyclicbool1 := FALSE;

PROC main()
  SetupCyclicBool cyclicbool1, di1=1 AND do2=1;
  WaitUntil cyclicbool1=TRUE;
  ! All is ok
  ...
  ! Remove connection when no longer in use
  RemoveCyclicBool cyclicbool1;
ENDPROC
```

Using `bool` variables

```
! Wait until all flags are TRUE
PERS bool cyclicbool1 := FALSE;
TASK PERS bool flag1 := FALSE;
PERS bool flag2 := FALSE;

PROC main()
  SetupCyclicBool cyclicbool1, flag1=TRUE AND flag2=TRUE;
  WaitUntil cyclicbool1=TRUE;
  ! All is ok
  ...
  ! Remove connection when no longer in use
  RemoveCyclicBool cyclicbool1;
ENDPROC
```

Using `num` and `dnum` variables

```
! Wait until all conditions are met
PERS bool cyclicbool1 := FALSE;
PERS bool cyclicbool2 := FALSE;
PERS num num1 := 0;
PERS dnum1 := 0;

PROC main()
  SetupCyclicBool cyclicbool1, num1=7 OR dnum1=10000000;
  SetupCyclicBool cyclicbool2, num1=8 OR dnum1=11000000;
  WaitUntil cyclicbool1=TRUE;
  ...
  WaitUntil cyclicbool2=TRUE;
  ...
  ! Remove all connections when no longer in use
  RemoveAllCyclicBool;
ENDPROC
```

Continues on next page

Using alias variables

```
! Wait until all conditions are met
ALIAS bool aliasBool;
ALIAS num aliasNum;
ALIAS dnum aliasDnum;

PERS bool cyclicbool1 := FALSE;
PERS aliasBool flag1 := FALSE;
PERS aliasNum num1 := 0;
PERS aliasDnum dnum1 := 0;

PROC main()
  SetupCyclicBool cyclicbool1, flag1=TRUE AND (num1=7 OR
       dnum1=10000000);
  WaitUntil cyclicbool1=TRUE;
  ! All is ok
  ...
  ! Remove connection when no longer in use
  RemoveCyclicBool cyclicbool1;
ENDPROC
```

Using user defined constants for comparison

```
! Wait until all conditions are met
PERS bool cyclicbool1;
PERS bool flag1 := FALSE;
PERS num num1 := 0;
PERS dnum dnum1 := 0;
CONST bool MYTRUE := TRUE;
CONST num NUMLIMIT := 10;
CONST dnum DNUMLIMIT := 10000000;

PROC main()
  SetupCyclicBool cyclicbool1, flag1=MYTRUE AND num1=NUMLIMIT AND
       dnum1=DNUMLIMIT;
  WaitUntil cyclicbool1=TRUE;
  ! All is ok
  ...
  ! Remove connection when no longer in use
  RemoveCyclicBool cyclicbool1;
ENDPROC
```

Continues on next page

Handing over arguments by reference

If the instruction `SetupCyclicBool` is used inside a called procedure, it is possible to hand over conditions as arguments to that procedure.

Using conditions passed by reference works only for `SetupCyclicBool`. Conditions passed by reference has the same restrictions as conditions for `SetupCyclicBool`.

This functionality works regardless if the modules are `Nostepin` or has any other module attributes.

```
MODULE MainModule
CONST robtarget p10 := [[600,500,225.3], [1,0,0,0], [1,1,0,0],
     [11,12.3,9E9,9E9,9E9,9E9]];
PERS bool m1;
PERS bool Flag2 := FALSE;

PROC main()
  ! The Expression (di_1 = 1) OR Flag2 = TRUE shall be
  ! used by SetupCyclicBool
  my_routine (di_1 = 1) OR Flag2 = TRUE;
ENDPROC

PROC my_routine(bool X)
  ! It is possible to pass arguments between several procedures
  MySetCyclicBool X;
ENDPROC

PROC MySetCyclicBool (bool Y)
  RemoveCyclicBool m1;
  ! Only SetupCyclicBool can pass arguments
  SetupCyclicBool m1, Y;
  ! If conditions passed by reference shall be used by any other
  ! instruction, the condition must be setup with SetupCyclicBool
  ! before it can be used.
  WaitUntil m1;
  MoveL p10, v1000, z30, tool2;
ENDPROC
ENDMODULE
```

2.3.3 System parameters

About the system parameters

This is a brief description of the system parameters used by *Cyclic bool*. For more information about the parameters, see *Technical reference manual - System parameters*.

Type *Cyclic bool settings*

The system parameters used by *Cyclic bool* belong to the type *Cyclic bool settings* in topic *Controller*.

Parameter	Description
Name	There can be only one instance of each allowed value, that is a maximum of three instances in the system. All three instances will be installed in the system (default) and cannot be removed. • *RemoveAtPpToMain* • *ErrorMode* • *RecoveryMode*
RemoveAtPpToMain	The action value *RemoveAtPpToMain* is used to configure if a connected Cyclic bool shall be removed or not when setting the program pointer to Main.
ErrorMode	The action value *ErrorMode* is used to configure which error mode to use when evaluation fails.
RecoveryMode	The action value *RecoveryMode* is used to configure which recovery mode to use when evaluation fails.

2.3.4 RAPID components

About the RAPID components

This is an overview of all RAPID instructions, functions, and data types in *Cyclic bool*.

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

Instructions

Instruction	Description
`SetupCyclicBool`	`SetupCyclicBool` connects a logical condition to a boolean variable.
`RemoveCyclicBool`	`RemoveCyclicBool` removes a specific connected logical condition.
`RemoveAllCyclicBool`	`RemoveAllCyclicBool` removes all connected logical conditions.

Functions

Function	Description
`GetMaxNumberOfCyclicBool`	`GetMaxNumberOfCyclicBool` retrieves the maximum number of cyclically evaluated logical condition that can be connected at the same time.
`GetNextCyclicBool`	`GetNextCyclicBool` retrieves the name of a connected cyclically evaluated logical condition.
`GetNumberOfCyclicBool`	`GetNumberOfCyclicBool` retrieves the number of a connected cyclically evaluated logical condition.
`IsCyclicBool`	`IsCyclicBool` is used to test if a persistent boolean is a Cyclic bool or not, i.e. if a logical condition has been connected to the persistent boolean variable with the instruction `SetupCyclicBool`.

Data types

Cyclic bool includes no data types.

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 routine 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 as a mechanical unit. See *Application manual - Additional axes and standalone 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 routine 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 the option *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 *Name* in the type *Joint*. 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 *Name* in the type *Process*.
Lock Joint in Ipol	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 *Name* in the type *Linked M Process*.

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 *Max Follower Offset* 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 *Max Offset Speed* is exceeded, emergency stop is activated.
Offset Speed Ratio	Defines how large part of the *Max Offset Speed* that can be used to compensate for position error.

Continues on next page

Parameter	Description
Ramp Time	Time for acceleration up to *Max Offset Speed*. The proportion constant for position regulation is ramped from zero up to its final value (*Master Follower kp*) during *Ramp Time*.
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. • 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 Follower 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 routine for a follower axis

About the service routine

When the follower axis is configured as a mechanical unit and connected to a master axis, the service routine can be used to:

- calibrate the follower axis
- reset follower after a position error
- tune a torque follower axis, see *Tuning a torque follower on page 75*.

Copy service routine file to HOME

Copy the file *linked_m.sys* from directory:
hd0a\<active system>\PRODUCTS\RobotWare_6.0x.xxxx\utility\LinkedMotors
to the *HOME* directory of the active system.

Load cfg files

Load the configuration files *LINKED_M_MMC.cfg* and *LINKED_M_SYS.cfg*. These are located in the directory:
...\utility\LinkedMotors.

Loading configuration files can be done with RobotStudio or FlexPendant. How to do this is described in:

Tool	Description of loading cfg files
RobotStudio	Section *Loading a configuration file* in *Operating manual - RobotStudio*.
FlexPendant	Section *Loading system parameters* in *Operating manual - IRC5 Integrator's guide*.

Restart the controller after loading the configuration files.

Data variables

When the service routine starts, it 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 78*.

Start service routine

Note

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

Step	Action
1	In the program view, tap **Debug** and select **Call Routine...**.
2	Select **Linked_m** and tap **Go to**.
3	Press and hold the enabling device.
4	Press the RUN button to start the service routine.

Continues on next page

Step	Action
5	Tap **Menu 1**. The follower axes that are set up in the system are shown in the task bar.
6	Tap the follower axis you want to use the service routine for. The main menu of the service program is now shown.

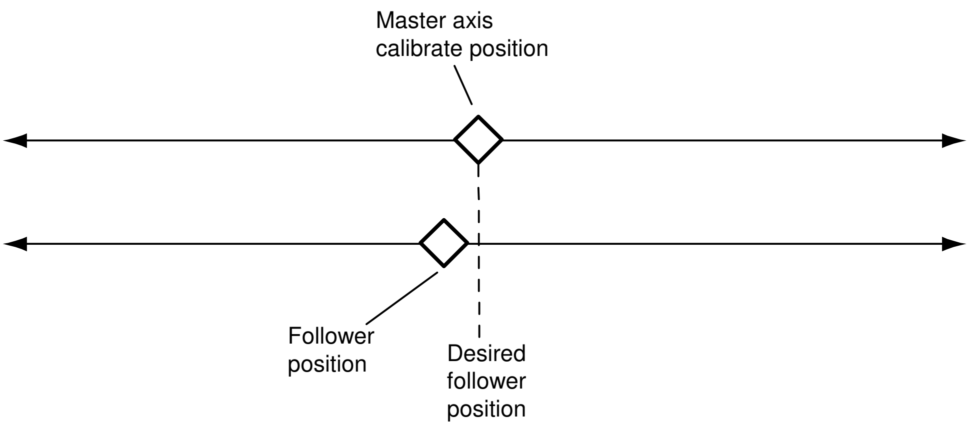
Menu buttons

Button	Description
AUTO	Automatically moves the follower axis to the position corresponding to the master axis, see *Reset follower automatically on page 74*.
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 72*. 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 *Unsynchronize on page 72*.
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 72*.
3. Jog the follower to the desired position. See *Jog follower axis on page 72*.
4. Fine calibrate follower axis. See *Fine calibrate on page 73*.

Unsynchronize

Step	Action
1	In the main menu of the service routine, 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

Fine calibrate

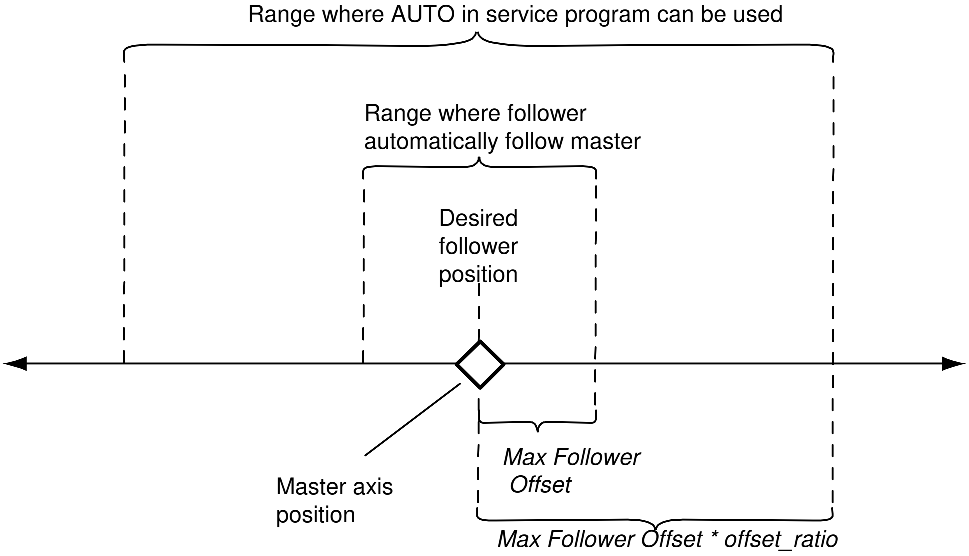
Step	Action
1	In the **ABB** menu, select **Calibration**.
2	Select the mechanical unit that the follower axis belongs to.
3	Tap the button **Calib. Parameters**.
4	Tap **Fine Calibration...**.
5	In the warning dialog that appears, tap **Yes**.
6	Select the axis that is used as follower axis and tap **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 routine must be used to move the follower back within the tolerance limits. This can be done automatically in the service routine 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 routine, 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 routine, 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 *Max Follower Offset* (or use AUTO when you are close enough).

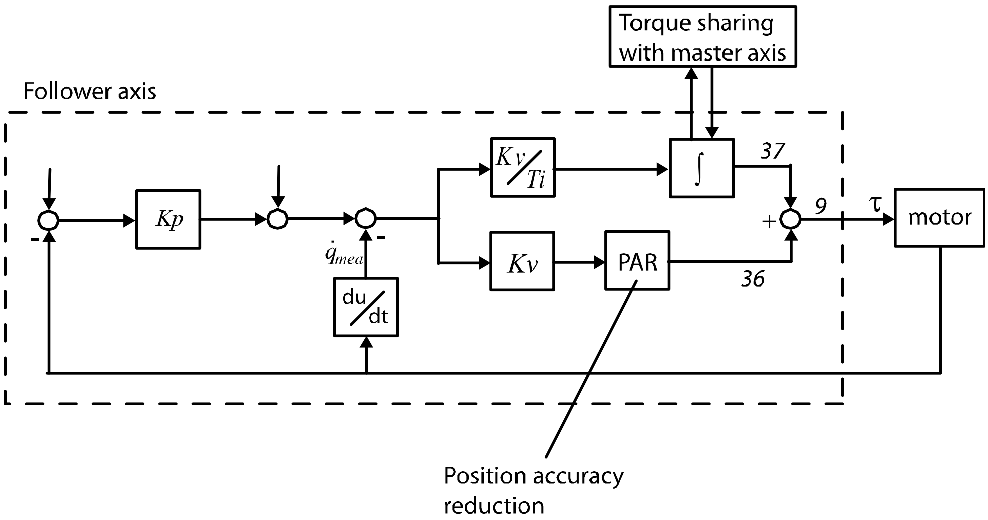
2.4.4 Tuning a torque follower

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 routine to tune a torque follower

About the service routine for torque follower

The service routine *Linked_M* can be used to find suitable values of some parameters for torque follower configuration. When the values are found, the system parameters are updated and a new fine calibration is done. 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 routine (as described by the first steps in *Start service routine on page 70*).	
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

	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 the service routine

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 routine. If they are not read correctly, the variables need to be edited in the service routine.

Data descriptions

Data variable	Description
l_f_axis_name	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.
l_f_mecunt_n	The name of the mechanical unit for the follower axis. Refers to the system parameter *Name* in the type *Mechanical Unit*. String array with 5 elements, one for each follower axis. If you only have one linked motor, use only the first element.
l_f_axis_no	Defines which axis in the mechanical unit (l_f_mecunt_n) 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.
l_m_mecunt_n	The name of the mechanical unit for the master axis. Refers to the system parameter *Name* in the type *Mechanical Unit*. String array with 5 elements, one for each master axis. If you only have one linked motor, use only the first element.
l_m_axis_no	Defines which axis in the mechanical unit (`l_m_mecunt_n`) 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.
offset_ratio	Defines the range where the AUTO function in the service program reset the follower axis. `offset_ratio` defines this range as a multiple of the range where the follower automatically follow the master (defined with the parameter *Max Follow Offset*). If the follower has a position error that is larger than *Max Follower Offset* * `offset_ratio`, the follower must be reset manually. For more information, see *Reset follower axis on page 74*.
speed_ratio	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

Data variable	Description
displacement	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. 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.

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 **l_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 **l_f_mecunt_n** and **l_m_mecunt_n**.
9	In the **Program Data** menu, select **num** and repeat step 3-7 for **l_f_axis_no**, **l_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.

l_f_axis_name

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

l_f_mecunt_n

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

l_f_axis_no

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

l_m_mecunt_n

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

Continues on next page

l_m_axis_no

Represented axis	Element and value in l_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 86*.

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
`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.
`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.

Continues on next page

Instruction	Description
`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).
`TriggCheckIO`	`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 `triggdata`. `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. `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. 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).
`TriggRampAO`	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 `triggdata`. `TriggRampIO` defines where the ramping of the signal is to start and the length of the ramping.
`TriggL`	`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. `TriggL` executes up to 8 position events stored as `triggdata`. These must be defined before calling `TriggL`.
`TriggC`	`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. `TriggC` executes up to 8 position events stored as `triggdata`. These must be defined before calling `TriggC`.
`TriggJ`	`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. `TriggJ` executes up to 8 position events stored as `triggdata`. These must be defined before calling `TriggJ`.
`TriggLIOs`	`TriggLIOs` is a move instruction, similar to `MoveL`. In addition to the movement the `TriggLIOs` instruction can set output signals at fixed positions. `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 `triggios`, `triggiosdnum`, or `triggstrgo`.

Functions

Fixed Position Events includes no RAPID functions.

Continues on next page

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 *Event preset time*. *Event preset time* belongs to the type *Motion System* in the topic *Motion*.

2.5.3 Code examples

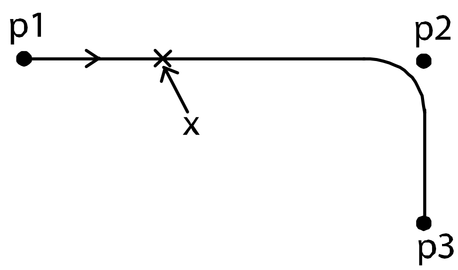
Example without Fixed Position Events

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

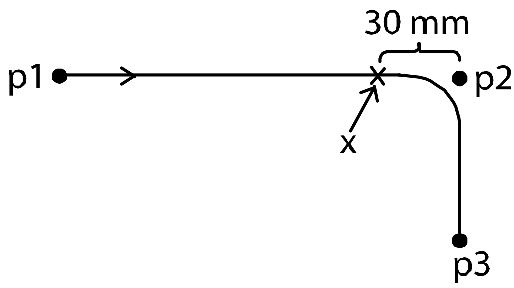
Setting the output signal 30 mm from the target can be arranged by defining the position event and then moving the robot while the system is executing the position event.

```
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

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

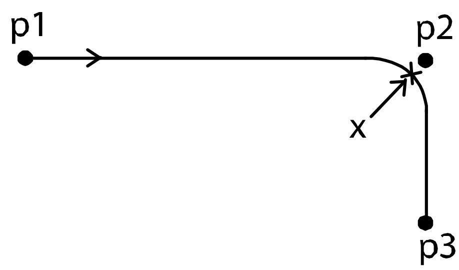
Example with MoveLSync instruction

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 I/O device handling

2.6.1 Introduction to file and I/O device handling

About file and I/O device handling

The RobotWare file and I/O device 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 for file and I/O device 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 I/O devices.
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, RobotWare gives you access to:

- instructions for manipulations of a file or I/O device
- instructions for writing to file or I/O device
- instruction for reading from file or I/O device
- functions for reading from file or I/O device.

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 91*.

1. Open a file or I/O device.
2. Read or write to the file or I/O device.
3. Close the file or I/O device.

Limitations

Access to files and I/O devices 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 I/O device. It can be linked to the physical unit with the instruction `Open` 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 I/O device for reading or writing.
`Close`	`Close` is used to close a file or I/O device.
`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 I/O device.
`WriteBin`	`WriteBin` is used to write a number of bytes to a binary I/O device or file.
`WriteStrBin`	`WriteStrBin` is used to write a string to a binary I/O device or file.
`WriteAnyBin`	`WriteAnyBin` is used to write any type of data to a binary I/O device or file.
`ReadAnyBin`	`ReadAnyBin` is used to read any type of data from a binary I/O device 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 I/O device.
`ReadStr`	`ReadStr` is used to read a string from a character based file or I/O device.
`ReadBin`	`ReadBin` is used to read a byte (8 bits) from a file or I/O device. This function works on both binary and character based files or I/O devices.
`ReadStrBin`	`ReadStrBin` is used to read a string from a binary I/O device or file.

2.6.2.3 Code examples

Communication with character based file

This example shows 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 file

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

```
PROC write_bin_chan()
  VAR iodev file1;
  VAR num out_buffer{20};
  VAR num input;
  VAR robtarget target;

  Open "HOME:" \File:= "FILE1.DOC", file1 \Bin;

  ! Write control character enq
  out_buffer{1} := 5;
  WriteBin file1, out_buffer, 1;

  ! Wait for control character ack
  input := ReadBin (file1 \Time:= 0.1);
  IF input = 6 THEN
    ! Write "Hello" followed by new line
    WriteStrBin file1, "Hello\0A";
```

Continues on next page

```
    ! Write current robot position
    target := CRobT(\Tool:= tool1\WObj:= wobj1);
    WriteAnyBin file1, 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 file
    WriteBin file1, out_buffer, 5;
  ENDIF
  Close file1;
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 I/O device, and to read and unpack data. This is particularly useful when communicating via a fieldbus, such as DeviceNet.

What is included

To handle raw data communication, RobotWare 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 95*.

1. Pack data into a `rawbytes` variable (data of type `num`, `byte` or `string`).
2. Write the `rawbytes` variable to a file or I/O device.
3. Read a `rawbytes` variable from a file or I/O device.
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 and I/O devices 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`	`rawbytes` is used as a general data container. It can be filled with any data of types `num`, `byte`, or `string`. It also stores the length of the valid data (in bytes). `rawbytes` can contain up to 1024 bytes of data. The supported data formats are listed in the instruction `PackRawBytes`, in *Technical reference manual - RAPID Instructions, Functions and Data types*.

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`	`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. `ClearRawBytes` can also be used to clear only the last part of a `rawbytes` variable.
`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 or I/O device.
`ReadRawBytes`	`ReadRawBytes` is used to read data of type `rawbytes` from any binary file or I/O device.

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 103*.

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, RobotWare 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 99*.

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 `OpenDir`.

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
`OpenDir`	`OpenDir` is used to open a directory.
`CloseDir`	`CloseDir` is used to close a directory.
`MakeDir`	`MakeDir` is used to create a new directory.
`RemoveDir`	`RemoveDir` is used to remove an empty directory.
`CopyFile`	`CopyFile` is used to make a copy of an existing file.
`RenameFile`	`RenameFile` 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.
`RemoveFile`	`RemoveFile` 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
`ReadDir`	`ReadDir` is used to retrieve the name of the next file or subdirectory under a directory that has been opened with the instruction `OpenDir`. Note that the first items read by `ReadDir` are `.` (full stop character) and `..` (double full stop characters) symbolizing the current directory and its parent directory.
`FileSize`	`FileSize` is used to retrieve the size (in bytes) of the specified file.
`FSSize`	`FSSize` (File System Size) is used to retrieve the size (in bytes) of the file system in which a specified file resides.`FSSize` can either retrieve the total size or the free size of the system.
`IsFile`	`IsFile` 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 lsdir(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 though 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 freefsyssize;
VAR num f_size;

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

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

! Copy file if enough space free
IF f_size < freefsyssize 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 93*.

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 103*.

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 require 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 93*.

```
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 106*.

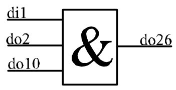
These parameters belong to the type *Cross Connection* in the topic *I/O System*.

Parameter	Description
Name	Specifies the name of the cross connection.
Resultant	The I/O signal that receive the result of the cross connection as its new value.
Actor 1	The first I/O signal to be used in the evaluation of the *Resultant*.
Invert actor 1	If *Invert actor 1* is set to *Yes*, then the inverted value of *Actor 1* is used in the evaluation of the *Resultant*.
Operator 1	Operand between *Actor 1* and *Actor 2*. Can be either of the operands: • `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. **Note** The operators are calculated left to right (*Operator 1* first and *Operator 4* last).
Actor 2	The second I/O signal (if more than one) to be used in the evaluation of the *Resultant*.
Invert actor 2	If *Invert actor 2* is set to *Yes*, then the inverted value of *Actor 2* is used in the evaluation of the *Resultant*.
Operator 2	Operand between *Actor 2* and *Actor 3*. See *Operator 1*.
Actor 3	The third I/O signal (if more than two) to be used in the evaluation of the *Resultant*.
Invert actor 3	If *Invert actor 3* is set to *Yes*, then the inverted value of *Actor 3* is used in the evaluation of the *Resultant*.
Operator 3	Operand between *Actor 3* and *Actor 4*. See *Operator 1*.
Actor 4	The fourth I/O signal (if more than three) to be used in the evaluation of the *Resultant*.
Invert actor 4	If *Invert actor 4* is set to *Yes*, then the inverted value of *Actor 4* is used in the evaluation of the *Resultant*.
Operator 4	Operand between *Actor 4* and *Actor 5*. See *Operator 1*.
Actor 5	The fifth I/O signal (if all five are used) to be used in the evaluation of the *Resultant*.
Invert actor 5	If *Invert actor 5* is set to *Yes*, then the inverted value of *Actor 5* is used in the evaluation of the *Resultant*.

2.8.3 Examples

Logical AND

The following logical structure...



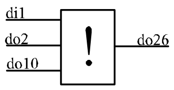
xx0300000457

... 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...



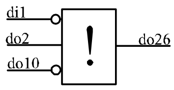
xx0300000459

... 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)...



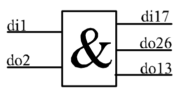
xx0300000460

... 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

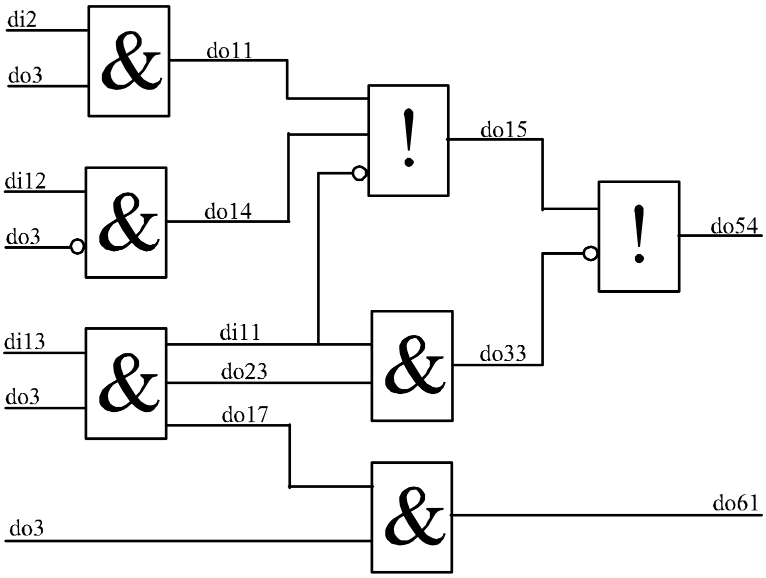
Continues on next page

... 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.

2.9 Connected Services

2.9.1 Overview

Description

Connected Services (was known as Remote Service previously) is a functionality available for ABB robot controllers that connects to ABB cloud.

Earlier the Connected Services functionality had been implemented on an external hardware (Remote Service Box) connected to the Service port of the controller. Remote Service Box had provided service data collection and the external connectivity (Wireless GPRS, 3G, or wired).

Connected Services is the software version of Remote Service Box inside RobotWare.

Purpose

The primary purpose of Connected Services is to remove the need of external hardware if the robot controller are connected to Internet by the customer on its WAN port.

Connected Services is then available natively as a plug and connect solution in RobotWare. The setup concept will be:

- Provide internet connectivity to the controller.
- Enable and register the connected controller to Connected Services.

An ABB 3G/4G/WiFi gateway or other external devices will be made available in the future to use wireless connectivity.

What is included

The RobotWare base functionality Connected Services gives you access to:

- a Connected Services Agent software to manage the connectivity and the Service data collection.
- System Parameters used to enable and configure the connectivity.
- dedicated event logs for key events of Connected Services.
- status and information pages available in System Info.

Prerequisites

The Connected Services function requires the controller to be defined in a Service Agreement. Contact the local ABB Service to create a Service Agreement with the Connected Services and get access to MyRobot website to perform the registration after the connection.

> **Note**
>
> MyRobot is the ABB website which gives access to the Service information of a Robot Controller under a Service Agreement.

Continues on next page

Basic workflow

Following is the basic workflow for setting up Connected Services.

1. Configure Internet connectivity to the robot controller.
2. Enable Connected Services and startup connection.
3. Register the controller through MyRobot registration page.

Once Connected Services is connected and registered, the service data collection will run transparently in the background.

Note

Use **System Info Connected Services** pages for information and local registration.

Use **MyRobot** website for all Connected Service features and connected service side registration

Limitations

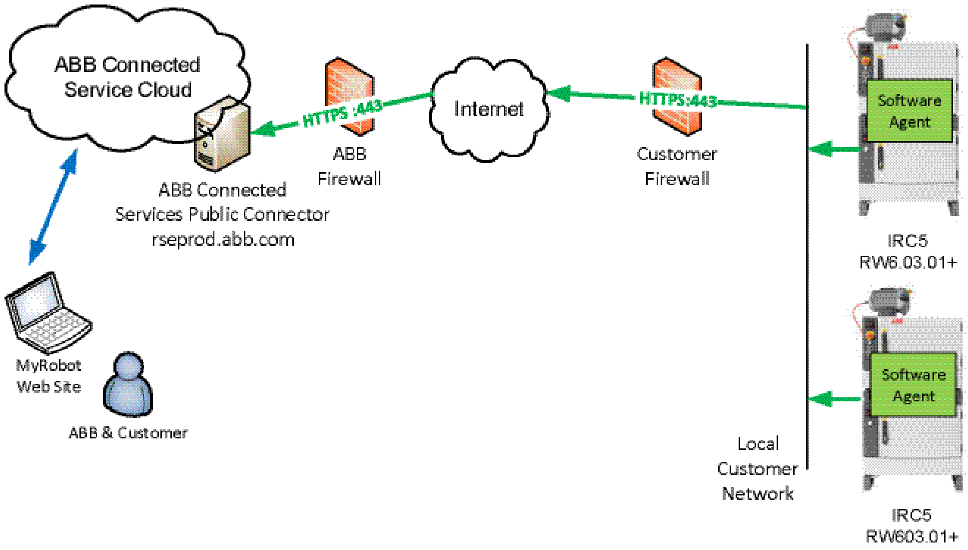
Following are the limitations of Connected Services:

- The controller identification is done using the controller serial number and must match the serial number defined in the Service Level Agreement.
- The customer must also provide for the robot controller the connectivity to public internet , use the ABB wireless gateway or third party supplier when available.

2.9.2 Connected Services connectivity

Connected Services connection concept

The concept of Connected Services is that a virtual Software Agent is implemented inside the controller and it communicates securely with the ABB Connected Services cloud through Internet. The communication is secured and encrypted using HTTPS (Secure HTTP) and only from the controller to ABB CSC connector to keep the customer network isolated from any external Internet access. The following figure describes these concepts:

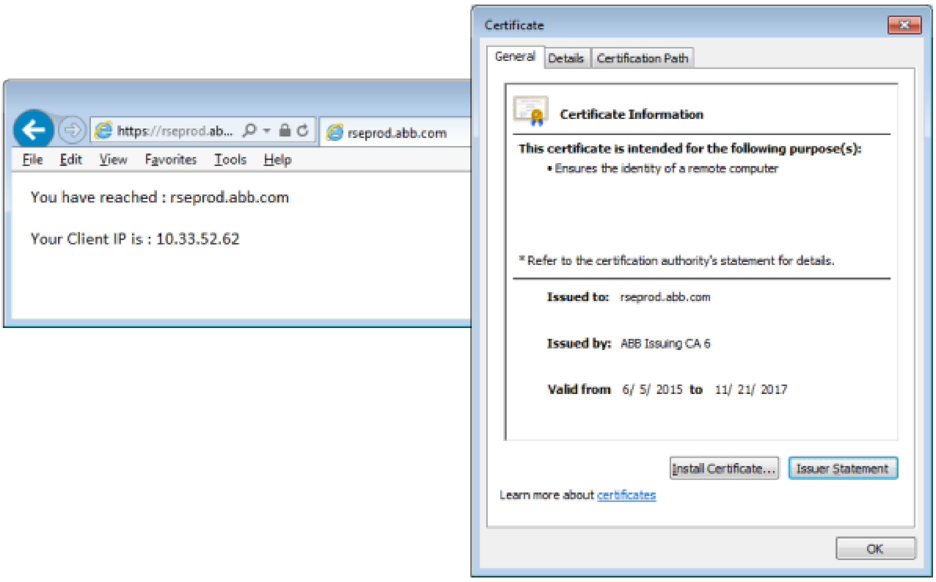


xx1500003224

Continues on next page

Troubleshooting

You can verify the connectivity from the controller to the Connected Services Public Connector server from your location. This is done by connecting a PC (instead of the controller) with the same network configuration (WAN IP/Mask, DNS, Route), and open the path to the root of the server (*https://rseprod.abb.com*) in a browser. The connectivity is validated if the DNS name has been resolved, the browser presents a page indicating the CS server, and secured with an ABB certificate as shown in the following figure.



xx1500003225

2.9.3 Configuration - system parameters

Connected Services Connection

The following parameters belong to the topic *Communication* and the type *Connected Services*. For more information, see *Technical reference manual - System parameters*.

Parameter	Description
Enabled	Enable or disable CS. If CS is disabled there will be no communication from the Controller.
Connection Type	Indicates if the communication is done on Customer Network or by using ABB Mobile Gateway Solution (to be implemented in future deliveries).
Connection Cost	Adapt the polling rates and traffic volume to the type of connectivity available: • Command polling (low) 1 min, (medium) 10 min, (high) 1 hour. • Register polling (low) 10 min, (medium) 30 min, (high) 2 hour.
Proxy Used, Name, Port	Indicates if a proxy is required to access Internet and its name and port.
Proxy Auth, User, password	Defines if the proxy is authenticated or not, with related credentials (user, password). **WARNING** The proxy password is stored in plain text.
Gateway IP Address	IP address of the ABB Mobile Gateway Solution if used (to come in future deliveries).

WAN configuration

The WAN IP/Mask/Gateway configuration is done in the **Boot Application Settings**. The WAN Ethernet port configuration which gives access to the Internet needs to be done on the controller. The port is defined by its IP, Mask, and possible Gateway. For details about WAN configuration, see *Hardware overview* in the *Application manual - EtherNet/IP Scanner/Adapter*.

DNS configuration

These parameters belong to the topic *Communication* and the type *DNS Client*. A DNS server need to be defined to resolve the name of the ABB Connected Services Connector (rseprod.abb.com) to its IP address if ABB Mobile Gateway is not used. For more details, see *Type DNS Client* in *Technical reference manual - System parameters*.

Note

For quick testing, use DNS as 8.8.8.8 (Google DNS) , then switch to customer recommended DNS server IP.

Continues on next page

IP Routing configuration

These parameters belong to the topic *Communication* and the type *IP Routing*. In some cases it is necessary to define some routing parameters to indicate which specific external device is used as a gateway to access the Internet on customer network. By default, an IP route is created based on the gateway defined on the WAN Port. But it is possible to add a specific route if the default gateway should not be used. For more details, see *Type IP Route* in *Technical reference manual - System parameters*.

Note

If the Internet Gateway is not the main Gateway, the traffic to rseprod.abb.com and the DNS must be defined as additional routes.

For example, if Internet Gateway has IP address 100.100.100.22, rseprod.abb.com has IP address 138.227.175.43 (verify by nslookup) and DNS has IP address 8.8.8.8, then you must define the following two routes:

- Route 138.227.175.43/31 to 100.100.100.22
- Route 8.8.8.8/31 to 100.100.100.22

2.9.4 Configuring Connected Services

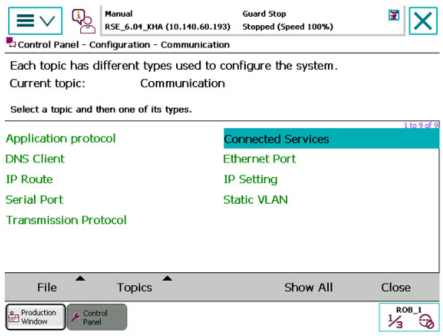
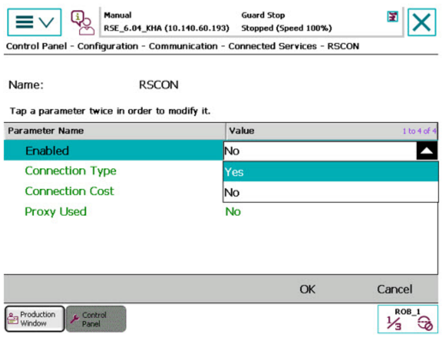
Overview

This section explains how the Connected Services is configured with the controller, when Internet is available on the default gateway. There are two separate network setups:

- Direct internet connection without proxy.
- Internet connectivity through a proxy.

Direct internet connection

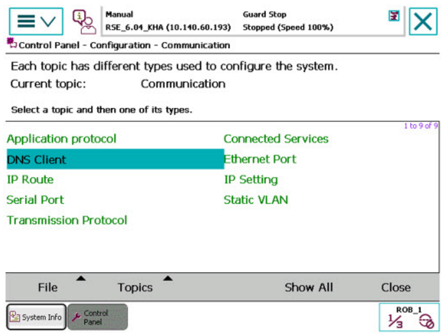
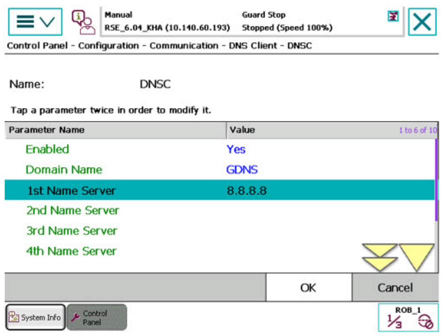
The following procedure provides information about configuring the Connected Services from the FlexPendant when there is direct internet connection from the controller.

	Action	Illustration
1	In the **ABB** menu, select **Control Panel**.	
2	Select **Configuration**.	
3	From **Topics**, select **Communication**.	
4	Select **Connected Services** and edit **RSCON**.	 xx1600001326
5	In **Enabled**, select *Yes*	 xx1600001327
6	Tap **OK** and restart the controller to take effect of the changes.	

Continues on next page

Direct internet connection with manual DNS

The following procedure provides information about configuring the Connected Services from the FlexPendant when there is direct internet connection with manual DNS.

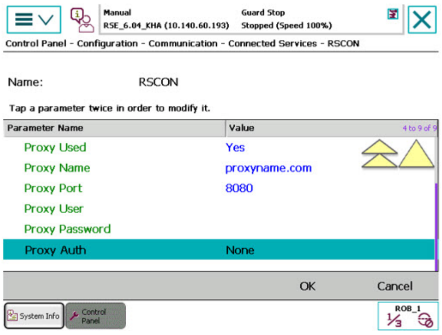
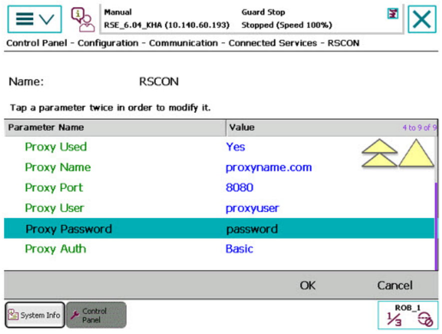
	Action	Illustration
1	In the **ABB** menu, select **Control Panel**.	
2	Select **Configuration**.	
3	From **Topics**, select **Communication**.	
4	Select **DNS Client** and edit **DNS Client**.	 xx1600001329
5	Edit 1st Name Server	 xx1600001330
6	Tap **OK** and restart the controller to take effect of the changes.	

Internet connection with proxy

The following procedure provides information about configuring the Connected Services from the FlexPendant when there is internet connection with proxy.

	Action	Illustration
1	In the **ABB** menu, select Control Panel.	
2	Tap **Configuration**.	
3	From **Topics**, select **Communication**.	
4	Select **Connected Services** and in **Proxy Used**, select *Yes*.	

Continues on next page

	Action	Illustration
5	In **Proxy Auth**, select *None* for no authentication from the drop-down list.	 xx1600001331
6	In **Proxy Auth** , select *Basic* for basic authentication from the drop-down list. • Define the proxy name, proxy port, user name, and password for the basic authentication.	 xx1600001332
7	Tap **OK** and restart the controller to take effect of the changes.	

Note

Manually define the DNS, if it is not provided automatically when proxy is used.

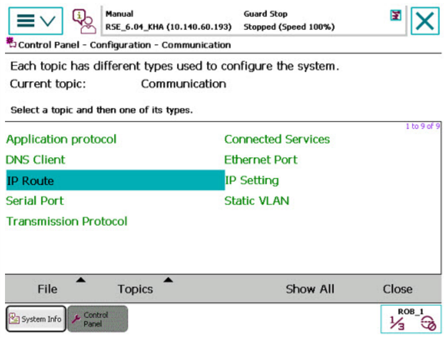
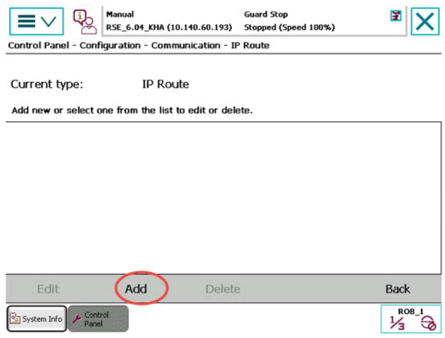
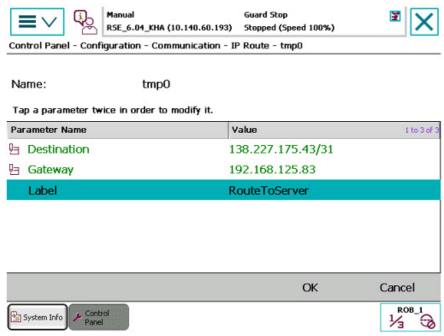
2.9.5 Configuring Connected Services using gateway box

Overview

This section explains how the Connected Services is configured using an external Internet gateway (3G/4G, WiFi, etc) not defined as default gateway in the controller. In this case, additional routes are needed to reach the external Internet gateway.

Controller with DHCP

The following procedure provides information about configuring the Connected Services from the FlexPendant when there is controller with DHCP.

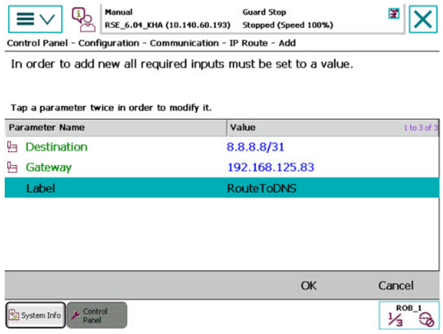
	Action	Illustration
1	In the **ABB** menu, select **Control Panel**.	
2	Select **Configuration**.	
3	From **Topics**, select **Communication**.	
4	Select **IP Route** and tap **Add**.	 xx1600001333  xx1600001334
5	Enter the details for **Destination**, **Gateway**, and **Label**. • In this example, **Destination**: 138.227.175.43/31 is the rsepro.abb.com IP • **Gateway**: 192.168.125.83	 xx1600001335

Continues on next page

	Action	Illustration
6	Tap **OK** and restart the controller to take effect of the changes.	

Controller with DHCP and manual DNS

The following procedure provides information about configuring the Connected Services from the FlexPendant for controller with DHCP and manual DNS.

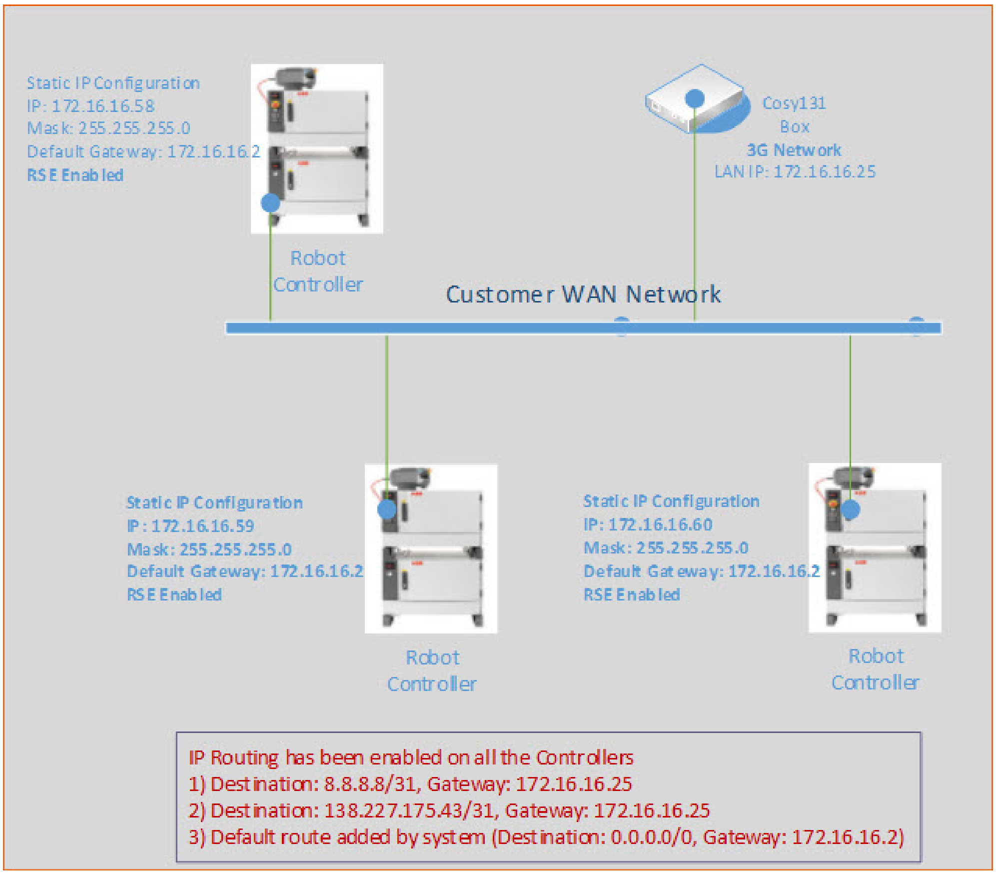
	Action	Illustration
1	In the **ABB** menu, select **Control Panel**.	
2	Select **Configuration**.	
3	From **Topics**, select **Communication**.	
4	Select **IP Route** and tap **Add**.	
5	Enter the details for **Destination**, **Gateway**, and **Label**. • If DNS IP is entered manually, add the routing for the DNS IP. • In this example, **Destination**: 8.8.8.8/31 is Google DNS.	 xx1600001337
6	Tap **OK** and restart the controller to take effect of the changes.	

Continues on next page

Gateway box on customer network

When gateway box is configured for multiple controllers, then the LAN IP of the gateway box changes. For more information about how to do setting for the gateway box for multiple controllers, see *Product manual - Connected Services*.

The gateway box should be connected to the customer network. And, the LAN IP should be modified to match with the customer network IP segment. A typical network infrastructure is shown below.



xx1600001338

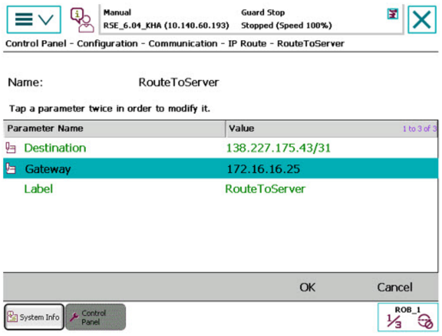
Note

The network infrastructure is an example to demonstrate the network topology.

Steps to configure DNS manually

	Action	Illustration
1	In the **ABB** menu, select **Control Panel**.	
2	Select **Configuration**.	
3	From **Topics**, select **Communication**.	
4	Select **IP Route** and tap **Add**.	

Continues on next page

	Action	Illustration
5	Enter the details for **Destination**, **Gateway**, and **Label**. • Enter the Gateway IP as box IP. In this example, it is 172.16.16.25.	 xx1600001339
6	Tap **OK** and restart the controller to take effect of the changes.	

Note

Manually define the DNS, if it is not provided automatically. Also, define a route to go through the gateway box for the DNS IP.

2.9.6 Connected Services on LAN 3

Overview

When internet is not provided on production WAN network, we can configure and use LAN 3 to connect with the Connected Services server.

LAN 3 (available on port X5) acts as a separate switch and its IP can be configured manually.

Note

There is a risk of conflict between PROFINET and LAN 3 in some configuration. It is not possible to use Connected Services in LAN 3, if PROFINET is set up in isolated mode. For more details, see section **Isolated LAN 3 or LAN 3 as part of the private network** in *Application manual - PROFINET Controller/Device*.

Note

It is not possible to use LAN 3 in RW 6.07. Only WAN port is supported for this release.

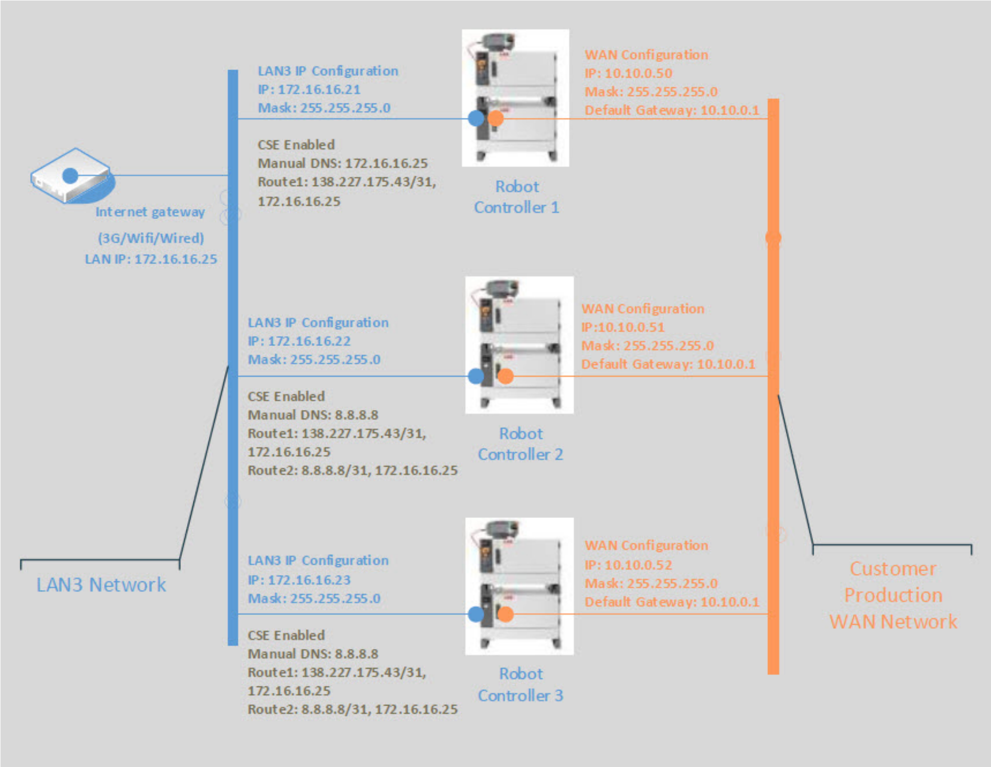
Steps to configure LAN 3

To configure the IP manually, follow the steps below:

Step	Action
1	In the **ABB** menu, select **Control Panel**.
2	Select **Configuration**
	From **Topics**, select **Communication**
3	Select **IP Settings** and tap **Add** • Enter the details for **IP Address**, **Interface**, and **Label**. • Change the **Interface** to *LAN3*.
4	Tap **OK** and restart the controller to take effect of the changes.

Continues on next page

The following diagram explains a sample with an Internet Gateway Box.



xx1700000061

As shown in the diagram above (for example robot controller 1), assign IP address to port X5 (LAN 3) as 172.16.16.21 and change the LAN IP of the Gateway Box to the same IP segment as 172.16.16.25.

A route may be needed to send the traffic to ABB Connected Services server (rseprod.abb.com:138.227.175.43) through the Internet Gateway on LAN 3 instead of the default Gateway on WAN.

Then the routing entry should be added as follows:

- Destination: 138.227.175.43/31
- Gateway: 172.16.16.25 (Box LAN IP)

In this example, configure LAN 3 of all the controllers to the same IP segment (172.16.16.xx) to connect multiple controllers together with the Gateway Box.

If there is no customer DNS on the production WAN network, configure the DNS manually as the Gateway IP. See *Steps to configure DNS manually on page 121*.

Note

If the Gateway Box only provides Internet access without DNS resolution then add an external DNS manually, for example 8.8.8.8. Then additional routing should be added as follows:

- Destination: 8.8.8.8/31
- Gateway: 172.16.16.25 (Box LAN IP)

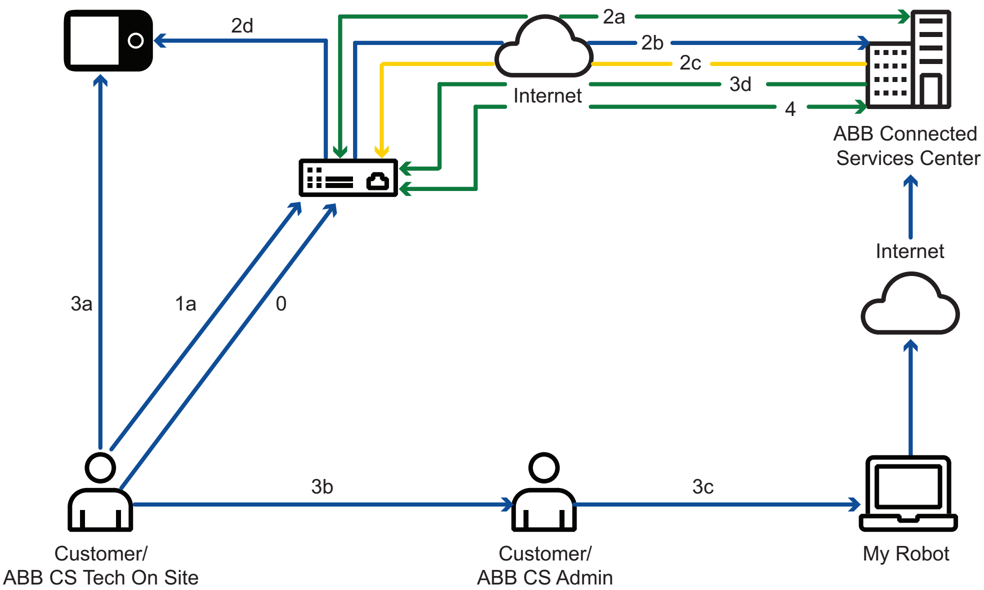
2.9.7 Connected Services registration

Connected Services startup

The Connected Services startup is based on the following steps:

- (0) Connected Services preparation
- (1) Connected Services configuration
- (2) Connected Services connectivity
- (3) Connected Services registration
- (4) Connected Services connected and registered

When these steps are done, the software Agent is securely connected and identified with a client certificate. The following figure describes these concepts:



xx1500003226

Step	Description
0	Check controller S/N and internet connectivity
1a	Enable CSE and set up connectivity configuration
2a	CS connectivity in place
2b	Low poll for registration
2c	Registration not trusted (get reg code)
2d	Display registration code
3a	Get registration code
3b	Give controller S/N and registration code
3c	Select controller S/N in SA and register with registration code
3d	Registration trusted (client certificate)
4	Connected and registered secure CS session

Continues on next page

Connected Services preparation

- Verify the controller serial number with the serial number found in the controller module cabinet.
- Verify and provide Internet connectivity to the robot controller.
- Verify that the service agreement for this controller is available with ABB Robotics Service.

Connected Services configuration

- Configure the connectivity parameters.
- Enable Connected Services

Connected Services connectivity

- Software Agent connects to the ABB Connected Services Center.
- An initial registration process starts at low polling rate.
- The initial registration is incomplete and not yet fully trusted.
- A registration code is received to finalize the trust relation.
- The registration code is made available on the Connected Services registration page.

Connected Services registration

- The customer/ABB on site provides the controller serial number and registration code to the Connected Services Administrator for registration.
- The Connected Services Administrator validates this registration code in MyRobot on its service agreement.
- The registration trust starts and implements a client certificate in the controller.

Connected Services connected and registered

- The controller is connected, registered, and identified in the service agreement.
- The connection is trusted with a client certificate.
- Connected Services is now actively running on the robot controller.

2.9.8 Connected Services information

Connected Services pages

Introduction

The Connected Services information pages are available under **System Info** > **Software resources** > **Communication** > **Connected Services**. The following are the 4 Connected Services information pages:

- **Overview**
- **Server Connection**
- **Registration**
- **Advanced**

> **Note**
>
> The information on a page can be refreshed by changing the page or by pressing the **Refresh** button. The **Refresh** button also forces a connection with the server if the software agent is waiting. (for example, wait for registration acknowledgement from MyRobot). This is useful in case of slow polling when connection cost is set to High.

Overview page

The **Overview** page provides a summary of the Connected Services status and information. If the status is not active then the other pages provide more detailed information.

Field	Description	Possible values	Example
Enabled	Displays the value of the master configuration switch for turning the Connected Services on/off.	Yes/No	Yes
Status	Displays the current status to see whether there is a need to navigate to the **Server connection** page or **Registration** page.	"-" Failed Initializing Shutdown Registration in progress Trying to connect Active	Active
Serial number	Displays the identifier that is used to identify the controller in Connected Service.	Controller Serial number	12-45678
RobotWare version	Displays the RobotWare version that is sent to the server.	RobotWare version name	6.03.0088
Restart counter	Displays the number of times the software Agent been auto-restarted. This is used to see if watchdog has restarted the by it.	0-N If not Enabled, then display: 0	2
Script version	Displays the downloaded data collector code version.	"Data Collector Script name" "-"	0116/ROBOTWARE-6.02.0000+/5196

Continues on next page

Field	Description	Possible values	Example
Service Agreement	To verify that the controller is associated to the expected service agreement.	"Name of the service agreement" "-"	SA_FR12_16
Customer name	To verify that the controller is associated to the expected service agreement.	"Customer Name of the service agreement" "-"	ABB Robotics
Country	To verify that the controller is associated to the expected service agreement.	"Country of the service agreement" "-"	France
Refresh button	On refresh, the software Agent replies with the current data and breaks the waiting state (if waiting) to contact the server and refreshes the information.		

Server Connection page

The **Server Connection** page provides a summary of the CS connectivity to the server.

Field	Description	Possible values	Example
Status	Displays the current status to see whether there is a need to navigate to the **Server connection** page or **Registration** page.	"-" Failed Initializing Shutdown Registration in progress Trying to connect Active	Active
Connection Status	Displays the status of communication with the server and the type of error.	Initializing Server not reachable Server not authenticated Server error (HTTP xxxx) Connected	Connected
Last updated	Displays the relative time since the information on the **Server connection** page has been generated.		"HH:MM:SS ago"
Server name	Displays the name of the server that software Agent is configured with.	"" Server name	rseprod.abb.com
Server IP	Displays the IP address of the server and the port number used for connection. The IP address is the result of DNS name resolution done by software Agent.	"" Server IP	138.227.175.43
Server certificate name	Displays the server certificate name information.	"" Server name Untrusted (Server)	rseprod.abb.com

Continues on next page

Field	Description	Possible values	Example
Server certificate issuer	Displays the name of the server certificate issuer.	"" Issuer Untrusted (Issuer)	ABB issuing CA 6
Server certificate valid until	Displays the server certificate date.	"" Issuer Expired (Date)	Nov 21 07:09:28 2017 GMT
Controller time	Displays the controller date and time details. **Note** It is important to set the correct time in the controller as this is needed for the certificate process.		16-01-08 13:52:33
DNS server	Displays the DNS information.	Not Available DNS value	10.0.23.45
Refresh button	On refresh, the software Agent responds with the current data and breaks the waiting state (if waiting) to contact the server and refreshes the information.		

Registration page

The **Registration** page provides a summary of the Connected Services registration.

Field	Description	Possible values	Example
Status	Displays the current status to see whether there is a need to navigate to the **Server connection** page or **Registration** page.	"-" Failed Initializing Shutdown Registration in progress Trying to connect Active	Active
Registration Status	Displays the registration status and code.	Register with code in MyRobot Registration in progress Registered Failed	Register with code in MyRobot
Registration code	Displays the registration code. This code can be used to login to MyRobot.	"-" Code value	456735
Refresh button	On refresh, the software Agent responds with the current data and breaks the waiting state (if waiting) to contact the server and refreshes the information.		

Continues on next page

Advanced page

The **Advanced** page provides advanced information about the dialog between software Agent and server.

Field	Description	Possible values	Example
Last HTTP message	Displays the last message sent.	Register CheckRegister GetLoginInfo GetMessage ...	GetMessage
Last HTTP message time	Displays the date and time when the last message was sent.		Sent hh:mm:ss ago
Last HTTP error	Displays the HTTP error when the last message was sent and the message ID if 4XX.	Not Available Error HTTP XXX + Message	Not Available
Next message	Displays the next message to send and the date to send the message.		GetMessage in 70 seconds
Last command	Displays the last command received from server.	Not Available Reboot Reset Ping Diagnostic ...	Not Available
Refresh button	On refresh, the software Agent responds with the current data and breaks the waiting state (if waiting) to contact the server and refreshes the information.		
Server Errors	Displays a count of the following servers errors: • Timeout errors • Request errors • Connection errors • Connection not Available errors • Unknown errors • Authentication errors • Proxy errors • Server errors	0-N for each server error	0/1/0/3/4/0/1/4

Connected Services logs

The software Agent generates some event logs in the central controller event log. Event logs are generated during starting, registering, unregistering, losing connectivity, and during other key events.

The events logs are in the range of 170XXX and are described with all the other controller event logs documentation. For more details, see *Operating manual - Troubleshooting IRC5*.

Continues on next page

Force a reset of the software agent

It is possible to reset the software agent. When you reset, the software agent erases all its internal information including the registration information, the data collector script, and all the locally stored service information. The configuration will not be reset, but a new registration is required to reactivate the Connected Services.

Use the following procedure to reset the software agent:

	Action
1	Tap the **ABB** button to display the ABB menu. Process applications are listed in the menu.
2	Tap **Program Editor** -> **Debug** -> **Call Routine**. **Note** Tap **PP to Main** if **Debug** is disabled.
3	Tap **Connected Services Reset** -> **Go to**. Press the **Motors on** button on the controller.
4	Press the Play button to execute the reset routine - > tap **Reset**.

2.10 User logs

2.10.1 Introduction to User logs

Description

The RobotWare base functionality *User logs* generates event logs for the most common user actions. The event logs are generated in the group *Operational events*, number series *10xxx*.

For more information on handling the event log, see *Operating manual - IRC5 with FlexPendant* and *Operating manual - Troubleshooting IRC5*.

Purpose

The purpose of *User logs* is to track changes in the robot controller related to user actions. This can for example be helpful to find the root cause if a production stop occurs.

What is included

The RobotWare base functionality *User logs* generates event logs for the following changes related to user actions. All event logs are described in *Operating manual - Troubleshooting IRC5*.

Topic	User action	Event logs
Program execution	Changing the speed or run mode (single cycle/continuous). Making changes to the task selection panel. Setting or resetting non motion execution mode.	10140 10145 10146 10153 10154 10284 10285
Simulate wait instructions	Simulating wait instructions, for example `WaitTime`, `WaitUntil`, `WaitDx`, etc.	10144
RAPID changes	Opening or closing RAPID programs or modules, editing RAPID code, or modifying robot positions.	10040 10041 10061 10062 10063 10064 10069 10078 10079 10147
Program pointer movements	Moving the program pointer to main, to a routine, to a position, or to a service routine (call routine).	10141 10142 10143 10149

Continues on next page

Topic	User action	Event logs
Changes on the mechanical unit	Updating the revolution counters or performing a calibration.	10205 10206 10290 10292
Jogging	Changing the tool, the work object, the payload, the coordinate system, or go to a position.	10280 10281 10282 10283 10286 10287 10288 10289 10291
Supervision	Setting or resetting the jog or path supervision. Setting the level of supervision.	10293 10294 10295 10296 10297 10298
Change of configuration	Loading configuration data or changing a configuration attribute.	10250
System changes	Clearing the event log or changing date and time.	10200 10201 10202
Serial measurement board	Changing the data in the serial measurement board or changing the data in the robot memory.	10115 10116 10117 10118
I/O	Setting or pulsing I/O signals.	10148 10160 10161

This page is intentionally left blank

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 improves TCP accuracy. 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.

Here are some examples of when this accuracy is important:

- Exchangeability of robots
- Offline programming with no or minimum touch-up
- Online programming with accurate movement and reorientation of tool
- Accurate cell alignment for MultiMove coordinated motion
- Programming with accurate offset movement in relation to eg. vision system or offset programming
- Re-use of programs between applications

The option *Absolute Accuracy* is integrated in the controller algorithms and does not need external equipment or calculation.

Note

The performance data is applicable to the corresponding RobotWare version of the individual robot.

Note

Singularities might appear in slightly different positions on a real robot compared to RobotStudio, where *Absolute Accuracy* is off compared to the real controller.

What is included

Every *Absolute Accuracy* robot is delivered with:

- compensation parameters saved in the robot memory
- a birth certificate representing the *Absolute Accuracy* measurement protocol for the calibration and verification sequence.

A robot with *Absolute Accuracy* calibration has a label with this information on the manipulator.

Absolute Accuracy supports floor mounted, wall mounted, and ceiling mounted installations. The compensation parameters that are saved in the robot memory differ depending on which Absolute Accuracy option is selected.

Continues on next page

When is *Absolute Accuracy* being used

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.

If the robot is inverted, the Absolute Accuracy calibration must be performed when the robot is inverted.

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

Note

In a robot system with, for example, an additional axis or track motion, the Absolute Accuracy is active for the manipulator but not for the additional axis or track motion.

RAPID instructions

There are no RAPID instructions included in this option.

Absolute Accuracy and MultiMove

If the main robot in a MultiMove system has the Absolute Accuracy option, it opens up Absolute Accuracy capability for all the robots in the system. However, each robot needs to be calibrated individually.

Note

Note that this is the only RobotWare option that is relevant for an additional robot.

Note

It is possible to mix robots with and without the option Absolute Accuracy arbitrarily in a MultiMove system.

3.1.2 Useful tools

Overview

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

- Load Identification
- 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*.

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.

3.1.3 Configuration

Activate Absolute Accuracy

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

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. For the parameter *Use Robot Calibration*, change the value to *r1_calib*.
5. For a MultiMove system, configure the parameter *Use Robot Calibration* for each robot. It should be set to *r2_calib* for robot 2, *r3_calib* for robot 3, and *r4_calib* for robot 4.
6. No restart is required.

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):

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 *Use Robot Calibration* and change the value to "r1_uncalib".
5. For a MultiMove system, repeat step 3 and 4 for each robot. *Use Robot Calibration* is then set to "r2_uncalib" for robot 2, "r3_uncalib" for robot 3 and "r4_uncalib" for robot 4.
6. No restart is required.

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 memory 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**.

Continues on next page

	Action
2	Tap on the robot you wish to update.
3	Tap the tab **Robot Memory**.
4	Tap **Advanced**.
5	Tap **Clear Controller Memory**.
6	Tap **Clear** and then confirm by tapping **Yes**.
7	Tap **Close**.
8	Tap **Update**.
9	Tap **Cabinet or robot has been exchanged** and confirm by tapping **Yes**.

3.1.4 Maintenance

3.1.4.1 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

Note

If the RobotWare version on the controller must be downgraded, then contact your local ABB for support regarding compatible versions of Absolute Accuracy.

Tool recalibration

For information about tool recalibration, see *Tool calibration on page 154*.

Motor replacement

Replacement of all motors requires a re-calibration of the corresponding resolver offset parameter using the standard calibration method for the respective robot. This is described in the product manual for the robot.

If the motor replacement requires disassembly of the arm, then see *Arm replacement or disassembly on page 140*.

Wrist replacement

Replacement of the wrist unit requires a re-calibration of the resolver offsets for axes 5 and 6 using the standard calibration method for the respective robot.

Arm replacement or disassembly

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.

Continues on next page

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.
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. **Note** An update of the defined work objects will make the deviation less in positioning.
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 138*. 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 152*.

3.1.4.2 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	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 the standard calibration for the robot.
the robot's relationship to the fixture(s) has changed	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	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).

3.1.5 Compensation theory

3.1.5.1 Error sources

Types of errors

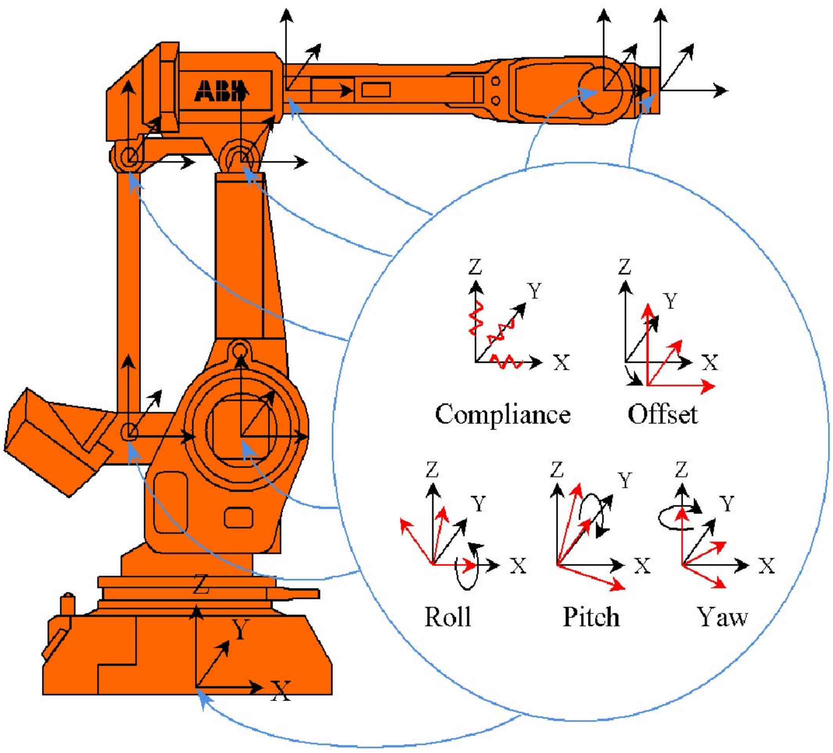
The errors compensated for in the controller derive from the mechanical tolerances of the constituent robot parts. A subset of these are detailed in the illustration below.

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

3.1.5.2 Absolute Accuracy compensation

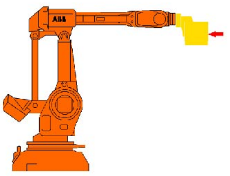
Introduction

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

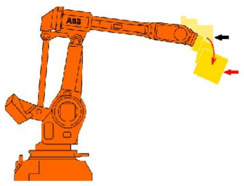
The following illustration shows the position you want the robot to have.



xx0300000225

Position due to deflection

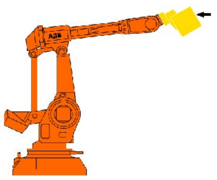
The following illustration shows the position the robot will get without *Absolute Accuracy*. The weight of the robot arms and the load will make a deflection on the robot. Note that the deflection is exaggerated.



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.

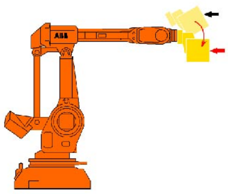


xx0300000226

Continues on next page

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

3.1.6 Preparation of Absolute Accuracy robot

3.1.6.1 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 in the robot memory
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.

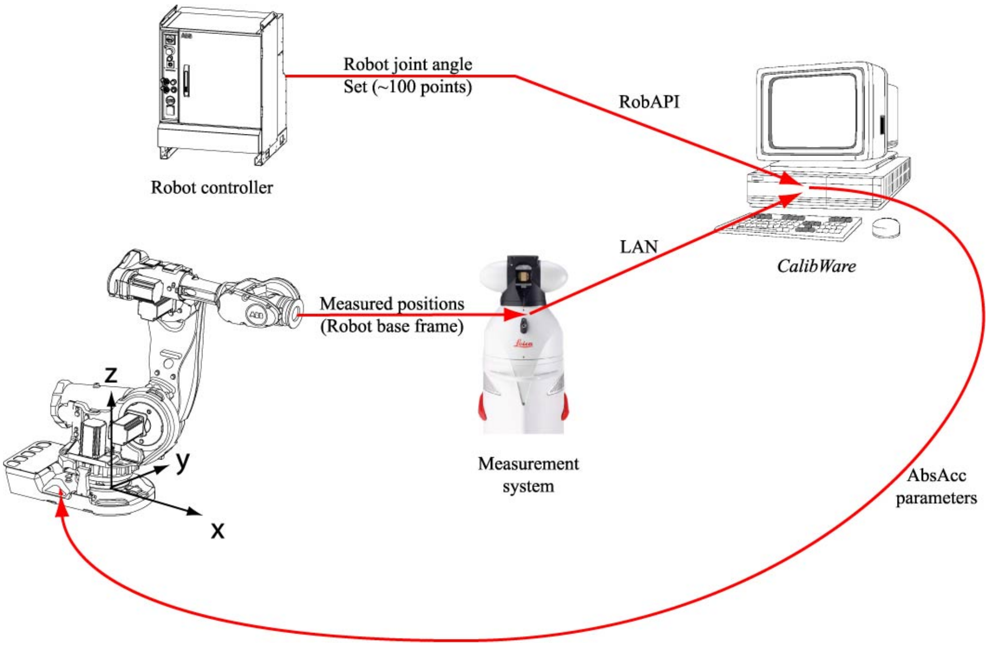
Absolute Accuracy calibration

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

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, see typical performance data in the product specification for the respective robot.

Compensation parameters and birth certificate

The compensation parameters are saved in the robot memory (see *Compensation parameters on page 149*).

A birth certificate is created representing the Absolute Accuracy measurement protocol for the calibration and verification sequence (see *Birth certificate on page 148*).

3.1.6.2 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 contains the following information:

- Robot information (robot type, serial number, version of Absolute Accuracy)
- 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)

3.1.6.3 Compensation parameters

About the compensation parameters

All Absolute Accuracy robots are shipped with a set of compensation parameters, as part of the system parameters (configuration). As the resolver offset calibration is integral in the Absolute Accuracy calibration, the resolver offset parameters are also stored in the robot memory.

The compensation parameters

The compensation parameters are defined in the following configuration types:

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

The type ROBOT_CALIB defines the top level of the calibration structure. The instance *r1_calib* activates the Absolute Accuracy functionality by specifying the flag -*absacc*. See *Activate Absolute Accuracy on page 138*.

The types ARM_CALIB, JOINT_CALIB, PARALLEL_ARM_CALIB, and MOTOR_CALIB are reserved by the system and are only shown when the Absolute Accuracy option is selected in the **Modify Installation** dialog. The parameter values can be changed by importing a new configuration file.

The compensation parameters are included in a backup, in the file *moc.cfg*.

3.1.7 Cell alignment

3.1.7.1 Overview

About cell alignment

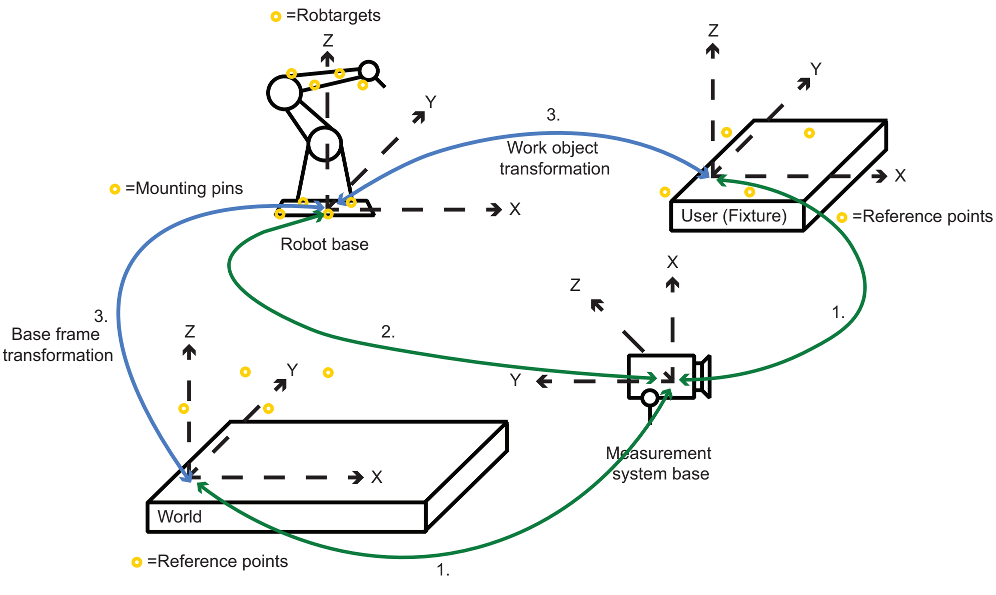
The compensation parameters for the Absolute Accuracy robot are determined from the physical base plate to the robot tool. For many applications this is enough, the robot can be used as any other robot. However, it is common that Absolute Accuracy robots are aligned to the coordinates in their cells. 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 151*.
2	Measure robot alignment	Determine the relationship between the measurement system and the robot. See *Measure robot alignment on page 152*.
3	Calculate frame relationships	Determine the relationship between, for example, the robot and the fixture. See *Frame relationships on page 153*.
4	Calibrate tool	Determine the relationship between the robot tool and other cell components. See *Tool calibration on page 154*.

Illustration



en0300000239

3.1.7.2 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.

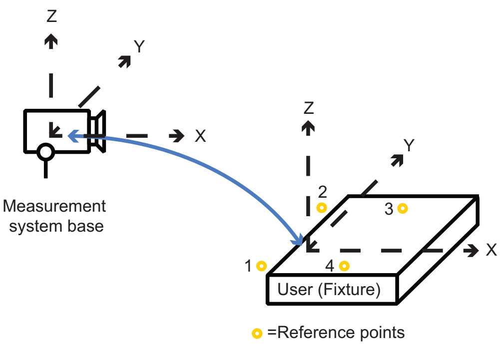
Absolute Accuracy fixtures must be equipped with at least three (preferably four) reference points, each with clearly marked position information.

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.
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

Measurement positions	Reference positions	Frame relationship
Pos1: 100, 100, 200	Pos1: 100, 100, 100	1) RobotStudio work object (0,0,-100,0,0,0) (x,y,z,roll,pitch,yaw
Pos2: 100, 200, 200	Pos2: 100, 200, 100	
Pos3: 200, 200, 200	Pos3: 200, 200, 100	
Pos4: 200, 100, 200	Pos4: 200, 100, 100	

3.1.7.3 Measure robot alignment

Select method

The relationship between the measurement system and the robot can be determined in the following 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 the product manual for the respective robot.
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 is 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 have 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.

3.1.7.4 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.

3.1.7.5 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.

3.2 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 156*.
- Changing *Motion Process Mode* from RAPID, see *Motion Process Mode [included in 687-1] on page 164*.
- *Wrist Move*, see *Wrist Move [included in 687-1] on page 172*.

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 161*.
2. Perform automatic tuning of the joint friction levels using the instructions `FricIdInit` and `FricIdEvaluate`. See *Automatic friction tuning on page 157*.
3. Compensate for the friction using the instruction `FricIdSetFricLevels`.

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 162*.
- It is not possible to view any test signals with TuneMaster 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.

3.3.3 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 `TuneServo` with the argument `TUNE_FRIC_LEV` 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 *Friction FFW Level*.
TUNE_FRIC_RAMP	By calling the instruction `TuneServo` with the argument `TUNE_FRIC_RAMP` 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 *Friction FFW Ramp*. 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 `TuneServo` and the tune type `TUNE_FRIC_LEV`. The level is given in percent of the *Friction FFW Level* value. Example: The instruction for increasing the friction level with 20% looks like this: `TuneServo MHA160R1, 1, 120 \Type:= TUNE_FRIC_LEV;`
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 *Friction FFW Level* is 0.5 and the final tune value (`TUNE_FRIC_LEV`) is 120%. Set *Friction FFW Level* 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 162*.

3.3.4 System parameters

3.3.4.1 System parameters

About the system parameters

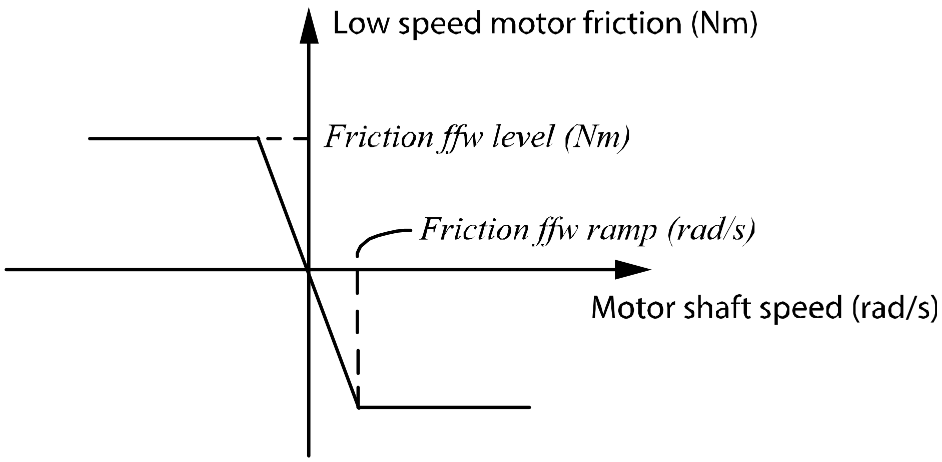
This is a brief description of each parameter in the option *Advanced Shape Tuning*. For more information, see the respective parameter in *Technical reference manual - System parameters*.

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 *Friction FFW On* is set to TRUE.
Friction FFW Level	*Friction FFW Level* is the friction level for the robot joint. See illustration below.
Friction FFW Ramp	*Friction FFW Ramp* is the speed of the robot motor shaft, at which the friction has reached the friction level defined by *Friction FFW Level*. See illustration below. There is normally no need to make changes to *Friction FFW Ramp*.

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 162*.

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 *Friction FFW Level* with the tuning value. Set this value as the new *Friction FFW Level* and set the tuning value (`TUNE_FRIC_LEV`) to 100%. Example: The *Friction FFW Level* is 0.5 and the final tune value (`TUNE_FRIC_LEV`) is 120%. Set *Friction FFW Level* 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 an 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 *Friction FFW Level* 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.

3.3.5 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.

There following modes are predefined:

- *Optimal cycle time mode* – this mode gives the shortest possible cycle time and is normally the default mode.
- *Accuracy mode* – this mode improves path accuracy. The cycle time will be slightly increased compared to *Optimal cycle time mode*. This is the recommended choice for improving path accuracy on small and medium size robots, for example IRB 2400 and IRB 2600.
- *Low speed accuracy mode* – this mode improves path accuracy. The cycle time will be slightly increased compared to *Accuracy mode*. This is the recommended choice for improving path accuracy on large size robots, for example IRB 4600.
- *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*.
- *Press tending mode* – Changes the *Kv Factor*, *Kp Factor* and *Ti Factor* in order to mitigate tool vibrations. This mode is primarily intended for use in press tending applications where flexible grippers with a large extension in the y-direction are used.

There are also four modes available for application specific user tuning:

- *MPM User mode 1 – 4*

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
```

Continues on next page

```
MoveL *, vmax, ...;
...

MotionProcessModeSet ACCURACY_MODE;
! Do cutting with high accuracy
MoveL *, v50, ...;
...
```

Limitations

- The *Motion Process Mode* concept is currently available for all six- and seven-axes robots except paint robots with TrueMove1.
- 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 *Accset* and the geometric accuracy parameters are available.
- The following robot models do not support the use of *World Acc Factor* (i.e. only *World Acc Factor = -1* is allowed):

  IRB 340, IRB 360, IRB 540, IRB 1400, IRB 1410

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. The predefined modes and the user modes can all 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.

- *Use Motion Process Mode Type* - selects predefined parameters for a user mode.
- *Accset Acc Factor* – changes acceleration
- *Accset Ramp Factor* – changes acceleration ramp
- *Accset Fine Point Ramp Factor* – changes deceleration ramp in fine points
- *Joint Acc Factor* - changes acceleration for a specific joint.
- *World Acc Factor* - activates dynamic world acceleration limitation if positive, typical value is 1, deactivated if -1.
- *Geometric Accuracy Factor* - improves geometric accuracy if reduced.
- *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

Most parameters 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`.

Continues on next page

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]*

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.

Accuracy Mode uses a dynamic world acceleration limitation (*World Acc Factor*) and increased geometric accuracy (*Geometric Accuracy Factor*) to improve the path accuracy.

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.

3.4.3 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 four 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 168*.

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 171*.

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.

WARNING

Incorrect tuning for a very low mounting stiffness can cause oscillating movements or torques that can damage the robot.

If accuracy still needs to be improved

- For applications with high demands on path accuracy, for example cutting, *Advanced Shape Tuning* and *Accuracy mode/Low speed accuracy mode* should be used. The choice of motion mode depends both on the robot type and the specific application. In general, *Accuracy mode* is recommended for small and medium size robots (up to *IRB 2400/2600*) and *Low speed accuracy mode* is recommended for larger robots.
- If the path accuracy still needs improvement, the accuracy modes can be adjusted with the tune parameters, some examples:
  - Tuning of *Accuracy mode* for improved accuracy:

    1) Reduce *World Acc Factor*, for example from 1 to 0.5.

    2) Reduce *Dh Factor* to 0.5 or lower. Note that a low value of *Dh factor* can change the corner zones at high speed.
  - Tuning of *Low speed accuracy mode* for improved accuracy:

    1) Set *World Acc Factor* to 1, and set *Geometric Accuracy Factor* to 0.1.

    2) Reduce *Dh Factor* to 0.5 or lower.
- The programmed speed must sometimes be reduced for best possible accuracy, e.g. in cutting applications. For example, a circle with radius 1 mm should not be programmed with a higher speed than 20 mm/s.
- For contact applications, for example milling and pre-machining, *Low speed stiff mode* is recommended. This mode can also be useful for large robots 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.

Continues on next page

- 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.

3.4.4 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- and seven-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 *Accset* and the geometric accuracy parameters are available.
- The following robot models do not support the use of *World Acc Factor* (i.e. only *World Acc Factor = -1* is allowed):

  IRB 340, IRB 360, IRB 540, IRB 1400, IRB 1410

Related information

For information about	See
Configuration of *Motion Process Mode* parameters.	*Technical reference manual - System parameters*
RAPID instructions: • `AccSet` - Reduces the acceleration • `MotionProcessModeSet` - Set motion process mode • `TuneServo` - Tuning servos	*Technical reference manual - RAPID Instructions, Functions and Data types*

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

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.

WristMove cannot be used on robots with non-spherical wrist, for example, GoFa or YuMi

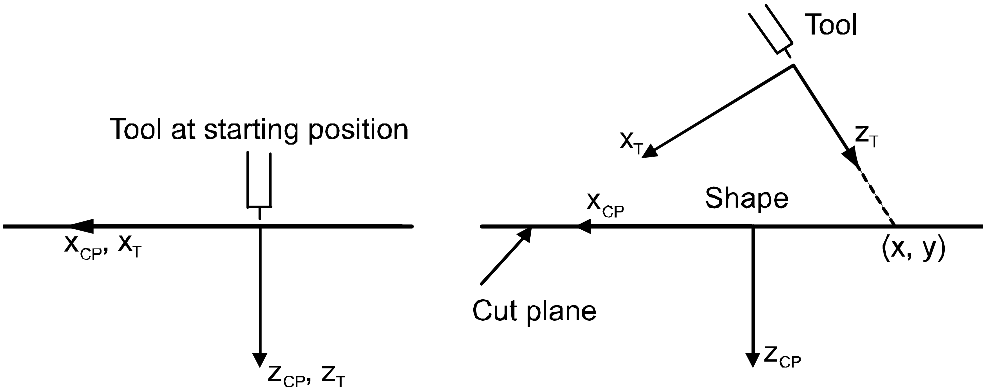
3.5.2 Cut plane frame

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.



en0900000118

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

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

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

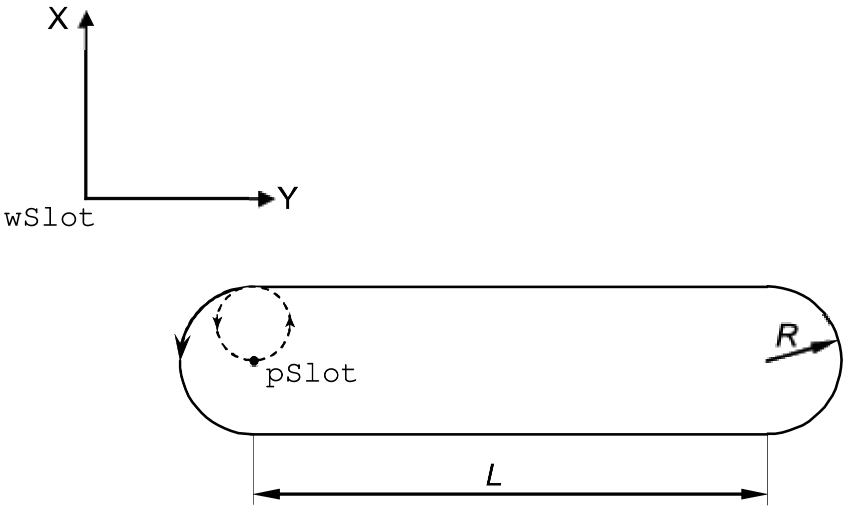
This example shows how to cut a slot with end radius `R` and length `L+2R`, using wrist movement. See *Illustration, `pSlot` and `wSlot` on page 178*. The slot both begins and ends at the position `pSlot`, which is the center of the left semi-circle. To avoid introducing oscillations in the robot, the 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;
```

Continues on next page

```
! Right semi-circle
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

3.5.5 Troubleshooting

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.

This page is intentionally left blank

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 `robtarget` when a *MODPOS* operation is performed
- the mechanical unit may be activated, and deactivated

Continues on next page

Basic approach

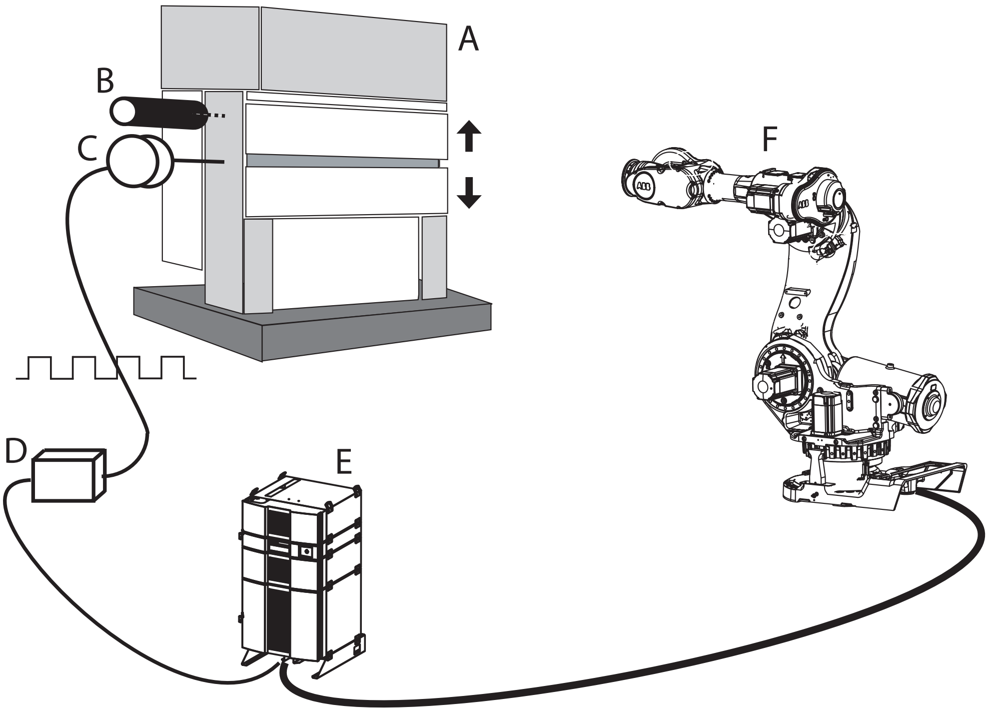
This is the general approach for setting up the synchronization option. For a more detailed description of how this is done, see the respective section.

- 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:



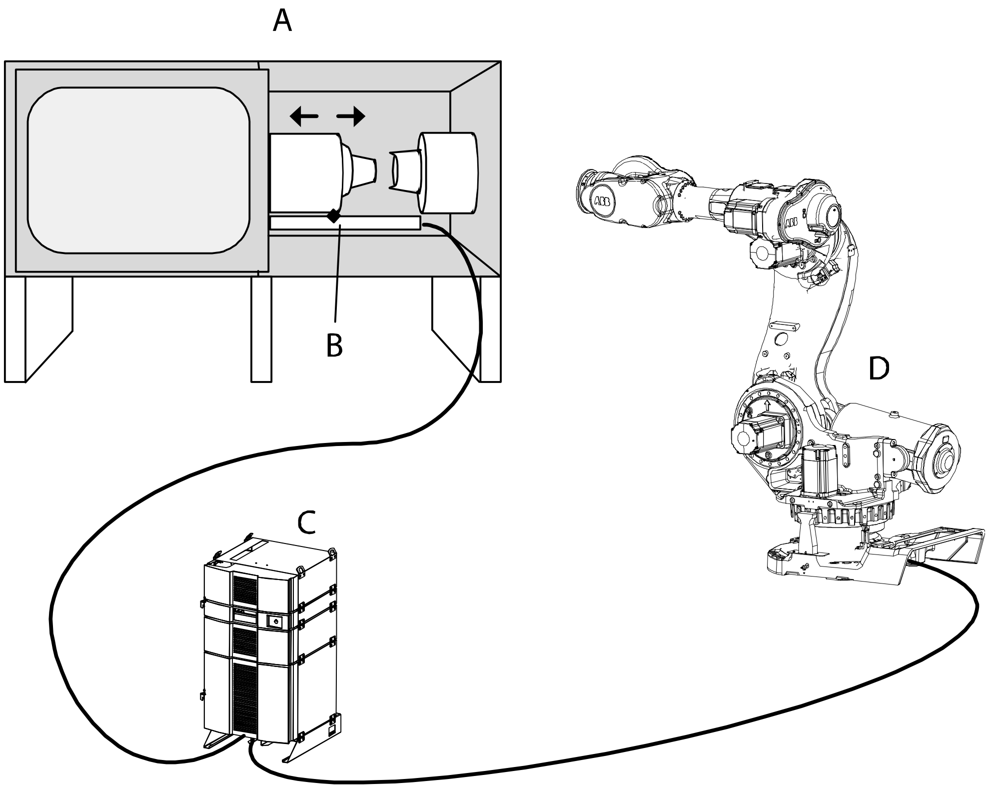
en0400000655

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

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	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: • +/- 50 ms for Sensor Synchronization • +/- 100 ms for Analog Synchronization 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.
Object queue	Only for Sensor Synchronization: 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.
RAPID access to sensor data	A RAPID program has access to the current position and speed of the external device, via the sensor.
Multiple sensors	Up to 2 sensors are supported. For Sensor Synchronization, each sensor must have a DSQC 377.

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	For Sensor Synchronization, the synchronization switch is triggered and a sensor object is created in the object queue. The robot connects to the object. 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.
the press is open enough for the robot to enter	the robot places (or removes) a work piece in the press. The synchronization is ended. For Sensor Synchronization, the sensor object is then dropped (removed from the object queue).

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.

4.1.6 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

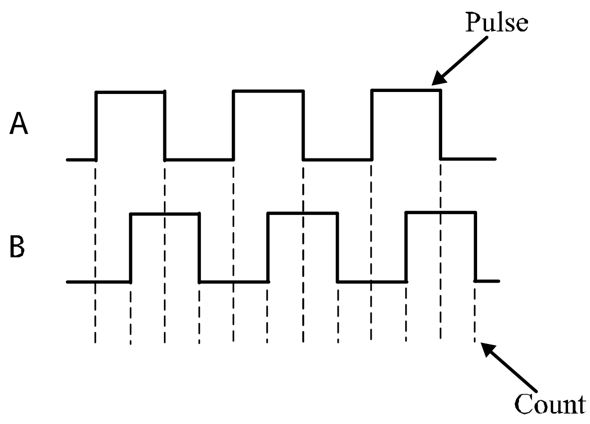
The encoder provides a series of pulses indicating the motion detected by the sensor. This is used to synchronize the motion between the robot and the external device.

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).

4.1.6.3 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 μF, depending on the length of the cable.

Encoder power supply

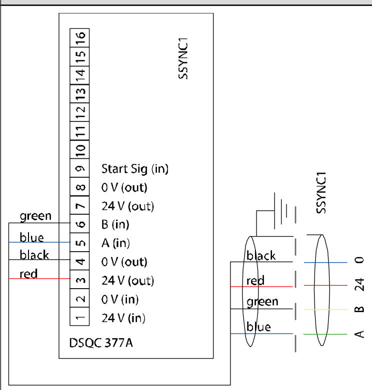
The encoder is normally supplied with 24 VDC from the encoder interface unit.

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.

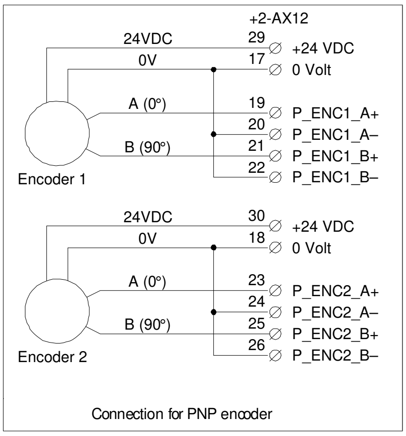
	Action	Illustration
1	Connect the encoder to the encoder interface unit (DSQC 377) on the controller.	 en0300000611

Continues on next page

	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: • No action is required. If the number counts down: • the connection of the two encoder faces (0° and 90°) must be interchanged.	 en0300000584

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.

4.1.8 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 197*.

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 *Connected to Bus* for the unit from "Virtual1" to the correct bus, for example "DeviceNet1".
2	Specify the correct address for the unit, parameter *DeviceNet Address*.
3	If the parameter *DeviceNet Master Address* (in topic *I/O*, type *Bus*) is changed, then the parameter *Default Value* (in topic *I/O*, type *Fieldbus Command Type*) for the instance *TimeKeeperInit* 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 *Type of Unit*, for the unit from "Virtual" to the correct unit type, for example "d355A".
2	Change the parameter *Connected to Bus* for the unit from "Virtual1" to the correct bus, for example "DeviceNet1".
3	Specify the correct address for the unit, parameter *DeviceNet Address*.
4	Change the communication interval for the unit type (e.g d355A) from 50 to 20 ms, parameter *Connection 1 Interval*. For more information about this parameter, see *Application manual - DeviceNet Master/Slave*.

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: *Load Parameters if no duplicates* 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: • *syvm1_eio.cfg* • *syvm1_prc.cfg* • *syvm1_moc.cfg*
5	Installation of a **slave sensor**, connected to DeviceNet2 (second board). Load the following files one by one from the OPTIONS/CNV directory: • *syvs1_eio.cfg* • *syvs1_prc.cfg* • *syvs1_moc.cfg*
6	Restart the system.
7	If necessary, correct the address for the new encoder interface units. The default addresses in the file *syvxx_eio.cfg* 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

4.1.8.2 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 *Motion*.
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: *Load Parameters if no duplicates* 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: • for second sensor: *syvm2_eio.cfg*, *syvm2_prc* and *syvm2_moc.cfg* • for third sensor: *syvm3_eio.cfg*, *syvm3_prc.cfg* and *syvm3_moc.cfg* • for fourth sensor: *syvm4_eio*.cfg, *syvm4_prc.cfg* and *syvm4_moc.cfg*
6	Installation of a **slave sensor**, connected to DeviceNet2 (second board). Load the following files one by one from the OPTION/CNV directory: • for second sensor: *syvs2_eio.cfg*, *syvs2_prc.cfg* and *syvs2_moc.cfg* • for third sensor: *syvs3_eio.cfg*, *syvs3_prc.cfg* and *syvs3_moc.cfg* • for fourth sensor: *syvs4_eio.cfg*, *syvs4_prc.cfg* and *syvs4_moc.cfg*
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/O*. The default addresses in the file *syvxx_eio.cfg* 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.

4.1.9 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 233*.

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:
	`ActUnit SSYNC1;`
	`MoveL waitp, v1000, fine, tool;`
	`WaitSensor SSYNC1;`	

2	Single-step the program past the `WaitSensor` instruction.	The instruction will return if there is an object in the object queue. If the is no object, the execution will stop while waiting for an object (i.e. a sync signal).

Continues on next page

	Action	Information
3	Run the external device until a sync signal is generated by the synchronization switch.	The program should exit the `WaitSensor` 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 `SyncToSensor SSYNC1\On` instruction. See *Programming examples on page 200*.	
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 204*.
7	End the synchronized motion with a `SyncToSensor SSYNC1\Off` instruction. See *Programming examples on page 200*.	
8	Only for Sensor Synchronization: Program a `DropSensor SSYNC1;` instruction. See *Programming examples on page 200*.	
9	Program a `DeactUnit SSYNC1;` instruction if this is the end of the program, or if the sensor is no longer needed. See *Programming examples on page 200*.	

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).

4.1.9.2 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;
```

4.1.9.3 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;
```

4.1.9.5 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.

4.1.9.7 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 204*.

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 `robtarget` position or the sensor value in the `robtarget` according to the warning text.

Speed warnings

If programmed `sensor_speed` is higher than:

- `(max_sync_speed*sensor_nominal_speed)/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:

- `(min_sync_speed*sensor_nominal_speed)/robot_tcp_speed`

a similar warning will appear:

- Programmed_sensor_speed equals sensor_distance/robot_interpolation_time.

Continues on next page

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 SSYNC1;
WaitSensor SSYNC1;
SyncToSensor SSYNC1\On;
! move instructions
...
SyncToSensor SSYNC1\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 three-position 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

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*.

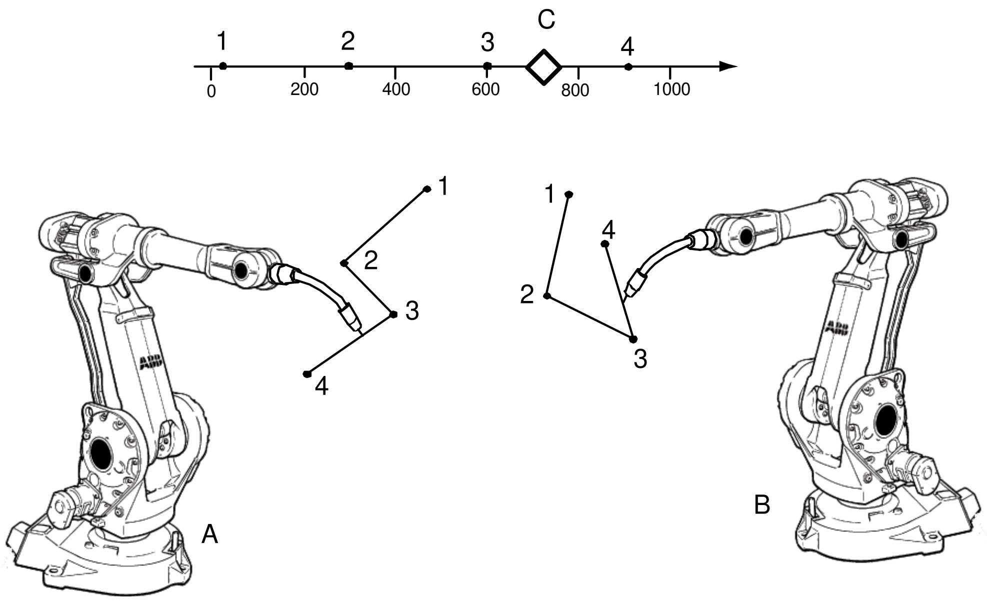
4.1.10.2 The concept of robot to robot synchronization

Description

The basic idea of robot to robot synchronization is that two robot 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



xx0400001145

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
Name	SSYNC2
mechanics	SS_LIN
process_name	SSYNC2
use_path	PSSYNC

Topic: Process

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

Topic: I/O

EIO_UNIT

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

EIO_SIGNAL

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

Continues on next page

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

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

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

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

Continues on next page

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

4.1.10.4 Slave robot configuration parameters

Overview

For default configuration, see *System parameters on page 231*.

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
Name	SSYNCS1
sensor_type	CAN
use_sensor	CAN1
adjustment_speed	1000
min_dist	600
max_dist	20000
correction_vector_ramp_length	10
nominal_speed	1000

CAN_INTERFACE

CAN_INTERFACE/Parameters	Value
Name	CAN1
Signal delay	34
Connected signal	c1Connected
Position signal	c1Position
Velocity signal	c1Speed
Null speed signal	c1NullSpeed
Data ready signal	
Waitwobj signal	c1WaitWObj
Dropwobj signal	c1DropWobj
Data Time stamp	c1DTimestamp
RemAllPObj signal	c1RemAllPObj
Virtual sensor	NO
Sensor Speed filter	0,33

Continues on next page

Topic: I/O

EIO_UNIT

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

EIO_SIGNAL

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

EIO_SIGNAL/Parameters	Value
Name	ai1Speed
SignalType	AI
Unit	SLAVE1
UnitMap	16-31
MaxLog	10.0
MaxPhys	1
MaxPhysLimit	1
MaxBitVal	32767
MinLog	-10.0
MinPhys	-1
MinPhysLimit	-1
MinBitVal	-32767

EIO_SIGNAL/Parameters	Value
Name	ai1PredTime
SignalType	AI

Continues on next page

EIO_SIGNAL/Parameters	Value
Unit	SLAVE1
UnitMap	32-47
MaxLog	10.0
MaxPhys	1
MaxPhysLimit	1
MaxBitVal	32767
MinLog	-10.0
MinPhys	-1
MinPhysLimit	-1
MinBitVal	-32767

EIO_SIGNAL/Parameters	Value
Name	di1Dready
SignalType	DI
Unit	SLAVE1
UnitMap	48

EIO_SIGNAL/Parameters	Value
Name	di1Sync2
SignalType	DI
Unit	SLAVE1
UnitMap	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 SSYNC1;

!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

Considerations

The following is to be considered

- The values of `extax.eax_e` should increase for every robtarget 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.

4.1.11 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 223* If the sensor is using group input, see *Settings for sensor using Group input on page 224*. Descriptions of the system parameters are found in *System parameters on page 231*.

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

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 SSYNC1;
WaitSensor SSYNC1;
! Set up a recording for 2 seconds
PrxStartRecord SSYNC1, 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 SSYNC1, 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;
```

4.1.12 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 230*.

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 227* If the sensor is using group input, see *Settings for sensor using Group input on page 228*. Descriptions of the system parameters are found in *System parameters on page 231*.

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

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 SSYNC1;
WaitSensor SSYNC1;
! Set up a recording for 2 seconds
PrxStartRecord SSYNC1, 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 SSYNC1, 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;
```

4.1.13 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 `WaitSensor`. 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 *Adjustment Speed*.
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

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 *Max Advance Distance*. During this ramping period, the position error may be 5 times *Max Advance Distance*.
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	*CPU load equalization* 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.

4.1.15 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 *c1DropWObj* in RAPID code. Use the `DropWobj` 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
`PrxSetSyncalarm`	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 of World Zones

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 241*.

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

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`).

World Zones supervision is not accessible when lead-through is active.

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	`wztemporary` is used to identify a temporary world zone and can be used anywhere in the RAPID program. 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.
wzstationary	`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 *Operating manual - IRC5 with FlexPendant*. 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. Stationary world zones shall be used if security is involved.
shapedata	`shapedata` is used to describe the geometry of a world zone. World zones can be defined in 4 different geometrical shapes: • 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	`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`. The volume can also be defined as the inverse of the box (all volume outside the box).
WZCylDef	`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`. The volume can also be defined as the inverse of the cylinder (all volume outside the cylinder).
WZSphDef	`WZSphDef` is used to define a volume that has the shape of a sphere. The definition is stored in a variable of type `shapedata`. The volume can also be defined as the inverse of the sphere (all volume outside the sphere).

Continues on next page

Instruction	Description
WZLimJointDef	`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. 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. The definition is stored in a variable of type `shapedata`.
WZHomeJointDef	`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. 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. The definition is stored in a variable of type `shapedata`.
WZLimSup	`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. When calling `WZLimSup` you specify whether it is a stationary world zone, stored in a `wzstationary` variable, or a temporary world zone, stored in a `wztemporary` variable.
WZDOSet	`WZDOSet` is used to define, and enable, setting a digital output signal when the TCP reaches the world zone. When calling`WZDOSet` you specify whether it is a stationary world zone, stored in a `wzstationary` variable, or a temporary world zone, stored in a `wztemporary` variable.
WZDisable	`WZDisable` is used to disable the supervision of a temporary world zone.
WZEnable	`WZEnable` is used to re-enable the supervision of a temporary world zone. A world zone is automatically enabled on creation. Enabling is only necessary after it has been disabled with `WZDisable`.
WZFree	`WZFree` is used to disable and erase a temporary world zone.

Functions

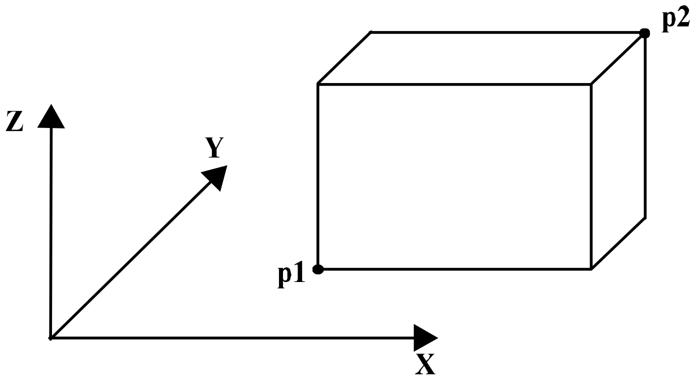
World Zones does not include any RAPID functions.

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
```

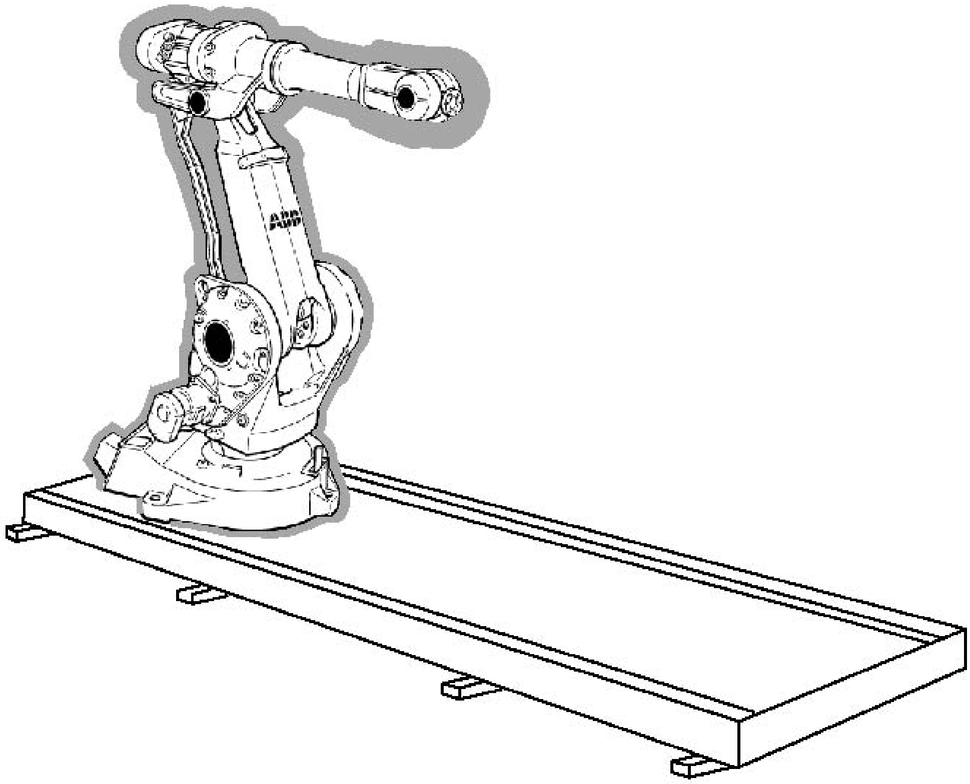
Signal when robot is in position

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

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
  WZDOSet \Temp, home \Inside, joint_space, do_home, 1;
ENDPROC
```

6 Motion functions

6.1 Independent Axis [610-1]

6.1.1 Overview

Purpose

The purpose of Independent Axis 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 Axis 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 247*.

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 loose 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

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.

Internal and customer cabling and equipment may limit the ability to use independent axis functionality on axis 4 and 6.

The option is not possible to use in combination with:

- SafeMove[i]
- Track Motion (IRBT)
- Positioners (IRBP) on Interchange axes
- Tool change

i *Independent Axis* can in some cases be combined with SafeMove2 if the additional axis does not move the robot, and the additional axis is not monitored by SafeMove. Contact your local ABB sales office team for additional information.

The following is deactivated when option Independent Axes is used:

- Collision detection

> **Note**
>
> The collision detection is deactivated on all axes in a motion planner if one of them is run in independent mode.

6.1.2 System parameters

About the system parameters

This is a brief description of each parameter in the option *Independent Axis*. 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 *Transmission Gear High* divided by *Transmission Gear Low*. If no smaller number can be used, the transmission gear ratio will be correct if *Transmission Gear High* is set to the number of cogs on the robot axis side, and *Transmission Gear Low* is set to the number of cogs on the motor side.
Transmission Gear Low	See *Transmission Gear High*.

6.1.3 RAPID components

Data types

There are no data types for Independent Axis.

Instructions

This is a brief description of each instruction in Independent Axis. 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 Axis. 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

```
  !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 (±180°).

```
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`.

6.2.2 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 occur 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 robtarget 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

```
    ...
    !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 syncronized 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

```
  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 robtarget 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 syncronized 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

```
  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 p1clean_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

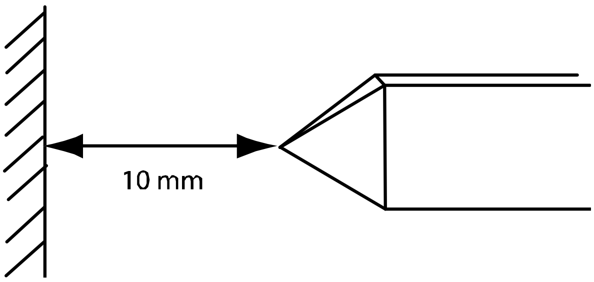
This is the general approach for using the path recorder:

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

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 loosing 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

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 robtarget p100 := ...
  CONST robtarget 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

```
  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
```

6.3 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 268*.

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

The instruction `CorrWrite` is intended with low speed and moderate values of correction. Too aggressive values will be clamped. The correction values should be tested in RobotStudio to confirm the performance.

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.

Continues on next page

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 the option *Path Offset*. For more information, see the respective data type in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
corrdescr	`corrdescr` 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 the option *Path Offset*. For more information, see the respective instruction in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
CorrCon	`CorrCon` activates path correction. Calling `CorrCon` 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	`CorrDiscon` deactivates path correction. Calling `CorrDiscon` will disconnect a correction generator.
CorrClear	`CorrClear` deactivate path correction. Calling `CorrClear` will disconnect all correction generators.
CorrWrite	`CorrWrite` sets the path correction values. Calling `CorrWrite` will set the offset values to a correction generator.

Functions

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

Function	Description
CorrRead	`CorrRead` 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_no1;
VAR corrdescr id;
VAR pos sens_val;
PROC PathRoutine()
  !Connect to the correction generator
  CorrCon id;

  !Setup a 5 Hz timer interrupt.
  CONNECT int_no1 WITH UpdateCorr;
  ITimer\Single, 0.2, int_no1

  !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_no1;
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_no1;
  CONNECT int_no1 WITH UpdateCorr;
  ITimer\Single, 0.2, int_no1;
ENDTRAP
```

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.

Continues on next page

Collision detection for YuMi robots

As default YuMi will have collision detection active at stand still. It also has another stop ramp compared to other robots to be able to release clamping forces.

> **Note**
>
> If the tool data is wrong, false collisions might be triggered and the robot arm might drop a short distance during the stop ramp.

Collision detection for MultiMove robots

The default behavior when a collision is detected for one robot in a MultiMove configuration is that all robots are stopped.

One reason for this behavior is that when a collision is detected, there is a big risk that it was two robots that collided. Another reason is that if one robot stops and another continues, this might cause another collision.

This behavior can be changed with the system parameter *Ind collision stop without brake*. If this parameter is set to TRUE and the robots are running in independent RAPID tasks when a collision is detected, only the robot that detected the collision will be stopped.

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.

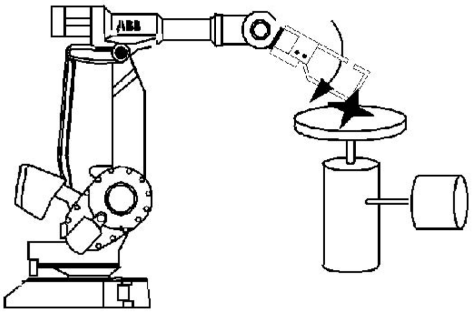
7.1.3 What happens at a collision

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

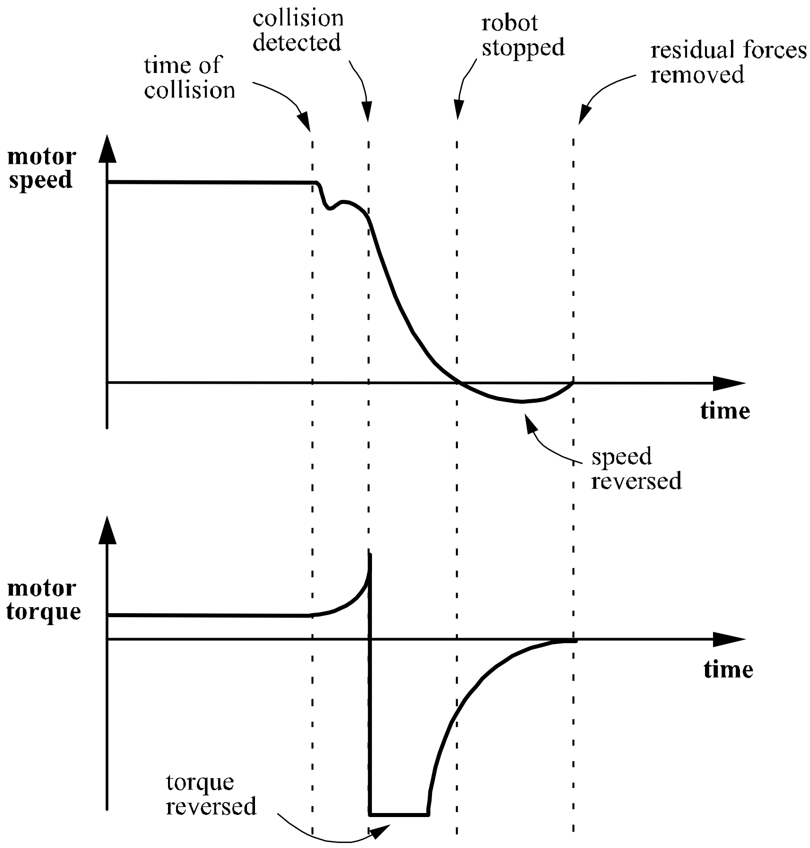
Robot behavior after a collision

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 *motors on* state

Continues on next page

Speed and torque diagram



en0300000360

7.1.4 Additional information

Motion error handling

For more information regarding error handling for a collision, see *Technical reference manual - RAPID kernel*.

7.1.5 Configuration and programming facilities

7.1.5.1 System parameters

About system parameters

Most of the system 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. *Path Collision Detection* is by default set to On.
Jog Collision Detection	Turn the collision detection On or Off for jogging. *Jog Collision Detection* 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. *Path Collision Detection Level* 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. *Jog Collision Detection Level* 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. *Collision Detection Memory* 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. *Manipulator Supervision* 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. *Manipulator Supervision Level* 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

Motion System

This parameter belongs to the type *Motion System* in the topic *Motion*.

Parameter	Description
Ind collision stop without brake	This parameter is only valid for systems using the MultiMove option. If this parameter is set to TRUE, detected collisions will be handled independently in RAPID tasks that are executed independently. A restart is required for this parameter to take effect.

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. *Collision Error Handler* is default set to Off. For more information regarding error handling for a collision, see *Technical reference manual - RAPID kernel*

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	`MotionSup` is used to: • activate or deactivate Collision Detection. This can only be done if the parameter *Path Collision Detection* 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	*MotSupOn* is high when Collision Detection is active and low when it is not active. Note that a change in the state takes effect when a motion starts. Thus, if Collision Detection is active and the robot is moving, *MotSupOn* is high. If the robot is stopped and Collision Detection is turned off, *MotSupOn* is still high. When the robot starts to move, *MotSupOn* switches to low. Before the first Motors On order after a restart of the robot controller, *MotSupOn* will reflect the value of the corresponding system parameter *Path Collision Detection*: • If *Path Collision Detection* is set to *On*, *MotSupOn* will be high. • If *Path Collision Detection* is set to *Off*, *MotSupOn* will be low.
MotSupTrigg	*MotSupTrigg* goes high when the collision detection triggers. It stays high until the error code is acknowledged from the FlexPendant.

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 275*.

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.

7.1.6.3 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 *Operating manual - IRC5 with FlexPendant*.
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 `MotionSup` to raise the supervision level or turn the function off temporarily.
everything else fails	turn off Collision Detection.

7.1.7 Collision Avoidance

Introduction

The function *Collision Avoidance* monitors a detailed geometric model of the robot. By defining additional geometrical models of bodies in the robot workarea, the controller will warn about a predicted collision and stops the robot if two bodies come too close to each other. The system parameter *Coll-Pred Safety Distance* determines at what distance the two objects are considered to be in collision.

The function *Collision Avoidance* is useful for example when setting up and testing programs, or for programs where positions are not static but created from sensors, such as cameras (non-deterministic programs). By using trigger-signals (see *Trigger signals on page 284*), *Collision Avoidance* can be used for implementing safe workspace sharing between multiple robots.

Besides the robot itself the function will monitor up 10 objects that is created via the configurator in RobotStudio. Typical objects to be monitored are tool mounted on the robot flange, additional equipment mounted on the robot arm (typically axis 3) or static volume around the robot.

The geometric models are set up in RobotStudio.

The functionality is activated by the system input *Collision Avoidance*. A high signal will activate the functionality and a low signal will deactivate the functionality. The functionality is by default active if no signal has been assigned to the system input *Collision Avoidance*.

Collision Avoidance is active both during jogging and when running programs. Also, the RAPID function `IsCollFree` provides a way to check possible collisions before moving to a position.

CAUTION

Always be careful to avoid collisions with external equipment, since a collision could damage the mechanical structure of the arm.

Collision Avoidance is no guarantee for avoiding collisions.

Tip

How to configure *Collision Avoidance* is described in *Operating manual - RobotStudio*.

Tip

Collision Avoidance adds the user configuration in the folder *CA* under the *HOME* folder. This is created when adding a configuration in RobotStudio.

If disk space is needed, the *rsgfx* files can be removed.

Continues on next page

False collision warning

There are different ways to lower the sensitivity of the function *Collision Avoidance* to avoid false warnings.

- Temporarily disable *Collision Avoidance*, see *Disabling Collision Avoidance on page 285*.
- For IRB 14000, decrease the safety distance for the arm or geometric model that triggers the false collision warning, see *Decrease sensitivity between links for IRB 14000 on page 285*.
- Decrease the general safety distance with the system parameter *Coll-Pred Safety Distance*.

Activation/deactivation of objects

By default, a defined collision object is active all the time. However, it is possible to configure a collision object with an activation signal, which basically connects it to a digital input that determines whether the object is active or not. This is useful, for example, for modelling multiple tools, where only one tool at a time is active. Another use case is modelling of objects that can be present or absent in the robot cell, for example a pallet.

Note that changing the state of an activation signal will immediately change the activation state of the connected collision object, and no synchronization to the robot path planning is made. Activating a collision object while the robot is moving towards the object can thus lead to a collision because the planned path may already have passed by the collision object while it was inactive. If synchronization is important, then activation signals should either be changed in finepoints when the robot is standing still or using trigg instructions like `TriggL` or `TriggJ`.

Trigger signals

A non-moving collision object can be configured with a trigger signal. The value of the trigger signal reflects which robots are in contact with the collision object. More specifically, the value of a trigger signal should be interpreted as a bit pattern, where bit *k* is high if robot *k* is in contact with the collision object. For example, if the trigger signal has the value 6, which is 110 in binary, it means that ROB_2 and ROB_3 are in contact with the collision object. Trigger signals can be used to implement safe workspace sharing between multiple robots.

A trigger signal can be configured with two timing behaviors: *immediate* or *on-arrival*. If configured with *immediate* behavior, then the trigger signal is changed as quickly as possible, well before the robot has physically reached the position where it comes into contact with the collision object. If configured with *on-arrival* behavior, then the trigger signal changes state when the robot physically reaches the position where it comes in contact with the zone.

Limitations

> **CAUTION**
>
> *Collision Avoidance* shall not be used for safety of personnel.

Continues on next page

- *Collision Avoidance* is a function included in the option *Collision Detection*.
- Paint robots, IRB 6620LX, and delta robots are not supported.
- *Collision Avoidance* cannot be used in manual mode together with responsive jogging. The system parameter *Jog Mode* must be changed to *Standard*.
- Only stationary/non-moving objects can be configured with a trigger signal. A trigger signal must correspond to a group signal. Furthermore, each collision object must have its own trigger signal.
- There is no support for applications that do corrections to the path, such as conveyor tracking, WeldGuide, Force Control, SoftMove, SoftAct etc.
- The *Collision Avoidance* functionality between 2 robots (or more) can only be achieved when using a MultiMove system.

Disabling *Collision Avoidance*

It is possible to temporarily disable the function *Collision Avoidance* if the robot has already collided or is within the default safety distance, or when the robot arms need to be very close and the risk of collision is acceptable.

Set the system input signal `Collision Avoidance` to 0 to disable *Collision Avoidance*. It is recommended to enable it (set `Collision Avoidance` to 1) as soon as the work is done that required *Collision Avoidance* to be disabled.

Decrease sensitivity between links for IRB 14000

For dual arm robots, the sensitivity can be decreased between individual robot arm links. This is useful if two links come close to each other, but the general safety distance should be maintained.

Open the file *irb_14000_common_config.xml* located in the folder *<SystemName>\PRODUCT\ROBOTWARE_6.XX.XXXX\robots\CA\irb_14000*.

For example, to decrease the safety distance between the left arm's link 3 and the right arm's link 4 to 1 mm, add the following row:

```
<Pair object1="ROB_L_Link3" object2="ROB_R_Link4"
      safetyDistance="0.001"/>
```

To decrease the safety distance between the left arm's link 5 and the robot base to 2 mm, add the following row:

```
<Pair object1="ROB_L_Link5" object2="Base" safetyDistance="0.002"/>
```

To disable collision avoidance between the left arm's link 2 and the right arm's link 3, add the following row:

```
<Pair object1="ROB_L_Link2" object2="ROB_R_Link3" exclude="true"/>
```

Note

The safety distance between two links can be decreased by adding a row to this XML file, but it cannot be increased to a higher value than defined by the system parameter *Coll-Pred Safety Distance*.

7.2 SafeMove Assistant

Purpose

SafeMove Assistant is a functionality in RobotWare that helps users to program their application when there is an active SafeMove configuration. The assistant will read the active configuration and plan the trajectories according to the limits and settings in that configuration. It will set the speed so that SafeMove will not trigger violations etc. It will also stop with error message in case the robot is programmed to enter a forbidden zone etc.

SafeMove Assistant will automatically adjust robot behavior to adopt to the active SafeMove configuration, the robot will adopt to speed limited zones and stop before entering forbidden zones.

CAUTION

SafeMove Assistant is not a safety function.

For example, if using a fence, then a safety distance is required between the safe cartesian zone and the fence.

Note

In case of SafeMove Assistant fails, the SafeMove supervision will trigger an emergency stop.

Description

SafeMove Assistant will check if any SafeMove speed limit is active for any Cartesian speed checkpoint (TCP, tool points, and elbow). If this is the case, a corresponding speed limit is applied in the path planner. For technical reasons, only the speed of the TCP, the wrist center point (WCP), and the elbow are limited by the path planner. Therefore, in cases where other tool points move faster than the TCP, SafeMove may trigger a Tool Speed violation. To avoid this, change the program or decrease the value of the parameter *SafeMove assistance speed factor* (see below).

SafeMove Assistant is not active in manual mode.

SafeMove Assistant does not take path corrections generated at lower level into account. It is therefore an increased risk of SafeMove violations when running applications like Externally Guided Motion or conveyor tracking.

Continues on next page

System parameters

SafeMove Assistant can be disabled for the SafeMove validation etc. This is done with the parameter *Disable SafeMove Assistance*, in the type in *Motion System*.

There are some parameters that can be changed in case robot system has minor overshoot or in any other way triggers SafeMove violations.

Parameter	Description
SafeMove Assistance Speed Factor	That has a default setting of 0.96 which corresponds to 96% of speed supervision will be the speed that path planner will use. This parameter can be decreased to reduce that risk but can in most cases be left at default value.
SafeMove assistance zone margin	When robot is running on a zone border there is a small risk that SafeMove can trigger violations when going in and out of the zone. This parameter can be increased to reduce that risk but can in most cases be left at default value.

For more information, see the parameters in the type *Motion System* described in *Technical reference manual - System parameters*.

This page is intentionally left blank

8 Communication

8.1 FTP Client [614-1]

8.1.1 Introduction to FTP Client

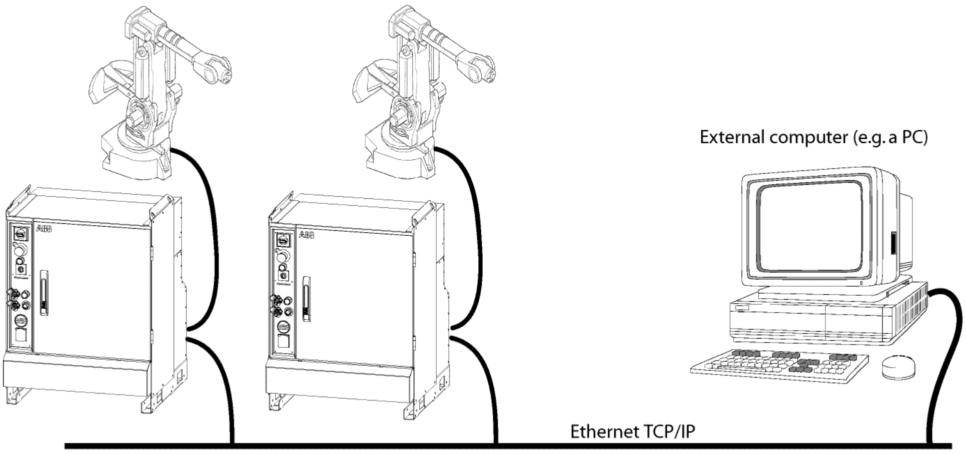
Purpose

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



en0300000505

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 and NFS Client* gives you access to the system parameter type *Application protocol* and its parameters: *Name*, *Type*, *Transmission protocol*, *Server address*, *Server type*, *Trusted*, *Local path*, *Server path*, *Username*, *Password*, and *Show Device*.

Basic approach

This is the general approach for using FTP Client. For more detailed examples of how this is done, see *Examples on page 292*.

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

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

Directory listing style on FTP server

The FTP server must list directories in a UNIX style.

Example:

```
drwxrwxrwx 1 owner group 25 May 18 16:39 backups
```

The MS-DOS style does not work.

Tip

For Internet Information Services (IIS) in Windows, the directory listing style is configurable.

Welcome Message from FTP server

The welcome message from the FTP server can only consist of one line. For the FileZilla FTP server, change the custom welcome message to "FileZilla".

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 below.

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 example "TCPIP1").
Server address	The IP address of the computer with the FTP server.
Server type	The type of FTP server the FTP client is connected to.
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.mod on this unit would be pc:test.mod
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.
Show Device	Shall the device be visible on external clients, e.g. on the FlexPendant?

Transmission protocol

For network devices, the connection instance is configured by setting the parameter *Type* to "TCP/IP" and the parameter *Name* to, for example, "TCPIP1".

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
Server type	NotSet
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

The following examples show how to open the file C:\robot_1\files\file1.txt 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.

For the home directory on IRC5:

```
Open "HOME:" \FILE:="file1.txt", file;
```

For the directory on the PC (e.g. C: \ ABB which is specified in the server):

```
Open "pc:" \FILE:="file1.txt", file;
```

8.2 SFTP Client [614-1]

8.2.1 Introduction to SFTP Client

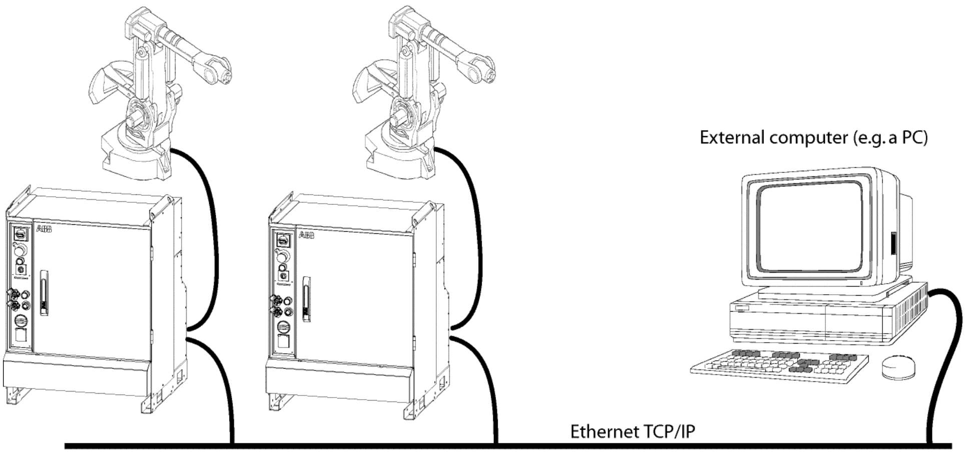
Purpose

The purpose of SFTP 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



en0300000505

Description

Several robots can access the same computer over an Ethernet network.

Once the SFTP 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 and 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*, *Username*, *Password*, *Show Device*, and *FingerPrint*.

Basic approach

This is the general approach for using SFTP Client. For more detailed examples of how this is done, see *Examples on page 292*.

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.

Continues on next page

SFTP supports the following servers:

- Rebex version 1.0.3
- CompleteFTP version 11.0.0
- Cerberus version 9.0.4.0

In certain SFTP servers, as Complete SFTP server, there is a configuration setting, **Timeout for idle sessions**, which defines the time that the connection can be idle. If no client requests are made during this time interval, the connection is closed. Setting the value as **No timeout** will keep the connection alive, even though client requests are not made.

Requirements

The external computer must have:

- TCP/IP stack
- SFTP Server

Limitations

When using the SFTP Client the maximum length for a file name is 99 characters.

When using the SFTP 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:

- A backup is saved to *pc:/Backups/Backup_20130109*
  (27 characters)
- The path on the PC will be *\Backups\Backup_20130109*
  (24 characters)
- The longest file path inside this backup is *\Backups\Backup_20130109\RAPID\TASK1\PROGMOD\myprogram.mod*
  (44+13 characters)

The maximum path length for this example first looks like 27 characters but is actually 57 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 below.

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 "SFTP".
Transmission protocol	Name of the transmission protocol the protocol should use (for example "TCPIP1").
Server address	The IP address of the computer with the SFTP 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.mod on this unit would be pc:test.mod
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 SFTP 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.
Show Device	Shall the device be visible on external clients, e.g. on the FlexPendant?
FingerPrint	To guarantee that the controller connects to the expected SFTP server, and not a malicious server, a server fingerprint can be used.

Transmission protocol

For network devices, the connection instance is configured by setting the parameter *Type* to "TCP/IP" and the parameter *Name* to, for example, "TCPIP1".

8.2.3 Examples

Example configuration

This is an example of how an application protocol can be configured for SFTP.

Parameter	Value
Name	my_SFTP_protocol
Type	SFTP
Transmission protocol	TCPIP1
Server address	100.100.100.100
Trusted	No
Local path	pc:
Username	Robot1
Password	robot1
Show Device	Yes
FingerPrint	A2:3E:41:90:4C:F6:32:BD:0A:7E:FB:57:89:D4:8E:13:20:07:B6:AF

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*.

Example with RAPID code

This example shows how to open the file *files\file1.txt* on the remote PC from a RAPID program on the controller.

For the home directory on IRC5:

```
Open "HOME:" \FILE:="file1.txt", file;
```

For the directory on the PC (e.g. C: \ ABB which is specified in the server):

```
Open "pc:" \FILE:="file1.txt", file;
```

8.3 NFS Client [614-1]

8.3.1 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.

> **Note**
>
> The controller has no antivirus software to check the data transferred to/from the controller via the remote mounted disk. It is up to the customer to secure the external data storage.

Description

Several robots can access the same computer over an Ethernet network.

The NFS mounted device is accessed by its name, as specified in the `Name` system parameter.

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 *FTP and NFS Client* gives you access to the system parameter type *Application protocol* and its parameters: *Name*, *Type*, *Transmission protocol*, *Server address*, *Server type*, *Trusted*, *Local path*, *Server path*, *User ID*, *Group ID*, and *Show Device*.

Basic approach

This is the general approach for using NFS Client. For more detailed examples of how this is done, see *Examples on page 292*.

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

Continues on next page

Limitations

When using the NFS Client the maximum length for a file path including the file name is 248 characters. The whole path is included in the 248 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.

8.3.2 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 below.

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 example "TCPIP1").
Server address	The IP address of the computer with the NFS server.
Server type	The type of FTP server the FTP client is connected to.
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.mod on this unit would be pc:test.mod
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 *User ID* 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 *Group ID* must be the same for all mountings on one robot controller.
Show Device	Shall the device be visible on external clients, e.g. on the FlexPendant?

Transmission protocol

For network devices, the connection instance is configured by setting the parameter *Type* to "TCP/IP" and the parameter *Name* to, for example, "TCPIP1".

8.3.3 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
Server type	NotSet
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.txt 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.

For the home directory on IRC5:

```
Open "HOME:" \FILE:="file1.txt", file;
```

For the directory on the PC (e.g. C: \ ABB which is specified in the server):

```
Open "pc:" \FILE:="file1.txt", file;
```

8.4 PC Interface [616-1]

8.4.1 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 option PC Interface is not required for RobotStudio and ABB software.

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*.

8.4.2 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
      }
    }
  }
```

8.4.3 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 WAN 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*.

8.5 Socket Messaging [616-1]

8.5.1 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 functionality Socket Messaging gives you access to RAPID data types, instructions and functions for socket communication between computers.

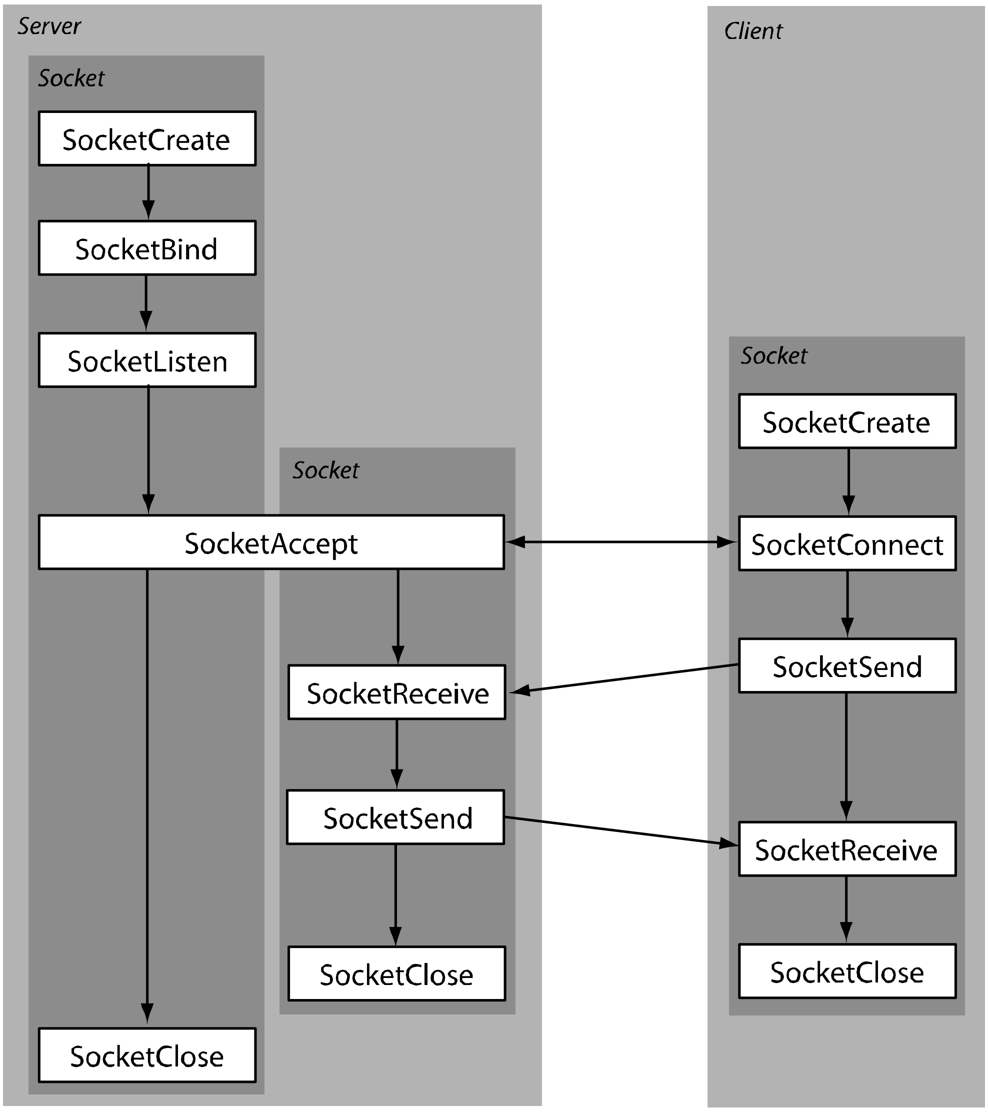
Basic approach

This is the general approach for using Socket Messaging. For a more detailed example of how this is done, see *Code examples for Socket Messaging on page 310*.

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.

8.5.2 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).

8.5.3 Technical facts about Socket Messaging

Overview

When using the functionality Socket Messaging to communicate with a client or server that is not a RAPID task, the following information can be useful.

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 them, 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*.

8.5.4 RAPID components

Data types

This is a brief description of each data type in Socket Messaging. For more information, see *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 `socketdev` variable.

Instructions for client

This is a brief description of each instruction used by the a Socket Messaging client. For more information, see *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
SocketCreate	Creates a new socket and assigns it to a `socketdev` 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 `string` or `rawbytes` variable, or a `byte` array.
SocketReceive	Receives data and stores it in a `string` or `rawbytes` variable, or in a `byte` 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 uses 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 `SocketBind`.
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 *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). `SocketGetStatus` does not detect changes from outside RAPID (such as a broken connection).

8.5.5 Code examples for Socket Messaging

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_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_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_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_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
```

8.6 RAPID Message Queue [included in 616-1, 623-1]

8.6.1 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 319*.

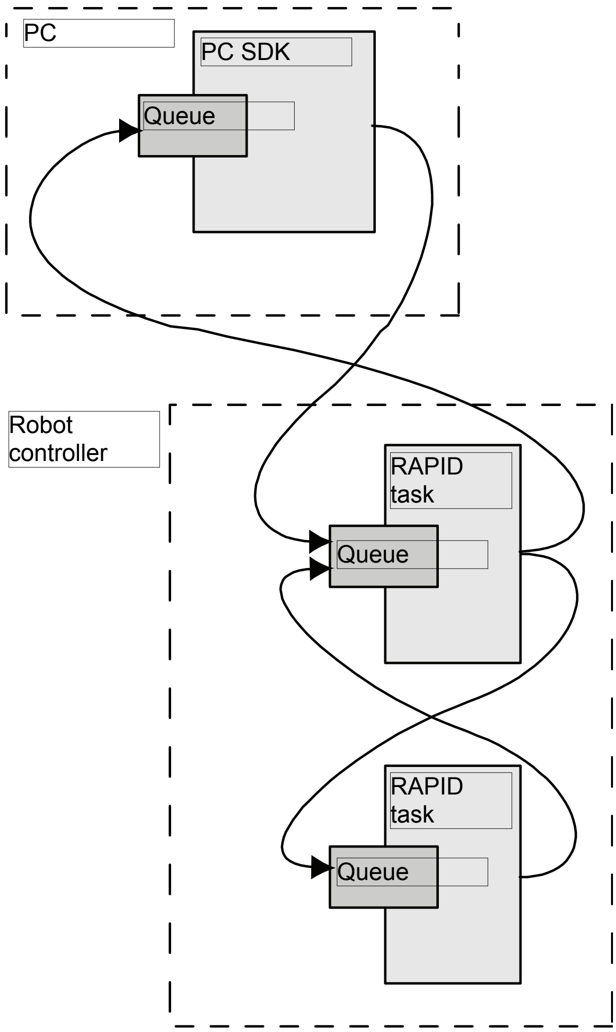
1. For *interrupt* mode: The receiver sets up a trap routine that reads a message and connects an interrupt so the trap routine is called when a new message appears.

   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.

8.6.2 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

- `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:

```
"robtarget;[[930,0,1455],[1,0,0,0],[0,0,0,0],
     [9E9,9E9,9E9,9E9,9E9,9E9]]"
"string;"A message string""
"stringarr:["string1","string2"]
"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

Related information

For more information on queues and messages, see *Technical reference manual - RAPID kernel*.

8.6.3 System parameters

About the system parameters

This is a brief description of each parameter in the functionality *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*.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>RMQ Type</td><td>Can have one of the following values:
None - Disable all communication with RAPID Message Queue for this RAPID task.
Internal - 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.
Remote - Enable communication with RAPID Message Queue for this task, both with other tasks on the controller and external clients (FlexPendant and PC applications).
The default value is None.</td></tr>
<tr><td>RMQ Mode</td><td>Defines the mode of the queue.
Can have one of the following values:
Interrupt - A message can only be received by connecting a trap routine to a specified message type.
Synchronous - A message can only be received by executing an <code>RMQReadWait</code> instruction.
Default value is Interrupt.</td></tr>
<tr><td>RMQ Max Message Size</td><td>The maximum data size, in bytes, for a RAPID Message Queue message.
An integer between 400 and 5000. The default value is 448.
Note
The value cannot be changed in RobotStudio or on the FlexPendant. The only way to change the value is to edit the sys.cfg file by adding the attribute RmqMaxMsgSize with the desired value.</td></tr>
<tr><td>RMQ Max No Of Messages</td><td>The maximum number of RAPID Message Queue messages in the queue to this task.
An integer between 1 and 10. The default value is 5.
Note
The value cannot be changed in RobotStudio or on the FlexPendant. The only way to change the value is to edit the sys.cfg file by adding the attribute RmqMaxNoOfMsg with the desired value.</td></tr>
</table>

8.6.4 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 *RMQ Mode* is defined as *Interrupt*.
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 *RMQ Mode* is defined as *Interrupt*.
RMQReadWait	Wait for a message. Can only be used if *RMQ Mode* is defined as *Synchronous*.
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 `rmqslot`.

Data types

Data type	Description
rmqslot	Slot identity of a RAPID task or Robot Application Builder client.
rmqmessage	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. **Note:** `rmqmessage` is a large data type. Declaring too many variables of this data type can lead to memory problems. Reuse the same `rmqmessage` variables as much as possible.
rmqheader	The `rmqheader` describes the message and can be read by the RAPID program.

8.6.5 Code examples

Example using RMQSendMessage and RMQGetMessage with PC SDK

This is an example using RMQSendMessage and RMQGetMessage with PC SDK. The PC SDK, creates data (a string) with a request to receive current position of the mechanical unit. The T_ROB1 task receives the request and creates data containing the position and sends it back to the PC SDK.

Example of RAPID with RMQ

```
MODULE MainModule
    RECORD position
      num x;
      num y;
      num z;
    ENDRECORD

    RECORD message
      string msg1;
      string msg2;
  ENDRECORD
  VAR position posData;
  VAR message request;
  VAR intnum rmqMsg;
  VAR rmqslot clientSlot;
  VAR pos currPosition;
  CONST string unknownRequest := "Unknown request";
  PROC main()
      RMQFindSlot clientSlot, "RMQ_PC_SDK";
      CONNECT rmqMsg WITH rmqMessageHandler;
      IRMQMessage request, rmqMsg;
      WHILE TRUE DO
          ...
          currPosition := CPos(\Tool:=tool0 \WObj:=wobj0);
          ...
      ENDWHILE
      IDelete rmqMsg;
      EXIT;
  ENDPROC
  TRAP rmqMessageHandler
    VAR rmqmessage rmqMsg;
    VAR rmqheader header;
    RMQGetMessage rmqMsg;
    RMQGetMsgHeader rmqMsg\Header := header\SenderId := clientSlot;
    IF header.datatype = "message" THEN
      RMQGetMsgData rmqMsg, request;
      IF request.msg1 = "Get current position" THEN
        posData.x := currPosition.x;
        posData.y := currPosition.y;
        posData.z := currPosition.z;
        RMQSendMessage clientSlot, posData;
```

Continues on next page

```
        ELSE
          RMQSendMessage clientSlot, unknownRequest;
        ENDIF
      ENDIF
    ENDTRAP
  ENDMODULE
```

Example of PC SDK with RMQ

```
class Messaging
{
    private static Controller ctrl;
    private static IpcQueue pcsdkQueue;
    private static IpcQueue trob1Queue;
    private static float X;
    private static float Y;
    private static float Z;
    private static string message1 = "\"Get current position\"";
    private static string message2 = "\"\"";

    static void Main(string[] args)
    {
      ...
      //Connect and login to selected controller.
      ...

      if (ctrl != null)
      {
        trob1Queue = ctrl.Ipc.GetQueue("RMQ_T_ROB1");

        string pcsdkQueueName = "RMQ_PC_SDK";

        if (ctrl.Ipc.Exists(pcsdkQueueName))
        {
        ctrl.Ipc.DeleteQueue(ctrl.Ipc.GetQueueId(pcsdkQueueName));
        }
        pcsdkQueue = ctrl.Ipc.CreateQueue(pcsdkQueueName, 16,
             ctrl.Ipc.GetMaximumMessageSize());

        SendMessage(message1, message2);

        ...

        ctrl.Ipc.DeleteQueue(ctrl.Ipc.GetQueueId(pcsdkQueueName));
      ctrl.Logoff();

        ...
    }

  }
  public static void SendMessage(string message1, string message2)
```

Continues on next page

```
{
  IpcMessage message = new IpcMessage();
  byte[] data;
  if (pcsdkQueue != null && trob1Queue != null)
  {
    data = new UTF8Encoding().GetBytes("message;[" + message1 +
         " , " + message2 + "]");
    message.SetData(data);
    message.Sender = pcsdkQueue.QueueId;
    trob1Queue.Send(message);
    System.Threading.Tasks.Task.Run(() => { receiveMessage();
         });
  }
}
private static void receiveMessage()
{

    IpcMessage message = new IpcMessage();
    IpcReturnType ret;
    int timeout = 5000;

    if (pcsdkQueue != null)
    {
      ret = pcsdkQueue.Receive(timeout, message);
      if (ret == IpcReturnType.OK)
      {
        string answer = new
             UTF8Encoding().GetString(message.Data);
        string[] answerStructure = answer.Split(';');
        if (answerStructure[0] == "position")
        {
          string pos = answer.Substring((answer.IndexOf('[')) +
               1, answer.IndexOf(']') - ((answer.IndexOf('['))
               + 1));
          string [] array=pos.Split(',');
          X = float.Parse(array[0]);
          Y = float.Parse(array[1]);
          Z = float.Parse(array[2]);
          ...
        }
        else if(answerStructure[0] == "string")
        {
          string valueCharacter = "\"";
          int valueStartIndex = answer.IndexOf(valueCharacter);
          int valueEndIndex = answer.IndexOf(valueCharacter,
               valueStartIndex + 1);
          string returnText = answer.Substring(valueStartIndex
               + 1, valueEndIndex - (valueStartIndex + 1));
          ...
        }
      }
```

Continues on next page

```
            else
            {
          //No message recieved within time limit.
              ...
            }
          }
          else
          {
            //No queue found
            ...
          }
        }
      }
    }
```

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.

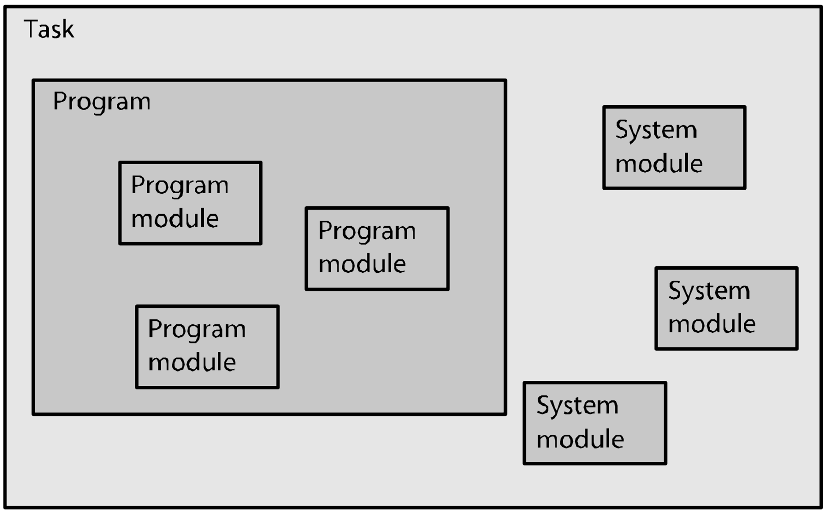
Examples of applications to run in parallel with the main program:

- 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. This includes tasks from add-ins and options, that might be running in the background.

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.

Continues on next page

- The data types: `taskid`, `syncident`, and `tasks`.
- 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 328*, and *RAPID components on page 327*.

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 the option *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	The name of the task. 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. 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.
Task in foreground	Used to set priorities between tasks. *Task in foreground* 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. If *Task in foreground* is set to empty string for a task, it runs at the highest level.
Type	Controls the start/stop and system restart behavior: • Normal (NORMAL) - The task program is manually started and stopped (e.g. from the FlexPendant). The task stops at emergency stop. • Static (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 (SEMISTATIC) - The task program restarts from the beginning at restart. The task program is normally not stopped by the FlexPendant or by emergency stop. A task that controls a mechanical unit must be of the type *normal*.
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	*TrustLevel* defines the system behavior when a static or semistatic task program is stopped (e.g. due to error): • 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

Parameter	Description
MotionTask	Indicates whether the task program can control robot movement with RAPID move instructions. Only one task can have *MotionTask* 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 *Task*, and cannot be defined in the RAPID program. However, the data type `taskid` can be used as a parameter when declaring a routine. For code example, see *taskid on page 345*.
syncident	`syncident` is used to identify the waiting point in the program, when using the instruction `WaitSyncTask`. The name of the `syncident` variable must be the same in all task programs. For code example, see *WaitSyncTask example on page 339*.
tasks	A variable of the data type `tasks` contains names of the tasks that will be synchronized by the instruction `WaitSyncTask`. For code example, see *WaitSyncTask example on page 339*.

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	`WaitSyncTask` is used to synchronize several task programs at a special point in the program. A `WaitSyncTask` 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 339*.

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	`TestAndSet` 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 343*.
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 344*.
TaskRunRob	Check if the task program controls any robot with TCP. For code example, see *Test if task controls mechanical unit on page 344*.

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.

1. Define the new task by adding an instance of the system parameter type *Task*, in the topic *Controller*.
2. Set the parameter *Type* 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 *Controller* and type *Automatic loading of Modules*, 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 *Type* and *TrustLevel* 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 *TrustLevel* 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 333*.

Continues on next page

	Action
5	Press the STOP button to stop the selected STATIC and SEMISTATIC tasks.
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 *TrustLevel* back to desired value (e.g. SysFail).
10	Restart the system.

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 .

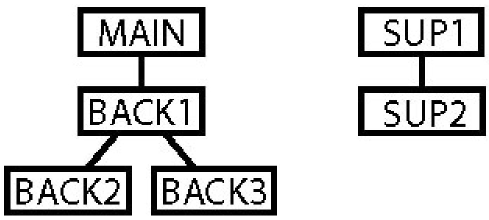
To set a task in the background of another task, use the parameter *Task in foreground*.

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:



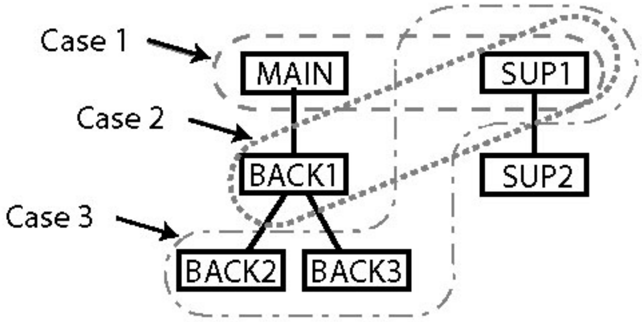
en0300000451

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 333*.

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 *Trust-Level*NoSafety can be selected, while STATIC and SEMISTATIC tasks with TrustLevel 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

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

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

Interrupt example

Main task program:

```
MODULE module1
  PROC main()
    SetDO do1,1;
    ...
  ENDPROC
ENDMODULE
```

Background task program:

```
MODULE module2
  VAR intnum intno1;

  PROC main()
    CONNECT intno1 WITH wait_trap;
    ISignalDO do1, 1, intno1;
    WHILE TRUE DO
      WaitTime 10;
    ENDWHILE
  ENDPROC

  TRAP wait_trap
    ! This is the point where the execution
    ! continues after do1 is set in main task
    ...
    IDelete intno1;
  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

```
  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()
    ...
```

Continues on next page

```
    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 the type *MotionTask* must be set to YES. See *System parameters on page 325*.

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 suffix "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";
```

9.1.6.4 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 or Ethernet channel.

The sensor may be accessed using a package of RAPID instructions that provide the ability to read and write raw 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:

- ABB supported sensor protocols.
- Instruction used to connect to a sensor device: `SenDevice`.
- Instruction used to set up interrupt, based on input from the sensor:`IVarValue`.
- Instruction used to write to a sensor: `WriteVar`.
- Function for reading from a sensor: `ReadVar`.
- Laser Tracker Calibration (LTC) functionality for optical sensor calibration.

Basic approach

This is the basic approach for using Sensor Interface.

1. Configure the sensor. See *Configuring sensors on page 348*.
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 354*.

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*.

9.2.2 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 on serial channels on page 349*.
- Sensors connected to Ethernet using the RoboCom Light protocol from Servo-Robot Inc, the LTAPP or the LTPROTOBUF protocol from ABB. For configuration, see *Configuring sensors on Ethernet channels on page 350*.

9.2.2.2 Configuring sensors on serial channels

Overview

Sensor Interface communicates with a maximum of one sensor 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 must be "RTP1".
Serial Port	The name of the serial port that will be used for the sensor. This refers to the parameter *Name* in the type *Serial Port*. For information on how to configure a serial port, see *Technical reference manual - System parameters*.

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 on Ethernet channels

Overview

Sensor Interface can communicate with a maximum of six sensors over Ethernet channel using the RoboCom Light protocol version E04 (from Servo-Robot Inc), the LTAPP or the LTPROTOBUF 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 has the default TCP port 6344 on the external sensor side, and LTAPPTCP has the default 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 *Name* in the type *Serial Port*. For information on how to configure a serial port, see *Technical reference manual - System parameters*. For IP based transmission protocols (i.e. *Type* has value TCP/IP, SOCKDEV, LTAPPTCP or UDPUC), *Serial Port* is not used and has the value N/A.
Remote Address	The IP address of the sensor. This refers to the type *Remote Address*. For information on how to configure Remote Address, see *Technical reference manual - System parameters*.
Remote Port	Remote Port specifies the port number on the network node identified by Remote Address with which the connection shall be established. The default value for SOCKDEV is 6344, and for LTAPPTCP it is 5020.

Configuration examples

These are examples of how a transmission protocol can be configured for a sensor.

Name	Type	Serial Port	Remote Address	Remote Port
laser2:	SOCKDEV	N/A	192.168.125.101	6344
laser3:	LTAPPTCP	N/A	192.168.125.102	5020

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 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 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 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

Constants

Name	Number	Read/write (R/W)	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 *Not ready yet* 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 *Unit*.
LTAPP__Y	9	R	Measured Y value, unsigned word. The units are determined by the variable *Unit*.
LTAPP__Z	10	R	Measured Z value, unsigned word. The units are determined by the variable *Unit*.
LTAPP__GAP	11	R	The gap between two sheets of metal. The units are determined by the variable *Unit*, -32768 if not valid.
LTAPP__MISMATCH	12	R	Mismatch, unsigned word. The units are determined by the variable *Unit*. -32768 if not valid.
LTAPP__AREA	13	R	Seam area, units in mm2, -32768 if not valid.
LTAPP__THICKNESS	14	RW	Plate thickness of sheet that the sensor should look for, LSB=0.1mm.
LTAPP__STEPDIR	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 (R/W)	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
LTAPP__X0	54	R	Scansonic sensors only. Measured X value line 1, unsigned word. The units are determined by the variable `Unit`.
LTAPP__Y0	55	R	Scansonic sensors only. Measured Y value line 1, unsigned word. The units are determined by the variable `Unit`.
LTAPP__Z0	56	R	Scansonic sensors only. Measured Z value line 1, unsigned word. The units are determined by the variable `Unit`.
LTAPP__X1	57	R	Scansonic sensors only. Measured X value line 2, unsigned word. The units are determined by the variable `Unit`.
LTAPP__Y1	58	R	Scansonic sensors only. Measured Y value line 2, unsigned word. The units are determined by the variable `Unit`.
LTAPP__Z1	59	R	Scansonic sensors only. Measured Z value line 2, unsigned word. The units are determined by the variable `Unit`.
LTAPP__X2	60	R	Scansonic sensors only. Measured X value line 3, unsigned word. The units are determined by the variable `Unit`.
LTAPP__Y2	61	R	Scansonic sensors only. Measured Y value line 3, unsigned word. The units are determined by the variable `Unit`.
LTAPP__Z2	62	R	Scansonic sensors only. Measured Z value line 3, unsigned word. The units are determined by the variable `Unit`.

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,[0,0,0,0,0,0,0,0,0],
     [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:=ArrIndx(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
```

Continues on next page

```
  ArcRefresh;
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 Robot Reference Interface [included in 689-1]

9.3.1 Introduction to Robot Reference Interface

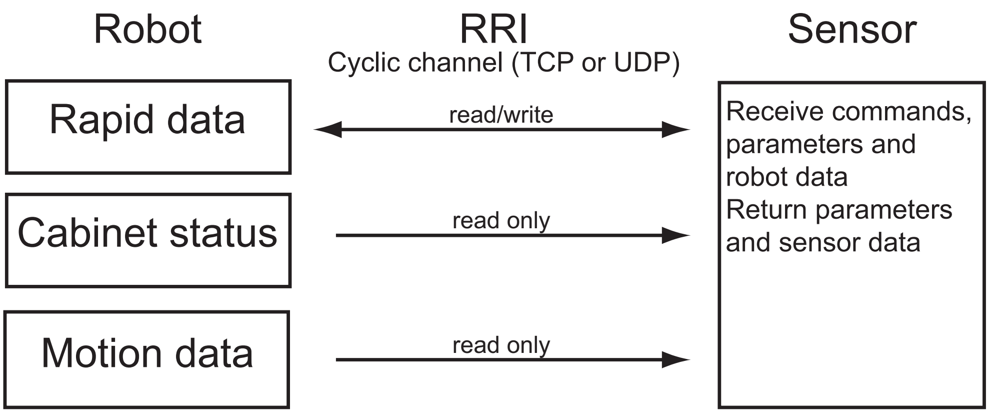
Introduction

Robot Reference Interface is included in the RobotWare option *Externally Guided Motion*. It can be used for 4-axis, 6-axis, and 7-axis robots.

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.



xx0800000128

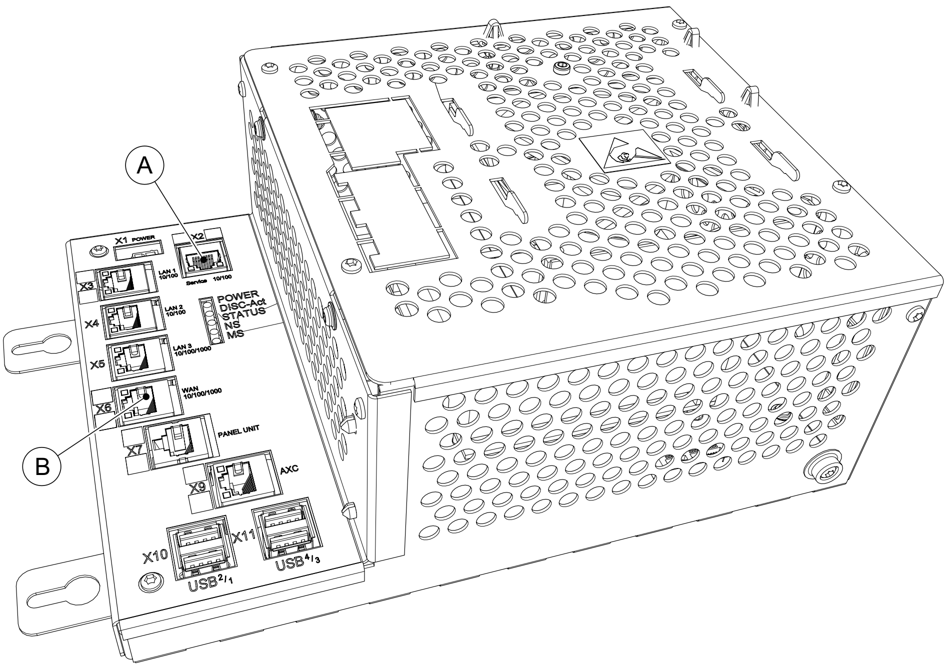
9.3.2 Installation

9.3.2.1 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).	**Note** The service connection can only be used if it is free.

9.3.2.2 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.

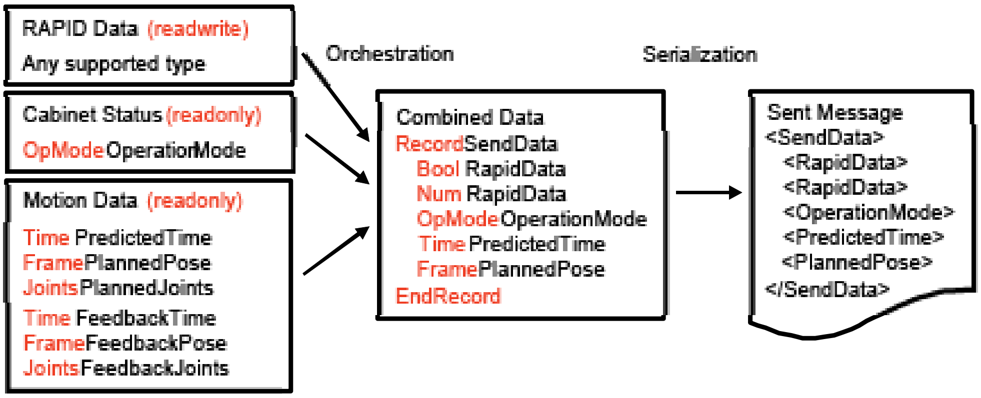
Therefore we recommended using a dedicated Ethernet link between the external system and the robot controller to provide the required performance for real-time applications. *Robot Reference Interface* can be used to communicate with any processor-based devices, that support IP via Ethernet and can serialize data into XML format.

9.3.2.3 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. **Note** The work object data needs to refer to a fixed work object. For example, it will not work with conveyor tracking. For more information about `wobjdata`, see *Technical reference manual - RAPID Instructions, Functions and Data types*.
FeedbackJoints	Joints	Robot joint values gathered from drive feedback.	

Continues on next page

Name	Type	Description	Comment
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.	

9.3.2.4 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-Jaw).	`pose`
`joint`	Robot joint values.	`robjoint`

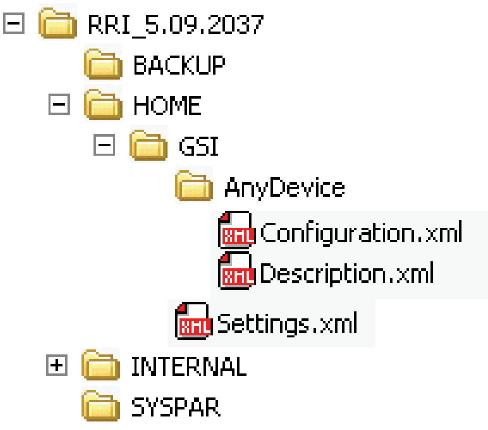
In addition, user-defined records can also be transferred from the external system to the robot controller, which are composed from the supported simple data types. User defined record types must be specified in the configuration file of the external device. See *Device configuration on page 367* for a description on how to create user-defined record types.

9.3.3 Configuration

9.3.3.1 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 363*.

For more detailed information of the *Description.xml* file see *Device description on page 364*.

For more detailed information of the *Configuration.xml* file see *Device configuration on page 367*.

9.3.3.2 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.

9.3.3.3 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 374*.

9.3.3.4 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	*Intern*	Optional. Can be omitted if members only have RAPID linkage.

Continues on next page

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 359*.
	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 359*.
	Type	Data type of the field	All supported data types	Described in section *Supported data types on page 361*.
	Size	Dimensions of the field (size of array)	Integer	Optional. Only basic types can be defined as array.
	Link	Linkage of complex type field	*Intern*	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 361*.
	Size	Dimension (Size of array)	Integer	Optional. Only basic types can be defined as array.
	Flag	Access Flag	None *ReadOnly* *WriteOnly* *ReadWrite*	Optional. Can be omitted if property is read and write enabled.
	Link	Linkage of property	*Intern*	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.

9.3.4 Configuration examples

9.3.4.1 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.

9.3.4.2 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 364*. 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 364*.

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

```
    <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="Real" Size="18"
         Alias="AppData" />
    </Record>
    <Record Name="SensorData">
      <Field Name="ErrorString" Type="String" Alias="EStr" />
      <Field Name="ApplicationData" Type="Real" 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 defined according 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 RobotData
  applicationdata AppData;
ENDRECORD
RECORD SensorData
  string ErrString;
  applicationdata AppData;
ENDRECORD
! Sensor Declarations
PERS sensor AnyDevice := [1,4,0];
PERS RobotData RobData := [[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]];
PERS SensorData SensData :=
     ["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, SensData, 64;
  SiSetCyclic AnyDevice, RobData, 64;
ENDPROC
! Close Interface Procedure
PROC RRI_Close()
  ! Close the connection
  SiClose RsMaster;
ENDPROC
ENDMODULE
```

Continues on next page

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>
```

9.3.5 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

9.4 Auto Acknowledge Input

Description

The RobotWare base functionality *Auto Acknowledge Input* is an option that enables a system input which will acknowledge the dialog presented on the FlexPendant when switching the operator mode from 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.

Remote control of operating mode

For information about using the safety module and a PLC for remote control of operating mode, see *Application manual - Functional safety and SafeMove2*.

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

The robot system must be installed with the option *Auto Acknowledge Input* using the **Modify Installation** function.

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

	Action
1	Save a copy of the I/O configuration file, *eio.cfg*, using the FlexPendant or RobotStudio.
2	Edit the I/O configuration file, *eio.cfg*, 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.

This page is intentionally left blank

10 Tool control options

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 online.

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:

- changing tool online
- 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 385*.

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.

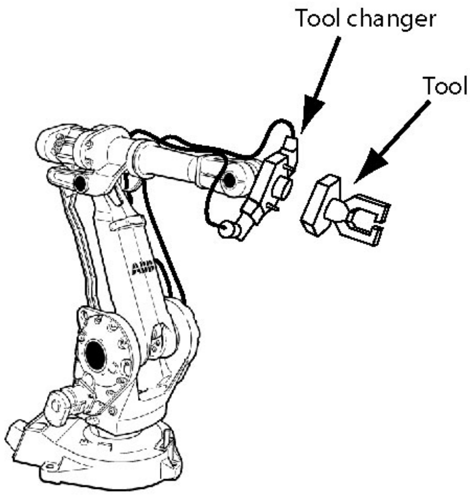
10.1.2 Requirements and limitations

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 *Application manual - Additional axes and standalone controller*.

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, use 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 to only activate a mechanical unit that is 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 383*.

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 *Application manual - Additional axes and standalone controller*.

The parameter *Deactivate PTC superv. at disconnect*, in the type *Mechanical Unit*, must be set to Yes.

The parameter *Disconnect at 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 manufacturer.

For a detailed description of the respective parameter, see *Technical reference manual - System parameters*.

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 *Technical reference manual - System parameters*.

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 *Name* in the type *Relay*.

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 *Use Connection Relay* in the type *Mechanical Unit*.
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

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: `DeactUnit gun1;`
2	Disconnect gun1 from the tool changer.
3	Connect gun2 to the tool changer.
4	Activate gun2 with the instruction: `ActUnit gun2;`
5	Optional but recommended: Calibrate gun2 with the instruction: `STCalib gun1 \ToolChg;` Note that this calibration requires option Servo Tool Control or Spot Servo.

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 385*.
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 Tool Control [1180-1]

10.2.1 Overview

Purpose

Tool Control can be used to control a servo tool, for example in a spot weld or Servo Gripper Application. *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

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

Basic approach

This is the general approach for using *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. Required hardware, such as drive module and measurement board, is specified in *Application manual - Additional axes and standalone controller*.

10.2.2 Servo tool movements

Closing and opening of a servo tool

The servo tool can be closed to a predefined thickness and tool force. When the tool reaches 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 tool force.

The force remains constant until an opening is ordered. Opening of the tool will reduce the tool 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

Note

This is not needed when controlling a gripper.

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 using the RAPID instruction `STCalib` (see *Instructions on page 392*). 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, for example when spot welding, 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 needs to be replaced.

Tip change calibration

The tip change calibration is to be used after mounting a new pair of tips, for example when spot welding. 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.

Continues on next page

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 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 395*.

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 396*.

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 393*.

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 394*.

10.2.5 RAPID components

About the RAPID components

This is an overview of all instructions, functions, and data types in *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. **Note** This is normally not needed when controlling a gripper. An argument determines which type of calibration will be performed: • `\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.

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

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 standalone controller*.

In this section, the parameters used in combination with *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, such as a gripper).

For gripper control, most of these parameters can be set to default values from the template files.

Parameter	Description
Close Time Adjust	Adjustment of the ordered minimum close time of the gun.
Close Position Adjust	Adjustment of the ordered position (plate thickness) where force control should start, when closing the gun.
Force Ready Delay	Delays the close ready event after achieving the ordered force.
Max Force Control Motor Torque	Max allowed motor torque for force control. Commanded force will be reduced, if the required motor torque is higher than this value.
Post-synchronization Time	Anticipation of the open ready event. This can be used to synchronize the gun opening with the next robot movement.
Calibration Mode	Defines the number of times the servo gun closes during a tip wear calibration.
Calibration Force Low	The minimum tip force used during a tip wear calibration.
Calibration Force High	The maximum tip force used during a tip wear calibration.
Calibration Time	The time that the servo gun waits in closed position during calibration.
Number of Stored Forces	Defines the number of points in the force-torque relation specified in *Tip Force 1 - 10* and *Motor Torque 1 - 10*.
Tip Force 1 - 10	*Tip Force 1* defines the tip force that corresponds to the motor torque in *Motor Torque 1*. *Tip Force 2* corresponds to *Motor Torque 2*, etc.
Motor Torque 1- 10	*Motor Torque 1* defines the motor torque that corresponds to the tip force in *Tip Force 1*. *Motor Torque 2* corresponds to *Tip Force 2*, etc.
Squeeze Position 1 - 10	Defines the joint position at each force level in the force calibration table.
Soft Stop Timeout	Defines how long the force will be maintained if a soft stop occurs during constant force.

Continues on next page

Parameter	Description
Automatic Open Disabled	This parameter should only be used for gripper control. Keeps the gripper closed even during and after a stop. The gripper can only be opened by the `STOpen` instruction.
Sync Check Off	This parameter should normally only be used for gripper control. Makes it possible to run the gripper without the `STCalib` instructions that otherwise are needed.

Force Master

These parameters belong to the type *Force Master* in the topic *Motion*.

Force Master is used to define how a servo tool, typically a servo gun, behaves during force control. The parameters only affect the servo tool when it is in force control mode.

Parameter	Description
References Bandwidth	The frequency limit for the low pass filter for reference values.
Use ramp time	Determines if the ramping of the tip force should use a constant time or a constant gradient.
Ramp when Increase Force	Determines how fast force is built up while closing the tool when *Use ramp time* is set to No.
Ramp time	Determines how fast force is built up while closing the tool when *Use ramp time* is set to Yes.
Collision LP Bandwidth	Frequency limit for the low pass filter used for tip wear calibration.
Collision Alarm Torque	Determines how hard the tool tips will be pressed together during the first gun closing of new tips calibrations and tool change calibrations.
Collision Speed	Determines the servo gun speed during the first gun closing of new tips calibrations and tool change calibrations.
Collision Delta Position	Defines the distance the servo tool has gone beyond the contact position when the motor torque has reached the value specified in *Collision Alarm Torque*.
Max pos err. closing	Determines how close to the ordered plate thickness the tool tips must be before the force control starts.
Delay ramp	Delays the starting of torque ramp when force control is started.
Ramp to real contact	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
No. of speed limits	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.
torque 1 - torque 6	The torque levels, corresponding to the ordered tip force, for which the speed limit and speed loop gain values are defined.
Speed Limit 1 - 6	*Speed Limit 1* to *Speed Limit 6* are used to define the maximum speed depending on the ordered tip force.

Continues on next page

Parameter	Description
Kv 1 - 6	*Kv 1* to *Kv 6* 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
Upper Joint Bound	Defines the upper limit of the working area for the joint.
Lower Joint Bound	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
Nominal Acceleration	Worst case motor acceleration.
Nominal Deceleration	Worst case motor deceleration.
Acceleration Derivate Ratio	Indicates how fast the acceleration can be increased.
Deceleration Derivate Ratio	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
Pole Pairs	Defines the number of pole pairs for the motor.
Inertia	The inertia of the motor, including the resolver but excluding the brake.
Stall Torque	The continuous stall torque, i.e. the torque the motor can produce at no speed and during an infinite time.
ke Phase to Phase	Nominal voltage constant. The induced voltage (phase to phase) that corresponds to the speed 1 rad/s.
Max Current	Max current without irreversible magnetization.
Phase Resistance	Nominal winding resistance per phase at 20 degrees Celsius.
Phase Inductance	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
Commutator Offset	Defines the position of the motor (resolver) when the rotor is in the electrical zero position relative to the stator.

Continues on next page

Parameter	Description
Calibration Offset	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
Speed Absolute Max	The absolute highest motor speed to be used.
Torque Absolute Max	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 continuos supervision of position, speed and torque.

Parameter	Description
Max Force Control Position Error	When a servo gun is in force control mode it is not allowed to move more than the distance specified in *Max Force Control Position Error*. This supervision will protect the tool if, for instance, one tip is lost.
Max Force Control Speed Limit	Speed error factor during force control. If the speed limits, defined in the type *Force Master Control*, multiplied with *Max Force Control Speed Limit* 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
Rotating Move	Defines if the axis is rotating or linear.
Transmission Gear Ratio	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
FFW Mode	Defines if the position regulation should use feed forward of speed and torque values.
Kp, Gain Position Loop	Proportional gain in the position regulation loop.
Kv, Gain Speed Loop	Proportional gain in the speed regulation loop.
Ti Integration Time Speed Loop	Integration time in the speed regulation loop.

Continues on next page

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
Kp, Gain Position Loop	Proportional gain in the position regulation loop.
Kv, Gain Speed Loop	Proportional gain in the speed regulation loop.
Ti Integration Time Speed Loop	Integration time in the speed regulation loop.
Speed Max Uncalibrated	The maximum allowed speed for an uncalibrated axis.
Acceleration Max Uncalibrated	The maximum allowed acceleration for an uncalibrated axis.
Deceleration Max Uncalibrated	The maximum allowed deceleration for an uncalibrated axis.

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 *Application manual - Additional axes and standalone controller*.
2	Load a *.cfg* file with the servo tool configuration. For detailed description on how to do this, see *Operating manual - RobotStudio*. 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 398*.
3	Use the RAPID instruction `STTune` 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 400*.
5	Unless force calibration was included in a loaded *.cfg* file, perform a force calibration.

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 400*.

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’.

For a gripper, it is sufficient with a normal calibration at a position where the fingers are touching, but are not squeezed together. In this case, `STCalib` instructions are not needed.

For this, it is recommended to create a service routine using the following instructions:

```
STCalib "ToolName" \TipChg;
STCalib "ToolName" \TipWear;
```

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’.

For this, it is recommended to use the same service routine as for the fine calibration.

10.2.9 RAPID code example

How to use the code package

The normal programming technique for *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 \WOIbj=WObj;
  STClose Gun, Thickness;
  rWeld;
  STOpen 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
```

10.2.10 Using tool control for gripper applications

Templates

There are no specific template files for grippers, but the Servo Gun files can be used as a foundation.

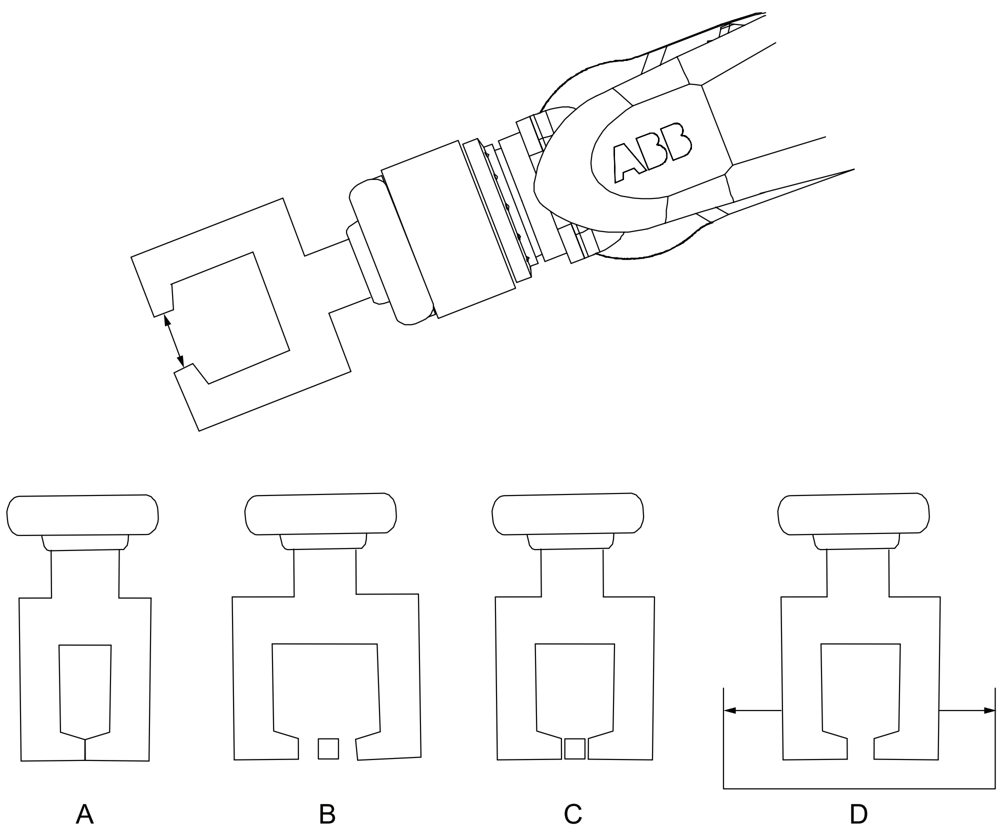
Parameters

When using the tool for gripper application, there are two key parameters that must be set. These parameters belong to the type *SG Process* in the topic *Motion*:

- *Automatic Open Disabled* keeps the gripper closed even during and after a stop. The gripper can only be opened by the `STOpen` instruction.
- *Sync Check Off* makes it possible to run the gripper without the `STCalib` instructions that otherwise are needed.

Instructions and positions

When using the tool control for gripper applications, the definition of zero position is when the fingers are closed.

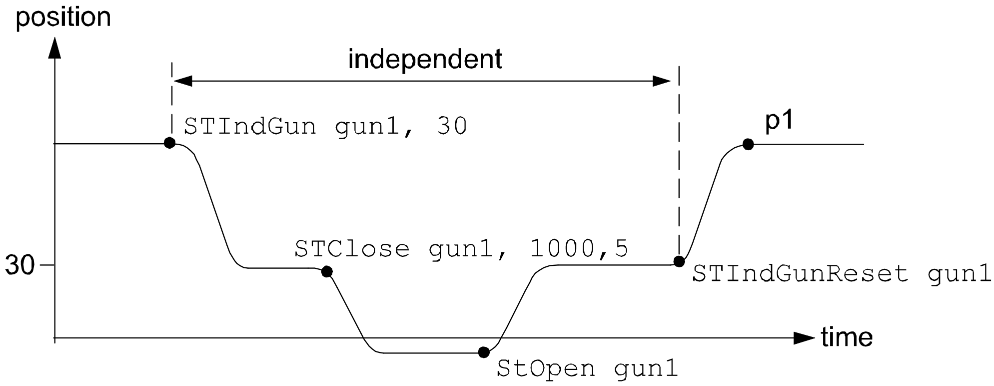


xx2000000214

A	Zero position
B	Example: `STIndGun` grip1,30
C	Example: `STClose` grip1,1000,5
D	Example: `STClose` grip1,-1000,20

Continues on next page

`STIndGun` instructions can be used to move the gripper independent of the normal movement instructions.



xx0500002342

If the gripper should squeeze in the opposite direction, the sign of the force should be negative.

10.3 I/O Controlled Axes [included in 1180-1]

10.3.1 Overview

Purpose

The purpose of *I/O Controlled Axes* is to control an axis from the robot controller by using an I/O interface instead of having the axis integrated into the IRC5 drive system.

For operation and programming, an I/O controlled axis acts just like an integrated process axis. The difference is that the drive unit of the I/O controlled axis is not directly connected to the drive system of the robot controller. The motion configuration provides an I/O interface, which connects the robot controller to an external servo regulator.

The robot controller can take and release control of the additional axis during program execution. The additional axis can be moved synchronously to the robot (while controlled by the robot controller) or independently of the robot (while controlled by an external PLC).

Some examples of applications are:

- Servo guns
- Grippers

What is included

The RobotWare option *I/O Controlled Axes* gives you access to system parameters for configuring I/O controlled axes.

Basic approach

This is the general approach for setting up *I/O Controlled Axes*.

1. Configure the system parameters for the axis to be controlled via I/O. See *Configuration on page 409*.
2. Operate the axis (jog, program etc.) just like any additional axis. See *RAPID programming on page 413*.

   For additional axis in general, also see *Operating manual - IRC5 with FlexPendant* and *Application manual - Additional axes and standalone controller*.

10.3.2 Contouring error

What is a contouring error

A contouring error is generated if an I/O controlled axis on the programmed robot path of the `robtarget` is not reached based on the bus delay and acceleration. If this event occurs, the robot's movement stops on the path. An error entry is made in the error log.

Possible causes for the occurrence of a contouring error:

- Robot collisions
- An external axis that is difficult to move or faulty
- Incorrect value of system parameter *Bus delay time in ms*

Error handling

1. Error – acknowledgement at the external process unit.

   For that, each application needs to provide a reset button. The process unit needs to be ready before the program can be started.
2. Motors On / Program start

   If automatic movement back to path is allowed, the robot will move back automatically to path before the program continues with the instruction that was canceled. In case automatic movement is not allowed, a error message occurs. A selection menu provides possibilities to accept the movement or to cancel the start event.

   In case the start event is canceled, the operator needs to change the operation mode to manual.

   Now the operator can specify a further procedure before the robot program can be restarted. For example:

   - move the robot manual out of collision area
   - move to a previous move instruction

For more information, see topic *Controller*, type *Path Return Region* in *Technical reference manual - System parameters*.

10.3.3 Correcting the position

Correcting the position

Correcting (teaching) a robot position (`robtarget`) is done using the button **Modify Position** in the program editor (as for the robot axes).

For the following states, the modified position of the I/O controlled axis will not be the current position, but the last valid feedback position:

- Axis is not referenced
- Servo regulator is not operative
- Actual position of the I/O interface invalid
- Position is outside the operating range

The position correction is adopted for activated axes only. If an available axis is not activated, this axis is ignored. This means the `robtarget` substitute symbol for the axis in question remains unchanged. This state does not lead to an error.

10.3.4 Tool changing

Tool changing

If a tool is deactivated with the instruction `DeactUnit`, it is necessary to set the signal unit disable. When the tool is disabled (can be verified with signal *unit_disabled*), it is possible to disconnect the power supply to the tool, for example undock a spotwelding gun.

It is possible to configure the same logical axis number for different tools, but this requires the RobotWare option *Servo Tool Change*.

10.3.5 Installation

Installation

After installation of the robot system, the I/O controlled axes needs to be loaded in the system parameters.

Each required axis needs to be loaded separately. The specific motion file includes default motion parameters. Parameterization and adjustments of the loaded axis is described in more detail in *Configuration on page 409*.

10.3.6 Configuration

Template configuration files

Template configuration files are available for setting up the I/O controlled axes. The files can be loaded to the controller, using RobotStudio or the FlexPendant, to facilitate and speed up the configuration.

The template configuration files can be obtained from RobotStudio or the IRC5 controller.

- **In the RobotWare installation folder in RobotStudio**: *...\RobotPackages\RobotWare_RPK_<version>\utility\ioctrlaxis*
- **On the IRC5 Controller**: *<SystemName>\PRODUCTS\<RobotWare_xx.xx.xxxx>\utility\ioctrlaxis*

> **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**.

Adding the I/O controlled axis

Loading the template configuration files for I/O controlled axis will install a mechanical unit called EXTCTL1 with default signal names defined in the type *External Control Process Data*, topic *Motion*.

1. Load one of the template motion configuration files for axis 1, select between logical axis number 7, 8, or 9.

   (ioctrl1_mn7_l7_moc.cfg, ioctrl1_mn7_l8_moc.cfg, ioctrl1_mn7_l9_moc.cfg)
2. Load one of the template I/O configuration files depending on the industrial network.

   (ioctrl1_eio.cfg, ioctrl1_pnet_eio.cfg)
3. Edit the I/O configuration and change from virtual signals to real signals according to the current setup.

Mandatory settings for the I/O controlled axis

The following configuration must be done with data for the mechanical unit that should be used as an I/O controlled axis.

1. In type *Transmission*, set *Transmission Gear Ratio*. See *Type Transmission on page 412*.
2. In type *Acceleration Data*, set *Nominal Acceleration*, *Nominal Deceleration*, *Acceleration Derivate Ratio* and *Deceleration Derivate Ratio*. See *Type Acceleration Data on page 411*.
3. In type *Arm*, set *Upper Joint Bound* and *Lower Joint Bound*. See *Type Arm on page 412*.
4. In type *Stress Duty Cycle*, set *Speed Absolute Max*. See *Type Stress Duty Cycle on page 412*.

Continues on next page

5. In type *Supervision Type*, set *static_position_limit* and *dynamic_position_limit*. See *Type Supervision Type on page 412*.
6. In type *External Control Process Data*, set *Bus delay time in ms*. See *Type External Control Process Data on page 411*.

Optional customization settings

If other values than the default values are preferred, any of the following settings can be changed.

- To change the logical axis number, change the value for *Logical Axis*. See *Type Joint on page 412*.
- To change the names of the signals used to communicate with the I/O controlled axis, change the settings in the type *External Control Process Data*, see *Type External Control Process Data on page 411*.
- To use an activation relay, set the parameter *Use Activation Relay*. See *Type Mechanical Unit on page 412*.

Adding another axis

For a second or third I/O controlled axis, EXTCTL2 and EXTCTL3, the corresponding configuration files must be loaded from the template folder.

1. Load one of the template configuration files for axis 2 or 3.
2. Make the same configurations as for the first I/O controlled axis.

Note

Several mechanical units may use the same logical axis number, but this requires the RobotWare option *Servo Tool Change*.

Settings for PROFINET

If a PROFINET bus is used, the parameter *Reduction ratio* should be set to 4 ms or 2 ms for the I/O controlled unit. See *Application manual - PROFINET Controller/Device*.

10.3.7 System parameters

About the system parameters

This is a brief description of each parameter in the option *I/O Controlled Axes*. For more information, see the respective parameter in *Technical reference manual - System parameters*.

Type External Control Process Data

These parameters belongs to the type *External Control Process Data* in the topic *Motion*.

Parameter	Description
Bus delay time in ms	Parameter for bus delay time.
Regulator activation signal	Output signal for activation of the I/O controlled unit.
Ext Controller output signal	Output signal for allowing external control of the unit.
Pos_ref output signal	Output signal with positioning reference for the I/O controlled axis.
Pos_ref sign signal	Output signal with sign (+ or -) of the positioning reference for the I/O controlled axis.
Pos_ref valid signal	Output signal that signals that the positioning reference is a valid signal and the axis needs to follow the reference signal.
Regulator is activated signal	Input signal that indicates if the I/O controlled unit is enabled and ready.
Req pos is out of range input signal	Input signal that signals if the required positioning reference is out of range.
Pos_fdb input signal	Input signal with position feedback from the I/O controlled axis.
Pos_fdb sign signal	Input signal with with sign (+ or -) of the position feedback from the I/O controlled axis.
Pos_fdb_valid signal	Input signal that indicates that the position feedback signal is valid.
Unit_ready input signal	Input signal from I/O controlled unit indicating that it is ready.
Ext Controller input signal	Input signal indicating that the external unit is in control of the movement. The robot controller is not allowed to move the external unit.
No program pointer move after error	The program pointer does not need to be moved after the an error.

Type Acceleration Data

These parameters belongs to the type *Acceleration Data* in the topic *Motion*.

Parameter	Description
Nominal Acceleration	Worst case motor acceleration.
Nominal Deceleration	Worst case motor deceleration.
Acceleration Derivate Ratio	Defines how fast the acceleration can build up, i.e. an indication of the derivative of the acceleration.

Continues on next page

Parameter	Description
Deceleration Derivate Ratio	Defines how fast the deceleration can build up, i.e. an indication of the derivative of the deceleration.

Type Arm

These parameters belongs to the type *Arm* in the topic *Motion*.

Parameter	Description
Upper Joint Bound	Defines the upper limit of the working area for this joint.
Lower Joint Bound	Defines the lower limit of the working area for this joint.

Type Joint

These parameters belongs to the type *Joint* in the topic *Motion*.

Parameter	Description
Logical Axis	Defines the axis number as seen by a RAPID program. Two mechanical units can have the same value set for *Logical Axis*, but then they cannot be activated at the same.

Type Mechanical Unit

These parameters belongs to the type *Mechanical Unit* in the topic *Motion*.

Parameter	Description
Use Activation Relay	Points out a relay that will be activated or deactivated when the mechanical unit is activated or deactivated.

Type Stress Duty Cycle

These parameters belongs to the type *Stress Duty Cycle* in the topic *Motion*.

Parameter	Description
Speed Absolute Max	The absolute highest motor speed to be used in meters/second.

Type Supervision Type

These parameters belongs to the type *Supervision Type* in the topic *Motion*.

Parameter	Description
static_position_limit	Position error limit at zero speed, in meters on motor side.
dynamic_position_limit	Position error limit (max lag) at max speed, in meters on motor side.

Type Transmission

These parameters belongs to the type *Transmission* in the topic *Motion*.

Parameter	Description
Transmission Gear Ratio	Defines the transmission gear ratio between motor and joint. For most axis this parameter is set to 1.

10.3.8 RAPID programming

Data types

This is a brief description of specific considerations regarding RAPID data types when using I/O Controlled Axes.

General descriptions of the data types are found in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Data type	Description
robtarget	The position of the I/O controlled axis is set as an additional axis in a `robtarget`. Example, where the I/O controlled axis is logical axis 7 and should be moved to position 100: `p1 := [[20,50,-80], [1,0,0,0], [1,1,0,0], [100,9E+09,9E+09,9E+09,9E+09,9E+09]];`

Instructions

This is a brief description of specific considerations regarding RAPID instructions when using I/O Controlled Axes.

General descriptions of the instructions are found in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Instruction	Description
MoveL MoveC MoveJ	Regular move instructions are used to move an I/O controlled axis. The position value of the I/O controlled value is included in the `robtarget`, see *Data types on page 413*. The I/O controlled axis can be moved simultaneously with the robot.

RAPID example

```
PROC Sequence123()
  ...
  MoveJ pHome, v1500, fine, tGun1;
  ActUnit EXTCTL1;
  MoveJ p100, v1000, z10, tGun1 \Wobj:=wobj1;
  MoveL p101, v1000, fine, tGun1 \Wobj:=wobj1;
  ...
  ! Application-specific commands
  ...
  MoveL p102, v1000, z10, tGun1 \Wobj:=wobj1;
  MoveJ p100, v1000, fine, tGun1 \Wobj:=wobj1;
  DeactUnit EXTCTL1;
  MoveJ pHome, v1500, fine, tGun1;
ENDPROC
```

This page is intentionally left blank

Index

ABB AB
Robotics & Discrete Automation
S-721 68 VÄSTERÅS, Sweden
Telephone +46 10-732 50 00

ABB AS
Robotics & Discrete Automation
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 22 87 2000

ABB Engineering (Shanghai) Ltd.
Robotics & Discrete Automation
No. 4528 Kangxin Highway
PuDong New District
SHANGHAI 201319, China
Telephone: +86 21 6105 6666

ABB Inc.
Robotics & Discrete Automation
1250 Brown Road
Auburn Hills, MI 48326
USA
Telephone: +1 248 391 9000

abb.com/robotics

3HAC050798-001, Rev V, en