Application manual

Robot communication and I/O control

Robot Controller
RobotWare 5.12

Application manual

Robot communication and I/O control

RobotWare 5.12

Document ID: 3HAC020435-001

Revision: F

ABB AB
Robotics Products
SE-721 68 Västerås
Sweden

Overview

About this manual

This manual explains the basics of when and how to use the following RobotWare base functionality and options:

- *File and Serial Channel Handling*
- *Fieldbus Command Interface*
- *Logical Cross Connections*
- *Analog Signal Interrupt*
- *PC Interface and Socket Messaging* (616-1)
- *FTP and NFS Client* (614-1)
- *RAPID Message Queue*

Usage

This manual can be used either as a reference to find out if a base functionality or option is the right choice for solving a problem, or as a description of how to use a base functionality or 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 mainly 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.

Organization of chapters

The manual is organized in the following chapters:

Chapter	Contents
1.	Describes the base functionality *File and Serial Channel Handling*.
2.	Describes the base functionality *Fieldbus Command Interface*.
3.	Describes the base functionality *Logical Cross Connections*.
4.	Describes the base functionality *Analog Signal Interrupt*.
5.	Describes the option *PC Interface*.
6.	Describes the option *FTP Client*.
7.	Describes the option *NFS Client*.
8.	Describes the option *Socket Messaging*.
9.	Describes the base functionality *RAPID Message Queue*.

Continues on next page

References

Reference	Document Id
Technical reference manual - RAPID overview	3HAC16580-1
Technical reference manual - RAPID Instructions, Functions and Data types	3HAC16581-1
Operating manual - IRC5 with FlexPendant	3HAC16590-1
Technical reference manual - System parameters	3HAC17076-1
Operating manual - RobotStudio	3HAC032104-001

Revisions

Revision	Description
-	First edition
A	*Socket Messaging* has been added.
B	*File and Serial Channel Handling*: Limitations added. *Socket Messaging*: Improved example to avoid `SocketClose` directly after `SocketSend`. *Fieldbus Command Interface*: Corrected code example.
C	*Socket Messaging*: Added tips and note.
D	*RAPID Message Queue* has been added.
E	*File and Serial Handling*, *Fieldbus Command Interface*, *Logical Cross Connections*, *Analog Signal Interupt*, and *Socket Messaging* are now part of the RobotWare base functionality.
F	*RAPID Message Queue* functionality extended with synchronous message handling.

Product documentation, M2004

General

The robot documentation is divided into a number of categories. This listing is based on the type of information contained within the documents, regardless of whether the products are standard or optional. This means that any given delivery of robot products **will not contain all** documents listed, only the ones pertaining to the equipment delivered.

However, all documents listed may be ordered from ABB. The documents listed are valid for M2004 robot systems.

Product manuals

All hardware, robots and controllers, will be delivered with a **Product manual** that contains:

- Safety information
- Installation and commissioning (descriptions of mechanical installation, electrical connections)
- Maintenance (descriptions of all required preventive maintenance procedures including intervals)
- Repair (descriptions of all recommended repair procedures including spare parts)
- Additional procedures, if any (calibration, decommissioning)
- Reference information (article numbers for documentation referred to in Product manual, procedures, lists of tools, safety standards)
- Part list
- Foldouts or exploded views
- Circuit diagrams

Technical reference manuals

The following manuals describe the robot software in general and contain relevant reference information:

- **RAPID Overview**: An overview of the RAPID programming language.
- **RAPID Instructions, Functions and Data types**: Description and syntax for all RAPID instructions, functions and data types.
- **System parameters**: Description of system parameters and configuration workflows.

Application manuals

Specific applications (for example software or hardware options) are described in **Application manuals**. An application manual can describe one or several applications.

An application manual generally contains information about:

- The purpose of the application (what it does and when it is useful)
- What is included (for example cables, I/O boards, RAPID instructions, system parameters, CD with PC software)
- How to use the application
- Examples of how to use the application

Continues on next page

Operating manuals

This group of manuals is aimed at those having first hand operational contact with the robot, that is production cell operators, programmers and trouble shooters. The group of manuals includes:

- **Emergency safety information**
- **General safety information**
- **Getting started, IRC5**
- **IRC5 with FlexPendant**
- **RobotStudio**
- **Introduction to RAPID**
- **Trouble shooting**, for the controller and robot

Safety

Safety of personnel

A robot is heavy and extremely powerful regardless of its speed. A pause or long stop in movement can be followed by a fast hazardous movement. Even if a pattern of movement is predicted, a change in operation can be triggered by an external signal resulting in an unexpected movement.

Therefore, it is important that all safety regulations are followed when entering safeguarded space.

Safety regulations

Before beginning work with the robot, make sure you are familiar with the safety regulations described in the manual *General safety information*.

1 File and Serial Channel Handling

1.1. Introduction to File and Serial Channel Handling

About File and Serial Channel Handling

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

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

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

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

1.2 Binary and character based communication

1.2.1. Overview

Purpose

The purpose of binary and character based communication is to:

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

What is included

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

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

Basic approach

This is the general approach for using binary and character based communication. For a more detailed example of how this is done, see *Code examples on page 14*.

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

Limitations

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

1.2.2. RAPID components

Data types

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

Data type	Description
iodev	`iodev` contains a reference to a file or serial channel. It can be linked to the physical unit with the instruction `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 serial channel for reading or writing.
Close	`Close` is used to close a file or serial channel.
Rewind	`Rewind` sets the file position to the beginning of the file.
ClearIOBuff	`ClearIOBuff` is used to clear the input buffer of a serial channel. All buffered characters from the input serial channel are discarded.
Write	`Write` is used to write to a character based file or serial channel.
WriteBin	`WriteBin` is used to write a number of bytes to a binary serial channel or file.
WriteStrBin	`WriteStrBin` is used to write a string to a binary serial channel or file.
WriteAnyBin	`WriteAnyBin` is used to write any type of data to a binary serial channel or file.
ReadAnyBin	`ReadAnyBin` is used to read any type of data from a binary serial channel or file.

Functions

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

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

1.2.3. Code examples

Communication with character based file

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

```
PROC write_to_file()
  VAR iodev file;
  VAR num number:= 8;

  Open "HOME:" \File:= "FILE1.DOC", file;
  Write file, "The number is :"\Num:=number;
  Close file;
ENDPROC

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

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


Continues on next page

Communication with binary serial channel

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

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

  Open "com1:", channel\Bin;

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

  ! Wait for  control character ack
  input := ReadBin (channel \Time:= 0.1);
  IF input = 6 THEN

    ! Write "Hello" followed by  new line
    WriteStrBin channel, "Hello\0A";

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

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

    ! Set character "H" (72="H")
    out_buffer{2} := 72;

    ! Set character "i"
    out_buffer{3} := StrToByte("i"\Char);

    ! Set new line character (10=new line)
    out_buffer{4} := 10;

    ! Set end text character (3=end text)
    out_buffer{5} := 3;

    ! Write the buffer with the line "Hi"
    ! to the channel
    WriteBin channel, out_buffer, 5;

  ENDIF
  Close channel;
```

Continues on next page

```
ENDPROC
```

In this example, the same sequence as above is read from the channel com2.

```
PROC read_bin_chan()
  VAR iodev channel;
  VAR string text;
  VAR num bindata;
  VAR robtarget target;

  Open "com2:", channel \Bin;

  ! Clear input buffer for com2
  ClearIOBuff channel;

  ! Wait for input from com2 and then
  ! write the first 5 characters to FlexPendant
  TPWrite ReadStrBin (channel, 5);

  ! Read the new line character
  ReadStrBin channel, 1;

  ! Read the next input, interpreted as  robtarget
  ! and move robot to that target
  ReadAnyBin channel, target;
  MoveJ target, vmax, fine, tool1;

  ! Read text one character at the time
  ! until end of file
  bindata := ReadBin(channel);
  WHILE bindata <> EOF_BIN DO
     text := text + ByteToStr(bindata\Char);
     bindata := ReadBin(channel);
  ENDWHILE
  TPWrite text;
  Close channel;
ENDPROC
```

1.3 Raw data communication

1.3.1. Overview

Purpose

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

What is included

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

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

Basic approach

This is the general approach for raw data communication. For a more detailed example of how this is done, see *Write and read rawbytes on page 19*.

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

Limitations

Fieldbus communication also require the base functionality Fieldbus Command Interface and the option for the fieldbus in question.

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

1.3.2. 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 type `num`, `byte` or `string`. It also store the length of the valid data (in bytes). `rawbytes` can contain up to 1024 bytes of data. The supported data formats are: • Hex (1 byte) • long (4 bytes) • float (4 bytes) • ASCII (1-80 characters)

Instructions

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

Instruction	Description
ClearRawBytes	`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, serial channel or fieldbus.
ReadRawBytes	`ReadRawBytes` is used to read data of type `rawbytes` from any binary file, serial channel or fieldbus.

Functions

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

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

1.3.3. 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 27*.

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
```


Continues on next page

```
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
  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);
```

1.4 File and directory management

1.4.1. Overview

Purpose

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

What is included

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

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

Basic approach

This is the general approach for file and directory management. For more detailed examples of how this is done, see *Code examples on page 23*.

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

1.4.2. 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 **.** and **..** 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.

1.4.3. 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
```


Continues on next page

```
ENDWHILE
CloseDir directory;

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

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 Fieldbus Command Interface

2.1. Introduction to Fieldbus Command Interface

Purpose

Fieldbus Command Interface provides an interface to talk to a DeviceNet device.

This interface is used together with raw data communication, see *Raw data communication on page 17*.

What is included

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

- Instruction used to create a DeviceNet header.

Basic approach

This is the general approach for using Fieldbus Command Interface. For a more detailed example of how this is done, see *Write rawbytes to DeviceNet on page 27*.

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 device.
4. Read data from the DeviceNet device to a `rawbytes` variable.
5. Extract the data from the `rawbytes` variable.

Limitations

Fieldbus communication also require the base functionality File and Serial Channel Handling and the option for the fieldbus in question.

2.2. RAPID components and system parameters

Data types

There are no data types for Fieldbus Command Interface.

Instructions

This is a brief description of each instruction in Fieldbus 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 board.

Functions

There are no functions for Fieldbus Command Interface.

System parameters

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

Parameter	Description
Explicit Messaging	Enables DeviceNet Explicit connection to the device. *Explicit Messaging* must be set to Enabled for the DeviceNet unit type that you want to use Fieldbus Command Interface to communicate with. *Explicit Messaging* belongs to the type *Unit Type* in the topic *I/O System*.

2.3. Code example

Write rawbytes to DeviceNet

In this example, data packed as a `rawbytes` variable is written to a DeviceNet board. For more details regarding `rawbytes`, see *Raw data communication on page 17*.

```
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 board
  PackDNHeader "10", "6,20 1D 24 01 30 64,8,1",
     rawdata_out;

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

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

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

  ! Read the answer from dev
  ReadRawBytes dev, rawdata_in;

  ! Close FCI device
  Close dev;

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


Continues on next page

Continued

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

3 Logical Cross Connections

3.1. Introduction to Logical Cross Connections

Purpose

The purpose of Logical Cross Connections is to check combinations of digital signals. This can be used to check 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 signals, a lot of RAPID code execution can be avoided. Logical Cross Connections can replace the process of reading signal values, calculate new values and writing the values to 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

Cross connections are used to define a digital signal's dependencies to other digital signals. Without the base functionality Logical Cross Connections, a signal can only be defined to have the same value as another signal.

With the base functionality Logical Cross Connections, more complex dependencies can be configured with the logical operators AND, OR, and inverted signal values.

The signals that constitute the logical expression (actor signals) and the signals that are the result of the expression (resultant signal) can be either digital input signals or digital output signals.

What is included

The RobotWare base functionality Logical Cross Connections allows you to build logical expressions with up to 5 actor signals and the logical operations AND, OR, and inverted signal values.

3.2. 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 *Technical reference manual - System parameters*.

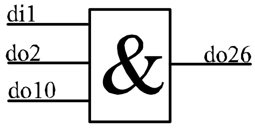
These parameters belong to the type *Cross Connection* in the topic *I/O System*.

Parameter	Description
Resultant	The signal that receive the result of the cross connection as its new value.
Actor 1	The first 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 the *Actor 1* signal 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 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 the *Actor 2* signal is used in the evaluation of the *Resultant*.
Operator 2	Operand between *Actor 2* and *Actor 3*. See *Operator 1*.
Actor 3	The third 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 the *Actor 3* signal is used in the evaluation of the *Resultant*.
Operator 3	Operand between *Actor 3* and *Actor 4*. See *Operator 1*.
Actor 4	The fourth 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 the *Actor 4* signal is used in the evaluation of the *Resultant*.
Operator 4	Operand between *Actor 4* and *Actor 5*. See *Operator 1*.
Actor 5	The fifth 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 the *Actor 5* signal is used in the evaluation of the *Resultant*.

3.3. Examples

Logical AND

The following logical structure...



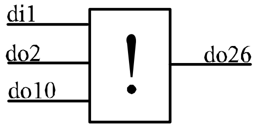
xx0300000457

... is created as shown below.

Resultant	Actor 1	Invert actor 1	Opera-tor 1	Actor 2	Invert actor 2	Opera-tor 2	Actor 3	Invert actor 3
do26	di1	No	AND	do2	No	AND	do10	No

Logical OR

The following logical structure...



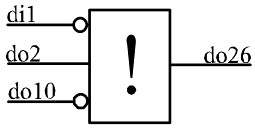
xx0300000459

... is created as shown below.

Resultant	Actor 1	Invert actor 1	Opera-tor 1	Actor 2	Invert actor 2	Opera-tor 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) ...



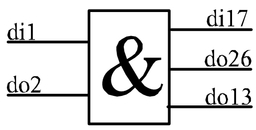
xx0300000460

... is created as shown below.

Resultant	Actor 1	Invert actor 1	Opera-tor 1	Actor 2	Invert actor 2	Opera-tor 2	Actor 3	Invert actor 3
do26	di1	Yes	OR	do2	No	OR	do10	Yes

Several resultant signals

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



xx0300000462

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

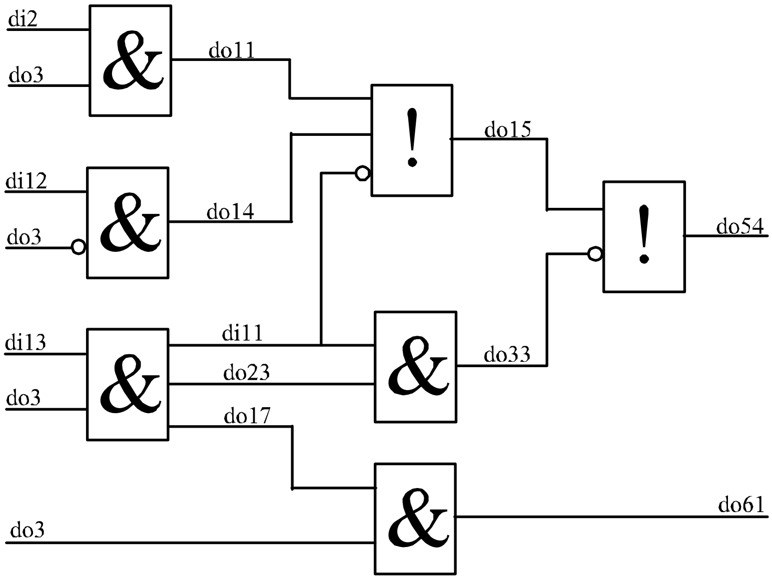
Continues on next page

Continued

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	Opera-tor 1	Actor 2	Invert actor 2	Opera-tor 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			

3.4. Limitations

Evaluation order

If more than two actor 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.

Max actor signals

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

Max cross connections

The maximum number of cross connections is 100.

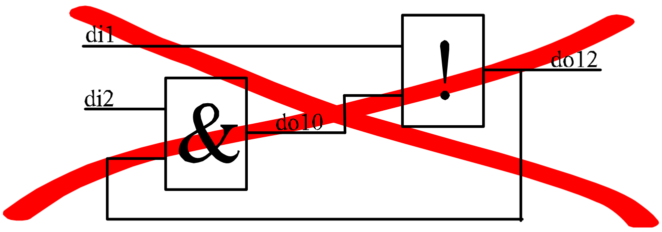
Max depth

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

A resultant signal from a cross connection cannot be used as an actor signal for the same cross connection. Nor can a signal be used as an actor signal in a cross connection if its value depends on that cross connection via a number of other cross connections.



xx0300000458

An error message will inform you if the cross connections form a loop.

4 Analog Signal Interrupt

4.1. 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 37*.

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 a fieldbus option (for example DeviceNet or Profibus).

4.2. 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.

4.3. 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
```

5 PC Interface [616-1]

5.1. Introduction to PC Interface

Purpose

PC Interface is used for communication between the controller and a PC.

PC Interface is required for some software products from ABB, such as WebWare and some functionality in RobotStudio.

With PC Interface, data can be sent to a PC. This is, for example, used for:

- Backup.
- Production statistics logging.
- Operator information presented on a PC.

What is included

The RobotWare option PC Interface gives you access to:

- A RAPID instruction used to send data to a PC: `SCWrite`.
- An Ethernet communication interface, which is used by some ABB software products.

Basic approach

This is the general approach for using PC Interface.

1. Set up a WebWare client application on a PC. For more information, see documentation for WebWare.
2. Use the instruction `SCWrite` to send data from the RAPID program to the client application on the PC. For more information, see *Send variable from RAPID on page 40*.

5.2. Send variable from RAPID

SCWrite instruction

`SCWrite` (Superior Computer Write) is 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*.

Client application

The PC must have a client application that can subscribe to the information sent from the controller with `SCWrite`.

This client application can be either:

- a WebWare custom database
- a client application developed with WebWare SDK

For more information, see documentation for WebWare.

Continues on next page

Code example

In this example the robot moves objects to a position where they can be treated by a process controlled by the PC. When the object is ready the robot moves it to its next station.

The program use `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.

```
PERS bool in_position:=FALSE;
PERS num nbr_objects:=0;

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
  in_position:=TRUE;
  SCWrite in_position;

  ! 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;
  SCWrite in_position;

  ! Inform PC how many object has been handled
  nbr_objects:= nbr_objects+1;
  SCWrite nbr_objects;
ENDWHILE
```

5.3. 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.

WebWare

WebWare is a software product created to give extra functionality to the industrial robot user. PC Interface is required to run WebWare.

Functionality	Description
Backup	Backup of all programs and system parameters.
Metrics	Access to status data for the robot and logging of error messages and other data.
Service	Allows a service organization to get remote access to the robot.
OPC Server	OLE for Process Control is an industry standard for uniform interfaces.
SDK	Software Development Kit for creating customized applications that subscribe to data from the controller.

For more information, see documentation for WebWare.

RobotStudio

RobotStudio is a software product delivered with the robot. However, all its functionality is not accessible by default. Some of the functionality requires PC Interface.

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*.

6 FTP Client [614-1]

6.1. Introduction to FTP Client

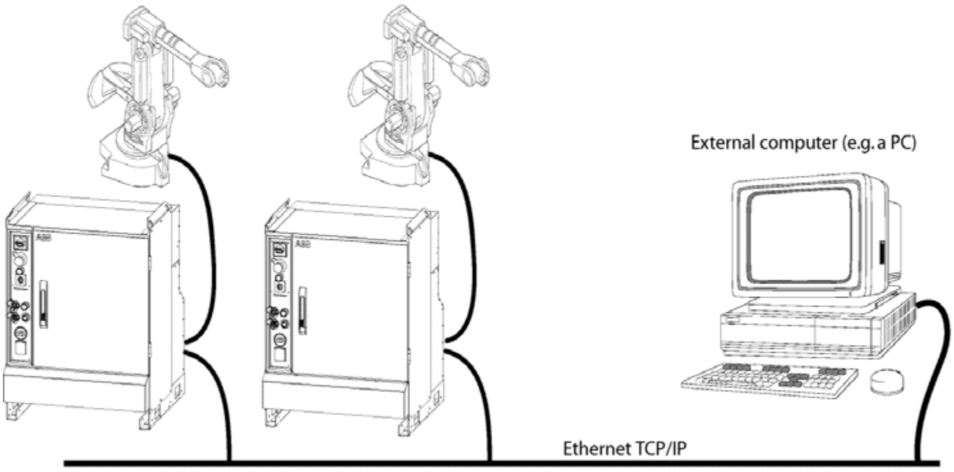
Purpose

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

Basic approach

This is the general approach for using FTP Client. For more detailed examples of how this is done, see *Examples on page 46*.

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

Requirements

The external computer must have:

- TCP/IP stack
- FTP Server

6.2. System parameters

Application protocol

This is a brief description of the parameters used to configure an application protocol. For more information, see the respective parameter in *Technical reference manual - System parameters*.

These parameters belongs to the type *Application protocol* in the topic *Communication*.

Parameter	Description
Name	Name of the application protocol.
Type	Type of application protocol. Set this to "FTP".
Transmission protocol	Name of the transmission protocol the protocol should use. For FTP, this is always "TCP/IP".
Server address	The IP address of the computer with the FTP server.
Trusted	This flag decides if this computer should be trusted, i.e. if losing the connection should make the program stop.
Local path	Defines what the shared unit will be called on the robot. The parameter value must end with a colon (:). If, for example the unit is named "pc:", the name of the test.prg on this unit would be pc:test.prg
Server path	The name of the disk or folder to connect to, on the remote computer. If not specified, the application protocol will reference the directory that is shared by the FTP server. **Note:** If communicating with an FTP server of type Distinct FTP or MS IIS, the exported path should not be specified.
Username	The user name used by the robot when it logs on to the remote computer. The user account must be set up on the FTP server.
Password	The password used by the robot when it logs on to the remote computer. Note that the password written here will be visible to all who has access to the system parameters.

Transmission protocol

There is a configured transmission protocol called TCP/IP, but no changes can be made to it. This is used by the FTP application protocol.

6.3. Examples

Example configuration

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

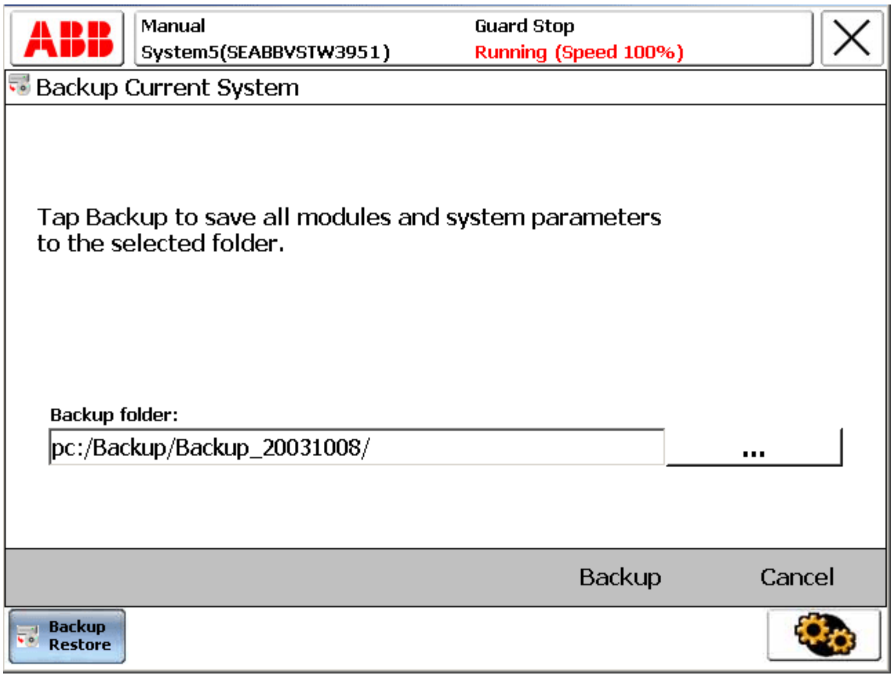
Parameter	Value
Name	my FTP protocol
Type	FTP
Transmission protocol	TCP/IP
Server address	100.100.100.100
Trusted	No
Local path	pc:
Server path	C:\robot_1
Username	Robot1
Password	robot1

Note that if communicating with an FTP server of type Distinct FTP or MS IIS, the value of *Server path* should exclude the exported path.

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).



en0300000504

Continues on next page

Example with RAPID code

This example shows how to open the file C:\robot_1\files\file1.doc on the remote PC from a RAPID program on the controller. We assume that the configuration is done according to the example configuration shown above.

```
Open "HOME:" \File:= "pc:/files/file1.doc", file;
```

7 NFS Client [614-1]

7.1. Introduction to NFS Client

Purpose

The purpose of NFS Client is to enable the robot to access remote mounted disks, for example a hard disk drive on a PC.

Here are some examples of applications:

- Backup to a remote computer.
- Load programs from a remote computer.

Description

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

Once the NFS application protocol is configured, the remote computer can be accessed in the same way as the controller's internal hard disk.

What is included

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

Basic approach

This is the general approach for using NFS Client. For more detailed examples of how this is done, see *Examples on page 46*.

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

7.2. System parameters

Application protocol

This is a brief description of the parameters used to configure an application protocol. For more information, see the respective parameter in *Technical reference manual - System parameters*

These parameters belongs to the type *Application protocol*in the topic *Communication*.

Parameter	Description
Name	Name of the application protocol.
Type	Type of application protocol. Set this to "NFS".
Transmission protocol	Name of the transmission protocol the protocol should use. For NFS, this is always "TCP/IP".
Server address	The IP address of the computer with the NFS server.
Trusted	This flag decides if this computer should be trusted, i.e. if losing the connection should make the program stop.
Local path	Defines what the shared unit will be called on the robot. The parameter value must end with a colon (:). If, for example the unit is named "pc:", the name of the test.prg on this unit would be pc:test.prg
Server path	The name of the exported disk or folder on the remote computer. For NFS, Server Path must be specified.
User ID	Used by the NFS protocol as a way of authorizing the user to access a specific server. If this parameter is not used, which is usually the case on a PC, set it to the default value 0. Note that *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.

Transmission protocol

There is a configured transmission protocol called TCP/IP, but no changes can be made to it. This is used by the NFS application protocol.

7.3. Examples

Example configuration

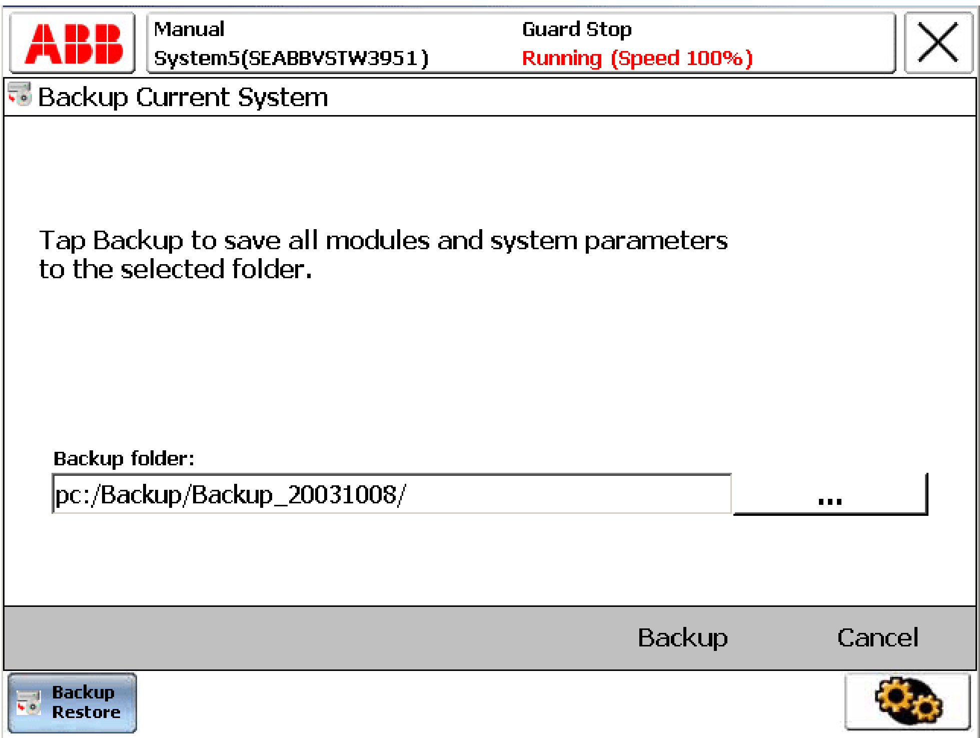
This is an example of how an application protocol can be configured for NFS.

Parameter	Value
Name	my NFS protocol
Type	NFS
Transmission protocol	TCP/IP
Server address	100.100.100.100
Trusted	No
Local path	pc:
Server path	C:\robot_1
User ID	Robot1
Group ID	robot1

Example with FlexPendant

This example shows how to use the FlexPendant to make a backup to the remote PC. We assume that the configuration is done according to the example configuration shown above.

1. Tap **ABB** and select **Backup and Restore**.
2. Tap on **Backup Current System**.
3. Save the backup to pc:/Backup/Backup_20031008 (the path on the PC will be C:\robot_1\Backup\Backup_20031008).



en0300000504

Continues on next page

Example with RAPID code

This example shows how to open the file C:\robot_1\files\file1.doc on the remote PC from a RAPID program on the controller. We assume that the configuration is done according to the example configuration shown above.

```
Open "HOME:" \File:= "pc:/files/file1.doc", file;
```

8 Socket Messaging [616-1]

8.1. Introduction to Socket Messaging

Purpose

The purpose of Socket Messaging is to allow a RAPID programmer to transmit application data between computers, using the TCP/IP network protocol. A socket represents a general communication channel, independent of the network protocol being used.

Socket communication is a standard that has its origin in Berkeley Software Distribution Unix. Besides Unix, it is supported by, for example, Microsoft Windows. With Socket Messaging, a RAPID program on a robot controller can, for example, communicate with a C/C++ program on another computer.

What is included

The RobotWare option Socket Messaging gives you access to RAPID data types, instructions and functions for socket communication between computers.

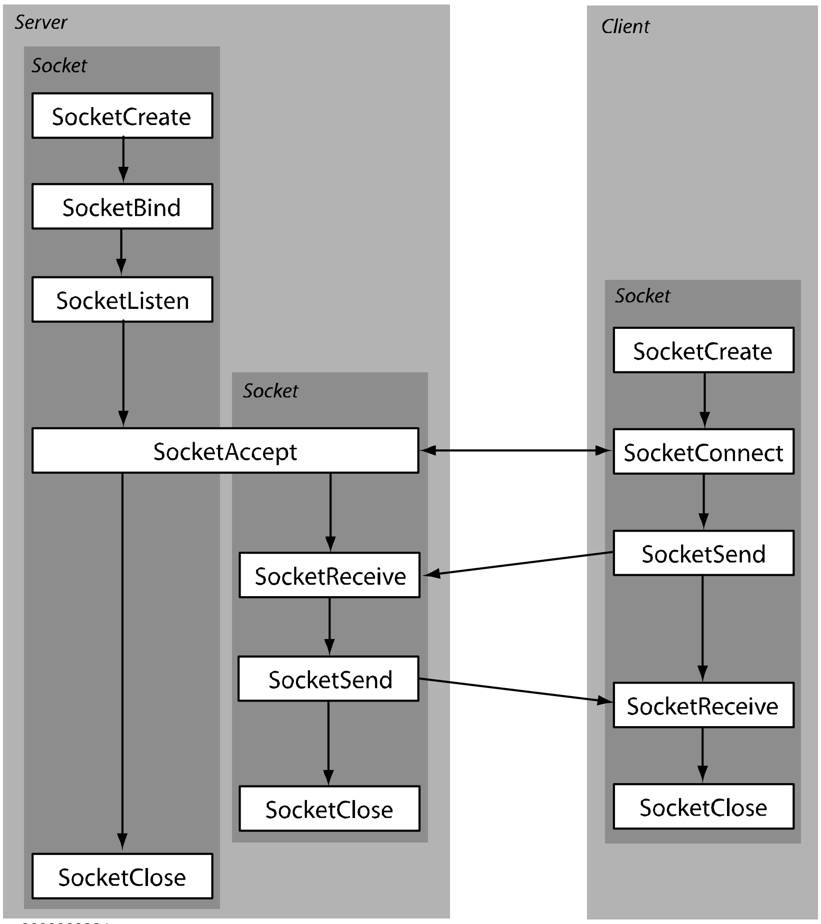
Basic approach

This is the general approach for using Socket Messaging. For a more detailed example of how this is done, see *Code examples on page 58*.

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.

8.2. 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).

8.3. Technical facts about Socket Messaging

Overview

When using RAPID functionality Socket Messaging to communicate with a client or server that is not a RAPID task, it can be useful to know how some of the implementation is done.

No string termination

When sending a data message, no string termination sign is sent in the message. The number of bytes sent is equal to the return value of the function `strlen(str)` in the programming language C.

Unintended merge of messages

If sending two messages with no delay between the sendings, the result can be that the second message is appended to the first. The result is one big message instead of two messages. To avoid this, use acknowledge messages from the receiver of the data, if the client/server is just receiving messages.

Non printable characters

If a client that is not a RAPID task needs to receive non printable characters (binary data) in a string from a RAPID task, this can be done by RAPID as shown in the example below.

```
SocketSend socket1 \Str:="\0D\0A";
```

For more information, see *Technical reference manual - RAPID kernel*, section *String literals*.

8.4. RAPID components

Data types

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

Data type	Description
socketdev	A socket device used to communicate with other computers on a network.
socketstatus	Can contain status information from a `socketdev` variable.

Instructions for client

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

Instruction	Description
SocketCreate	Creates a new socket and assigns it to a `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 use the same instructions as the client, except for `SocketConnect`. In addition, the server use the following instructions:

Instruction	Description
SocketBind	Binds the socket to a specified port number on the server. Used by the server to define on which port (on the server) to listen for a connection. The IP address defines a physical computer and the port defines a logical channel to a program on that computer.
SocketListen	Makes the computer act as a server and accept incoming connections. It will listen for a connection on the port specified by `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 the respective function in *Technical reference manual - RAPID Instructions, Functions and Data types*.

Function	Description
SocketGetStatus	Returns information about the last instruction performed on the socket (created, connected, bound, listening, closed). `SocketGetStatus` does not detect changes from outside RAPID (such as a broken connection).

8.5. Code examples

Example of client/server communication

This example shows program code for a client and a server, communicating with each other.

The server will write on the FlexPendant:

```
Client wrote - Hello server
Client wrote - Shutdown connection
```

The client will write on its FlexPendant:

```
Server wrote - Message acknowledged
Server wrote - Shutdown acknowledged
```

In this example, both the client and the server use RAPID programs. In reality, one of the programs would often be running on a PC (or similar computer) and be written in another program language.

Code example for client, contacting server with IP address 192.168.0.2:

```
! WaitTime to delay start of client.
! Server application should start first.
WaitTime 5;
VAR socketdev socket1;
VAR string received_string;
PROC main()
  SocketCreate socket1;
  SocketConnect socket1, "192.168.0.2", 1025;
  ! Communication
  SocketSend socket1 \Str:="Hello server";
  SocketReceive socket1 \Str:=received_string;
  TPWrite "Server wrote - " + received_string;
  received_string := "";
  ! Continue sending and receiving
  ...
  ! Shutdown the connection
  SocketSend socket1 \Str:="Shutdown connection";
  SocketReceive socket1 \Str:=received_string;
  TPWrite "Server wrote - " + received_string;
  SocketClose socket1;
ENDPROC
```


Continues on next page

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;
    ! 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
```


Continues on next page

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
```

9 RAPID Message Queue

9.1. Introduction to RAPID Message Queue

Purpose

The purpose of RAPID Message Queue is to communicate with another RAPID task or Robot Application Builder client.

Here are some examples of applications:

- Sending data between two RAPID tasks.
- Sending data between a RAPID task and a PC application.
- Sending data between a RAPID task and the FlexPendant.

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:

- FlexPendant Interface
- 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 67*.

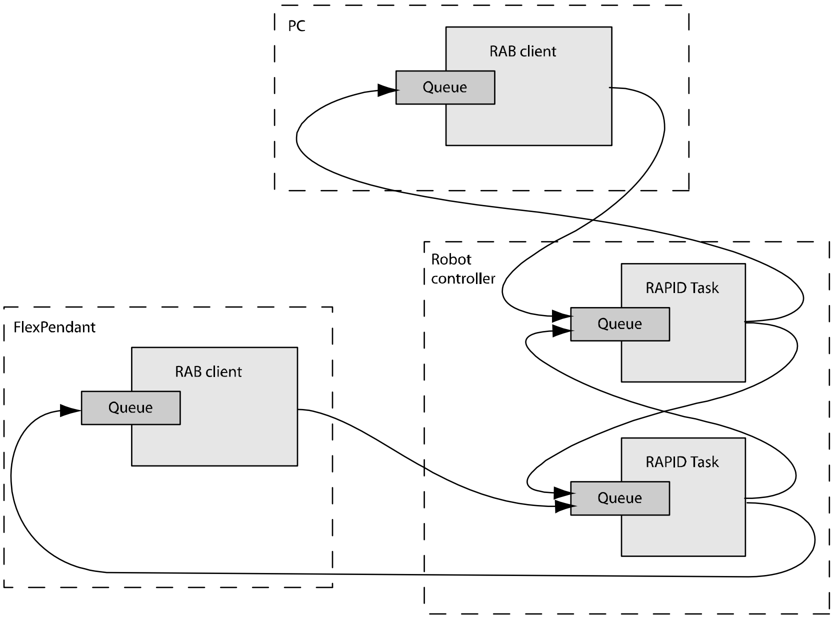
1. For *interrupt* mode: The receiver sets up a trap routine that reads a message and connects an interrupt so the trap routine is called when a new message appears.

   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.

9.2. 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 Robot Application Builder client

This manual only describes how to use RAPID Message Queue to make a RAPID task communicate with other RAPID tasks and Robot Application Builder clients. For information about how to set up the communication on a Robot Application Builder client, see *Application manual - Robot Application Builder*.

What can be sent in a message

The data in a message can be any data type in RAPID, except:

- non-value
- semi-value
- `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.

Continues on next page

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.

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""
"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.

Continues on next page

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 Robot Application Builder 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.

Related information

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

9.3. System parameters

About the system parameters

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

Type Task

These parameters belong to the type *Task* in the topic *Controller*.

Parameter	Description
RMQ Type	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*.
RMQ Mode	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 `RMQReadWait` instruction. Default value is *Interrupt*.
RMQ Max Message Size	The maximum data size, in bytes, for a message. The default value is 400. This value cannot be changed in RobotStudio or on the FlexPendant. The value can only be changed by editing the sys.cfg file. The maximum value is 3000.
RMQ Max No Of Messages	Maximum number of messages in queue. Default is 5. This value cannot be changed in RobotStudio or on the FlexPendant. The value can only be changed by editing the sys.cfg file. The maximum value is 10.

9.4. 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.

9.5. Code examples

Example with RMQSendMessage and RMQGetMessage

This is an example where the sender creates data (x and y value) and sends it to another task. The receiving task gets the message and extract the data to the variable named `data`.

Sender

```
MODULE SenderMod
  RECORD msgrec
    num x;
    num y;
  ENDRECORD

  PROC main()
    VAR rmqslot destinationSlot;
    VAR msgrec data;
    VAR robtarget p_current;

    ! Connect to queue in other task
    RMQFindSlot destinationSlot "RMQ_OtherTask";

    ! Perform cycle
    WHILE TRUE DO
      ...
      p_current := CRobT(\Tool:=tool1 \WObj:=wobj0);
      data.x := p_current.trans.x;
      data.y := p_current.trans.y;
      ! Send message
      RMQSendMessage destinationSlot, data;
      ...
    ENDWHILE
  ERROR
    IF ERRNO = ERR_RMQ_INVALID THEN
      WaitTime 1;
      ! Reconnect to queue in other task
      RMQFindSlot destinationSlot "RMQ_OtherTask";
      ! Avoid execution stop due to retry count exceed
      ResetRetryCount;
      RETRY;
    ELSIF ERRNO = ERR_RMQ_FULL THEN
      WaitTime 1;
      ! Avoid execution stop due to retry count exceed
      ResetRetryCount;
      RETRY;
    ENDIF
  ENDPROC
ENDMODULE
```


Continues on next page

Receiver

```
MODULE ReceiverMod
  RECORD msgrec
    num x;
    num y;
  ENDRECORD

  VAR intnum intnum_msgrec;
  VAR msgrec data;

  TRAP msghandler
    VAR rmqmessage message;
    VAR rmqheader header;

    RMQGetMessage message;
    RMQGetMsgHeader message \Header:=header;
    ! It is a good habit to always check data type and dimensions
    ! since it is possible to handle several data types in one trap
    IF header.datatype = "msgrec" AND header.ndim = 0 THEN
      RMQGetMsgData message, data;
      ! Handle message
      ...
    ELSE
      ! Fatal error since we received a message that we cannot
      ! handle. Can happen if we have set up the interrupts wrong
      ! or if we received an array with dimensions that we can
      ! not handle.
      Stop;
    ENDIF
  ERROR
    IF ERRNO = ERR_RMQ_NOMSG THEN
      ! A message has been lost, most probably a power fail has
      ! occurred.
      RETURN;
    ENDIF
  ENDTRAP

  PROC main()
    ! Setup interrupt
    CONNECT intnum_msgrec WITH msghandler;
    IRMQMessage data, intnum_msgrec;

    ! Perform cycle
    WHILE TRUE DO
      ...
    ENDWHILE
```


Continues on next page

```
      IDelete intnum_msgrec;
    ENDPROC
  ENDMODULE
```

Example with RMQSendWait

This is an example of a RAPID program that sends a message and wait for an answer before execution continues by getting the answer message.

```
MODULE SendAndReceiveMod
  VAR rmqslot destinationSlot;
  VAR rmqmessage recmsg;
  VAR string send_data := "How many units should be produced?";
  VAR num receive_data;

  PROC main()
    ! Connect to queue in other task
    RMQFindSlot destinationSlot "RMQ_OtherTask";

    ! Send message and wait for the answer
    RMQSendWait destinationSlot, send_data, recmsg, receive_data
          \Timeout:=30;

    ! Handle the received data
    RMQGetMsgData recmsg, receive_data;
    TPWrite "Units to produce: " \Num:=receive_data;

  ERROR
    IF ERRNO = ERR_RMQ_INVALID THEN
      WaitTime 1;
      ! Reconnect to queue in other task
      RMQFindSlot destinationSlot "RMQ_OtherTask";
      ! Avoid execution stop due to retry count exceed
      ResetRetryCount;
      RETRY;
    ELSIF ERRNO = ERR_RMQ_FULL THEN
      WaitTime 1;
      ! Avoid execution stop due to retry count exceed
      ResetRetryCount;
      RETRY;
    ELSEIF ERRNO = ERR_RMQ_TIMEOUT THEN
      ! Avoid execution stop due to retry count exceed
      ResetRetyCount;
      RETRY;
    ENDIF
  ENDPROC
ENDMODULE
```


Continues on next page

Example with RMQReadWait

This is an example of a RAPID program that waits for messages and handles them if the type is correct.

```
MODULE ReceiverMod
  RECORD msgrec
    num x;
    num y;
  ENDRECORD

  PROC main()
    VAR rmqmessage message;
    VAR rmqheader header;
    VAR msgrec data;

    ! Perform cycle
    WHILE TRUE DO
      RMQReadWait message;
      RMQGetMsgHeader message \Header:=header;
      ! It is a good habit to always check data type and
           dimensions
      IF header.datatype = "msgrec" AND header.ndim = 0 THEN
        RMQGetMsgData message, data;
        ! handle message
        ...
      ELSE
        ! Skip any message of unknown type.
      ENDIF
    ENDWHILE
  ERROR
    IF ERRNO = ERR_RMQ_TIMEOUT THEN
      ! No message was received, try again.
      ResetRetryCount;
      RETRY;
    ELSEIF ERRNO = ERR_RMQ_INVMSG
      ! A received message has been unpacked and found to have a
           corrupt header
    ELSEIF ERRNO = ERR_RMQ_MSGSIZE
      ! A received message sent as a package has been unpacked
           and found to be too large
      RETURN;
    ELSEIF ERRNO = ERR_RMQ_VALUE
      ! A received message has been unpacked and found to have a
           corrupt data
      RETURN;
    ENDIF
  ENDPROC
ENDMODULE
```

3HAC020435-001, Revision F, en

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