Rexroth
Bosch Group

Rexroth Rho 4
BAPS3 Programming Instruction

1070072178
Ausgabe 06

Software manual

Title Rexroth Rho 4
BAPS3 Programming Instruction

Type of Documentation Software manual

Document Typecode DOK-RHO*4*-BAPSI*SOFTH-PR06-EN-P

Purpose of Documentation The present manual informs about:

- The programming language BAPS3

Record of Revisions

Description	Release Date	Notes
DOK-RHO*4*-BAPSI*SOFTH-PR06-EN-P	10.2003	Valid as of VO07

Published by Bosch Rexroth AG
Postfach 11 62
D-64701 Erbach
Berliner Straße 25
D-64711 Erbach
Tel.: +49 (0) 60 62/78-0
Fax: +49 (0) 60 62/78-4 28
Abt.: BRC/ESH (KW)

Contents

Contents

Contents

Contents

Contents

1 Safety Instructions

Please read this manual before you startup the rho4.
Store this manual in a place to which all users have access at any time.

1.1 Intended use

This instruction manual presents a comprehensive set of instructions and information required for the standard operation of the described products. The described products are used for the purpose of operating with a robot control rho4.

The products described

- have been developed, manufactured, tested and documented in compliance with the safety standards. These products normally pose no danger to persons or property if they are used in accordance with the handling stipulations and safety notes prescribed for their configuration, mounting, and proper operation.
- comply with the requirements of
  - the EMC Directives (89/336/EEC, 93/68/EEC and 93/44/EEC)
  - the Low-Voltage Directive (73/23/EEC)
  - the harmonized standards EN 50081-2 and EN 50082-2
- are designed for operation in industrial environments, i.e.
  - no direct connection to public low-voltage power supply,
  - connection to the medium- or high-voltage system via a transformer.

  The following applies for application within a personal residence, in business areas, on retail premises or in a small-industry setting:
  - Installation in a control cabinet or housing with high shield attenuation.
  - Cables that exit the screened area must be provided with filtering or screening measures.
  - The user will be required to obtain a single operating license issued by the appropriate national authority or approval body. In Germany, this is the Federal Institute for Posts and Telecommunications, and/or its local branch offices.

☞ **This is a Class A device. In a residential area, this device may cause radio interference. In such case, the user may be required to introduce suitable countermeasures, and to bear the cost of the same.**

The faultless, safe functioning of the product requires proper transport, storage, erection and installation as well as careful operation.

Safety Instructions

1.2 Qualified personnel

The requirements as to qualified personnel depend on the qualification profiles described by ZVEI (central association of the electrical industry) and VDMA (association of German machine and plant builders) in:
Weiterbildung in der Automatisierungstechnik
edited by: ZVEI and VDMA
MaschinenbauVerlag
Postfach 71 08 64
D-60498 Frankfurt.

The present manual is designed for RC technicans. They need special knowledge on handling and programming robots.

Interventions in the hardware and software of our products, unless described otherwise in this manual, are reserved to specialized Rexroth personnel.

Tampering with the hardware or software, ignoring warning signs attached to the components, or non-compliance with the warning notes given in this manual may result in serious bodily injury or damage to property.

Only electrotechnicians as recognized under IEV 826-09-01 (modified) who are familiar with the contents of this manual may install and service the products described.

Such personnel are

- those who, being well trained and experienced in their field and familiar with the relevant norms, are able to analyze the jobs being carried out and recognize any hazards which may have arisen.
- those who have acquired the same amount of expert knowledge through years of experience that would normally be acquired through formal technical training.

With regard to the foregoing, please note our comprehensive range of training courses. Please visit our website at
http://www.boschrexroth.com
for the latest information concerning training courses, teachware and training systems. Personal information is available from our Didactic Center Erbach,
Telephone: (+49) (0) 60 62 78-600.

Safety Instructions

1.3 Safety markings on products

Warning of dangerous electrical voltage!

Warning of danger caused by batteries!

Electrostatically sensitive components!

Warning of hazardous light emissions (optical fiber cable emissions)!

Disconnect mains power before opening!

Lug for connecting PE conductor only!

Functional earthing or low-noise earth only!

Connection of shield conductor only

Safety Instructions

1.4 Safety instructions in this manual

DANGEROUS ELECTRICAL VOLTAGE
This symbol is used to warn of a **dangerous electrical voltage.** The failure to observe the instructions in this manual in whole or in part may result in **personal injury**.

DANGER
This symbol is used wherever insufficient or lacking compliance with instructions may result in **personal injury**.

CAUTION
This symbol is used wherever insufficient or lacking compliance with instructions may result in **damage to equipment or data files**.

☞ This symbol is used to draw the user's attention to special circumstances.

★ This symbol is used if user activities are required.

Safety Instructions

1.5 Safety instructions for the described product

DANGER
Danger of life through inadequate EMERGENCY-STOP devices!
EMERGENCY-STOP devices must be active and within reach in all system modes. Releasing an EMERGENCY-STOP device must not result in an uncontrolled restart of the system!
First check the EMERGENCY-STOP circuit, then switch the system on!

DANGER
Danger for persons and equipment!
Test every new program before starting up a system!

DANGER
Retrofits or modifications may adversely affect the safety of the products described!
The consequences may include severe injury, damage to equipment, or environmental hazards. Possible retrofits or modifications to the system using third-party equipment therefore have to be approved by Rexroth.

DANGER
Do not look directly into the LEDs in the optical fiber connection. Due to their high output, this may result in eye injuries.
When the inverter is switched on, do not look into the LED or the open end of a short connected lead.

DANGEROUS ELECTRICAL VOLTAGE
Unless described otherwise, maintenance works must be performed on inactive systems! The system must be protected against unauthorized or accidental reclosing.

Measuring or test activities on the live system are reserved to qualified electrical personnel!

Safety Instructions

CAUTION
Danger to the module!
Do not insert or remove the module while the controller is switched ON! This may destroy the module. Prior to inserting or removing the module, switch OFF or remove the power supply module of the controller, external power supply and signal voltage!

CAUTION
use only spare parts approved by Rexroth!

CAUTION
Danger to the module!
All ESD protection measures must be observed when using the module! Prevent electrostatic discharges!

The following protective measures must be observed for modules and components sensitive to electrostatic discharge (ESD)!

- Personnel responsible for storage, transport, and handling must have training in ESD protection.
- ESD-sensitive components must be stored and transported in the prescribed protective packaging.
- ESD-sensitive components may only be handled at special ESD-workplaces.
- Personnel, working surfaces, as well as all equipment and tools which may come into contact with ESD-sensitive components must have the same potential (e.g. by grounding).
- Wear an approved grounding bracelet. The grounding bracelet must be connected with the working surface through a cable with an integrated 1 MΩ resistor.
- ESD-sensitive components may by no means come into contact with chargeable objects, including most plastic materials.
- When ESD-sensitive components are installed in or removed from equipment, the equipment must be de-energized.

Safety Instructions

1.6 Documentation, software release and trademarks

Documentation

The present manual provides information about programming the rho4 with BAPS3.

Overview of available documentation	Part no.	
	German	English
Rho 4.0 Connectivity Manual	1070 072 364	1070 072 365
Rho 4.0 System description	1070 072 366	1070 072 367
Application IndraControl VEH 30	1070 170 330	1070 170 331
Rho 4.1/BT155, Rho 4.1/BT155T, Rho 4.1/BT205 Connectivity manual	1070 072 362	1070 072 363
Rho 4.1, Rho 4.1/IPC300 Connectivity manual	1070 072 360	1070 072 361
Control panels BF2xxT/BF3xxT, connection	1070 073 814	1070 073 824
Rho 4.1 System description	1070 072 434	1070 072 185
ROPS4/Online	1070 072 423	1070 072 180
BAPS plus	1070 072 422	1070 072 187
BAPS3 Short description	1070 072 412	1070 072 177
BAPS3 Programming manual	1070 072 413	1070 072 178
Control functions	1070 072 420	1070 072 179
Signal descriptions	1070 072 415	1070 072 182
Status messages and warnings	1070 072 417	1070 072 181
Machine parameters	1070 072 414	1070 072 175
PHG2000	1070 072 421	1070 072 183
DDE-Server 4	1070 072 433	1070 072 184
DLL-Library	1070 072 418	1070 072 176
Rho 4 available documentation on CD ROM	1070 086 145	1070 086 145

☞ **In this manual the floppy disk drive always uses drive letter A:, and the hard disk drive always uses drive letter C:.**

Special keys or key combinations are shown enclosed in pointed brackets:

- Named keys: e.g., <Enter>, <PgUp>, <Del>
- Key combinations (pressed simultaneously): e.g., <Ctrl> + <PgUp>

Safety Instructions

Release

☞ **This manual refers to the following versions:**
Hardware version: rho4
Software release: ROPS4

Trademarks

All trademarks of software installed on Rexroth products upon delivery are the property of the respective manufacturer.

Upon delivery, all installed software is copyright-protected. The software may only be reproduced with the approval of Rexroth or in accordance with the license agreement of the respective manufacturer.

MS-DOS® and Windows™ are registered trademarks of Microsoft Corporation.

PROFIBUS® is a registered trademark of the PROFIBUS Nutzerorganisation e.V. (user organization).

MOBY® is a registered trademark of Siemens AG.

AS-I® is a registered trademark of AS-International Association.

SERCOS interface™ is a registered trademark of Interessengemeinschaft SERCOS interface e.V. (Joint VDW/ZVEI Working Committee).

INTERBUS-S® is a registered trade mark of Phoenix Contact.

DeviceNet® is a registered trade mark (TM) of ODVA (Open DeviceNet Vendor Association, Inc.).

2 Program structure

2.1 General information

BAPS3 is an abbreviation of **B**ewegungs- and **A**blauf **P**rogrammier-**S**prache, Version **3**, which means Movement and Sequence Programming Language, version 3, and is a task-oriented programming language for programming the rho4 control family.

As a programming language for robot and handling systems, BAPS3 is an extensive but easy-to-learn language. It allows a quick and maintenance-friendly realization of user tasks.

The language instructions can be written in the corresponding national language, currently in German and English.

In this document, the general syntax of each statement is given before every detailed statement.

The following symbols are used for the purpose of description:

CAPITAL LETTERS	means part of the language element, i. e. a BAPS keyword, and must be written
{ }	may be optionally specified several times
[ ]	may be optionally specified once

Program structure

2.2 Mode of compiler operation

The BAPS compiler is integrated in both the operating system of the rho4 control and the programming system ROPS4.

The qll file contains the source text of BAPS3 programs. The BAPS compiler generates the following files from the statements contained in the *.qll source file.

ird file

This file contains the program code executed by the rho4 control and the memory area required for the variables used in the program. This file is generated only if the program has been compiled without errors.

☞ **Memory area is also reserved in this file for point variables which are not declared with DEF and to which a value is assigned in the program.**

pkt file

The memory area in this file is reserved for the point variables which are declared in the program with DEF or which are not declared and to which no value is assigned in the program.

sym file

This file contains information on the variable names used in the program and is always required for testing BAPS3 programs.

err file

This file contains the errors detected during compilation of the BAPS3 program in plain text.

Source files

Programs are stored in files which are stored in the main memory of the control or on a data medium of your programming systems. The program files are identified with names to permit the location of the correct program from among the large number of programs. These files are also referred to as source files and must be identified with the file label (extension) qll.

The program name and the name of the file in which the program is stored must be identical.

Program structure

In the rho4 control, a distinction is made between main programs and subroutines. Main programs are programs which exist as files and which can be started as rho4 BAPS user process. It is possible to call other main programs which exist in the control's main memory from within a main program. We then speak of external subroutines that must be declared correspondingly in the declaration part.

Internal subroutines are part of the main program in which they are defined and can be called only from within this program.

Program structure

2.3 Compiler statements

BAPS3 contains compiler statements. They are destined for the control of the compiler and reduce the extent of typing. The compiler statement always begins with two semicolons.

The following compiler statements exist:

;;INCLUDE name
;;PROCESS_KIND = PERMANENT
;;[kinematic_name .] INT = CIRCULAR | PTP | LINEAR
;;CONTROL= rho4
;;KINEMATICS: ({ integer_const = kinematic_variable | , })
;;KINEMATICS = kinematic_variable
;;kinematic_variable.(JC_NAMES | WC_NAMES) = { name || , }
;;SER_IO_STOP [+/–]
;;DEBUGINFO [+/–]

The compiler statement for the JC names and for the target control must be placed in front of the first source symbol, i. e. before the program declaration.

2.3.1 Kinematic definition

The control can handle several kinematics simultaneously.

If more than one kinematic (robots, feeding units, etc.) is to be controlled in a BAPS program, they have to be defined first.

Syntax:

```
;;KINEMATICS:(1=SR6,2=kin2) ;KINEMATICS:(kinematic number=kinematic name)
```

It is now possible to distinguish the kinematics in the program and it is evident to which kinematic the instructions refer.

The kinematic definition must be made according to a control definition (if available) and before the PROGRAM statement.

Example:

```
;;CONTROL=rho4

;;KINEMATICS:(1=sr6,2=feeder)

PROGRAM main
```

Program structure

2.3.2 WC name definition

The world coordinate points contain the components for the position, the orientation and possibly for the belt coordinate(s). The component names can be defined.

Syntax:

```
;;WC_NAMES=x_k,y_k,z_k,a_k ;WC_NAMES=WC_name,...
```

If several coordinates have to be controlled by a BAPS program, they have to be preceded by the kinematic name.

Syntax:

```
;;kinematic name.WC_NAMES=x_k,y_k,z_k,a_k ;kinematic name.WC_NAMES=WC name,...
```

Example for WC name declarations

```
;;automat.WC_NAMES=x_k,y_k,z_k,u_k,v_k,w_k,b_k

;;sr6.WC_NAMES=x_k,y_k,z_k,a_k
```

☞ **The WC name declaration must be in one line and must not be interrupted by a carriage return.**

2.3.3 JC name definition

The joint coordinate points contain the components for the individual axes and possibly for the belt coordinate(s). The JC names can be defined.

Syntax:

```
;;JC_NAMES=a_1,a_2,a_3,a_4 ;JC_NAMES=JC name,...
```

If several coordinates have to be controlled by a BAPS program, they have to be preceded by the kinematic name.

Syntax:

```
;;JC_NAMES=a_1,a_2,a_3,a_4 ;kinematic name.JC_NAMES=JC name, ...
```

Example for JC name declarations:

```
;;automat.JC_NAMES=a_1,a_2,a_3,a_4,a_5,a_6,bnd

;;sr6.JC_NAMES=a_1,a_2,a_3,a_4
```

☞ **The JC name declaration must be in one line and must not be interrupted by a carriage return.**

2.3.4 Kinematic-related statements and data

If several kinematics are controlled by a BAPS program, a distinction has to be made in the program as to which kinematic the statements or data refer. This applies to the

- point variables
- movement instructions
- tool statements
- workspace limitations

The point variables are preceded by the kinematic name.

Syntax:

```
sr6.POINT:corner      ;kinematicname.POINT

kinematicname.POINT   ;kinematicname.JC_POINT
```

If no kinematic is indicated, the currently preselected kinematic is assigned.

The movement instructions can contain the kinematic indication additionally or the preselected kinematic is controlled, see chapter 8.

In the reference point statement, the kinematic name has to be entered immediately after the REF_PNT key value.

Syntax:

```
REF_PNT sr(1,2,3,4) ;REF_PNT kinematicname(axisnumber)
```

The same applies analog to the TOOL and LIMIT_OFF, LIMIT_MIN and LIMIT_MAX statement.

Syntax:

```
TOOL automat innergripper ;TOOL kinematicname toolname

LIMIT_OFF sr6             ;LIMIT_OFF kinematic name
                          ;LIMIT_MIN kinematic name(parameter)
                          ;LIMIT_MAX kinematicname(parameter)
```

Program structure

2.3.5 Inclusion of files

With the compiler statement ;;INCLUDE 'filename', parts of source programs can be included into the program.

Example:

```
;;INCLUDE baps                                ;baps.qll, file for the
                                              ;automatic inclusion of the
                                              ;declaration part during compilation

INPUT REAL:  1=grip_contact,4=measheight      ;declaration inputs

INPUT:       11=gateswitch_1,15=li_barrier

OUTPUT REAL: 1=print,2=met_unit               ;declaration outputs

OUTPUT:      7=alarm
```

The baps.qll file contains e. g. the declarations of your inputs and outputs. These are defined with regard to their

- file type
- channel number
- variable name of the signals

2.3.6 Selectable extension within the include statement

From the compiler version 3.0 on, the extension of include files can be freely selected. If only the file name is indicated, the extension .qll will be used for reasons of compatibility. If a file without extension is to be included, the file name has to be ended with a dot, see example for include statement.

Syntax:

```
;;INCLUDE filename     ;compiler statement INCLUDE
```

In the above syntax applies:

- filename: name of the file to be included during compilation.

Example for include statement:

```
;;INCLUDE head.inc

PROGRAM demo

;;INCLUDE constant      ;The file constant.qll is included

;;INCLUDE types.INC     ;The file types.INC is included

;;INCLUDE variable.     ;The file variable. (without extension) is included

BEGIN
```

Program structure

```
PROGRAM_END
```

2.3.7 Process kind

By means of the compiler statement ;;PROCESS_KIND = PERMANENT, a process can be declared as permanent.

Example:

```
;;PROCESS_KIND=PERMANENT

PROGRAM tmp_cont

...

PROGRAM_END
```

This means that this process is not ended by a reset or by automatic/manual switching.

☞ **Permanent processes must not contain any movement instructions. The compiler statement must precede the PROGRAM statement.**

2.3.8 Debug information

With the compiler statement ;;DEBUGINFO–, the creation of debug information to the Irdata code can be switched off for the comfortable test operation. This is normally only reasonable for fully tested application programs.

Since the information is then missing in the Irdata code, the Irdata code is shorter and processed faster, but a test with the debugging system is then no longer possible.

Syntax:

```
;;DEBUGINFO- ;Generation of debug information is switched off
```

Further restrictions with DEBUGINFO–

- No line numbers will be shown on the process display, mode 10.3 and mode 7.3.2.
- Program errors possibly occurred will also be displayed without line numbers.

With ;;DEBUGINFO+, the generation of debug information is switched on again.

Program structure

2.3.9 Compiler statement SER_IO_STOP

With this compiler statement it is possible to avoid the abort of a user program in case of an interface error.

The statement is only permitted at the start of the program. It applies to the whole program and may only appear once. If this compiler statement is not used, an interface error will lead to a program break. Explaining example see section 7.13.

Syntax:

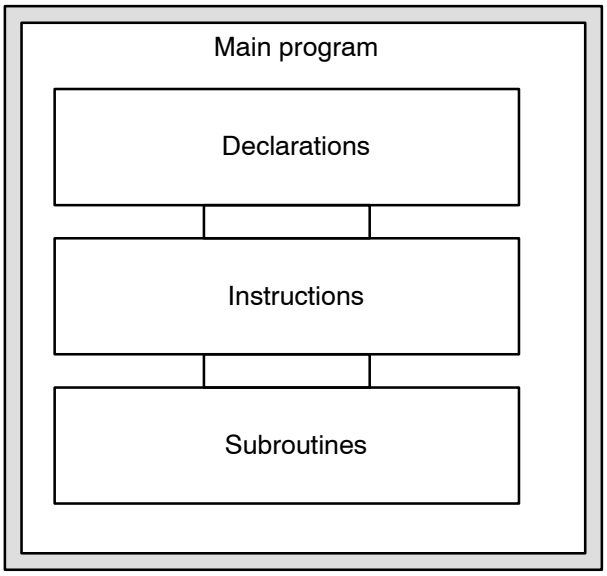
```
;;SER_IO_STOP- ;No program break in case of error

;;SER_IO_STOP+ ;Program break in case of error

;;SER_IO_STOP  ;Program break in case of error
```

Program structure

2.4 Main program structure

Each main program consists of

- declaration part
- statement part and optionally
- subroutine declaration(s)



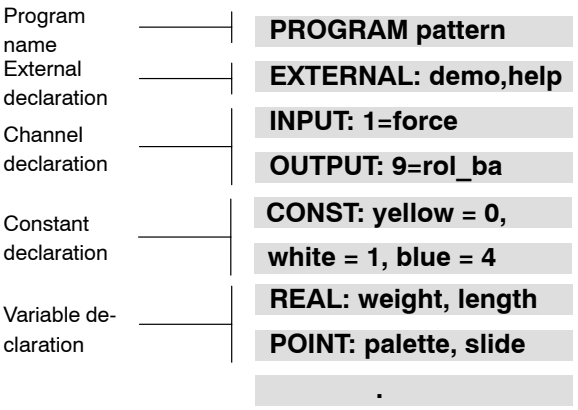
2.4.1 Declaration part

The declaration part is at the beginning of the main program. In the declaration part, the names are declared which occur in the main program.

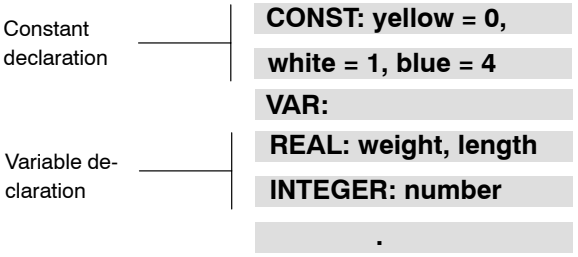
This applies to:

- program head, the name of the main program
- external declaration, the names of the external main programs called in the main program
- channel declaration, the names of the input and output channels used in the program
- variable and constant declaration, the names of the variables and constants which occur in the program

Program structure



The declaration of variables can optionally be introduced by the keyword VAR.



The declaration part has to be separated from the statement part by the keyword BEGIN.

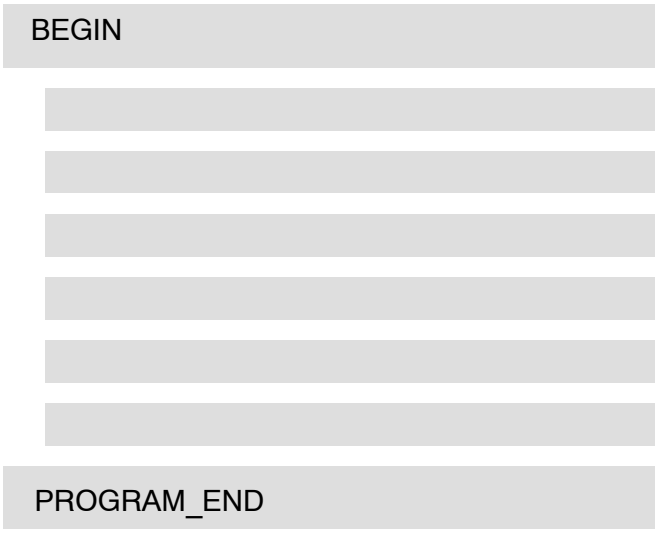
☞ **Undeclared variables are considered to be variables of the type POINT or JC_POINT.**

2.4.2 Statement part

In the statement part, the statements to be carried out are programmed. These are for example:

- movement statements
- decelerations and halt
- main program calls
- subroutine calls
- program part repetitions
- program jumps
- arithmetical operations
- function calls

Program structure



The statement part is placed between the keywords BEGIN and PROGRAM_END.

2.4.3 Subroutine declaration

At the end of the main program, the subroutines are listed, if provided.

Program structure

2.5 Program declaration

A main program is identified at the start by the BAPS word PROGRAM and its program name.

The program name consists of a maximum of eight characters. Letters, digits and underlines are permitted.

☞ **The first character must be a letter. Upper-case and lower-case letters are deemed equivalent.**

The program end is identified by the BAPS word PROGRAM_END or SUB_END, if subroutines are listed.

☞ **The program name and the file name must be identical.**

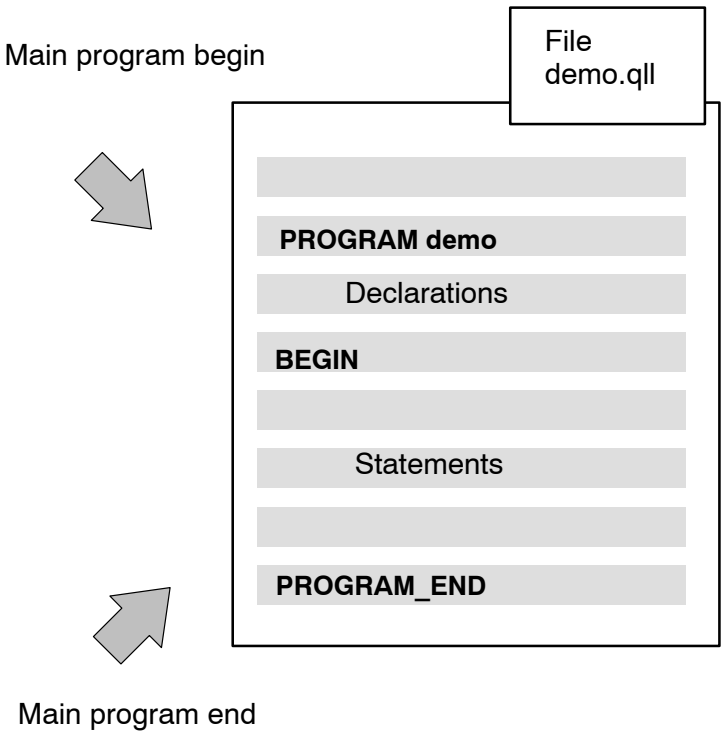
Example:

A program is to be given the name ’demo’

```
PROGRAM demo
```

End of the program ’demo’

```
PROGRAM_END
```



Program structure

2.6 Main program call in the main program

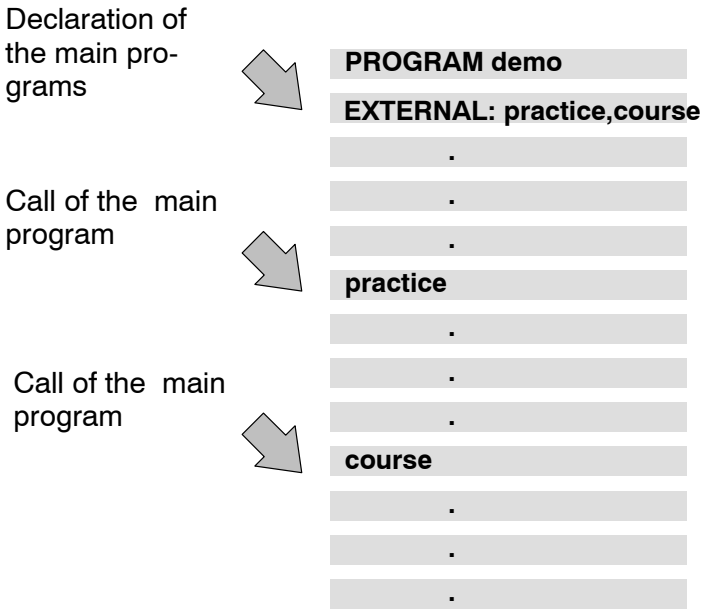
A main program can consist of several external individual main programs.

External declaration: The external main programs must be declared with EXTERNAL at the beginning of the declaration part and be available in the control as ird file.

External main programs can be optionally provided with transfer parameters. The number, order and data types must agree with the declaration upon parameter transfer. All variables are permitted as parameters of the types array and channels.

External main programs with transfer parameters cannot be started as independent programs but only by a program call from a higher-order main program.

It is then sufficient to specify to declared program name in order to call external programs in the active main program.



Program structure

☞ **The main program and the external program to be called are compiled independently of each other. No check of the transferred parameters with respect to agreement with the declaration in the external main program is thus possible at the time of compilation. This is performed during the program run. The number, types, order and nature (VALUE or addressing) of the transfer parameters must correspond to the declaration of the called external main program.**

Programming

The program names of the external main programs are declared by the statement EXTERNAL:.

Example:

```
EXTERNAL: drill,course
```

External declaration with parameter transfer

```
EXTERNAL: withpar(VALUE INTEGER: number)
```

The main program must be declared correspondingly

```
PROGRAM withpar(VALUE INTEGER: l)
```

Main program call in the main program

The control executes the active main program up to the external program call.

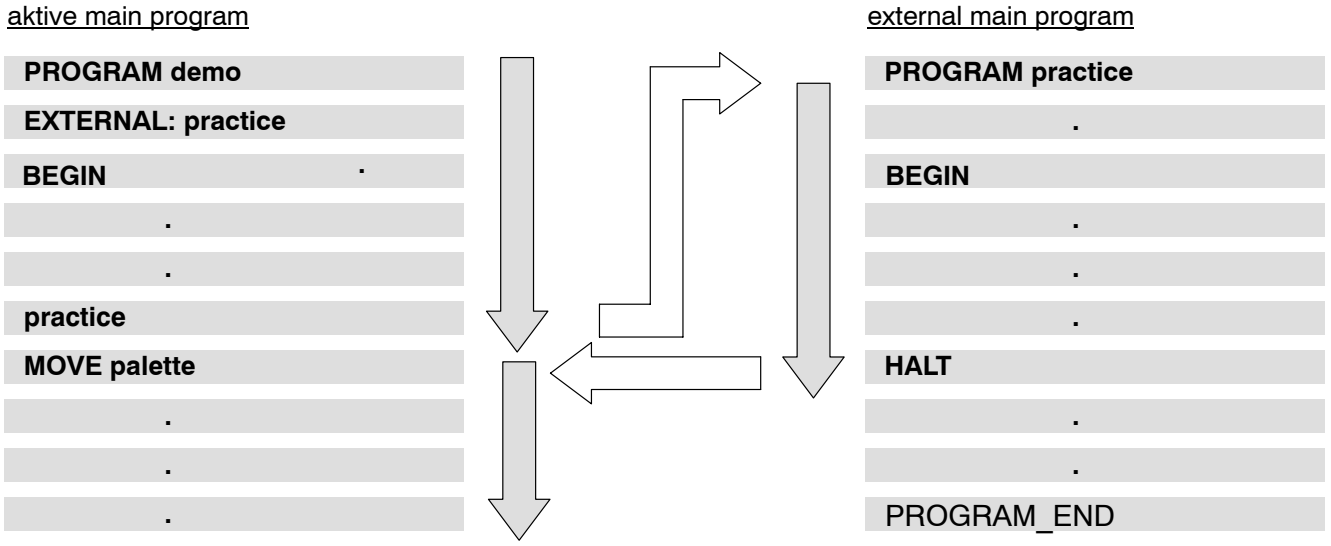
This if followed by a jump to the start of the program drill.

The program drill is executed up to the HALT statement.

HALT results in a return to the main program demo.

The control continues the program run with the statement following the call.

Program structure



Program structure

2.7 Subroutine declaration

If the same work steps have to be performed at different points in the program, it is possible to combine these steps to subroutines. Use of subroutine techniques saves on memory space and also increases the clarity of your program. Variables which are defined in the main program, i. e. global variables, can also be processed in the subroutine. Variables which are declared in the subroutine, i. e. local variables, can only be processed in the subroutine. A transfer to the main program does not take place. The subroutine declarations are located after the main program after the PROGRAM_END statement.

Identification

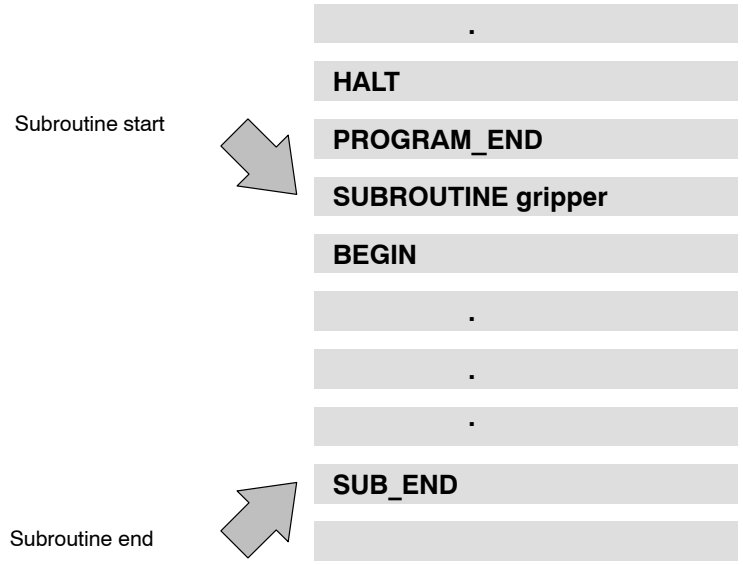
A subroutine is identified at the start by the word SUBROUTINE and the subroutine name. The subroutine name may consist of a maximum of twelve characters. Letters and numbers are permitted.

☞ **The first character must be a letter. Upper-case and lower-case letters are deemed equivalent.**

The subroutine is ended by the BAPS keyword SUB_END.

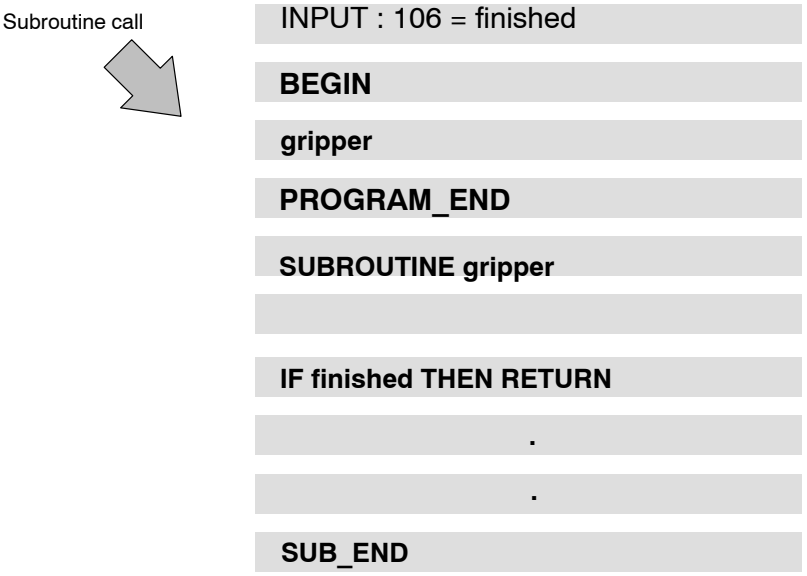
Programming

A subroutine contains statements for the gripper and is to be given the namen ’gripper’:



Program structure

☞ **RETURN can be used several times within a subroutine, e. g. with program jumps and conditional statements.**



If the subroutine return is recognizable from the program structure, e. g. at the subroutine end, the compiler generates the instruction RETURN automatically.

Subroutine call

It is sufficient to specify the declared subroutine name for the subroutine call, in the example ’gripper’.

Program structure

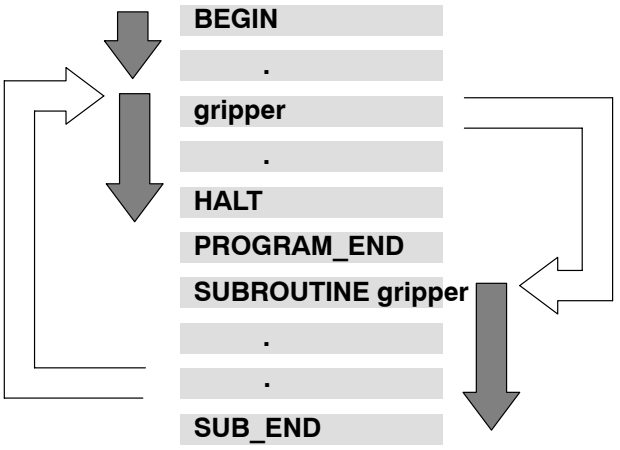
2.8 Program run

The control executes the main program up to a subroutine call.

This is followed by a jump to the start of the called subroutine. The subroutine is executed up to RETURN.

The instruction RETURN results in a return to the main program.

The control continues the program run with the statement following the subroutine call.



It is possible to transfer variables or values with a subroutine call. The variables must be correspondingly declared in the subroutine declaration for this purpose.

Example:

```
gripper (1.5,6.0)                              ;value assignment

SUBROUTINE grip(VALUE REAL: force1,force2) ;in the subroutine call, the variable
                                           ;force1 is assigned the value 1.5 and
                                           ;variable force2 the value 6.0
```

The value is transferred if the declaration is made with the preceding BAPS instruction VALUE, otherwise the address of the variable is transferred, i. e. the variable must be declared in the main program.

If the declaration is preceded by VALUE, the calling program transfers information to the called subroutine. However, the called subroutine does not return any information in this way.

If the address is transferred, on the other hand, the assignments in the subroutine also act on this variable after return to the calling program.

Program structure

Example:

```
;;INCLUDE head             ;Include compiler statements, such as control type,
                           ;DEBUGINFO+ etc. from the file head into the program

program demo               ;Test program for subroutine with parameter transfer

REAL: x                    ;Variable declaration

BEGIN

   x=5

   WRITE PHG,x

   anix(X)                 ;Call of the SUBROUTINE anix

   WRITE PHG,x

   awas(X)                 ;Call of the SUBROUTINE awas

   WRITE PHG,x

   HALT

PROGRAM_END

SUBROUTINE awas(REAL: aw)  ;Subroutine declaration awas. Transfer of the value and
                           ;the variable. The subroutine offsets the variable and
                           ;then returns it to the main program. The subroutine
                           ;outputs the number series 5, 10, 5, 10, 10.

BEGIN

   aw=aw*2

   WRITE PHG,aw

   RETURN

SUB_END

SUBROUTINE anix(VALUE REAL: aw)  ;Subroutine declaration anix. Transfer of value,
                                 ;the variable x is unchanged. The subroutine
                                 ;processes the variable. The variable in the main
                                 ;program remains unchanged after leaving the
                                 ;subroutine.

BEGIN

   aw=aw*2

   WRITE PHG,aw

   RETURN

SUB_END
```

Program structure

Nesting

Additional main program calls and subroutine calls can be programmed within called main programs or subroutines. In these cases, we speak of ’nesting’.

Program examples for subroutine nesting:

- A call of the subroutine ’stacker’ is programmed in the main program.
- In the subroutine ’stacker’ a further subroutine call is programmed, the subroutine ’gripper’.
- The control executes the main program up to the call ’stacker’.
- A jump then takes place to the subroutine ’stacker’.
- The control executes the subroutine ’stacker’ up to the call ’gripper’.
- A jump then takes place to the next subroutine ’gripper’.
- The subroutine ’gripper’ is executed completely in the example shown here.

The control jumps back to the subroutine ’stacker’ after the instruction RETURN, continues the program run up to RETURN and then finally jumps back to the main program.

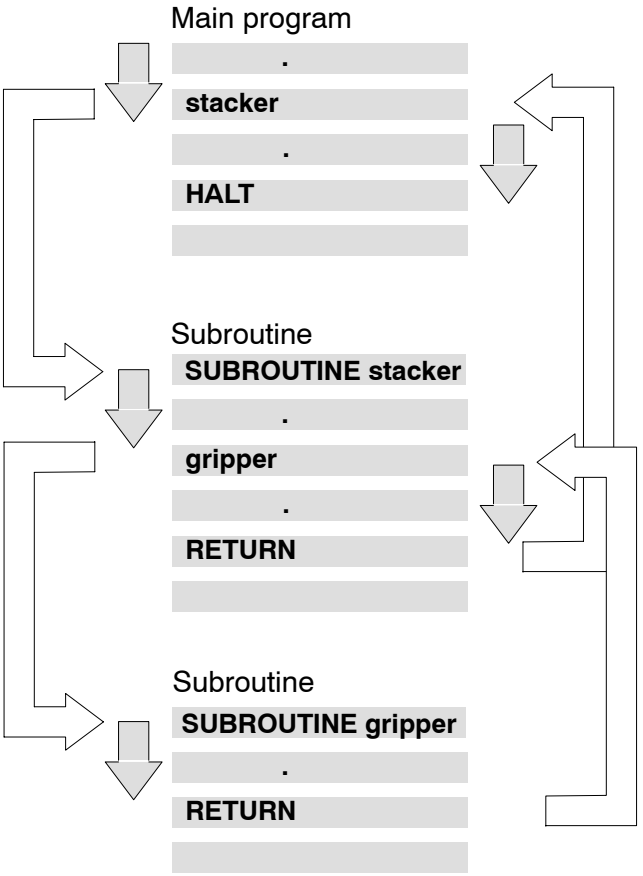
☞ **When programs and subroutines are nested, it must be ensured that no endless loops are created.**

Any nesting depth is possible. The depth is limited only by the available memory place.

☞ **The memory size can be defined by the machine parameter P16, see manual ’Machine parameters’.**

Program structure

Example 1



Program structure

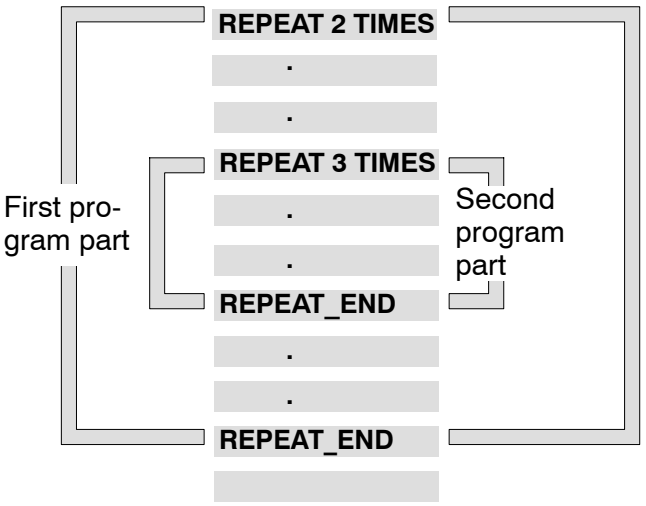
Example 2

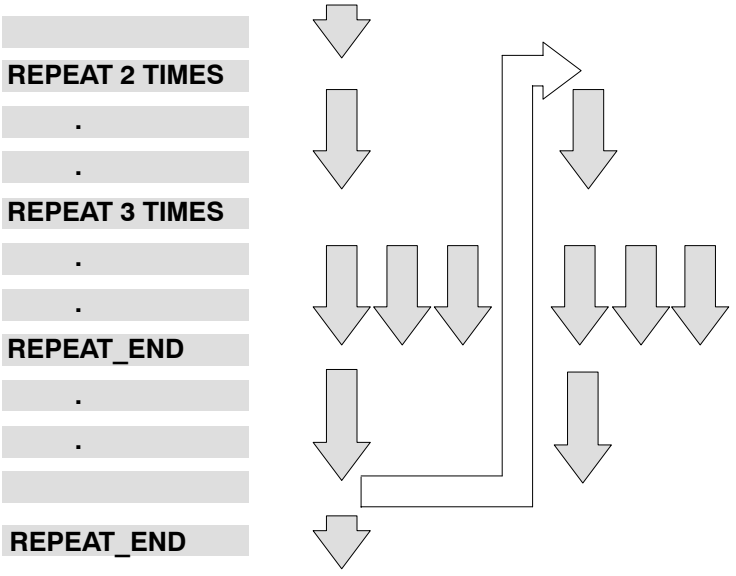
Program part repetitions can also be nested; a second repetition is programmed within a program part repetition.

The program is executed as follows:

- The control executes the program part once up to the start of the second program part.
- The second program part is repeated three times.
- The control then continues the program run up to the end of the first program part; the first program part has thus been executed once.
- The control then jumps back to the start of the first program part for the second run.
- The whole sequence is repeated a second time.

Any nesting depth is possible. The depth is limited only by the available memory place.





Program structure

Notes:

Constants

3 Constants

3.1 Constant declaration

Within a program, numeric values, e.g. the number of repetitions, character sequences or characters can be indicated in form of constants.

The value of a constant can also be specified by a computing rule. The constants are to be defined with the constant declaration before the variables of a program.

Example of a constant declaration:

```
CONST: red=1,yellow=2*red,green=2*yellow,lines=4,
                    columns=20,limit_value=2.5*7.25      ;Constant declaration

INTEGER: color, l_index, c_index                         ;Variable declaration

REAL: meas_value

ARRAY[lines .. columns] CHAR: phg_display

IF meas_value>limit_value THEN ....                      ;Program part
```

Constants

3.2 Standard constants

In BAPS3 standard constants are available. These constants are contained in the language scope and need therefore not be declared.

CLS

```
WRITE PHG,CLS  ;By the output of constant CLS the PHG display is cleared.
```

VERSION

```
IF VERSION <3.00 THEN
WRITE PHG, 'old version of compiler'     ;Via the constant VERSION, the compiler
                                         ;version can be determined. The constant
                                         ;is of the type REAL.
```

Variables

4 Variables

Numeric values, e. g. coordinate values for positions or the number of repetitions, can be replaced by variables in a program.

A variable reserves a memory location under its name.

A numeric value can be assigned to this memory location any number of times. The control stores the last-assigned numeric value in each case.

If a variable name occurs during the program run, the control replaces the variable by the value stored under its name at this point of time.

Every variable has a name. Different variables must have different names. The names should be chosen so that it is possible to recognize the meaning of these variables wherever this is possible, e. g. position designations: instead of ’b1’, it is better to use ’bore_1’ or arithmetic variables: instead of ’i’ , it is better to use ’counter’ etc.

The name consists of a maximum of 12 characters.

It is permitted to use letters, numbers and the ’underline’. Upper-case and lower-case letters are deemed equivalent.

☞ **The first character must be a letter, blanks are not permitted. The point variables of the data type JC_POINT form an exception. These variables must start with the character @.**

Variables

4.1 Data types

The data type of a variable determines its value range and the permitted assignment and arithmetic operations. The permitted operations are described in the section 'Value assignment'.

A distinction must be made for data types between simple data types and composed or structured data types.

Simple data types
- INTEGER
- REAL
- BINARY
- CHAR

Structured data types
- POINT
- JC_POINT
- BELT
- TEXT
- ARRAY
- WC_FRAME
- SEMAPHORE
- FILE

A structured data type consists of two or more simple data types.

4.1.1 Simple data types

INTEGER

Only integral values, i. e. numbers without decimal point, positive or negative, must be assigned to variables of the type INTEGER.

Value range -2^{31} to $+(2^{31}-1)$

REAL

Only real numbers, i. e. numbers with decimal point, positive or negative, must be assigned to variables of the type REAL.

Value range: approx. -10^{37} to -10^{-38}, real zero, 10^{-38} to 10^{37}

☞ **Value inputs for variables of the type REAL must be made with a decimal point and not with a comma.**

Variables

BINARY

Only the digits 0 or 1 may be assigned to variables of the type BINARY.

☞ **No integral arithmetic operations may be performed with variables of the type BINARY.**

The digits 0 and 1 do not represent any values in the conventional sense. They describe two defined states, e. g.:

Variable designation	0	1
switch	off	on
question	no	yes
bowl	empty	full
filled	false	true

Variables of the type BINARY are signals of the binary input and output channels.

CHAR

Only ASCII characters in accordance with DIN 66003 may be asigned to variables of the type CHAR.

4.1.2 Structured data types

POINT

Only positions in world coordinates may be assigned to variables of the type POINT.

The individual coordinate values of a position of the type POINT must be of the type REAL.

The kinematic must be specified for point variable declarations where appropriate, e. g. 'robot_1.corner'.

If no kinematic is specified, the valid kinematic is the first-specified kinematic in the kinematic declaration, kinematic number one or the kinematic last selected by the compiler statement ;;KINEMATICS, e. g. ;;KINEMATICS = robot_1.

JC_POINT

Only positions in joint coordinates may be assigned to variables of the type JC_POINT.

Variables

The individual coordinate values of a position of the type JC_POINT must be of the type REAL.

☞ **Variable names of the type JC_POINT must start with the character @.**

Variables of the types POINT and JC_POINT are also called point variables.

The kinematic must be specified in point variable declarations where appropriate, e. g. robot_2.@corner. The number of components depends on the number of axes of the specified kinematic.

If no kinematic is specified, the valid kinematic is the first-specified kinematic in the kinematic declaration, kinematic number one or the kinematic last selected by the compiler statement ;;KINEMATICS, e. g. ;;KINEMATICS = robot_2.

TEXT

Only texts, consisting of up to 80 ASCII characters, may be assigned to variables of the type TEXT. The individual characters can be addressed directly like array elements with an index.

```
TEXT: char_string                ;Declaration of text variables

char_string[1]='A'               ;Assignment of individual components

char_string[2]='B'

char_string='This is ASCII text' ;Assignment of a text
```

ARRAY

It is possible to combine variables of the same type in an array. These variables all have the same name and differ only with respect to the index. For this reason, these variables are also called indexed variables.

Syntax:

```
ARRAY[([±]integer_constant)..([±]integer_constant)] <Typ>: variablename
```

Example:

```
ARRAY[-10..10] INTEGER: variablename
```

SEMAPHORE

Syntax:

```
SEMAPHORE: sema_name
```

Variables

Variables of the type SEMAPHORE are used as parameters in the EXCLUSIVE statement.

FILE

Syntax:

```
FILE: cad_dat
```

A file name is defined with data type FILE. This is used as a parameter for access with WRITE or READ.

4.1.3 User-defined types

Through the expansion of the declaration part of a BAPS program by the type definition part it is possible for the user to declare in BAPS his own types.

It is thus possible to asign new names to any data type. This applies to both the standard types, e. g. BINARY, INTEGER, and the structured types formed from them, e. g. arrays. This leads to a clearer structure of BAPS programs.

Example:

```
PROGRAM types                                   ;type definition part

TYPE:
tmatrix=ARRAY[0..7] ARRAY[0..7] INTEGER         ;2-dimensional array under a new name
                                                ;new name for INTEGER (not used
                                                ;here)

myinteger=INTEGER                               ;Variable definition part

VAR:
ARRAY [0..7] ARRAY [0..7] INTEGER: matrixone
                          tmatrix: matrixtwo

myinteger:integervar

BEGIN

   initold(matrixone)

   initnew(matrixtwo)

PROGRAM_END

SUBROUTINE initold(ARRAY[0..7] ARRAY[0..7] INTEGER: matrix) ;long writing variable
                                                            ;matrix initialization

BEGIN

SUB_END
```

Variables

```
SUBROUTINE initnew(tmatrix: matrix)                    ;short, clear writing
                                                       ;variable matrix
                                                       ;initialization

BEGIN

SUB_END
```

Record types

A further improvement of the clarity and structure of programs is achieved by using record types.

A record is an accumulation of one or several variables, the so-called components, possibly of different data types, which are for a comfortable handling combined under one single name. In PASCAL these record types are also known as ’record’, in C they are called ’struct’.

The traditional example of a record is an entry into a file for payroll accounting: A ’staff member’ is written as a block of attributes, such as for example name, address, social security number, salary category etc. Some of these attributes can be record types themselves. A name has several components, first and last names, exactly as an address or a rate class.

Records are useful to organize complicated data, especially in large programs. In many situations they permit the processing of a group of associated variables as a unit and not separately.

The name appearing in the type declaration on the left side of the assignment character, before the reserved word RECORD, represents the complete record and can in the following be used as an abbreviation for the detailed declaration.

Variables mentioned in a record are called components in BAPS. A component can have the same name as an ordinary variable without creating a conflict. They can also be distinguished from the context.

A component is exclusively accessed through the name of the record. The component name is in this case separated from the record name by a dot. If the component represents a record itself, its components are equally separated by a dot, see example of record types.

When using record variables within the WRITE or READ statement, the following has to be observed:

- General dat files, e. g. PHG, V24_1, FILE:dana, etc. record variables can only be read or written by components.
- Binary files, e. g. PLC, WIN_1, BNR_FILE: BinFile etc. record variables can be read or written completely or by components.

Variables

Example record types:

```
PROGRAM records

Type:                                    ;Type definition part

tname=RECORD

          TEXT: firstname,lastname,title

      RECORD_END

taddress=RECORD

            TEXT:    street

            INTEGER: ZIP_code

            TEXT:    town

         RECORD_END

tstaffmember=RECORD

                tname:    name

                taddress: address

                TEXT:     ss_number

                BINARY:   salary

             RECORD_END

tstaffmember: worker                 ;Variable declaration part declaration
                                     ;of a staff member

BEGIN

   ..

   worker.ss_nummer=0815             ;Access to the component

   worker.name.firstname='Rainer'    ;Access to the component of a

   worker.name.title='Dr.'           ;component

PROGRAM_END
```

Variables

Example record statements

Syntax:

```
BEGIN
   {statements}
END
```

By means of the record statement, several statements can be combined:

```
BEGIN
   statement
   statement
END
```

A statement sequence can be at the place of a statement:

```
INPUT: 1=valve_1,2=valve_2
IF ready THEN
   BEGIN
      valve_1=1
   END
ELSE
   BEGIN
      valve_2=1
   END
```

Variables

4.2 Declaration of variables

Syntax:

```
[DEF] Type: [channel no.=]name{ ,[channel no.=]name}
```

DEF is possible only for point variables (POINT and JC_POINT) and channel number only for the types INPUT, OUTPUT or BELT.

The control must know which values or characters a variable may possess before execution of an statement with variables.

For this reason, every variable used in the program must be declared, i. e. it is necessary to define the data type of the variable.

☞ **Variables of the type POINT and JC_POINT need not be declared, i. e. the compiler interprets undeclared variables as point variables and reserves memory for these in der point file pkt.**

☞ **JC_point variables start with the character @. They are assigned to the last kinematic set by a compiler statement.**

☞ **BAPS3 standard variables (e. g. VFACTOR, AFACTOR) must not be used as user variables.**

Programming

The declaration consists of the data type and the variable name to be assigned. The data type is separated from the variable name by a colon. Several variable names of the same type are separated by commas.

Example:

```
INTEGER: counter

REAL:    divisor,xvalue,yvalue

TEXT:    message_1,message_2
```

Variables

4.3 Global variables

Global variables permit the simple data exchange between several independent BAPS user programs (processes). The basic idea is to combine programs working with the same global variable in a program group. Within this group, a program can export data, while the other programs of this group may only import these data. On the control, as many of these program groups as desired can be created (only limited by the available memory space).

Program example:

```
PROGRAM exp_var               ;Exporting program
                              ;Declaration part in the exporting program
GLOBAL DEF POINT: start_pos   ;Global data

GLOBAL INTEGER: index         ;Global variables are declared by adopting the

GLOBAL SEMAPHORE: write_prot  ;keyword GLOBAL in the type declaration
                              ;of these variables. The data range for these
                              ;variables is reserved in the IRD file.
                              ;Teachpoints, e. g. identified by the keyword
                              ;DEF, are stored in the PNT file.

REAL: mvalue                  ;Local data

BEGIN                         ;BAPS statements

   EXCLUSIVE write_prot

      start_pos=POS

      index=5

   EXCLUSIVE_END

   .
   .                          ;Additional statements
.

PROGRAM_END
```

Variables

```
PROGRAM imp_var                ;Importing program

                               ;Declaration part in the importing program
                               ;Global data
EXTERNAL exp_var: start_pos    ;The variables can be accessed from other BAPS

EXTERNAL exp_var: index        ;programs, if the variables are declared with

EXTERNAL exp_var: write_prot   ;EXTERNAL und the name of the exporting
                               ;program

POINT: end_pos                 ;Local data

BEGIN                          ;BAPS statements

   EXCLUSIVE write_prot

      REPEAT index TIMES

         MOVE TO start_pos

         MOVE TO end_pos

      REPEAT_END

   EXCLUSIVE_END

PROGRAM_END
```

Restrictions

When using global data, the following restrictions have to be taken into account:

- The exporting program of a program group must be compiled before the importing programs of this group. This is necessary because of the type verification by the compiler. The user must ensure that each importing program is of a more recent date than the exporting program. If this is not the case, an error message or a warning will be put out at the program start.
- In a program it is not possible to import and export data at the same time.
- Only data from one program may be imported.
- Global data may only be exported or imported in the declaration part of the main program but not in subroutines.
- To ensure a transfer of a complete data block without a break, the access to the variables must be protected by the EXCLUSIVE statement. The consistency, i. e. the validity of data of simple standard types, such as BINARY, INTEGER, REAL and CHAR, is ensured by the operating system.

Variables

4.4 Point variables

Point variables are composed, structured data types which consist of components.

The components are the coordinates or the axes of the point variables.

In addition to the complete value assignment, you can also assign to the point variables values by components, e. g. corner.z_k = height

You specify the component designation with the compiler statements

;;JC_NAMES = JC_name, ... and

;;kinematicname.JC_NAMES = JC_name, ...

as well as

;;WC_NAMES = WC_name, ... and

;;kinematicname.WC_NAMES = WC_name, ...

Example declaration of point variables:

```
DEF POINT:                  corner

ARRAY[1..4] POINT:          point_array

JC_POINT:                   @between_pnt

DEF ARRAY[1..8] JC_POINT:   @mk_pnt_array

S8.POINT:                   startpoint

DEF robot1.JC_POINT:        @placing
```

Variables

4.4.1 Identification of point variables

Names of point variables of the type POINT start with a letter.

Names of point variables of the type JC_POINT start with a special character @.

Permitted operations with point variables

	JC_POINT	POINT	REAL
JC_POINT	+	+ JC (...)	*
	–	– JC (...)	/
POINT	+ WC (...)	+	*
	– WC (...)	–	/
REAL	*	*	+
			–
			*
			/

4.4.2 Points and point file pkt

While all other variables have to be declared, point variables need not be declared explicitely.

The BAPS3.0 compiler interprets all undeclared variables as POINT or JC_POINT and reserves the corresponding space for them in the point file.

All points declared with DEF and all points to which no value has been assigned at any location in the program, are stored in a point file with the identification pnt.

The points that are transmitted as parameters to subroutines, Spc_Fct and rho-Fct, do not appear in the point file. That applies to the transfer of individual point components (e.g. 'Pos.X_C').

With the function 'Define' you can assign to these points direct values by means of 'Teach-In' or the input of values.

☞ **Values can only be assigned to points from the pnt file during the program run, i. e. overwrite the content of the point file, if they have been declared with DEF declared.**

Variables

☞ **Undeclared points to which, however, values have been assigned in the program, are not stored in the point file but in the ird file. No values can be assigned to these points with the 'Define' function.**

4.4.3 Complete value assignment

In case of complete value assignments, values will be assigned to all coordinates of the point variable. Only point variables of one data type may be within one assignment.

☞ **Exception: Mixed operations with the standard functions JC or WC.**

Assignment of numeric values

Example:

```
position=(50,0,100,0,15,10)

@edge=@(45,5.8,70,10,5.8,0)
```

4.4.4 Assignment of variables for individual components

Example:

```
hole=(xvalue,yvalue,zvalue,0,0,0)

@seat=@(50,95.8,height,40,38,0)

hole.z_c=height
```

The variables xvalue, yvalue, zvalue and height are of the type REAL.

Assignment by addition and subtraction

The total of the point variables 'position' and the constant has been assigned to the point variables 'shelf'.

Constants consist of the enumeration of their coordinates put into brakkets. Constants can contrary to variables not be changed during the program run.

In case of addition and subtraction, the individual components are added resp. subtracted each.

Example:

```
shelf=position+(10,-30,100,5,0,0,0)
```

Variables

Assignment with multiplication and division

By the multiplication and division with numeric values or variables of the type REAL, you can change any coordinate value of point variables or constants.

Multiplication:

```
pos_1=(10,10,10,0,0,0)

pos_2=pos_1*2
```

Each individual component of pos_1 is doubled by the multiplication with 2 and is allocated to the new point variable pos_2; pos_2 then has the coordinates: pos_2 = (20, 20, 20, 0, 0, 0).

Division:

```
div=4.0

pos_3=pos_2/div
```

pos_3 then has the coordinates: pos_3 = (5, 5, 5, 0, 0, 0)

Mixed operation with point variables

With the standard functions JC and WC you can execute mixed operations with point variables of the type POINT and JC_POINT.

If the result of the arithmetic operations is to be assigned to a point variable of the type POINT, the operation has to be made in world coordinates.

If the result of the arithmetic operations is to be assigned to a point variable of the type JC_POINT, the operation has to be made in joint coordinates.

Example:

```
@p3=@(0,100,0,0)

@p1=JC(p2)+@p3
```

The control converts the world coordinates of point p2 into joint coordinates, adding them to the coordinates of @p3. The result is assigned to the point variable @p1.

Reading of the actual position

With the standard point variables POS (world coordinates) and @POS (joint coordinates) you can assign point variables to the current position of the robot each during the program run.

```
act_pos=POS

@joint_pos=@POS
```

Variables

Programming

The standard point variables are located at the right-hand side of the assignment. It is also possible to separate the assignment by components.

```
cpos.c_3=POS.c_3

xvalue=POS.c_1
```

Assignment by components

In case of point variables you can assign new values to individual coordinates.

Vice versa it is also possible to assign coordinate values of point variables to variables of the type REAL.

The JC names are defined in the machine parameters or by compiler statement.

In the following examples, the coordinates of a position in world coordinates are named k1, k2, k3, etc. (c = coordinates), the coordinates of a position in joint coordinates a2, a3, etc. (a = axis).

The coordinate name is annexed to the name of the point variable with a dot.

Example:

```
position.c3=100     ;Value 100 is assigned to the coordinate c3

zvalue=anchor.c3    ;The third coordinate value of the point variable anchor
                    ;is assigned to the variable zvalue

@pal_pos.a4=radius ;The value of the variable radius of the type REAL is
                    ;assigned to coordinate a4, the position described in
                    ;joint coordinates
```

Variables

4.5 Text variables

Within a program it is possible to assign texts to text variables.

Example:

```
TEXT: message,note ;The variable message and note are of the type TEXT
```

Text assignment

The text to be assigned must be placed within inverted commas and may not have more than a maximal of 80 characters. The text must be in one line.

Example:

```
message='gripper is defective'

note='change pallet'
```

Use of variables

Text variables can be put out in the program onto an output channel, e. g. PHG, with the BAPS instructions WRITE and be read with READ.

Example:

```
WRITE PHG,note

READ PHG,entry
```

☞ **The variable itself must not be placed within inverted comma, such as e. g. WRITE 'note'. The control puts out otherwise the word 'note' instead of the declared text.**

Variables

4.6 Array variables

Variables of the same type can be combined to arrays.

Arrays consist of a freely selectable number of array places being designated with ascending numbers. By specifying a number (index), a variable can be assigned to each array place.

Variables of one array have all the same name and differ only in their index. The index is identical with the number of the assigned array place.

All types of variables can be stored in arrays.

Array declaration

The array declaration consists of:

- declaration instruction ARRAY
- array limits (place numbers)
- declaration of array variables

The array limits are indicated in angular brackets and formed from the first place number (first index = lower limit) and the last place number (last index = upper limit).

Examples:

```
ARRAY[1..9] POINT: placing_pos ;onedimensional arrays

ARRAY[0..10] TEXT: message

ARRAY[-10..10] INTEGER: number_pos


ARRAY[1..9] ARRAY[5..10] POINT: picking_pos ;multidimensional array
```

The lower limit has to be separated from the upper limit by two dots, e. g. [3..8]

The upper limit of the array must not be smaller than the lower limit.

The index is of the type INTEGER.

Example:

```
ARRAY[1..5] POINT: hole ;Declaration of an array with 5 places for the
                        ;point variable hole, the first index is to be 1
```

Variables

Value assignment

Each array variable is addressed by its name and index put in angular brackets.

For the index you can also insert a variable or an expression of the type INTEGER. A value assignment is then possible via the REPEAT statement or through the other program loops.

Example:

```
ARRAY[-5..0] INTEGER: number

number[-5]=1           ;The variable number on the array place –5 has the value 1,

number[-4]=2           ;the variable number on the array place –4 has the value 2,

number[-3]=4           ;etc.

number[-2]=8

number[-1]=16

number[0]=32

                           ;Example REPEAT statement
ARRAY[-5..0] INTEGER: number ;Array with 6 places

INTEGER: i,k               ;Index i and value k

i=-5                       ;First index

k=1                        ;First value

REPEAT 6 TIMES

   number[i]=k             ;Value assignment

   i=i+1                   ;Index increase

   k=k*2                   ;Value change
                           ;Any assignment

REPEAT_END
```

Variables

```
                              ;Example program loop
ARRAY[-5..0] INTEGER: number ;Array declaration

INTEGER: i,k                  ;Index i and value k

i=-5                          ;First index

k=1                           ;First value

mark:                         ;Jump mark

number[i]=k                   ;Value assignment

i=i+1                         ;Index increase

k=k*2                         ;Value change
                              ;Any assignment

IF i<=0 THEN JUMP mark        ;Conditional assignment for program loop
```

Determination of pallet positions:

```
;;INCLUDE head                          ;Required compiler statements
                                        ;are included

PROGRAM palpos                          ;Determination of pallet positions

INTEGER: line,column,c,l,k              ;Declarations

ARRAY[1..12] POINT: palpos              ;The pallet has four lines and three
                                        ;columns, i. e. 12 positions

BEGIN                                   ;Program start

   dx=(30,0,0,0,0)                      ;The spacing of the positions to each other

   dy=(0,20,0,0,0)                      ;is specified in the incremental dimension
                                        ;under dx and dy

   line=4

   column=3

   c=0

   l=0

   k=0

   mark_1:

      k=k+1                             ;statements, point assignments

      palpos[k]=pos+c*dx+l*dy           ;The position pos is a teach point,
                                        ;i. e. it is defined by being approached
                                        ;and stored, it is the first position
                                        ;palpos[1]. It is therefore placed in the
                                        ;initialization part c=0 and l=0
```

Variables

```
      c=c+1                              ;Increase of column number

   IF c<column THEN JUMP mark_1          ;The next position palpos[2] is placed in
                ELSE c=0                 ;the adjacent column. The column number c
                                         ;must thus be increased.
                                         ;At the same time, the column number c
                                         ;must not exceed the total number of
                                         ;columns. If c remains smaller than
                                         ;column, the control jumps to the
                                         ;jump mark mark_1, if the index k is
                                         ;increased by 1 and assigns the value
                                         ;pos+1*dx+0*dy to the position palpos[2].
                                         ;If c exceeds the column, the control
                                         ;assigns the value zero to variable c and
                                         ;increases the line number.

   l=l+1                                 ;Increase of line number

   IF l<line THEN JUMP mark_1           ;The increase of the line number is analog
                                        ;to the increase of the column number. In
                                        ;the IF-THEN statement the ELSE
                                        ;statement is missing; the control
                                        ;continues with the move statement of the
                                        ;condition l < line is not met

   ;;INT=LINEAR

   V=1000

   AFACTOR=9.999

   k=0

   REPEAT 12 TIMES

      k=k+1

      MOVE TO palpops[k]                ;Move statements

      MOVE_REL with V=36 EXACT (0,0,-20,0,0)

      WAIT 2

      MOVE_REL with V=59 EXACT (0,0,20,0,0)

   REPEAT_END

   MOVE_REL CIRCULAR((-50,-50,100,0,0),(-100,-100,0,0,0))

   HALT

PROGRAM_END
```

Variables

4.7 Channels

BAPS3 permits reading or writing of any digital or analog inputs or outputs present in the hardware configuration.

The respective input or output is addressed by specifying a channel number in the declaration of input or output variables.

The following channel numbers are available for rho4

Channel number	**Type and meaning**
1 to 199	BINARY inputs/outputs
201 to 216	REAL inputs/outputs
401 to 416	INTEGER inputs/outputs
501 to 516	Belt channels

4.7.1 Channel declaration

In the channel declaration, the data type BINARY, INTEGER or REAL and the variable name of the signal to be transferred are assigned to a channel number. It is necessary to define whether input or output signals are involved.

Example: Signals of the type BINARY

```
INPUT:

   1=gate_switch1,

   2=met_unit,

   5=li_barrier

OUTPUT:

   7=alarm
```

Please refer to the examples under item 4.7.3.

Variables

4.7.2 Data types

Depending on your control version, user channels are available to you by which you can transfer data of the following types.

BINARY: Interrogation and setting to state 0 or 1.

The control possesses 199 binary inputs and outputs.

INTEGER: Interrogation and setting integral numeric values in the range between 0 and 255. The control treats these numeric values as data of the type INTEGER.

The control possesses 16 inputs and outputs of the type INTEGER. Please also refer to the manuals Machine parameters, Status messages and warnings.

REAL: Interrogation and setting to analog voltage values. The control treats these voltages as data of the type REAL.

BELT: Belt channels serve the purpose of synchronization with conveyor belts or acquisition of values by means of standard position measuring systems.

Belt channels are (only) inputs of the type REAL and may be located only on the right side of an assignment.

Any measuring system input of the rho4 can be used as a hardware input. The parameterization takes place via machine parameter 501.

4.7.3 Programming

The individual channel assignments must be separated by a comma. There must be no comma after the last assignment.

If the data type is not specified, the control automatically assumes BINARY.

Example: Signals of the type INTEGER

```
INPUT INTEGER:

   401=grip_force,

   403=meas_height

OUTPUT INTEGER:

   401=pressure
```

Variables

Example: Signals of the type REAL

```
INPUT REAL:

   201=torque,

   206=force

OUTPUT REAL:

   203=speed
```

Interrogation of channels and signals

The interrogation and evaluation of the channels or their assigned names takes place in conditions. Only 1 interrogation is possible with a MOVE UNTIL instruction.

Example:

```
WAIT UNTIL gate_switch=1

IF grip_force>=26 THEN...

WAIT UNTIL meas_height>=212

MOVE LINEAR UNTIL meas_height>=200 TO pos        ; ok
```

Example:

```
MOVE UNTIL grip_force=100 AND meas_height=200    ; wrong

TO pallet
```

Wrong is:
MOVE UNTIL grip_force=100 AND meas_height=200 TO pallet

The compiler would interpret this instruction as follows:
MOVE UNTIL grip_force= ((100 AND meas_height) =200) TO pallet

It is not necessary to specify '=1' when interrogating binary signals for 1. The control then automatically interrogates for 1.

Example:

```
IF met_unit THEN ...
```

☞ **Output signals cannot be interrogated. Interrogation is only possible for input signals.**

Variables

Setting of signals

Signals are set in value assignments, see section 6.2.1.

Example:

```
print=75
```

☞ **Input signals cannot be set. Setting is possible only for output signals.**

☞ **Binary signals can be combined with AND, OR, NOT.**

Example:

```
IF NOT gate_switch1 AND li_barrier THEN alarm=1
```

Exceptions are the WAIT UNTIL condition and MOVE UNTIL condition. No combinations are permitted here.

Variables

Notes:

Program control

5 Program control

5.1 WAIT statement

Syntax:

```
WAIT expression
```

The WAIT statement allows programming delays and interruptions in the program execution.

Dwell time

A time can be specified directly if the robot is to dwell at a position for a specific time.

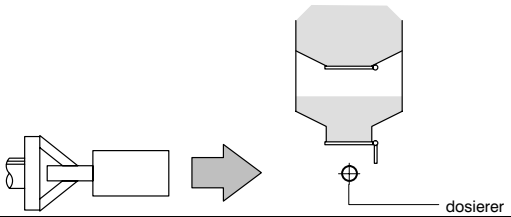
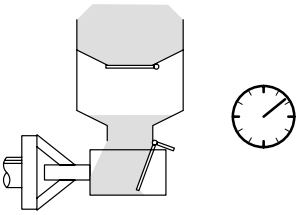
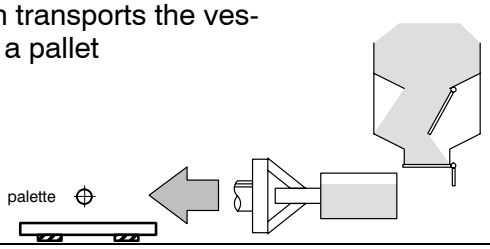
Dwell time for WAIT

Unit of measure:	Second
Input range:	0.01 to 3200 s

Programming: Time input takes place at a numeric value following the statement WAIT, e. g. WAIT 8.5

Program control

Program example: The robot transports a vessel to the metering unit to have it filled.

Syntax	**Description**
MOVE TO met_unit	
WAIT 8.5	The filling time is approx. 8.5 seconds 
MOVE TO pallet	It then transports the vessel to a pallet 

Waiting until condition occurs

Syntax:

```
WAIT UNTIL variable rel_operator expression [max_time=expression [ERROR
statement]]
```

Rel_operator: =, < >, <=, >=, <, >

If the robot is to wait at a position for a condition to occur, the condition can be specified together with the WAIT statement.

Program execution will then be interrupted, until the condition is satisfied.

☞ **The conditions can be set only by means of input channel variables, also see section 4.7.3.**

Programming: the condition is appended to the WAIT statement with the word UNTIL, e. g.:

```
WAIT UNTIL pal_empty=1
```

Program control

☞ **If the condition consists of several input channels, the WAIT statement must be divided into several steps.**

Example:

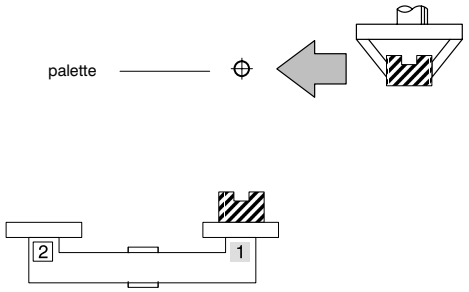
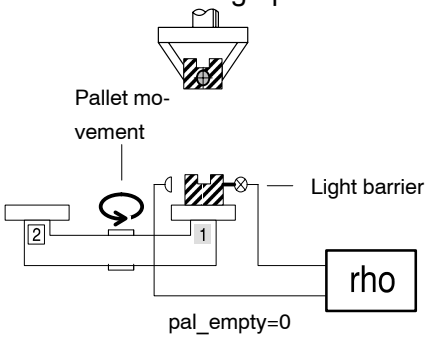
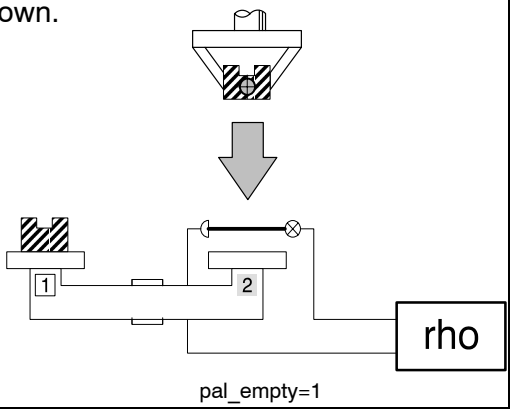
```
WAIT UNTIL signal=1

WAIT UNTIL light=1

MOVE TO pallet
```

What would be wrong:
WAIT UNTIL signal=1 AND light=1

The compiler would interpret this instruction as follows:
WAITE UNTIL signal= ((1 AND light) =1)

Program control

Program example: The robot transports a workpiece to a pallet changer.

Syntax	**Description**
MOVE TO pallet	
WAIT UNTIL pal_empty=1	It waits until the pallet is empty. The pallet changer then puts the signal pal_empty = 1 when an empty pallet has arrived at the change position. 
put_down	The deposit operation is programmed in a sub-routine put_down. 

Program control

Maximum wait time

Designation	max_time
Unit of measure	Second
Input range	0.5 to 32000

A maximum wait time can be defined In conjunction with a WAIT condition.

Program execution will then be interrupted, until the condition is satisfied or the specified maximum time is exceeded.

An error statement can be programmed with the maximum wait time. The control executes the error statement if the maximum wait time is exceeded.

Programming: The maximum wait time is specified with the BAPS instruction MAX_TIME. This is programmed after the wait condition.

Example:

```
WAIT UNTIL sig=1 max_time=60
```

The error statement is programmed after max_time.

Example:

```
WAIT UNTIL sig=1 max_time=60 ERROR PAUSE
```

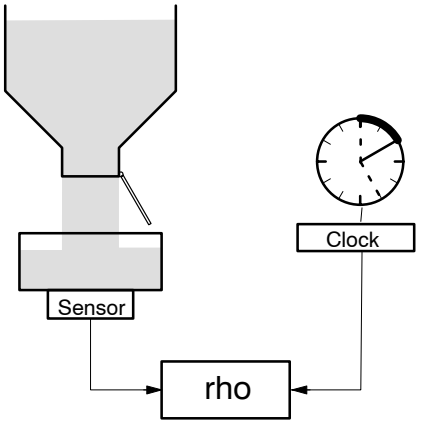
Sequence example

A container is filled with a liquid. A sensor measures the weight of the container. The sensor outputs the signal 'weight = 1'. The filling system then closes the valve and the robot transports the container away. The average filling time is approx. 25 s.

The container may remain under the filling device for a maximum of 45 s in order to ensure that the production sequence is not put at risk.

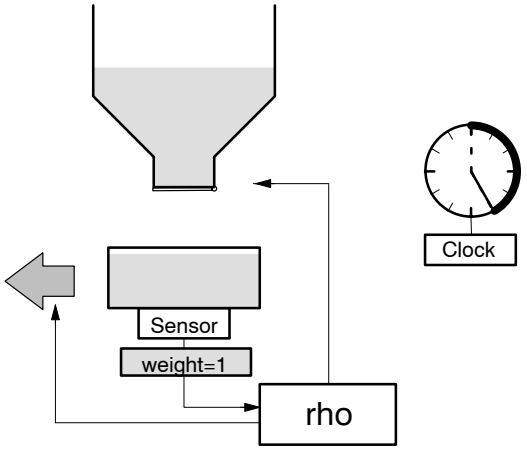
Syntax	Description
WAIT UNTIL weight=1 max_time=45 ERROR JUMP f_end . . f_end:	 Error processing

Program control



Case 1

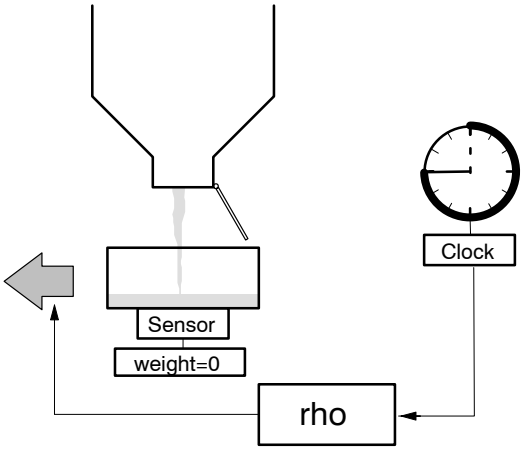
- The container is completely filled. The sensor issues the signal ’weight = 1’ to the rho.
- The robot transports the container away.
- The max_time has not been reached.



Case 2

- The filling system ist empty, and the container is not completely filled.
- The max_time is reached. The program is continued at f_end. The program f_end instructs the robot to isolate the partially filled container.

Program control



Program control

5.2 PAUSE statement

The program execution can be stopped with the PAUSE statement.

It is then necessary to issue the external start signal to continue the program run, see manual Status messages and warnings.

Programming: The PAUSE statement consists of the BAPS instruction PAUSE.

It is recommended to program a text output or to set an output before the PAUSE instruction in order to inform the operator about the program flow.

5.3 HALT statement

The HALT statement ends the execution of an statement string in the main program.

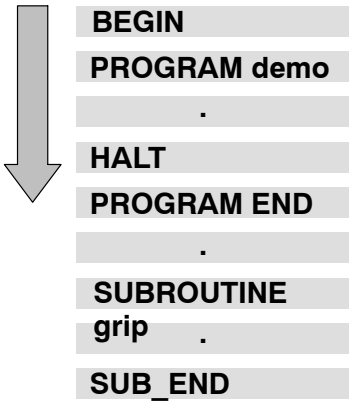
The control recognizes that the program has been terminated during the program run by way of the HALT instruction.

☞ **In case of called external subroutines, HALT results in a return to the calling active main program.**

Programming

The HALT instruction is entered before the subroutine declaration or before the program end. In the case of branched programs, the instruction is entered several times within a program.

If the program halt is evident from the program structure, e. g. in the case of the BAPS statement PROGRAM_END, the control generates the HALT instruction automatically during compilation.



Program control

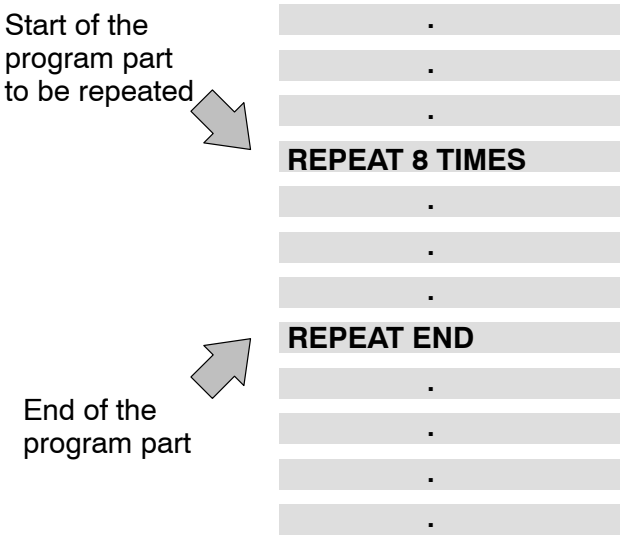
5.4 Repeat statement

Syntax:

```
REPEAT [expression TIMES]

   statement(s)

REPEAT_END
```

A program part can be executed several times with a repeat statement.

In this case, we speak of a program part repetition.



The program part is identified at the start by the repeat statement REPEAT number TIMES.

Numbers, variables or expressions of the type INTEGER, see section 4.1, can be entered for the number of repetitions.

The loop is not executed if number=0 or negative.

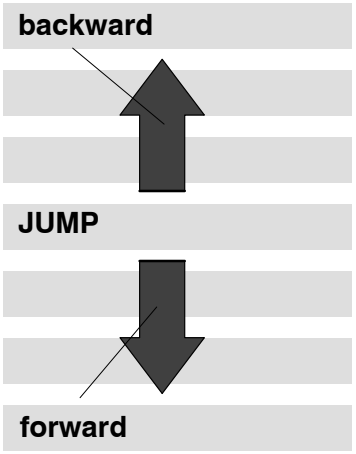
The program part is ended by the BAPS instruction REPEAT_END.

Programming: A certain program part is to be repeated eight times, i. e. the BAPS statement is REPEAT 8 TIMES. The program part is ended with REPEAT_END.

Program control

5.5 Jump statement

Labels can be set in main programs and subroutines to which it is possible to jump with a jump statement. Forward and back jumps are possible.



The jump statement consists of the BAPS instruction JUMP and the name of the set jump label. The labels (jump destinations) are identified with names.

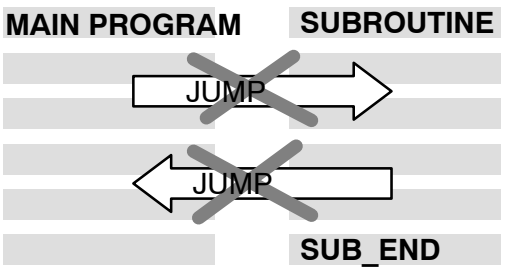
The name consists of a maximum of twelve characters.

☞ **The first character must be a letter. As special character only the underline '_' is permitted. Upper-case and lower-case letters are equivalent.**

Setting of a jump label must be identified with a colon ':' in order to distinguish it from subroutine calls.

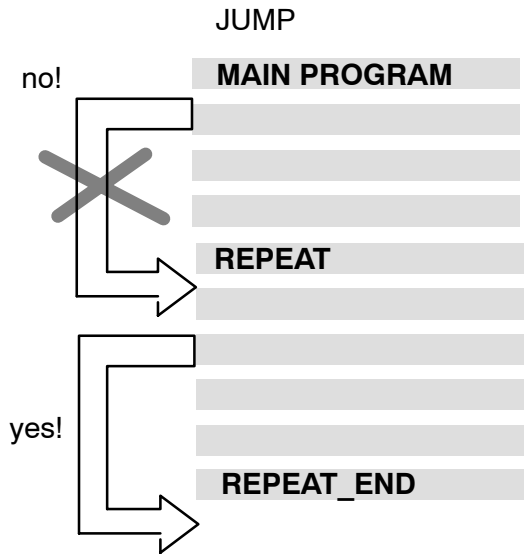
☞ **A specific jump label must be set only once! Any number of differently named jump labels is possible.**

☞ **Jumps from the main program into a subroutine and vice versa are not permitted!**



Program control

☞ **Jumps to program part repetitions are also not permitted.**



Jumps from program part repetitions, on the other hand, are permitted.

Programming

A jump to the jump label ’table’ (forward) is to take place in a program.

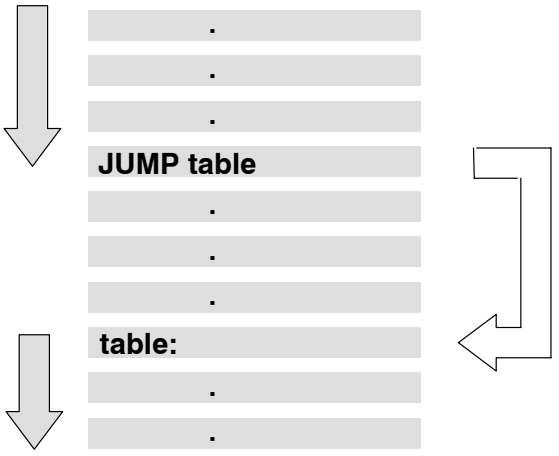
Syntax:

```
JUMP table    ;Jump statement for jump to jump label 'table'

table:        ;Setting of a jump label with the name 'table'
```

The control executes the program up to the jump statement, here JUMP table.

This is followed by a jump to the jump label table.

The control then continues the program from this label.



Program control

5.6 IF-THEN statement

Syntax:

```
IF condition THEN statement

             [ELSE statement]
```

The remaining execution of the program can be made to depend on a condition at freely selectable locations within a main program or subroutine. The statement dependent on the condition is therefore also referred to as conditional statement.

Condition is understood to mean an expression of the type BINARY.

The condition is satisfied if the statement is correct, i. e. the variable actually posesses the value or has a value within the specified value range.

The condition is not satisfied if the statement is false.

Example for conditions

channel	=	1
i	=	15
sig1	AND	sig2
word	=	j
force	>	115.0
torque	>=	thresholdval

☞ **Conditions must be put in brackets if necessary when combined with AND, OR, NOT, see section 6.2.3, in order to obtain the desired priorities of the condition operation.**

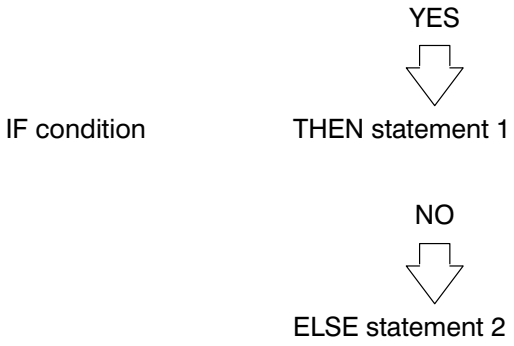
Program control

Programming

The conditional statement has the following form:

```
IF condition THEN statement 1

             ELSE statement 2
```

IF the condition is satisfied, THEN the control executes statement 1, ELSE, if the condition is not satisfied, the control executes statement 2.



Possible statements are, for example:

- Program calls (main program or subroutine calls)
- Jump statements (JUMP)
- Pause statements (PAUSE)
- Halt statements (HALT)
- Delays (WAIT)
- Repetitions (REPEAT)
- Movement instructions (MOVE, MOVE_REL)
- Conditional statements (IF...THEN...ELSE)
- etc.

If no jump is programmed in the THEN statement or ELSE statement, the control continues the program run with the program steps following the conditional statement.

The ELSE statement may be omitted. In this case, the control also continues the program run with the program steps following the conditional statement.

Example

The robot is to search for a pallet loaded with a workpiece on a shelf, weight approx. 200 kg. It has a sensor in its lifting device for this purpose which informs the control about the weight of the pallet.

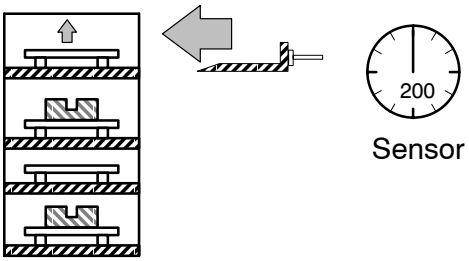
When it has found the workpiece, it is to transport it to the machine.

Starting position: The robot is positioned before the top pallet.

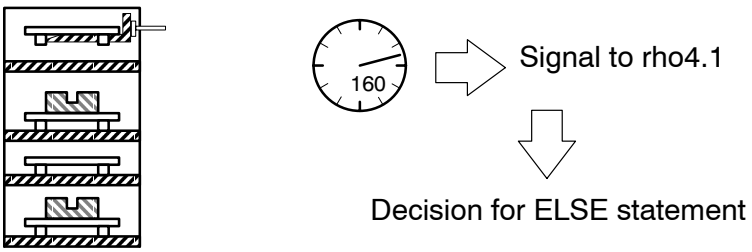
Program control

Syntax:

```
next:

   in                          ;call of subroutine 'in': travel in and
                               ;lift up pallet.

IF weight>=180.0

   THEN MOVE TO machine

   ELSE JUMP search

WRITE 'Workpiece found'

PAUSE

search:

   down                        ;SUBROUTINE deposit, travel out and travel down

   JUMP next

next:

   in                          ;SUBROUTINE move in and lift up
```

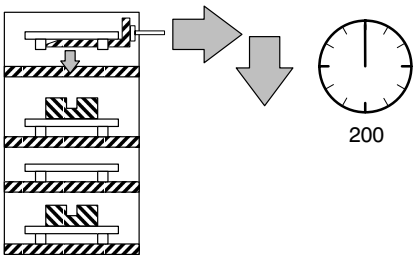



```
IF weight>=180.0               ;Evaluation of signal < 180.0 kg, e. g. only
                               ;weight of an empty pallet

   THEN MOVE TO machine

   ELSE JUMP search            ;Control decides for ELSE statement
                               ;Program jump to jump label 'search:'
```

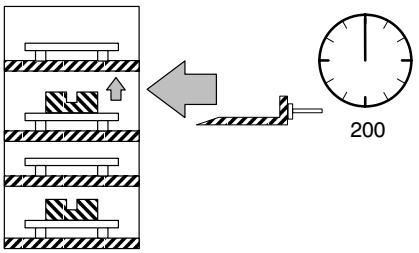


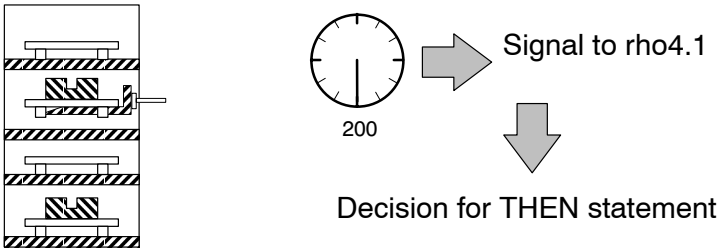
Program control

```
search:

   down       ;Call of subroutine 'down': lowering of pallet,
              ;travel out and lower.
```



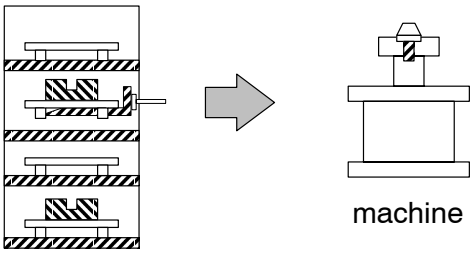

```
JUMP next    ;Program jump to jump label 'next:'

next:

   in         ;Call of subroutine 'in': travel in, and lift up pallet.
```



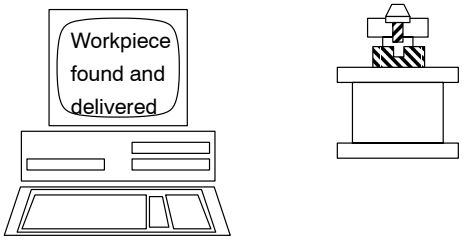


Program control

```
IF weight>=180.0          ;Evaluation of signal 200 kg. Control decides for
                          ;THEN statement

   THEN MOVE TO machine   ;Travels to position 'machine'

   ELSE JUMP search
```




```
WRITE 'Workpiece found and delivered' ;Output of text: 'Workpiece found and
                                      ;delivered' on monitor.

PAUSE
```



Program control

5.7 CASE statement

By using the branching statement CASE, a selection from several alternatives can be realized. Nested IF – THEN inquiries can be avoided and the execution time of the program can be reduced.

Syntax:

```
CASE selection_expression

   EQUAL: statement

   EQUAL: statement

   DEFAULT: statement

CASE_END
```

Admissible types for the selection expression are BINARY, INTEGER, CHAR and TEXT.

The constants of the constant list must be compatible with the selection expression. A constant may be used only once within the branching statement.

The DEFAULT part can be specified if desired. If the DEFAULT part is missing in the branching statement, no runtime error will be put out during the execution of the program. This is also valid if none of the statements applies.

Example:

```
PROGRAM case_ex

INTEGER: amount, offset        ;Declarations

BEGIN                          ;Program start

   amount=0

   offset=0

   READ amount

   CASE amount

      EQUAL 9,10:  offset=0

      EQUAL 11,12: offset=1

      EQUAL 13:    offset=2

      DEFAULT      offset=5

   CASE_END

PROGRAM_END
```

Program control

5.8 Parallel processes

The control rho4 can execute several user processes (programs) simultaneously. This feature is also known as multitasking capability.

The parallel processes are either defined within the same program (internal processes) or are each defined in a separate program (external processes).

5.8.1 External processes

An external process can be started or stopped by a program. Further synchronization does not take place.

In other words, the external process may still be active even if the program which has started the external process and has not stopped it again is terminated. The process may be stopped, for example, by a third process.

The external processes are included in the EXTERNAL statement.

```
        .
EXTERNAL: temp_control
        .
        .
START temp_control
        .
        .
STOP temp_control
        .
```

Program control

Starting and stopping external processes

External processes can be started either by:

- selection on the PHG2000
- program selection via the file Exprog.dat and corresponding interface signals
- BAPS statement START from a running BAPS process

The START statement can be extended by priority information

Syntax:

```
START temp_control PRIO=100 ;START process name PRIO = number
```

The process name is the name of the ird file and must be declared with EXTERNAL.

The priority code must lie between 100 (highest priority) and 150 (lowest priority). 100 is taken as the default value if no priority is specified.

An external process is stopped with the STOP statement.

Syntax:

```
STOP temp ;STOP process name
```

The process name is the name of the ird file and must be declared with EXTERNAL.

5.8.2 Internal processes

Internal processes are executed simultaneously within a program. In contrast to external processes, synchronization takes place here. The main program is started after a PARALLEL_END statement only after all parallel processes have been completed.

Internal processes are defined as follows.

Syntax:

```
PARALLEL

   MOVE sr8 TO corner               ;Process statements

ALSO

   MOVE_REL feeder EXACT (0,100)    ;Process statements

PARALLEL_END
```

Program control

5.8.3 Semaphores

If parallel processes access common resources, e. g. output devices, assignment can be managed by means of an EXCLUSIVE statement.

Syntax:

```
EXCLUSIVE sema_V24_1  ;EXCLUSIVE statement semaphore name

   WRITE V24_1, n     ;Statement

EXCLUSIVE_END         ;End of the EXCLUSIVE statement
```

The semaphore names are declared with the semaphore statement.

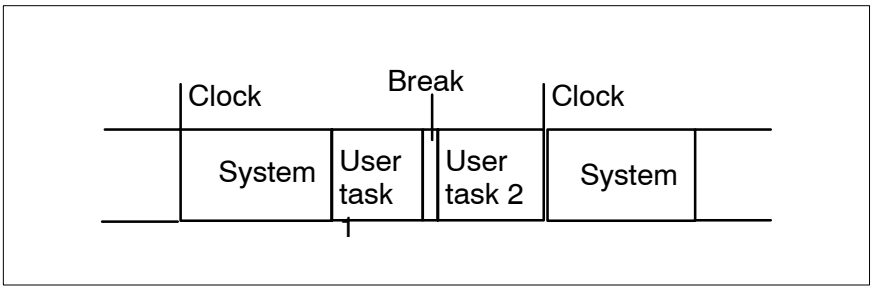
Syntax:

```
SEMAPHORE: sema_V24_1 ;SEMAPHORE: semaphore name
```

Program control

5.9 BREAK

A BAPS program normally needs the CPU until no further blocks have to be prepared or until the clock calls the CPU for the position control. 11 blocks or the number of blocks specified via the special function block preparation are prepared.



The user can force a break of the active user process at any location by using the BAPS function BREAK. The CPU is then be made available to other system processes of equal or higher priority during the same clock time. The interrupted process is appended to the end of the waiting queue of all processes of the same priority. After all processes of the same priority, which are waiting for the assignment of the CPU, have been active once, Round Robin-Algorithm, the interrupted process will be reactivated. Processes of higher priority can also be activated several times before the interrupted process is reactivated.

The BAPS function BREAK is realized as BAPS standard subroutine without parameters.

Syntax:

```
PROGRAM ex_1       ;Program name

INPUT: 1=i1        ;Declarations

OUTPUT: 1=o1

BEGIN

   loop:

      IF i1=1

      THEN o1=1

      BREAK

   JUMP loop

PROGRAM_END
```

The example process enables the CPU after each loop. The remaining time until the next clock interrupt is made available to other processes.

6 Value assignments and combinations

6.1 Value assignments

Value assignments are used to assign values to variables for speed and acceleration during the course of a program. This also applies to standard variables. The assigned value must be of the same data type as the variable.

Value assignment for general variables is described in the following section. Value assignment for positions can be found in the section 4.4.

Programming

Syntax:

```
variable=expression
```

Assignment takes place via the = symbol. The name of the variable to which a value is assigned must be stated on the left side of the assignment. An expression is located on the right side.

It is possible to enter

- numeric values (constants)
- variables
- arithmetic expressions
- standard functions

for the expression.

The sign + can be omitted.

The sign – is positioned directly before the variables or constants. The negative expression must be put in brackets if two operators follow each other directly.

All components must be multiplied by –1 if you want to negate all components of a variable of the type POINT or JC_POINT. –1.0 must also be put in brackets in this case.

Syntax:

```
counter=1             ;Variable and numeric value

xvalue=yvalue+2.5     ;Variable and arithmetic expression

distance=SIN(alpha)   ;Variable and standard function

value=-value          ;Variable of the type REAL and negation

d=d*(-2.0)            ;Variable and negative expression

corner=(-1.0)*corner ;Variable of the type POINT and negation
```

Value assignments and combinations

6.2 Combinations

6.2.1 Arithmetic expressions

Arithmetic expressions are combinations of

- numeric values and/or
- variables and/or
- standard functions and/or
- further arithmetic expressions.

The type of combination is defined by the operator. The rho4 knows five arithmetic operators:

+	Addition e. g. k = 1 + 5
–	Subtraction e. g. value = weight-1
*	Multiplication e. g. length = width x 2
/	Division e. g. height_new = height/2.0
MOD	Modulo calculation only for the data type INTEGER e. g. rest = number MOD divisor

☞ **The characters + and – can also be used as signs for variables and numeric values.**

☞ **Variable and numeric values with a sign must be put in brackets so that two operators do not follow each other directly.**

There must be no sign on the left side of an assignment.

The operations addition and subtraction can be performed with variables and numeric values (constants) of the type INTEGER, REAL, POINT and JC_POINT, while the operations multiplication and division can be performed with variables of the type INTEGER and REAL.

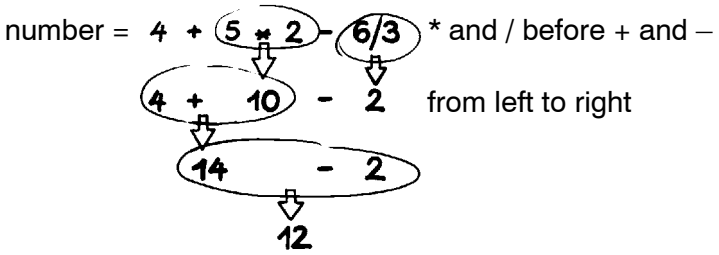
Variables and numeric values of the type REAL can also be used for multiplication and division of point variables, see section 4.4.

The operators *, / and MOD are executed before + and –

Calculation takes place from left to right within these classes.

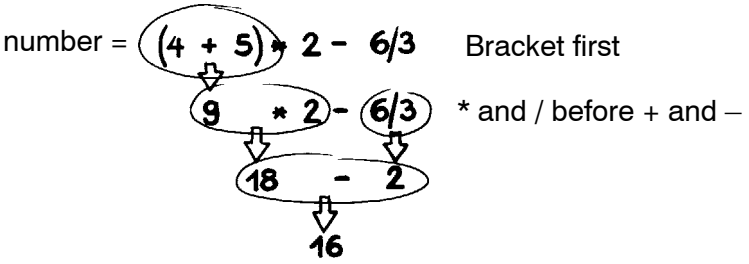
Value assignments and combinations

Example:



In addition, expressions which belong together can be put in brackets.

Example:



Modulo function

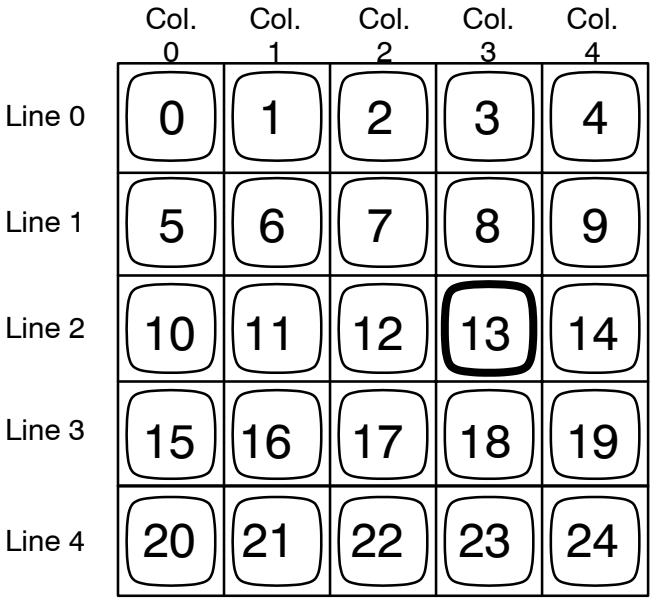
Modulo function: value1 MOD value2

The modulo function, data type INTEGER, calculates the integral remainder from division of value1 by value2. Value1, value2 and the result are of the type INTEGER.

Example:

Determination of the column of a pallet after specifying the position number and column number.

Value assignments and combinations



The pallet has 5 columns (0, 1, 2, 3, 4). Which column and which row does position 13 occupy?

The position number must be substituted for value1 and the column number for value2.

```
column=13 MOD 5 ;Calculation: 13/5 = 2 remainder 3. The remainder specifies the
                ;column, column 3 is thus the sought answer

line=13 MOD 5   ;Calculation: 13/5 = 2 remainder 3. The ratio indicates the line,
                ;line 2 is thus the sought answer
```

When calling the column, the modulo function determines the column in which the sought position is located by means of the value of the remainder, in the example = 3.

When calling the line, the modulo function determines the line in which the sought position is located by means of the value of the result, in the example = 2.

Value assignments and combinations

6.2.2 Comparison

The control inquires values and states in conditions, e. g. 'UNTIL condition' or 'IF condition'. This interrogation takes place via comparisons.

Programming

The following characters are available

=	equal to, e. g.: p = 1
<>	not equal to, e. g.: p <>1
>	greater than, e. g.: p > 1
>=	greater than or equal to, e. g.: p >= 1
<	less than, e. g.: p < 1
<=	less than or equal to, e. g.: p <= 1

☞ **Variables of the type POINT, JC_POINT and Record–Variables can be inquired only with respect to = (equal to) or <> (not equal to).**

6.2.3 Logic operations

The control checks conditions with respect to their truth value, see also section 6.2.2 and section 5.6. Conditions can thus only have one of two values.

Value 1 for true and value 0 for false.

This also applies to variables of the type BINARY.

These variables can also only ever have one of the two values, 0 or 1.

Combination of conditions

Often, the program sequence may depend on several conditions simultaneously.

☞ **Variables and expressions (conditions) of the type BINARY can be combined with the logic operations AND, OR and NOT.**

Value assignments and combinations

Combinations of two conditions cond_1 and cond_2 with AND

Truth values

cond_1	cond_2	cond_1 AND cond_2
1	1	1
1	0	0
0	1	0
0	0	0

Combinations of two conditions cond_1 and cond_2 with OR

Truth values

cond_1	cond_2	cond_1 OR cond_2
1	1	1
1	0	1
0	1	1
0	0	0

Negation of conditions

The truth content of conditions and variables of the type BINARY can be negated with the word NOT.

Example:

Condition cond_1 is true, thus: (cond_1) = 1

If the word NOT is placed before the condition cond_1, then the following is true for the truth content of NOT cond_1: (NOT cond_1) = 0

or

The condition cond_2 is false, thus (cond_2) = 0

then the following is true for the inverse function NOT: (NOT cond_2) = 1

Programmig of combinations of several conditions

Results of comparative operations are always of the data type BINARY. If several conditions are combined with each other, the following order of operators must be observed:.

1. NOT
2. *, /, MOD, AND
3. +, –, OR
4. =, < >, >, >=, <, <=

Example:

Interrogation of numeric values of the variables i and j of the type REAL

Value assignments and combinations

```
IF i = 10 AND j = 50 THEN...
```

In this example, the control first processes the expression ’10 AND j’. However, ’10 AND j’ represents a type conflict for the control, because the constant 10 is of the TYPE REAL and not of the data type BINARY.

Brackets are used to define the order for processing expressions.

```
IF (i = 10) AND (j = 50) THEN...
```

Value assignments and combinations

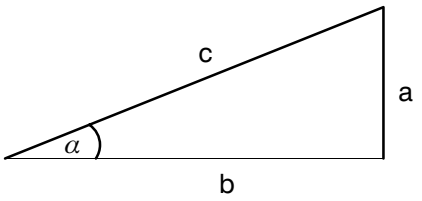
Notes:

Functions

7 Functions

7.1 Sine function

The sine function, data type REAL, establishes the mathematical relationship between an angle and the side lengths in a right-angle triangle.



$$\sin\alpha = \frac{\text{opposite leg}}{\text{hypothenuse}}$$

$$\sin\alpha = \frac{a}{c}$$

Programming

The angle α must be specified in radian measure 'rad'. The number is of the type REAL.

$$rad = \alpha \times \frac{\pi}{180^\circ} \quad ; \qquad \pi = 3.14$$

The radian is specified after SIN in brackets.

Syntax:

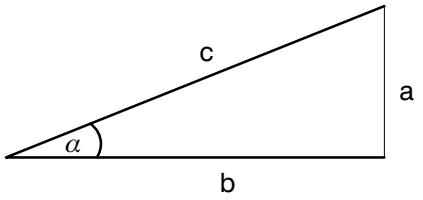
```
avalue=c*SIN(rad)
```

☞ **The designation a_{value} was chosen instead of only 'a' because 'A' has already been allocated as a reserved name for acceleration.**

Functions

7.2 Cosine function

The cosine function, data type REAL, establishes the mathematical relationship between an angle and the side lengths in a right-angle triangle.



$$\cos\alpha = \frac{\text{adjacent leg}}{\text{hypothenuse}}$$

$$\cos\alpha = \frac{b}{c}$$

Programming see Sine function.

Syntax:

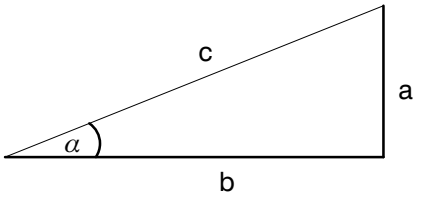
```
b=c*COS(rad)
```

Functions

7.3 Arc tangent function

The arc tangent function, data type REAL, determines the angle in a right-angle triangle by specification of the side length.

The arc tangent function is the inverse of the tangent function and is defined as follows:



$$\tan\alpha = \frac{\text{opposite leg}}{\text{adjacent leg}}$$

$$\tan\alpha = \frac{a}{b}$$

The inverse of the tangent function is then

$$\alpha = atan\ \frac{a}{b}$$

The angle is available as a radian value and can be converted.

$$\alpha = rad * \frac{180^\circ}{\pi}$$

Syntax:

```
alpha=ATAN(avalue/b)
```

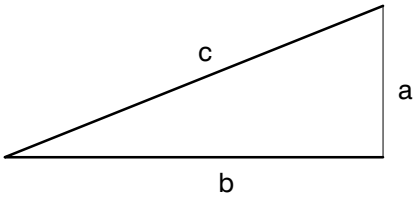
☞ **The designation a_{value} was chosen instead of only 'a' because 'A' has already been allocated as a reserved name for acceleration.**

Functions

7.4 Square root function

The root function, data type REAL, determines the value of the square root of an expression.

Calculation of the length c in a right-angle triangle.



Pythagorean theorem:

$a^2 + b^2 = c^2$ or $c = \sqrt{a^2 + b^2}$

Syntax:

```
c=SQRT(avalue*avalue+b*b)
```

☞ **The designation a_{value} was chosen instead of only 'a' because 'A' has already been allocated as a reserved name for acceleration.**

7.5 Absolute value

Syntax:

```
ABS(argument)
```

The result supplied by the function is the absolute value of the transferred argument. The argument may be of the type REAL or INTEGER. The result is of the same type as the argument.

Example:

```
deviation=ABS(delta)
```

Functions

7.6 TRUNC

Syntax:

```
TRUNC(argument)
```

The function transforms the argument of the type REAL into a value of the type INTEGER by truncation. In case of a positive argument, the result obtained is the largest integral number less than or equal to the argument, e. g. 3.7 becomes 3.

In case of a negative argument, the result obtained is the smallest integral number greater than or equal to the argument, e. g. –3.7 becomes –3.

7.7 ORD

Syntax:

```
ORD(char_var)
```

This function supplies the INTEGER value of variables of the TYPE CHAR

Example:

```
char_num=ORD(asc_char)

char_num=ORD('i')
```

7.8 CHR

Syntax:

```
CHR(integer)
```

This function supplies a value of the TYPE CHAR corresponding to the calculation integer MOD 256.

Example:

```
asc_char=CHR(34)
```

Functions

7.9 ROUND

Syntax:

```
ROUND(argument)
```

The function transforms an argument of the type REAL into a value of the type INTEGER.

Rounding takes place to the integral number closest to the argument. In the case of arguments which lie exactly between two neighboring integral numbers (e. g. 0.5, 1.5), rounding always takes place to the even integral number, i. e. 1.5 is rounded to 2 and 6.5 is rounded to 6.

Example:

```
value_3=ROUND(7.81)  ;Value_3 is assigned to 8

value_4=ROUND(-5.43) ;Value_4 is assigned to -5
```

7.10 Coordinate transformation

Syntax:

```
WC(@jc_point)

JC(wc_point)
```

Using the standard functions JC and WC, it is possible to calculate both with POINT and JC_POINT variables in an assignment.

WC	transforms the joint coordinates into world coordinates e. g. position = WC(@position)
JC	transforms the world coordinates into joint coordinates e. g. @corner = JC(corner)

WC and JC always supply values from the main area if joint or world coordinates could be ambiguous.

Functions

7.11 End of file

Syntax:

```
END_OF_FILE(argument)
```

This function allows interrogation of whether the end of the file has been reached when reading a .dat file. The argument is a variable of the type FILE.

The function supplies the value 1 (true) if the end of the file has been reached and the value 0 (false) if the end of the file has not been reached.

Example 1:

```
IF END_OF_FILE(dat_values)  ;Use of the function in a conditional statement

   THEN statements

   ELSE statements
```

Example 2:

```
EOF=END_OF_FILE(dat_values) ;Use of the function in an assignment
```

Functions

7.12 Integration of PLC program modules

The rho4 offers the possibility to activate program modules in the PLC from out of BAPS. Various standard subroutines have been designed for this purpose. The START, STOP and standard function 'status' can be expanded to achieve a flexible control of the time behavior of the program modules.

7.12.1 Standard subroutines and/or standard functions

PLC_process

The standard subroutine PLC_process permits the call of PLC program modules. It contains a parameter which specifies the number of the desired PLC program module.

PLC_time

The standard subroutine PLC_time permits the cyclical call of PLC program modules with a fixed time raster. It contains a parameter which specifies the number of the desired PLC program module.

☞ **Since the subroutine PLC_process (PLC_time) is a BAPS standard subroutine there is no need for declaring it by the user; it can simply be used, as e. g. INT_ASC. As in all BAPS standard subroutines, the name PLC_process (PLC_time) is no reserved word. If the user creates a BAPS subroutine with the name PLC_process (PLC_time), it will cover the standard subroutine PLC_process (PLC_time) and can thus no longer be called.**

Syntax of the the implicite declaration:

```
PLC_process=PLC_process(VALUE INTEGER: proc_no) ;Standard subroutine

PLC_time=PLC_time(VALUE INTEGER: task_no)       ;Standard subroutine
```

In the above syntax the following applies:

- proc_no:
  BAPS constant of the type INTEGER. Specifies the PLC program module number, 1 to 128.
- task_no:
  BAPS constant of the type INTEGER. Specifies the PLC task number, 1 to 8. The priority of the task decreases with ascending task number, 1 = highest priority, 8 = lowest priority.

Functions

7.12.2 Single activation of program modules

When calling the standard subroutine PLC_process, a program module is passed once on the PLC. The processing of the BAPS program is interrupted and will only be continued after having ended the PLC program module (synchronous processing).

☞ **The PLC contains 128 prepared program modules: PROC_001 to PROC_128. These constants are not predefined in BAPS. They have to be declared if required, e. g. in an include file.**

7.12.3 Cyclical activation of program modules

The cyclical activation of program modules on the PLC represents an expansion of the above single program module call.

There are two different types to distinguish:

- cyclical activation without fixed time raster
- cyclical activation with a fixed time raster

Cyclical activation without fixed time raster

The start statement START PLC_process serves in BAPS the cyclical activation of a program module in the PLC. Since the programs are always processed cyclically in the PLC, the program module activated by means of the START PLC_process is also processed cyclically in the PLC.

Cyclical activation of PLC programs with a fixed time raster

PLC program modules with a fixed time raster can be activated from out of BAPS user programs to permit a quick cyclical operation of the peripheral equipment. The start statement also serves the cyclical activation of a program module in the PLC. The standard subroutine PLC_time is contrary to the activation without fixed time raster used as a parameter.

The cyclical activation with a fixed time raster is realized by programming time-controlled modules on the PLC. They are called by means of prepared program modules in a fixed time raster.

The determination of a time raster is made by the extention of the start statement. The time value can in this case be specified in seconds. The specification of a time value is optional. If it is not specified, the time value set in the PLC is used. In this case, there will be no transfer of time values to the PLC. If a time is programmed, the execution of the BAPS program is stopped until the PLC has taken over the value.

Functions

7.12.4 Extension of the START statement

The START statement is extended by the following functions:

- Standard subroutine:
  Use of one of the standard subroutines PLC_process resp. PLC_time.
- Argument list:
  Specification of a parameter within the START statement. This is only permitted when using the standard subroutines PLC_process resp. PLC_time.
- Event:
  Specification of a time value for the cyclical activation in a fixed time raster. The specification is made in seconds.

Syntax:

```
start_statement=START name[(argument)][priority][event]
```

name = external_mp_name resp. name = PLC_process resp. name = PLC_time

argument = constant_expression

priority = PRIO = constant_expression

event = EVERY T = expression

In the previous syntax the following applies:

- name:
  External main program name, PLC_process or PLC_time. The standard subroutines PLC_process and PLC_time require the specification of the desired PLC program modules as parameter.
- argument:
  Constant expression; specifies the PLC program module resp. PLC task number. Only permitted for PLC_process or PLC_time.
- expression:
  Specification of the time value in seconds, only for PLC_time. The expression is of the type REAL.

Functions

7.12.5 Extension of the STOP statement

The execution of program modules is stopped on the PLC by means of the stop statement STOP PLC_process (PROC_xxx). The argument in the brackets of the instruction specifies the PLC program module to be stopped.

For this purpose, the stop statement is extended by the following functions:

- Standard subroutine:
  Use of one of the standard subroutines PLC_process resp. PLC_time.
- Argument list:
  Specification of a parameter within the stop statement. This is only permitted when using the standard subroutines PLC_process resp. PLC_time.

Syntax:

```
stop_statement=STOP name[(argument)](1)
```

name = external_mp_name resp. PLC_process resp. PLC_time

argument = constant_expression

(1) is only permitted for PLC_process resp. PLC_time

In the previous syntax the following applies:

- name:
  External main program name, PLC_process or PLC_time. The standard subroutines PLC_process and PLC_time require the specification of the desired PLC program modules as parameter.
- argument:
  Constant expression; specifies the PLC program module resp. PLC task number. Only permitted for PLC_process or PLC_time.

Functions

Example of PLC integration for rho4:

```
PROGRAM rho_plc

CONST:

   pm_005=5,

   pm_007=7,

   pm_cycl=1

BEGIN

   PLC_process(pm_007)        ;Single call of a PLC program module. The
                              ;program rho_plc is only continued, when the
                              ;program module pm_007 has been passed.

   START PLC_process(pm_005) ;Starts program module pm_005 on the PLC cyclically
                              ;with each PLC cycle.

   STOP PLC_process(pm_005)  ;The program module pm_005 is no longer
                              ;passed in the next PLC-cycle, the
                              ;program module 5 has stopped. i. e. STOP does
                              ;not act immediately as in BAPS processes.

   START PLC_time(pm_cycl) EVERY T=500 ;Starts program module pm_cycl every
                                       ;500 ms

   IF condition (PLC_time(pm_cycl))

      THEN STOP PLC_time(pm_cycl)      ;Program module pm_cycl has stopped

   START PLC_time(pm_cycl)             ;Starts program module pm_cycl,
                                       ;using the time value set
                                       ;in the PLC

PROGRAM_END
```

Functions

7.13 CONDITION interface, process, system signal, file

CONDITION

This function permits to determine the statuses of interfaces, processes, system signals and files.

The following interfaces are supported:

- V24_1 to V24_4
- PHG
- TTY
- WIN_1 to WIN_4
- PLC

Example:

```
IF CONDITION(V24_1)<0

   THEN statements

   ELSE statements
```

The possible conditions are made available to the program for evaluation via the return value of the function of the data type INTEGER. The function value can be evaluated by the BAPS programmer, and the corresponding reactions can be initiated.

Functions

The following values are supplied by the standard function condition:

Function value	Meaning
0	no error
–1	interface not available
–2	interface is occupied
–3	timeout
–4	parity error
–5	overrun error
–6	framing error
–7	general interface error
–8	length error of character string
–9	protocol error
–10	REAL value not permitted
–11	point not defined
–30	BUEP data error PCL
–31	DM on PCL not ok
–32	DM length to high
–33	DM length not assigned
–34	DM No. not assigned

Also files are permitted as arguments for the function CONDITION.

```
FILE: data              ;Declaration of the file. data can represent any filename

IF CONDITION(data)<0    ;Inquiry whether file is available
THEN JUMP n_dat
n_dat:
```

The following values are supplied by the function CONDITION:

Function value	Meaning
0	no error
–1	FILE not available

Functions

Also processes are permitted as arguments for the function CONDITION.

```
EXTERNAL: procname          ;Declaration of the process. procname can represent
                            ;any process name

IF CONDITION(procname) = -1 ;Inquiry whether process is available

THEN JUMP n_proc

n_proc:
```

The following values are supplied by the function CONDITION:

Function value	Meaning
–1	Process not found
1	Process runnning
2	Process idle
3	Process ready
4	Process has reached breakpoint (BAPS plus)
5	Process stopped (e.g. at Emergency–Stop)
11	1 Sub–Process active
12	2 Sub–Processes active
13	3 Sub–Processes active
14	4 Sub–Processes active
15	5 Sub–Processes active
10+n	n Sub–Processes active (n= 1 until 90)
>100	Process error (runtime error) error code see manual “Status messages and warnings”

Functions

Also system signales are permitted as arguments for the function CONDITION.

There are two programming possibilities:

– The corresponding signal address is directly inserted as parameter between inverted commas
– The desired signal address is assigned indirectly by means of a text constant

```
Examples:

CONST   :               ;Declaration of the system signal. I85.7 and I13 can
 txtconst   = 'I85.7'   ;represent any system signal
 txtbyconst = 'I13'

INTEGER: state


state = CONDITION('I0.0')       ;Input signal Byte0, Bit0
state = CONDITION('I0')         ;Input signal Byte 0

state = CONDITION(txtconst)     ;Input signal Byte85, Bit7
state = CONDITION(txtbyconst)   ;Input signal Byte13
```

The following values are supplied by the function CONDITION

Function value	**Meaning**
0/1	Signal condition (0 resp. 1)
0..255	Signal condition of a byte
–1	invalid signal group
–2	invalid bit address
–3	invalid character
–4	invalid signal address

Functions

Extension of the standard function 'CONDITION'

The standard function condition is extended to permit the two standard subroutines PLC_process and PLC_time as arguments, see section 7.12.1. The return values of the standard function condition when using the two standard subroutines are defined as follows:

0 program module not active

1 program module is active

Syntax of the implicite declaration:

```
CONDITION=INTEGER: CONDITION(argument) ; standard function
```

argument = file variable resp. interface resp. name resp. text expression.

name = external_mp_name resp. PLC_process (parameter) resp. PLC_time (parameter).

parameter = constant_expression.

In the previous syntax the following applies:

- Name:
  External main program name, PLC_process or PLC_time. PLC_process and PLC_time require the specification of the desired PLC program modules as parameter.
- Parameter:
  Constant expression; number of the desired PLC program modules.

Compiler statement SER_IO_STOP

The compiler statement can avoid that a user program is aborted when an interface error occurs.

Syntax:

```
;;SER_IO_STOP- ;No program break in case of error

;;SER_IO_STOP+ ;Program break in case of error

;;SER_IO_STOP  ;Program break in case of error
```

The statement is only permitted at the start of a program. It applies to the entire program and may only appear once. If the compiler statement is not used, an interface error will lead to a pogram break.

Example:

```
;;SER_IO_STOP-                          ;No program break

PROGRAM io_test                         ;Program name
```

Functions

```
INTEGER: index,number                    ;Declarations
BEGIN
   number=0
   READ V24_1,index
   IF CONDITION(V24_1)<0
   THEN
      BEGIN
         WRITE 'general read error V24_1'
         HALT
      END
   ELSE
      REPEAT index TIMES
         number=number+1
      REPEAT_END
PROGRAM_END
```

Functions

7.14 ASSIGN

ASSIGN (file variable, text expression)

The statement ASSIGN permits to change the names of files or the assignment of the standard channels V24_1 to V24_4, SER_1 to SER_4 and PHG at the program runtime by the user program.

☞ **The ASSIGN statement can only be applied to closed files and standard channels.**

Syntax:

```
   ASSIGN datvar,datname ;Assignment of the content of the text variable datname
                         ;to the file variable datvar.

   READ_BEGIN datvar     ;The following read access to the file is made with the
                         ;name read in the variable datname.

   READ datvar, oneline

   WRITE PHG, oneline

   CLOSE datvar

   channel_var='V24_2.'  ;Assignment of the output channel PHG to the serial
                         ;interface V24_2 by means of the text variable
                         ;channel_var.

   ASSIGN PHG, channel_var

   WRITE PHG, oneline    ;Output of a line on channel V24_2.

   ASSIGN PHG, 'PHG.'    ;Reassign the channel PHG to the PHG.

PROGRAM_END
```

☞ **The character '.' is important for the channel names as an identification of the end. If no dot follows as last character, the assignment will be made to a file name with the extension dat, e. g. PHG.dat.**

Functions

7.15 Conversion routine INT_ASC

The standard procedure INT_ASC converts an integral value into an array of characters.

☞ **The format is right-aligned with possibly leading space characters. If an error is detected during the conversion, the character array will remain unchanged and the corresponding error code will be returned.**

Syntax:

```
INTEGER: int_number             ;Default

INTEGER: index_na               ;int_number = number to be converted. Only

INTEGER: length_char            ;permitted of the type  INTEGER, e. g.

INTEGER: error_ret              ;integer value <= 2147483647

ARRAY[1..10] CHAR: ascii_array  ;Result return message ascii_array =  result array
                                ;of the type ARRAY [ ] character

int_number=-1234                ;Default

index_na=1                      ;index_na = start index in the character array

length_char=5                   ;Default
                                ;length_char = maximum number of characters
                                ;reserved for the number to be converted.

INT_ASC(int_number,ascii_array,index_na,length_char,error_ret)
                                ;The destination range is initialized with space
                                ;characters before the conversion.

IF error_ret<>0                 ;Result return message error_ret =
                                ;output of error number

   THEN WRITE 'INTEGER-ascii conversion error'

     ELSE WRITE ascii_array
```

Functions

Meaning of error number

0	no errors
–1	start index exceeds array limits
–2	end index, consisting of start index and length, exceeds array limits
–3	reserved length is too small
–4	range exceeded (value too high)
–5	array length < 0
–6	array length = 0

Functions

7.16 Conversion routine ASC_INT

The standard procedure ASC_INT converts an array of characters into an integral value.

The procedure reads characters beginning at the start position until a character is recognized which is not a number or until the maximum number of characters has been read resp. the end of the character array has been reached.

Syntax:

```
INTEGER: int_number              ;Result return message
                                 ;int_number = converted number (only
                                 ;permitted of the type INTEGER)

INTEGER: index_na                ;Default
                                 ;index_na = position in the array, from which the
                                 ;reading of the number should start

INTEGER: length_char             ;Default
                                 ;length_char = maximum number of characters to be
                                 ;read

INTEGER: error_ret

ARRAY [1..10] CHAR: ascii_array  ;ascii_array = array to be converted of the type
                                 ;ARRAY[ ] CHAR

ascii_array='AaBc'

index_na=1

length_char=5

ASC_INT(ascii_array,int_number,index_na,length_char,error_ret)

IF error_ret<>0                  ;Result return message error_ret =
                                 ;output of error number

   THEN WRITE 'ascii-INTEGER conversion error'

     ELSE WRITE int_number
```

Functions

Meaning of error number

0	no errors
–1	start index exceeds array limits
–2	end index, consisting of start index and length, exceeds array limits
–3	———
–4	range exceeded (value too high)
–5	array length < 0
–6	array length = 0
–7	character string does not start with a number or a sign

7.17 Call of rho4 library functions

In the rho4 library, library functions are made available for the OEM C functions which serve the communication with the basic operating system and permit the access to operating system variables. These functions are declared in the individual include files, divided by function groups, and are adopted into the BAPS program by means of INCLUDE statement if required. The rho4 library functions can then be called directly in the BAPS program.

The declaration of these functions is made similar to the special functions via a new function declaration part named RHO_FCT. The functions declared there, always send back a return code of the data type INTEGER which contains the error messages and warnings.

Further details, especially the names of the functions and function groups can be taken from the manual 'DLL libraries'.

☞ **Since RHO_FCT is a reserved word, it may only be used within the declaration part of the library functions.**

The rho4 library functions can in BAPS programs also be used at other places at which a subroutine is expected. In this case, the return value of the function is ignored.

Syntax the library function declaration:

```
RHO_FCT:
const_expression=name[parameterlist]{,const_expression=name[parameterlist]}
```

In the previous syntax the following applies:

- const_expression:
  Constant expression of the type INTEGER. Function number of the desired library function. This number is determined by Bosch.

Functions

- name:
  Name the desired library function, can be changed by the user ir required.
- parameterlist:
  Parameter of the desired library function. The number of parameters and their types are determined by Bosch and must not be changed.

Example of an include file for library functions (rhoKin.inc)

```
rhoKin.inc                                          ;file: rhoKin.inc

TYPE :TVB_AxPos=RECORD                              ;components
                RECORD_END

RHO_FCT: 0815=rKGAxPos(TVB_AxPos: AxPos),
         4711=rKGAllAxPos(TVB_AllAxPos: AllAxPos)
```

Example of library functions

```
PROGRAM rhoFct

;;INCLUDE rhoKin.inc                    ;Contains constants and record types of
                                        ;the rho4 functions, among others also the
                                        ;record type TVB_AxPos!

TVB_AxPos: VB_AxPos INTEGER: ReturnCode ;Record type TVB_AxPos is contained in
                                        ;rhoKin.inc

BEGIN

   ReturnCode=rKGAxPos(VB_AxPos)        ;rKGAxPos contained in rhoKin.inc

   IF ReturnCode <0

      THEN WRITE 'error':,ReturnCode

   IF rKGAxPos(VB_AxPos)<0              ;simplified writing

      THEN WRITE 'error:',ReturnCode

PROGRAM_END
```

Functions

7.18 rho4 special functions

With the special functions, the BAPS3 programmer will obtain access to special functions present in the control rho4 for which no BAPS3 language elements are reserved.

Special functions represent an extension of the BAPS3 language extent. They can be called in a program if they have been deifined before the call in a similar way to a variable.

☞ **Not all listed special functions are suggestive resp. practicable for each rho4 controllable kinematics. If you have a question, please apply to the technical support:**

mailto: Mounting–Handling–RC.brc@boschrexroth.de
Phone: +49 (0) 60 62 / 78–0

Functions

In the rho4, the following special functions are available:

Ft. No.	Function short description
1	Position exact switching of digital outputs on the path
2	Positionexact switching of decimal outputs on the path
3	Set machine position
4	Call operating system functions
15	Parametrization of the belt characteristic
16	Select point file
17	Mirroring
21	Belt type
23	System data and time
24	System counter
27	WC main counter
28	Set belt counter
29	Switch on set path registration
30	Switch off set path registration
31	Read set path values
43	Flying measurement On, only available for rho4.1
44	Flying measurement Off, only available for rho4.1
45	MOVE_FILE, run curve from .bnr-file
46	Adjust set advance
47	Define exception
48	Detect exception
51	Set of belt counter reset value (from version VO05 no more available)
52	Velocity adjustment for PTP movements
53	Belt range with beltkind 4
54	Questioning the belt velocity with beltkind 4
55	Changing the belt simulation velocity with beltkind 4
56	Accurate beltsynchronous position switching of digital outputs on the path
57	Accurate beltsynchronous position switching of decimal outputs on the path

Functions

Specifying the special functions

The specification of a special function contains code and name of the function as well as name and data type of the transfer parameters. Through the transfer parameters, you define when, where and how the function is to be effective. The specification must be done in the declaration part of the program.

The name of the special function and the names of the transfer parameters can be freely chosen.

☞ **The data types are fixed through the specification of the corresponding special function.**

Example

SPC_FCT : x = example (VALUE REAL: wap)

SPC_FCT : x	Function number
example	Special function name
VALUE REAL:	Data type designation of the transfer parameter, e.g. decimal value
wap	Name of the transfer parameter

Call of the special function

The call in the instruction part of the program is made through the entry of the special function name and the definition of the specified transfer parameters.

In the call, the name of the special function and the types of the transfer parameters must be kept in the way as they have been defined in the specification in the program.

General example

example (15.5)

with the meaning

example	Special function name
(15.5)	Definition of the transfer parameter

The special functions are described in the manual rho 4 ’Control functions’.

Functions

7.19 Standard function ’sizeof’

The standard function ’sizeof’ serves the determination of the required memory space for any BAPS variable resp. type. Especially when dimensioning transfer buffers, e. g. for the communication via TCP/IP connections, it is important to know the memory requirement for a given variable (data buffer). When programming under Windows it is a quasi standard to initialize the first element of a complex data structure with its size.

The function sends back the memory requirement of the specified type or the variable in bytes. The standard function ’sizeof’ can be used within the constant definition part.

Syntax of the implicite declaration:

```
sizeof=INTEGER: sizeof(type_or_variable)           ;standard function
```

In the previous syntax the following applies:

- Type_or_variable:
  Any type resp. variable name.

The following table shows the memory place of the standard types.

Functions

Table of memory requirement for the BAPS data types

Data type	sizeof (data type) [Bytes]	corresponding MS-C/C++ -type	Value range in BAPS
BINARY	4	long	[0, 1]
INTEGER	4	long	[–2147483648 , +2147483647]
REAL	4 (IEEE, simply exact)	float	3.4E +/–38 (7 digits)
CHAR	1	unsigned char	[CHR(0), CHR(255)]
POINT	axisnumber * sizeof (REAL)	typedef float POINT[axisnumber]	–
JC_POINT	axisnumber * sizeof (REAL)	typedef float JC_POINT[axisnumber]	–
BELT	4 + sizeof (REAL)	typedef struct BELT {float belt_value long belt_No } BELT	–
WC_FRAME	6 * sizeof (REAL)	typedef float WC_FRAME[6]	–
TEXT	80 * sizeof (CHAR)	typedef char TEXT[80]	–
INPUT <type>	4 + sizeof <type>	typedef struct IN<type> {<type> value; long address} IN<type>	see <type>
OUTPUT <type>	4 + sizeof <type>	typedef struct OUT<type> {<type> value; long address} OUT<type>	see <type>
FILE	28	–	–
BNR_FILE	28	–	–
SEMAPHORE	0 (no memory is reserved)	–	–

Example standard function sizeof

```
;;JC_names=a1,a2,a3,a4

;;WC_names=c1,c2,c3,c4

PROGRAM size

POINT: EndPos

BEGIN

   WRITE 'REAL-size:',sizeof(REAL)      ;output: 4

   WRITE 'POINT-size:',sizeof(EndPos)   ;output: 16 (4 axes * size of REAL)

PROGRAM_END
```

Functions

7.20 Workpiece coordinate system

7.20.1 General information

The function 'workpiece coordinate system' makes it possible for the user to define his own world coordinate systems, both in the automatic and the manual mode.

It is thus possible to adapt a BAPS processing program established in workpiece coordinates to the current position of the workpiece, e. g. a program created offline can be adapted to the actual position in the processing phase.

Example varnishing system:
The car body data supplied by a CAD–system refer to the car body (zero point of the coordinate system is placed in the front axle of the car). The function workpiece coordinate system permits to adapt the varnishing programs to all occurring carriage types.

Example palettising:
A corner of the pallet is defined as the zero point of the local coordinate system. All processing points have been teached. During assembly, the pallet can lie anywhere in the working area of the robot. The function workpiece coordinate system makes it possible to process the pallet in every position without changing the program.

7.20.2 Name determination of coordinate systems

Because we speak in the following of different coordinate systems, here are the name determinations:

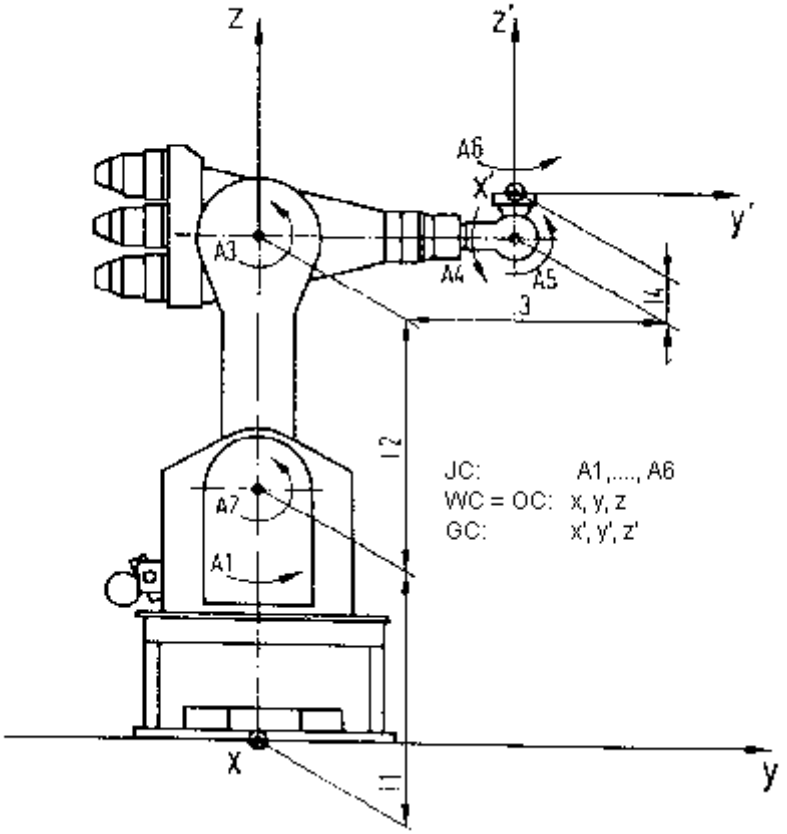
- **JC** = **J**oint **C**oordinates. Describe the several axis positions in [mm] (linear axis), resp. [degrees] (rotation axis)

The units of the following coordinate systems are in each case (x, y, z) in [mm] + 3 rotations at the coordinate axes in [degrees] :

- **WC** = **W**orld **C**oordinates. Is a free definable local world coordinate system, also called workpiece coordinate system
  Standard (= in the robot root point): WC = OC
  WC system programmed: WC = WC_System
- **OC** = **O**riginal **C**oordinates. Is a fix defined coordinate system related to the robot root
- **GC** = **G**ripper **C**oordinates. Is a co–moved coordinate system related to the TCP (only enable in manual operation)
- **CC** = **C**ell **C**oordinates. Is a fix defined coordinate system, related to the working cell

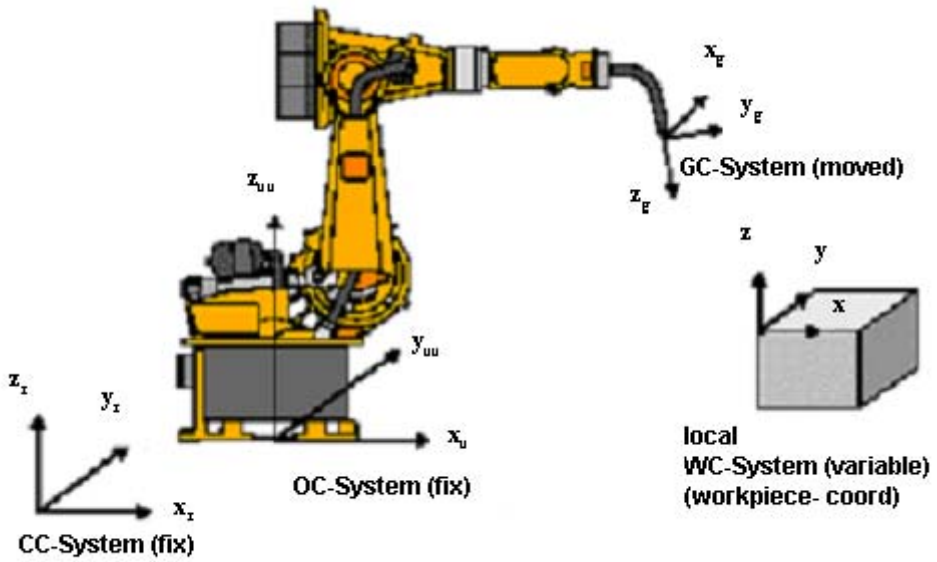
Functions

Fig. 1



Functions

Fig. 2



7.20.3 BAPS Syntax

For the definition of the individual coordinate systems, the following language elements are used in the programming language BAPS:

- **WC_FRAME**
- **WC_SYSTEM**
- **WC_UR**

WC_FRAME is a kinematic-specific standard BAPS type consisting of six components of type **REAL**. The 6 components describe the displacement and rotation of the workpiece coordinate system (local WC system) opposite to the original coordinate system (robot coordinate system). The parameter block specifies the position of the zero point of the local WC system in the original coordinate system. The order of the components is fixed as follows:

	Translation of the original coordinate system in
first component = Δx	x direction
second component = Δy	y direction
third component = Δz	z direction

Functions

	Rotation by the
fourth component = $\Delta o1$	first orientation axis
fifth component = $\Delta o2$	second orientation axis
sixth component = $\Delta o3$	third orientation axis

The special definition and order of the 3 orientations depend on the robot type and if necessary of the selected orientation definition.

☞ **This order has to be observed in any case.**

The displacements are entered in mm and the rotations in degrees.

Restriction:

The workpiece coordinate system can only be used with 6–axis kinematics. If the kinematic has less than 6 degrees of freedom, in machine parameter P313 must be determined, which axis should be considered. In this case the workpiece coordinate system can be displaced in all 3 directions and maximal 1 axis can be rotated. Around which axis the rotation can take place, can only be inquired directly from Bosch–Rexroth.

WC_SYSTEM is a kinematic-specific standard variable of the type WC_FRAME. The valid workpiece coordinate system is communicated to the control and activated via this standard variable.

WC_UR is a constant of the type WC_FRAME predefined in BAPS. Its six components have all the value zero. If this constant is assigned to WC_SYSTEM, all displacements and rotations will become zero relative to the original coordinate system. For example by

pallet_1.WC_SYSTEM = WC_UR

The workpiece coordinate system thus eqals the original world coordinate system.

Variable–declaration and –assignment

<Kinematic–name>.**WC_FRAME** : <WC–Varname>

<WC–Varname> is the name (variable name) of the local workpiece coordinate system that can be freely selected by the user. Its length may have a maximum of twelve characters.

The WC_FRAME variables declared in this way may be used in a BAPS program in three different ways:

- (1) per direct declaration of value
- (2) per assignment via variable

Functions

- (3) via the fixed defined component names
  Pose_x, Pose_y, Pose_z, PoseO1, PoseO2, PoseO3.

  This variant offers additionally the possibility, to readout the actual values of the variable WC_FRAME.

Example:

```
;;KINEMATICS: (1=Rob1)

REAL: V1, V2, V3, V4, V5, V6

Rob1.WC_FRAME: pallet1, pallet2, deposit_pal

     .

     .

(1)      pallet1 = WC_FRAME (10.1, 20.2 30 40 45 10.6)

(2)      pallet2 = WC_FRAME (V1, V2, V3, V4, V5, V6)

(3)      deposit_pal.Pose_x = V3
         V5 = depositpal.PoseO3
```

The WC_FRAME variables supplied with values are used below for switching the WC coordinate system. The keyword WC_FRAME is used at the assignment to distinguish the six REAL values from a 6–axis WC point.

7.20.4 System file WCSYST.DAT

Alternatively to the direct programming, the ASCII system file **WCSYST.DAT** (counterpart to TOOLS.DAT) in which all local coordinate systems are defined by name, is available.

WCSYST.DAT is the reserved name for the file to be created by the user himself. The name of the file depend on the selected language in machine parameter P10.

The individual local WC coordinate systems are named with a free selectable name by the user. Under this name the appropriate coordinate values are discarded.
The file WCSYST.DAT is edited as ASCII file in the robot control or offline. For each line, one WC sytem name and all associated coordinate values are entered as follows:

pallet3 = 11.1 22.2 33 44 55.5 66

The name of the local WC system must be at the beginning of the line and its length must not have more than twelve characters. It can be freely selected. The WC system name and the coordinate values are to be separated by equal signs '='.

The individual coordinate data are to be separated by space characters. They are decimal values, whereby the decimal point has not in any case to be set. For the coordinate values, only the entries '0', '1' to '9', '+', '–' and '.' are permitted.

Comments at the end of the line are permitted. They must start with ';'. Complete comment lines are also allowed. They too must start with ';'.

7.20.5 WC system selection in a BAPS program

The activation of a local workpiece coordinate system may take place in 3 kinds.

```
;;KINEMATICS: (1=Rob1)

TKSWcSystem:    PKSWcSystem

Rob1.WC_FRAME: pallet1, pallet2

     .

     .

(1)        Rob1.WC_SYSTEM = WC_FRAME (0, 0, 30.5, 0, 0, 0)

(2)        Rob1.WC_SYSTEM = pallet1


(3)        PKSWcSystem.ChannelId    = 0
           PKSWcSystem.System.KinNr = 1
           PKSWcSystem.WcSysName    = 'pallet2'
           Ret = rKSWcSystem (PKSWcSystem)
```

With variant 1 the coordinate values are programmed directly at the activation.
With variant 2 the values of the variables of "pallet1" are activated.

With variant 3 the WC system with name "pallet2" is activated by a library function. The name and the appropriate coordinate values must be defined in the system file "WCSYST.DAT" (see chapter 7.20.4).
Variant 3 ist name oriented. With variants 1 and 2, the name 'WCSys active' is assigned by the operating system.

☞ **Attention: Because movement informations may vary by instructions, the local WC system may only be switched over in a BAPS program in automatic mode in a non–permanent process.**

Functions

With the assignment of the BAPS standard variable WC_UR the activation of a special local WC system can be suspended. In this case the name 'WCur active' is assigned by the operating system.

```
Rob1.WC_SYSTEM =  WC_UR
```

7.20.6 Machine parameter P313: WCSYS-ROB assignment

For the general definition of the workpiece coordinate system, six components are used, as mentioned before. If the kinematic has less than six degrees of freedom, it is not possible to compensate for all displacements/rotations of the workpiece coordinate system.

In this case, the rotations of the workpiece coordinate system opposite the original coordinate system is defined as follows:

- Pose_O1 rotation around the x–axis
- Pose_O2 rotation around the y–axis
- Pose_O3 rotation around the z–axis

Via machine parameter P313 the control is informed about

- which of the 6 common workpiece coordinates are considered by the mechanics
- which robot coordinate corresponds to which workpiece coordinate

☞ **Machine parameter P313 is only necessary with kinematics with missing degrees of freedom.**

Example 1: 4–axis Scara

A 4–axis Scara may move in x, y, z – direction and may perform a rotation around the z–axis. Thus the machine parameter P313 has the following assignment:

```
WCSYS–x = 1
          y = 2
          z = 3
          a = 0
          b = 0
          c = 4
```

Functions

Example 2: 3–axis roof machine with belt

A 3–axis roof machine may move in y, z – direction and may perform a rotation around the y–axis. The belt moves to x–direction. A BAPS point has the following components:
P1 = (yc, zc, bc, blt)
The belt coordinate (= 4. coord) is added alternatively to the workpiece coordinates. In order that, the parameter P313 has the following assignment:

WCSYS–x = 4
y = 1
z = 2
a = 0
b = 3
c = 0

P313: WCSYS-ROB assignment		
WCSYS-x	1	The first WC-PNT component (SCx) corresponds to WCSYS-x
WCSYS-y	2	The second WC-PNT component (SCy) corresponds to WCSYS-y
WCSYS-z	3	The third WC-PNT component (SCz) corresponds to WCSYS-z
WCSYS-a	0	an orientation change Δa is not compensated
WCSYS-b	0	an orientation change Δb is not compensated
WCSYS-c	4	The fourth WC-PNT component (SCc) corresponds to WCSYS-c

Example: 3-axis roof machine

Faulty inputs in the machine parameter P313 can cause the following runtime messages:

WCS: P313 value not permitted	The entered value is negative or greater than the axis number (plus possibly belt number).
WCS: no pos offset	No position offset is entered, i. e. WCSYS-X = WCSYS-Y = WCSYS-Z = 0.
WCS: too many angles	More than one orientation offset WCSYS-a, WCSYS-b, WCSYS-c has been defined. At this place, a maximum of one orientation offset is permitted (see above restriction). In the BAPS program it is, however, possible to program several rotations via the WC_FRAME.

Functions

7.20.7 Library functions

The class 2000 (rhoKinematics [rK]) of the **rho4 library functions** contain 5 functions, which make the use of common WC systems easier. For detailed description of the parameter lists, see header files rk.h, resp. rk.inc.

- **2071 : rKxGWcSystem**
  supplies name and coordinates of the active WC system
- **2072 : rKSWcSystem**
  activates the WC system selected by name. The name and the corresponding coordinate values must be defined in the system file ”WCSYST.DAT”
- **2073 : rKCWpFrame**
  calculates from 3 pallet positions P0, P1, P2, recorded in coordinates of the original coordinate system WC_{ur}, the appropriate zero point of the local WC system WC_{loc} (inclusive rotation) opposite to the original coordinate system (see Fig. 3)

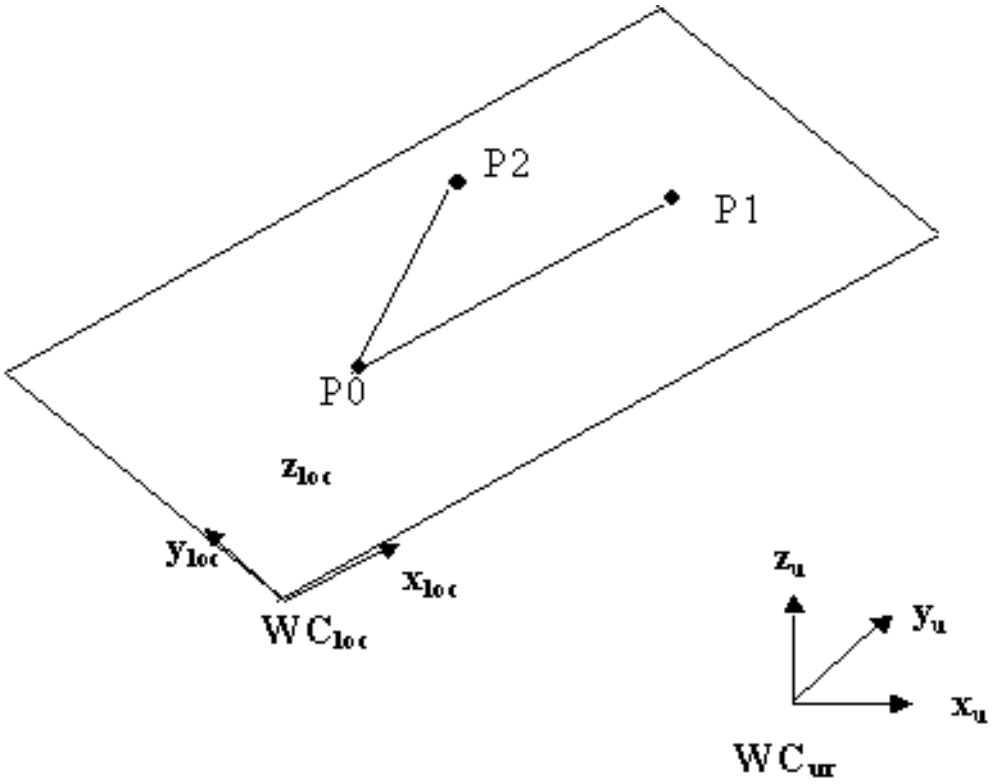
Input–Par.: P0, P1, P2 in WC_{ur}, P0loc in WC_{loc} (see Fig. 3)
Output–Par.: Variable of type WC_FRAME

For calculation, the following requirements must be fulfilled:

- The 3 points P0, P1, P2 must lie in the same pallet plane, i.e. they have the same z_{loc}–coordinate
- The vector P0P1 lies parallel to the x–axis
- The (x, y, z)–coordinates of the point P0 in the local WC system WC_{loc} must be known

Functions

Fig. 3



The two following library functions may be used with 6–axis kinematics. On missing degrees of freedom they are mathematical under determined and make no sense (see also Fig. 4):

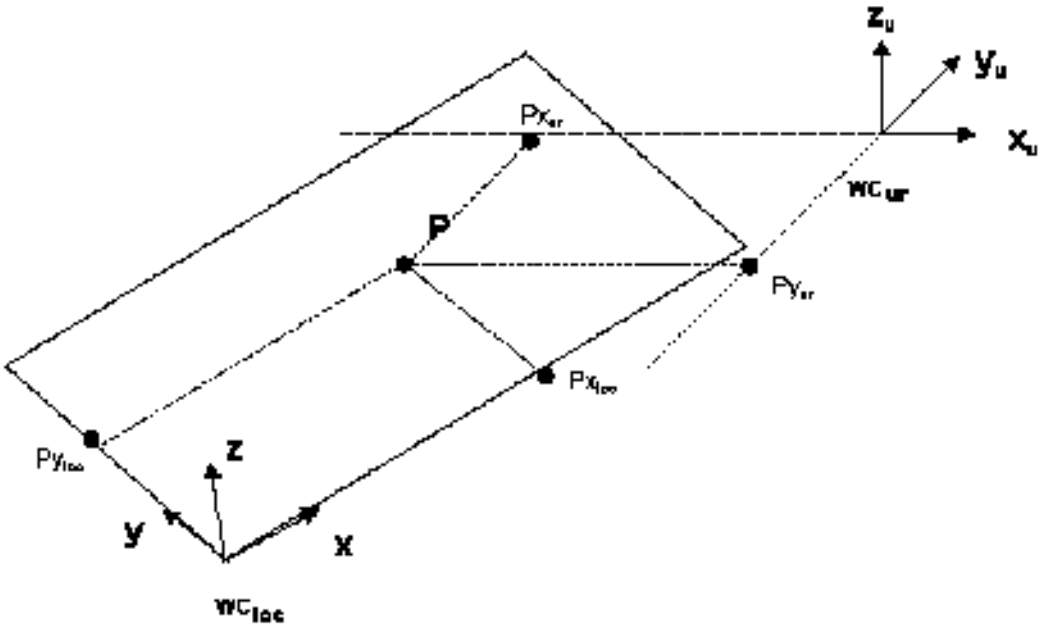
- **2074 : rKTrfLocOri**
  calculates of the local coordinates of a point the appropriate original coordinates
- **2075 : rKTrfOriLoc**
  calculates of the original coordinates of a point the appropriate local coordinates

Both functions work name oriented relating to the local WC system. The name with the appropriate coordinates is expected in the system file WCSYST.DAT of the control (both in BAPS– and in the DLL–function).

P_{loc} corresponds to point P in coordinates of the local WC system.
P_{ur} is the same point in coordinates of the original coordinate system WC_{ur}.

Functions

Fig. 4



Example :

The pallet positions PalPos1, PalPos2 of pallet ”deposit_Pal” are acquired offline (e.g. from a CAD–system). Thus they are coordinates of a local WC system. To move to this coordinates in a BAPS program, they must be converted into original coordinates. This is made with library function 2074.

Library functions are unaware of the standard type ’POINT’. For easier handling in the BAPS program, only a BAPS conversion program is needed, which copies a point to a REAL array and vice versa (see below, resp. C:\Bosch\rho4\Example\Baps\LocUr_K1.qll).

```
;external Subroutine 'Loc_Ur'

EXTERNAL;Loc_Ur (INTEGER: Ret Rob1.POINT: Ploc TEXT: WcName Rob1.POINT: Pur)

POINT:   PalPos1, PalPos2, Pur1, Pur2

TEXT:    deposit_Pal

INTEGER: RetCode

         .
         .

Loc_Ur (RetCode, deposit_Pal, PalPos1, Pur1)

MOVE LINEAR Pur1
```

Functions

```
Loc_Ur (RetCode, deposit_Pal, PalPos2, Pur2)

MOVE LINEAR Pur2
```

This functionality is implicit executed inside in the operating system, if a local coordinate system is activated by the WC_SYSTEM instruction.

```
WC_FRAME: pallet1

     .
     .

Rob1.WC_SYSTEM = pallet1

MOVE LINEAR PalPos1

MOVE LINEAR PalPos2



;-------------------------------------------------------------------
;-------------------------------------------------------------------
;---   External main program for kinematics 1:
;---   Tranformation WC-Loc -> WC-Ur:
;---
;---   Transforms an existing point in a local WC system (workpiece coord.)
;---   into the original coordinate system
;-------------------------------------------------------------------
;-------------------------------------------------------------------


;;CONTROL       = RHO4

;;KINEMATICS    : (1=Rob1, 2=Rob2)

;;Rob1.WC_NAMES = xx,yy,zz,o1,o2,o3

;;Rob2.WC_NAMES = xx,yy,zz,o1,o2,o3


PROGRAM Loc_Ur (INTEGER: Ret Rob1.POINT: PLoc  TEXT: WcName  Rob1.POINT: Pur)

;;INCLUDE RMAIN.INC

;;INCLUDE RK.INC


BEGIN

 LocUr (Ret, PLoc, WcName, Pur)

PROGRAM_END
```

Functions

```
;-------------------------------------------------------------------
SUBROUTINE LocUr (INTEGER: Ret  Rob1.POINT: PLoc  TEXT: WcName  Rob1.POINT: Pur)
;-------------------------------------------------------------------
; The real functionality is realized in a subroutine,so that the routine
; is reentrent. The following declared variables lie in this case on the
; stack; in an external main program it would be global variables.
;-------------------------------------------------------------------


INTEGER       : i

TKTrfLocOri   : PKTrfLocOri


BEGIN

 PKTrfLocOri.ChannelId = 0

 PKTrfLocOri.KinNr     = 1

 i=0

 REPEAT 12 TIMES

  i=i+1

  PKTrfLocOri.WcSysName[i] = WcName[i]

 REPEAT_END

 PKTrfLocOri.PntLoc[1] = PLoc.xx

 PKTrfLocOri.PntLoc[2] = PLoc.yy

 PKTrfLocOri.PntLoc[3] = PLoc.zz

 PKTrfLocOri.PntLoc[4] = PLoc.o1

 PKTrfLocOri.PntLoc[5] = PLoc.o2

 PKTrfLocOri.PntLoc[6] = PLoc.o3


 Ret = rKTrfLocOri (PKTrfLocOri)


 PUr.xx = PKTrfLocOri.PntOri[1]

 PUr.yy = PKTrfLocOri.PntOri[2]

 PUr.zz = PKTrfLocOri.PntOri[3]

 PUr.o1 = PKTrfLocOri.PntOri[4]

 PUr.o2 = PKTrfLocOri.PntOri[5]

 PUr.o3 = PKTrfLocOri.PntOri[6]

SUB_END
```

Functions

7.20.8 Workpiece coordinate system in a BAPS program

ATTENTION
When starting the program, it has in any case to be ensured that the correct workpiece coordinate system is activated. If programs are executed with false workpiece coordinate systems, unexpected movements can occur. The same effect can occur if different workpiece coordinate systems are used for a taught world coordinate point for the Teach In and for the execution of the program!

World coordinate points (WC points)

All world coordinate points used in a BAPS program refer to the local workpiece coordinate system, regardless of the fact whether they are textually programmed offline or taught online.

By activating a WC system, these points are transformed within the control into original coordinates.

☞ **In PNT–Files, WC points are basically stored in values of the active local workpiece coordinate system.**

Joint coordinate points (JC points)

All points programmed in joint coordinates (JC_POINT, @–points) remain unchanged when switching to a specific workpiece coordinate system. Joint coordinate points describe the axis position of the robot and are therefore independent from the active workpiece coordinate system, e. g. @homepos, @(0, 0, 0, 0, 0, 0).

POS, @POS, @MPOS

The values achieved by the assignment

- act_pos = POS

depend on the currently active workpiece coordinate system. POS describes the actual position of the robot in local workpiece coordinates.

@POS and @MPOS specify the axis position of the robot each. They are independent from the active workpiece coordinate system.

LIMIT_MIN, LIMIT_MAX

By means of the assignments

- kin1.LIMIT_MIN = min_position

Functions

- kin1.LIMIT_MAX = max_position

the workspace limits can be defined. In form, min_position and max_position are normal world coordinate points. They refer, as all the WC points, to the local workpiece coordinate system. It results that LIMIT_MIN and LIMIT_MAX are programmed in workpiece coordinates.

Standard functions JC( ), WC( )

The standard function JC supplies a conversion from world into joint coordinates, e. g.

- @start_pos = JC(start_pos).

As result, the arm position of the kinematics is achieved. Since the starting point is a world coordinate point, the result depends on the active workpiece coordinate system.

The standard function WC supplies a conversion from joint into world coordinates, e. g.

- start_pos = WC(@start_pos).

The result are coordinates referred to the local workpiece coordinate system, i. e. the result depends on the active workpiece coordinate system.

Special functions 1, 2, 56, 57: IO/PPO logic

By means of this special functions, position-specific outputs of process parameters can be programmed.

The programmed switching on/off coordinates refer to the local workpiece coordinate system, i. e. it is for example always switched at the same position. When looking at the original coordinate system (robot coordinate system), the switch points change with different workpiece coordinate systems.

Special function17: MIRROR

This option permits mirroring any points at the axes of the world coordinate system. As with all other programmed world coordinate points, mirroring refers to the local workpiece coordinate system.

Displacement of the world coordinate system (P310)

The zero point of the robot coordinate system is defined by means of the machine parameter P310, i. e. it defines the original coordinate system. The coordinates of the active workpiece coordinate system (i.e. the coordinates transfered by the instruction WC_SYSTEM) always refer to the original coordinate system.

Functions

Limit switch monitoring

In automatic operation, the software limit switches are monitored in joint coordinates (see machine parameter P204, P205). The monitoring is independent from the currently active workpiece coordinate system.

Program end, program break, run-up of control

A change of the coordinate system is only possible via the selection in a non–permanent BAPS program. After program end, the workpiece coordinate system remains last selected active.

In case of a program break, e. g. Emergency stop input, auto-manual switching, reset etc., the workpiece coordinate system active at the time of program break remains active.

After the run-up of the control, the original coordinate system is active, i. e. WC_SYSTEM = WC_UR.

Axis display

The axis display on the PHG2000 under mode 7.1 have the following effects:

- In joint coordinates (JC), always the current axis position is displayed. It is independent from the active workpiece coordinate system.
- In world coordinates (WC), the values are displayed in local workpiece coordinates. The display depends on the active workpiece coordinate system.
- In original coordinates (OC) the positions are displayed in the original coordinate system. The display is independent from the active workpiece coordinate system. This level is only offered when WC_SYSTEM <> WC_UR, i. e. when a local workpiece coordinate system is activated.

Display of the active WC_SYSTEM

On the PHG2000 under mode 7.1 a level exists

on which the coordinates of the zero point of the currently active workpiece coordinate system are displayed. If all coordinates equal zero, no workpiece coordinate system is activated, i. e. WC_SYSTEM = WC_UR.

Functions

7.20.9 Selection and function in manual mode

The function 'workpiece coordinate system' acts for each kinematic globally, i. e. exceeding program and process limits. If individual part programs are called at several places of the overall process, it has to be made sure, as far as the program technique is concerned, that they always work with the same workpiece coordinate system.

As mentioned before, a change of the workpiece coordinate system is only possible via the selection in a non-permanent BAPS program. Since after the program end the workpiece coordinate system actived last remains active.

In manual mode it is possible by the PHG2000, to display the active workpiece coordinate system, resp. to select a special one.

In PHG-Mode 2 (Manual) and 4.2 (Teach In), the left key of the second row (from the top) is used as a function key (softkey, keycode "WAIT"). Entering this key, the following mask is displayed:

WCSYST.DAT	_X O1	_Y O2	_Z O3
deposit_Pal1	100,123 40.345	200.123 50.567	300.234 60.321
pickup_Pal1	654,111 77.777	543.222 88.888	432.333 99.999
pickup_Pal2	454,111 77.777	343.222 88.888	332.333 99.999
pickup_Pal3	120,000 44.444	222.222 55.555	333.333 66.666
deposit_Pal2	254,111 77.777	143.222 88.888	232.333 99.999
deposit_Pal3	100,123 40.345	200.123 50.567	300.234 60.321
pallet2	110.123 45.345	220.123 55.567	330.234 65.321

In the last two lines the active local WC system (workpiece) is displayed.

In the middle part of the mask the content of the system file WCSYST.DAT is displayed.

The invers displayed WC system can be activated by the <Enter> - key. Thereby the permission key (deadman) must be released.

Using the following key combinations makes it possible to navigate with the cursor within the WC system list:

Functions

Standard key assignment

< Cursor–Up >	moves cursor up	< Shift > < 5 >
< Cursor–Down >	moves cursor down	< Shift > < . >
< '<' >	moves cursor one page up	< Alt > < 7 >
< '>' >	moves cursor one page down	< Alt > < 8 >
< BEGIN >	moves Cursor to begin of the list	< BEGIN >
< END >	moves cursor to end of the list	< END >
< Cursor–Left >	exit mask	< Shift > < 1 >

Movements in jog mode

Movements in jog mode (Mode 2) or Teach in (Mode 4.2) depend on the selected coordinate system (see chapter 7.20.2).

- **JC:** In joint coordinates in principle an axis–related movement take place
- **WC:** In world coordinates a linear movement in coordinates of the active local WC system (workpiece coordinates) take place
- **OC:** In original coordinates a linear movement in the fixed robot original coordinate system (independent from the active WC system) take place
- **GC:** In gripper coordinates a linear movement in the co–moved gripper coordinate system (independent from the active WC system) take place

7.20.10 Examples for special workpiece coordinates

The car body data of a car is supplied by a CAD system. The zero point of the local workpiece coordinate system (in this case the car body coordinate system) is placed in the front axle of the car. For processing, the car body is drawn on a carriage through the working area of the robot.

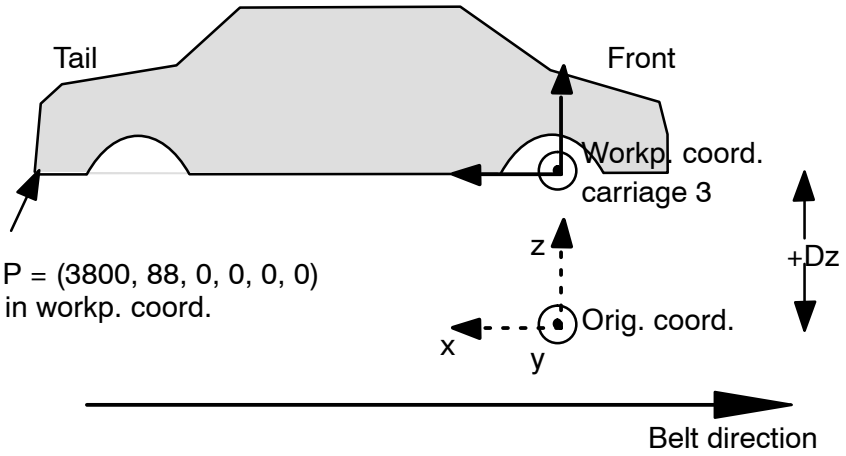
With the instruction WC_SYSTEM, the control is communicated the offset of the zero point of the local workpiece coordinate system in relation to the zero point of the original coordinate system (robot coordinate system).

Example 1: pure height offset

Example 1 shows the case of a mere height offset of +Dz, i. e. on carriage 3 the car body is higher than on the standard carriage. The workpiece coordinate system of the carriage 3 would have to be defined as follows:

```
carr3=WC_FRAME(0,0,+Dz,0,0,0)
```

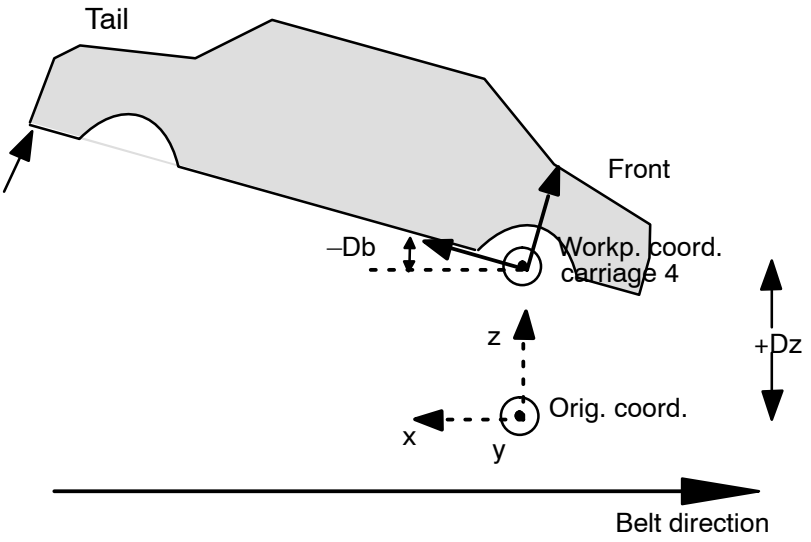
Functions



Example 2: height offset and rotation

In this example, the car body is compared with the standard carriage higher and is turned by –Db around the y–axis. The workpiece coordinate system of carriage 2 would have to be defined as follows:

```
carr2=WC_FRAME(0,0,+Dz,0,-Db,0)
```



8 Movement statements

Robot movements are initiated by movement statements. Movement statements describe the movement of the robot from a current position and orientation to a destination point.

In the rho4 a distinction is made between direct movement statements and movement statements which influence movement.

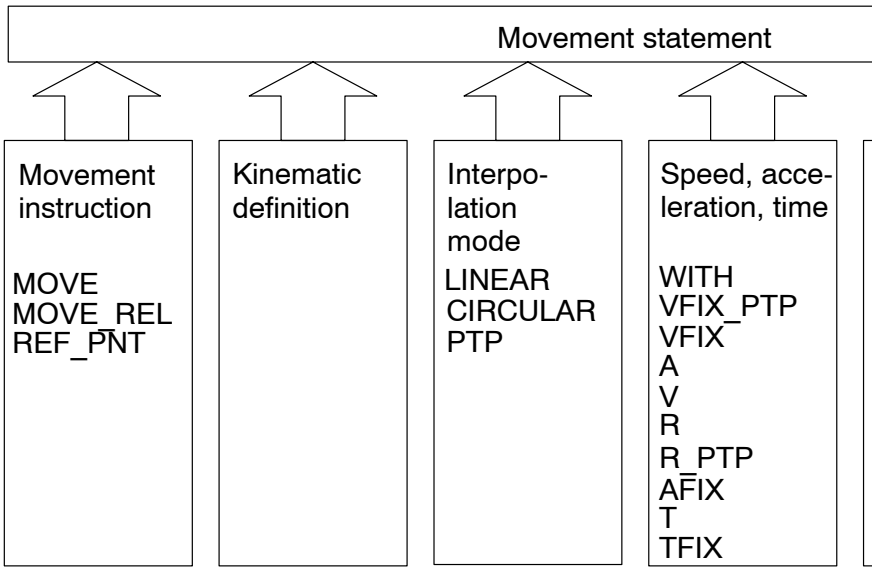
8.1 Direct movement statements

A direct movement statement is made up of the following individual statements:

- Movement instruction
- Kinematic definition
- Interpolation mode
- Speed/acceleration
- Abort condition
- Destination

The movement instruction and the destination must always be programmed.

Information on the kinematic, interpolation mode, speed/acceleration/time and abort conditions may be omitted. The control then automatically uses internally stored statements for default values.



Movement statements

8.1.1 Movement instructions

The control knows the following movement instructions:

- MOVE
- MOVE_REL
- REF_PNT

MOVE

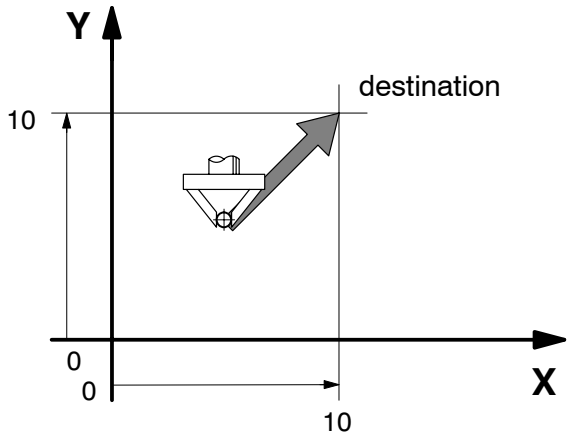
Syntax:

```
MOVE[kinematic][interpolation mode][additional info][abort condition]
    [TO]point string|VIA point string[TO point string]
```

The control interprets all position values programmed after MOVE as absolute dimensions. The coordinate values refer to the zero point of the world or joint coordinate system.

Example:

```
destination=(10,10)

MOVE destination
```



Within the movement instruction, additional information decides whether the robot approaches the programmed points exactly – i. e. within the defined tolerance – or whether it only travels past the points – without halt –. This information consists of VIA and TO for movements in absolute dimensions with MOVE.

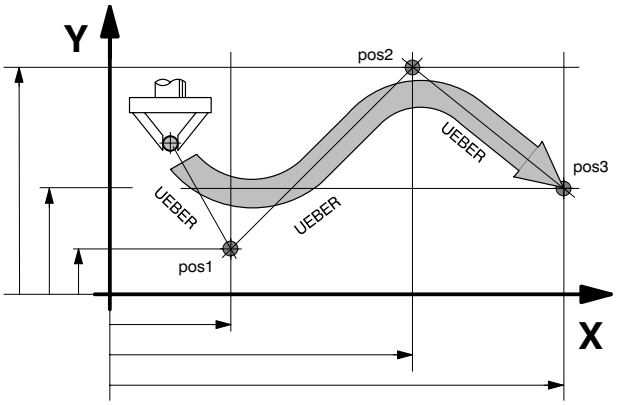
Movement statements

MOVE VIA ... (passing)

The robot travels past the positions without an intermediate halt.

Syntax:

```
MOVE VIA pos1,pos2,pos3
```



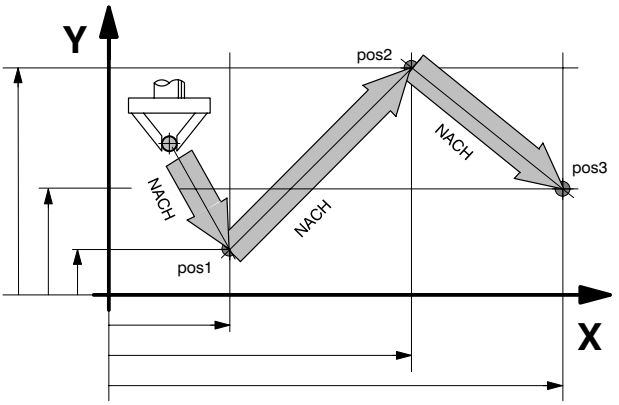
MOVE TO ...

The robot travels to the positions successively with an intermediate halt.

The word TO can be omitted when programming the movement instruction. The control generates the instruction TO automatically.

Syntax:

```
MOVE TO pos1,pos2,pos3
```



It is possible to link VIA and TO within a movement instruction:

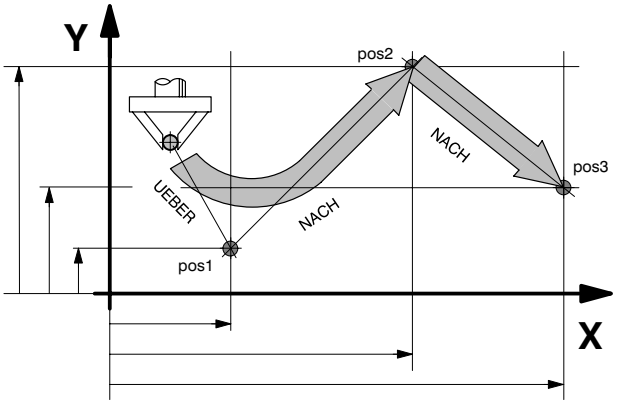
Movement statements

MOVE VIA ... TO ...

The robot travels past position 1 without an intermediate halt and then travels successively to the positions 2 and 3 with an intermediate halt.

Syntax:

```
MOVE VIA pos1 TO pos2,pos3
```



MOVE_REL

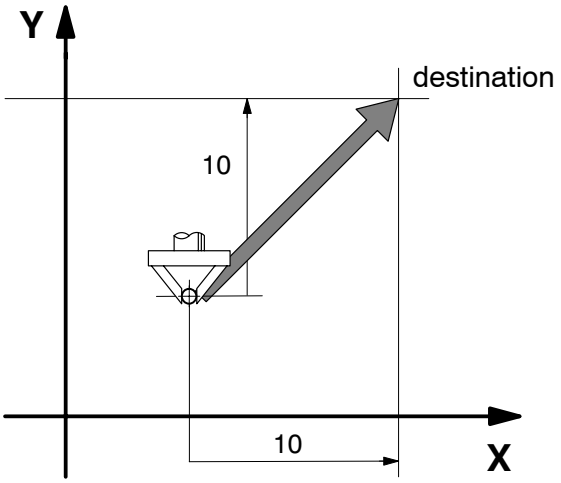
Syntax:

```
MOVE_REL[kinematic_variable][interpolation mode][additional info][abort condition]
        ([EXACT]point string|APPROX point string[EXACT point string] )
```

The control interprets all position information programmed after MOVE_REL as incremental dimensions. The coordinate values in this case represent distance values in the respective coordinate system and refer to the current actual position of the robot.

Syntax:

```
destination=(10,10)MOVE_REL destination
```



Movement statements

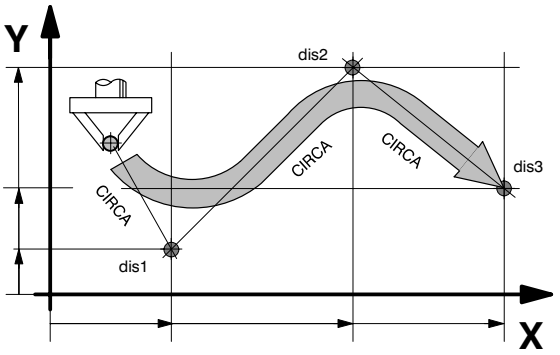
For movements in incremental dimensions with MOVE_REL the words APPROX and EXACT decide with respect to exact positioning or travel past.

MOVE_REL APPROX ...

The robot travels past the positions defined in incremental dimensions without an intermediate halt.

Syntax:

```
MOVE_REL APPROX dis1,dis2,dis3
```



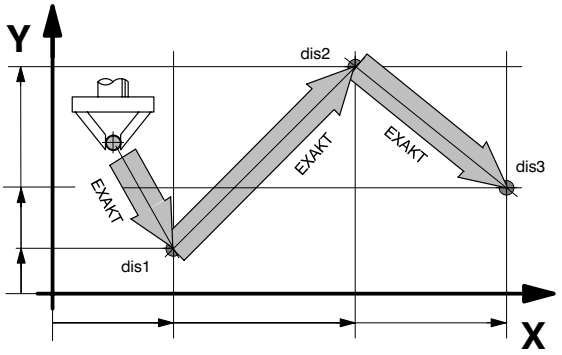
MOVE_REL EXACT ...

The robot travels to the positions defined in incremental dimensions successively with an intermediate halt in each case.

The word EXACT can be omitted when programming the movement instruction. The control generates the instruction automatically.

Syntax:

```
MOVE_REL EXACT dis1,dis2,dis3
```



It is possible to link APPROX and EXACT within the movement instruction.

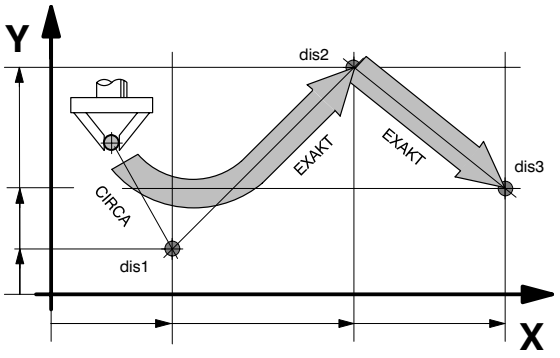
Movement statements

MOVE_REL APPROX ... EXACT ...

The robot travels past the first position without an intermediate halt and then travels successively to the next positions 2 and 3 with an intermediate halt in each case.

Syntax:

```
MOVE_REL APPROX dis1 EXACT dis2,dis3
```



REF_PNT

Syntax:

```
REF_PNT[kinematic_variable](axis_number{||,})
```

REF_PNT is a special movement statement which is used for programmed reference point travel of the axes without having to travel the axes to the reference point after a system start-up. The machine axes which are to travel simultaneously to their reference points are specified in brakkets after the REF_PNT statement.

The values in the bracket refer to the axis number of the respective kinematic, the kinematic itself can be specified before the bracket.

Example:

```
REF_PNT kin_1(1,2,3)

;;KINEMATICS=kin_2

REF_PNT(4,5)

REF_PNT kin_3(4,5)
```

Destination point designations

The designations of the points for position and orientation can be freely selected, see section 4.4. It is thus possible to assign the names pallet1, pallet2 ... to the pallet points, for example.

For simplicity's sake, point information in absolute dimensions is designated by 'pos' on the following pages. e. g. MOVE pos.

Point information in incremental dimensions is designated by 'dis' (distance information), e. g. MOVE_REL dis.

8.1.2 Kinematic definition

If the program controls more than one kinematic, it is necessary to specify in the movement statement which kinematic is to be moved.

Example:

```
MOVE sr6 TO corner

MOVE robot_2 VIA prepos TO home
```

☞ **Only one kinematic definition must be made in a movement statement.**

If the kinematic is missing in the movement statement, the kinematic stored in the operating system or the last selected kinematic in the program is moved.

Example:

```
;;KINEMATICS=robot_2

MOVE VIA prepos TO home
```

The kinematic preselection is performed using the compiler statement:

```
;;KINEMATICS=kinematic name
```

The kinematic defined in this way is then valid for all subsequent movement statements without kinematic definition. It continues to be valid until it is overwritten by another kinematic preselection.

☞ **The kinematic preselection refers to the following line, not to the program sequence (subroutine call, jumps).**

Kinematic names are defined using a compiler statement, see section 2.3.1.

8.1.3 Interpolation mode

The control must know on which path the robot must approach the next position. In order to define this path, there are three interpolation modes:

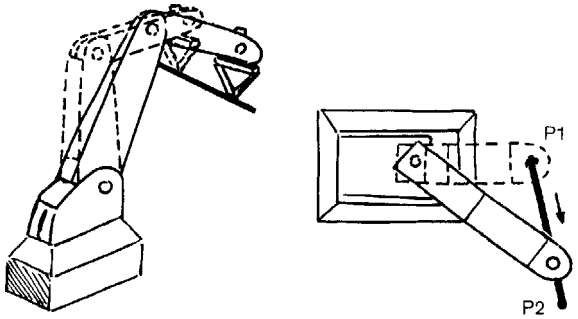
- LINEAR (straight line in space)

Movement statements

- CIRCULAR (circular path in space)
- PTP (synchronous point to point) the path depends on the robot design, synchronous means that all axes reach their programmed destination point at the same time)

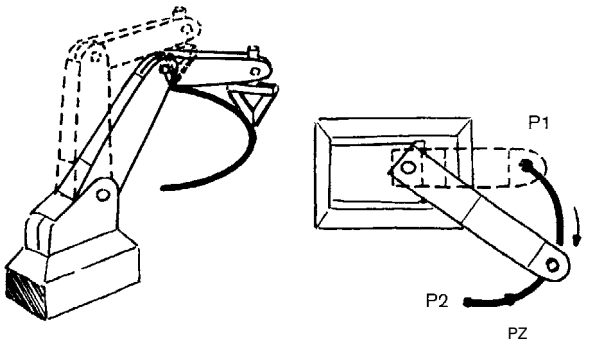
LINEAR interpolation mode

The robot travels to the destination point on a straight line. The straight line is geometrically defined by two points. Since the control knows the current position P1 of the robot, it is sufficient to specify a destination point P2.



CIRCULAR interpolation mode

The robot travels to the destination point on a circular path in space. The circle is geometrically defined by three points. In addition to the destination point PZ, it is thus also necessary to specify an intermediate point P2 so that the control can unambiguously calculate the circular path. The point P1 is the last-approached point and known to the control. The intermediate point P2 is a point on the arc , which is travelled by the robot.



☞ **The orientation of the intermediate point (PZ) does not have any influence on the movement sequence.**

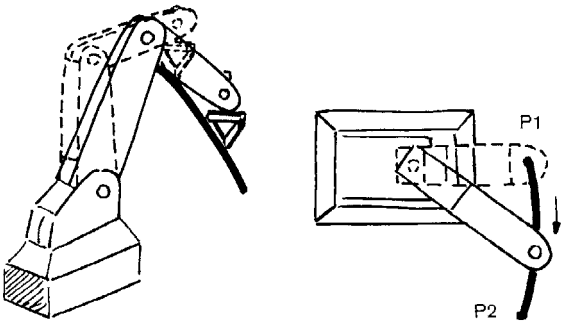
Movement statements

PTP interpolation mode

PTP = from point to point.

The control calculates the movement of all axes so that they simultaneously start and end movement . This is generally also referred to as synchronous PTP. This results in travel which is not further definded, dependent on the lever ratio of the robot arms and the points P1 and P2.

It is sufficient to specify a destination point for the PTP procedure



Statement-specific interpolation mode

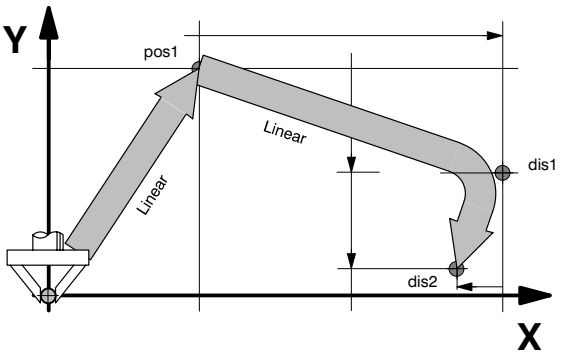
The interpolation mode is programmed in the movement statement if a specific interpolation mode is to be valid for one movement statement only.

The interpolation mode is contained in the movement instruction, directly following MOVE or MOVE_REL.

☞ **There must be only one interpolation mode within a movement function!**

Example LINEAR

```
MOVE LINEAR TO pos1

MOVE_REL LINEAR APPROX dis1 EXACT dis2
```



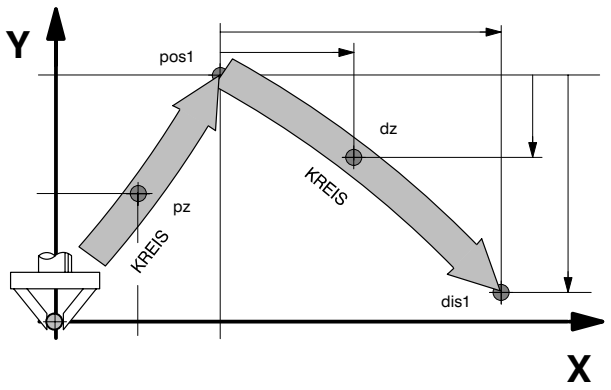
Movement statements

Example CIRCULAR

The point information, such as intermediate point pz or dz, end position pos1 or dis1, required for CIRCULAR interpolation must be written in brackets and separated by a comma.

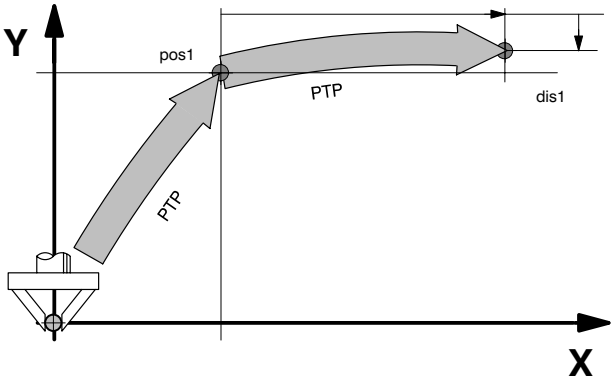
Syntax:

```
MOVE CIRCULAR TO (pz,pos1)

MOVE_REL CIRCULAR EXACT (dz,dis1)
```



Example PTP

```
MOVE PTP TO pos1

MOVE_REL PTP EXACT dis1
```



☞ **The control automatically selects PTP if no interpolation mode is specified and if no global interpolation mode has been specified.**

Movement statements

Global interpolation mode

If an interpolation mode is to be valid for several movement statements, it can be specified as a global interpolation mode.

The global interpolation mode is specified with the following compiler statement.

Syntax:

```
;;INT      ;Interpolation mode applies to all kinematics

;;KIN1.INT ;Interpolation mode applies to respective kinematic
```

The interpolation mode defined in this way then applies to all subsequent movement statements which do not contain any specific interpolation definitions.

Example

```
;;INT=LINEAR MOVE pos1 ;The robot travels to the position pos1 and the point
                       ;defined via dis1 on a straight line in each case.

MOVE_REL dis1 MOVE CIRCULAR(pz,pos2) ;Travel to point pos2, on the other hand,
                                     ;takes place on a circular path.
```

The global interpolation mode remains valid until it is replaced by another interpolation mode.

Example

```
;;KIN1.INT=LINEAR

MOVE pos1 ;The robot travels the axes of kinematic 1 to the positions pos1

MOVE pos2 ;and pos2 on a straight line in each case

;;KIN1.INT=CIRCULAR

MOVE (pz1,pos3) ;The positions pos3 and pos4 and all other positions are

MOVE (pz2,pos4) ;approached on a circular path after definition of CIRCULAR
                ;interpolation mode
```

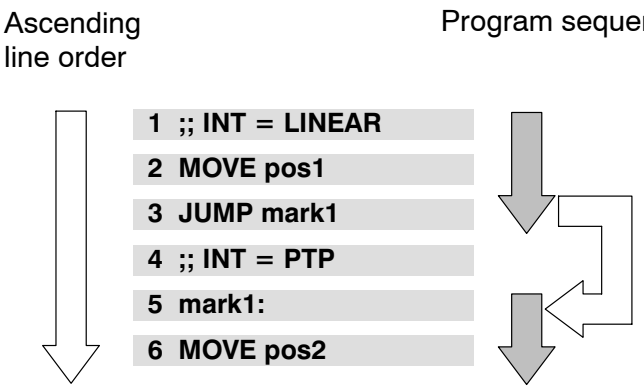
☞ **In case of a global definition of CIRCULAR interpolation, the point values within the movement statement for which the interpolation definition is to be valid must be point pairs.**

☞ **If several global interpolation modes are programmed, assignment of the interpolation mode takes place in ascending line order and not in accordance with the program sequence (subroutine, jumps etc.).**

Movement statements

Example

The position pos2 is approached with PTP, although the compiler statement ;;INT = PTP was equipped in the program run.



8.1.4 Destinations

The robot always travels from its current position to the programmed destination point. The path for this must be defined by the interpolation mode, so that the control knows how it is to calculate the path to the destination point.

In the same way, the path must be defined unambiguously in geometric terms. This is done by means of the point values. In the case of circular interpolation, for example, it is necessary to define an auxiliary point in addition to the destination point in order to clearly define the circular path.

8.1.5 Speed, acceleration and time

In addition to informing the control of the position of the destination point and the path, you must also tell how fast or in what time the robot should travel to the destination point.

The control therefore requires information about the duration of the movement or the path speed of the robot.

A value can be entered for acceleration in order to determine how quickly the robot is to reach this path speed.

Speed

The speed has different designations, units and input ranges, depending on the path to be travelled by the robot.

Movement statements

If you do not program any values for the speed, the robot travels with speed values which are internally stored in the control. Stored are 25 mm/s for path interpolation resp. 10 % of the maximum speed for synchronous PTP interpolation.

The speed V_PTP is specified as a decimal factor of the maximum axis speed for synchronous PTP interpolation. It is also possible to enter values in percent, e. g. V_PTP = 80 % is equivalent to V_PTP = 0.8.

Speeds for LINEAR and CIRCULAR

Designation	V (override active) VFIX (override not active)
Unit of measure	mm/s
Input range	1 to 2000 mm/s (depending on machine parameter P102)
Power-on condition	25 mm/s

Speeds for PTP

Designation	V_PTP (override active) VFIX_PTP (override not active)
Unit of measure	decimal factor
Input range	0.0001 to 9.9999 (0.01 % to 999.99 %) (depending on machine parameter P103)
Power-on condition	(0.1)

Programming possibilities

The speed can be programmed as a

- global speed definition,
- statement-specific speed definition,
- statement-specific time definition,
- with and without speed override.

Global speed definition

If the path speed remains the same for the whole program or for a large section, it is sensible to define the speed as a global speed.

The speed value remains valid until it is changed by a further global speed definition.

Movement statements

Example

```
V750                ;V = decimal expression. All subsequent
                    ;movement functions with the interpolation modes LINEAR and
                    ;CIRCULAR have the following speed value: V = 750 mm/s

MOVE LINEAR pos1

MOVE CIRCULAR (pz,pe)

V=750               ;The speed for linear approach to the points pos1

MOVE LINEAR pos1    ;and pos2 is V = 750 mm/s

MOVE LINEAR pos2

V=300

MOVE LINEAR place3 ;The points place3 and place4 and all other points are

MOVE LINEAR place4 ;approached at the speed V =300 mm/s
```

Statement-specific speed definition

If the path speed is to be valid for only movement statement, the speed is accordingly defined as a statement-specific speed, i. e. with the movement statement.

The speed designation and value assignment are part of the movement function for which the speed is to be valid.

Programming takes place following the interpolation mode with the key word WITH.

Example LINEAR:

```
MOVE_REL LINEAR WITH V=500 EXACT dis
```

Example PTP:

```
MOVE PTP WITH V_PTP=70% TO pos
```

☞ **The global speed inputs do not have any influence on statement-specific inputs.**

Movement statements

Speed override

Speed and time values can be changed with speed override VFACTOR.

The speed override is a factor by which the control automatically multiplies all speed definitions.

Time inputs are divided by the VFACTOR. The values calculated in this way then apply to the subsequent movement functions.

☞ **Inputs with VFIX and TFIX are not taken into account.**

The factor relates to the defined speed or time values in the program. The factor can also be specified in percent, e. g. VFACTOR = 180 % is equivalent to VFACTOR = 1.8.

Speed override

Designation	VFACTOR
Unit of measure	%
Input range	0.01 to 999.99 % (depending on machine parameters P23 and P119)
Power-on condition	1.0 (100 %)

☞ **AFACTOR and VFACTOR are reset to 1.0 (100 %) with RESET and by program abort.**

The global Vfactor (P23) acts on all kinematics, the local VFACTOR (P119) acts only on the respective kinematic.

Designation and value assignment form a separate statement.

Example

```
VFACTOR=180%                        ;The VFACTOR of 180% (= factor 1.8) acts
                                    ;on the path speed to the positions 1, 2,
                                    ;3 and 4. pos1 is approached with 180% of Vmax
                                    ;(power-on condition 10%, factor 1.8)

MOVE PTP TO pos1

V=200

MOVE LINEAR TO pos2                 ;pos2 is approached with 360 mm/s

MOVE LINEAR WITH V=100 TO pos3      ;pos3 is approached with 180 mm/s

MOVE LINEAR WITH T=10 TO pos4       ;pos4 is approached in a time of 5.6 seconds
                                    ;(10 / 1.8 = 5.6)
```

Movement statements

Acceleration

The control recognizes from the acceleration values how quickly the robot has to reach the speed defined for it.

☞ **Acceleration values can be programmed only for LINEAR and CIRCULAR interpolation.**

The designations, units of measure and input ranges are contained in the following table.

Acceleration vor LINEAR and circular interpolation

Designation	A (override possible) AFIX (override not possible)
Unit of measure	mm/s^2
Input range	0.001 to 32000 mm/s^2
Power-on condition	10 mm/s^2

In the PTP method, the robot accelerates with the maximum values defined in the machine parameters. It is possible to change the acceleration value with the AFACTOR.

Programming possibilities

In case of LINEAR and CIRCULAR interpolation, acceleration input is possible as a

- global definition
- statement-specific definition
- acceleration override

The acceleration override is also active for PTP interpolation.

Global acceleration definition

Interpolation modes: LINEAR, CIRCULAR

If the acceleration value remains the same for the whole program or a larger section of it, it is sensible to program a global acceleration value.

The designation and value assignments form a separate statement at the beginning of the program or program section.

The acceleration value remains valid until it is replaced by another global value.

Movement statements

Example

```
A=30                    ;All subsequent movement functions with the
                        ;interpolation modes LINEAR and CIRCULAR have the
                        ;value A=30 mm/s2 defined as acceleration

V=100                   ;The acceleration until the path speed V=100
                        ;is reached is A=30 mm/s2 when appproaching the points
                       ;pos1 to pos3

MOVE CIRCULAR (pz,pos1)

MOVE LINEAR pos2

MOVE LINEAR pos3

A=15                    ;The positions pos4, pos6 and all other positions with
                        ;LINEAR and CIRCULAR interpolation have an acceleration
                        ;value of A=15 mm/s2

MOVE LINEAR pos4

MOVE PTP pos5

MOVE LINEAR pos6
```

☞ **pos5 is approached with PTP interpolation. The acceleration value A = 15 mm/s² is not valid here.**

Statement-specific acceleration definition

The acceleration is defined as a local value if a path acceleration value is to be active only with one movement instruction.

Acceleration definition

Designation	A
Unit of measure	mm/s²
Input range	0.001 to 32000
Power-on condition	10 mm/s²

☞ **Local acceleration values are possible only for LINEAR and CIRCULAR interpolation.**

Movement statements

The designation and value assignments are part of the movement function, for which the acceleration is to be valid. Programming takes place following the interpolation mode with the key word WITH.

Example LINEAR:

```
MOVE LINEAR WITH A=30 TO pos
```

Example CIRCULAR:

```
MOVE_REL CIRCULAR WITH AFIX=25 EXACT (dz,dis1)
```

☞ **The global acceleration values do not have any influence on statement-specific values.**

It is possible to locally define the speed and acceleration together within a movement function. The two values are separated by a comma when programming. They may be entered in any order.

Example CIRCULAR:

```
MOVE_REL CIRCULAR WITH V=120, A=35 EXACT (dz,dis1)
```

Acceleration override

Acceleration values can be influenced once more with the acceleration override AFACTOR. The acceleration override is a factor by which the control automatically multiplies all acceleration inputs. The values calculated in this way then apply for all subsequent movement functions.

Example:

Programmed A = 300.00 mm/s^2 and AFACTOR = 90 % results in an active
A_{active} = A * AFACTOR = 270 mm/s^2

The acceleration override function is not active for programming with AFIX .

Acceleration override

Designation	AFACTOR
Unit of measure	decimal factor
Input range	0.01 to 999.99 % (depending on machine parameter P22 and P118)
Power-on condition	100 %

The decimal factor relates to the defined acceleration values.

The factor can also be specified in %, i. e. AFACTOR = 60 % is equivalent to AFACTOR = 0.6.

Designation and value assignments form a complete BAPS statement.

Example

```
AFACTOR=200%                                ;The AFACTOR of 200 % (=factor 2)
                                            ;acts on the path acceleration for
                                            ;the movement to the points pos2
                                            ;and pos3

V=200

MOVE PTP TO pos1

MOVE CIRCULAR TO (pz,pos2)

MOVE WITH A=80 TO pos3

MOVE LINEAR WITH V=100, AFIX=10 TO pos4     ;The point pos4 is approached with
                                            ;the fix acceleration value AFIX = 10
                                            ;[mm/s²]. The acceleration override
                                            ;does not have any influence here
```

In the same way as AFACTOR acts on the acceleration phase and can be programmed, the DFACTOR is used analogously for the deceleration phase of a movement. In this case also the BAPS word DFIX has to be used instead of AFIX.

Movement statements

8.2 Time definition, indirect speed programming

If the robot must approach the next position within a certain time, it is possible to define a time T, resp. TFIX.

Time for LINEAR

Designation	T (override active) TFIX (override not active)
Unit of measure	second
Input range	0.5 to 32000 s

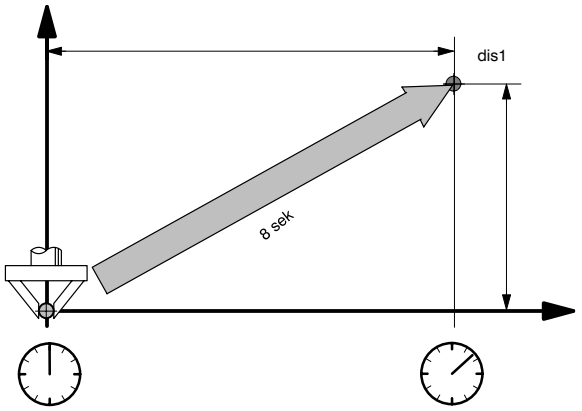
☞ **Time inputs are possible only on a statement-specific basis.**

The time value is part of the movement function for which it is to be valid.

The robot travels in a straight line in incremental dimensions. It has been allocated a time of 8 seconds for the distance dis1 to be covered.

Syntax:

```
MOVE_REL LINEAR WITH T=8 EXACT dis1
```

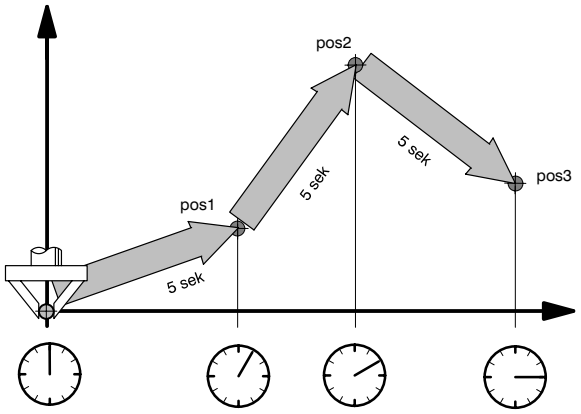


Movement statements

The robot successively travels to the positions pos1, pos2 and pos3 on a straight line with an intermediate halt. It is allocated a time of 5 seconds in each case to travel from position to position.

Syntax:

```
MOVE LINEAR WITH T=5 TO pos1,pos2,pos3
```



8.3 Statements influencing movement

In addition to the above described direct movement statements, there are also the following statements which have an influence on the movement sequences:

- synchronization statements SYNC, SYNCHRON, SYNCHRON_END
- acceleration, deceleration between movement statements (block transitions) BLOCK_SLOPE, PROGR_SLOPE

8.3.1 Belt synchronization

The belt synchronization function allows the robot movement to be synchronized with an assembly or conveyor belt with respect to position and orientation.

It does not matter whether the belt travels forward or backward, changes its speed or stops. This does not apply to belt_kind 2 and 3. The belt must be a straight line. This line may be arbitrarily positioned in space, see rho4 machine parameters. The belt movement is registered by a position measuring system.

The statements SYNC, SYNCHRON and SYNCHRON_END are available in BAPS for function programming.

Syntax:

```
SYNC belt variable[, variable] condition

SYNCHRON [kinematic name] belt variable

SYNCHRON_END [kinematic name] belt variable
```

The belt variable must be declared in the declaration part of the program, see also section 4.7.1, and must be assigned to a kinematic, see also rho4 machine paramters.

Example:

```
;Declaration of a belt variable

;several belts for one kinematic

sr6.belt: 501=belt1, 502=belt2

;same belt as belt1

screwdr.belt: 503=screw_belt
```

Movement statements

☞ **Up to 16 belts can be declared . The belt names and belt numbers must be different, even if several kinematics use the same measuring system for a belt variable.**
The movement on belts can only be synchronous that are also declared for this kinematic.

The component names and axis names must be specified correspondingly for the assigned belt.

Programming belt synchronozation

The belt variable, which is of the data type REAL, contains the counter value of the belt measuring system. The belt variable can be interrogated only via compare operations, such as >= and <=.

The belt variable can be used in the program with WAIT UNTIL, IF and SYNC.

Example:

```
;belt variable in WAIT UNTIL

WAIT UNTIL belt_1>=6
```

The instruction SYNC sets the belt variable to zero, resp. to the reset value, see special function 28 in the manual Control functions. Zeroing can take place dependent on a condition. The following example shows the possibilities of how the SYNC instruction can be used.

Example belt variable and SYNC statement:

```
SYNC belt_1>=300       ;Zeroing takes place if belt_1>=300

SYNC belt_1, li_scr=1 ;Zeroing takes place dependent on a condition
```

Belt synchronization is switched on with the instruction SYNCHRON. From now on, all programmed movements are synchronized with respect to position and orientation.

☞ **The instruction SYNCHRON should therefore directly follow the statement SYNC. If this is not the case, a set value may result due to a belt movement, which the control then attempts to compensate for with the SYNCHRON statement. This may in turn result in a jerky movement of the robot.**

Synchronization is switched off with SYNCHRON_END.

Example:

```
SYNCHRON sr6 belt_1   ;Belt variable and synchronous statement
```

Movement statements

```
MOVE LINEAR TO pos_1

SYNCHRON_END belt_1
```

☞ **Only LINEAR and CIRCULAR interpolation must be programmed in the belt synchronization mode.**

Program example

```
;;CONTROL = rho4

;;KINEMATICS: (1=robi_1,2=robi_2)          ;Definition of the kinematic names

;;robi_1.JC_NAMES=a1,a2,a3,b1              ;B1 is a dummy for belt values

;;robi_1.WC_NAMES=k1,k2,k3,b_c1

;;robi_2.JC_NAMES=a1,a2,a3,b2              ;b2 is a dummy for belt values

;;robi_2.WC_NAMES=k1,k2,k3,b_c2

PROGRAM beltsyn

INPUT: 1=i1,2=i2

robi_2.POINT: start_pos

robi_1.belt: 501=belt1

robi_2.belt: 502=belt2

BEGIN

   SYNC belt1,e1=1

   SYNCHRON robi_1 belt1                  ;Synchronization of kinematic robi_1
                                          ;with belt1

      MOVE robi_1 LINEAR TO robi_1.POS

      WAIT UNTIL belt1>=1000

   SYNCHRON_END robi_1 belt1

   SYNC belt2>=200

   SYNCHRON robi_2 belt2                  ;Synchronization of kinematic robi_2
                                          ;with belt2

      MOVE robi_2 LINEAR TO start_pos

      WAIT UNTIL e2=1

   SYNCHRON_END robi_2 belt2

PROGRAM_END
```

Movement statements

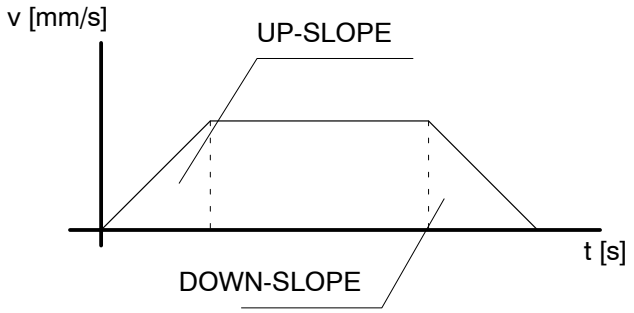
8.3.2 Block transitions (slope mode)

Syntax:

```
BLOCK_SLOPE

PROGR_SLOPE
```

In the normal movement sequence, the controlled axes are accelerated to the programmed speed with every MOVE statement, traversed at the programmed speed and then decelerated again to V = 0 when the programmed position is reached. The speed change is referred to as the SLOPE. The acceleration phase is thus known as the up-slope and the deceleration phase as down-slope.

The slope mode can be activated on a kinematic-specific basis.



General

If it is wished to execute several MOVE statements coherently without changing the speed to V = 0 and accelerating to the programmed speed, this can be done by using PROGR_SLOPE. Switch-back to block-by-block acceleration and deceleration is possible with BLOCK_SLOPE.

If PROGR_SLOPE is activated, the block transitions are executed at a constant required speed if no speed changes are programmed. Otherwise, the required speed is changed in jump and/or ramp form.

SLOPE mode activation

The slope mode can be switched in the BAPS program. The BAPS standard functions BLOCK_SLOPE and PROGR_SLOPE are defined for this purpose.

PROGR_SLOPE is switched off by activating BLOCK_SLOPE. The slope function is then active block-by-block.

Robot acceleration control

The robot is accelerated at the start of every block in accordance with the slope form and is then decelerated again correspondingly at the end. This means: jump to slope point, then start with defined acceleration.

See also rho4 machine parameters P120 to P124.

Movement statements

PROGR_SLOPE

The robot is accelerated by means of the slope function at the start of a coherent movement sequence and is decelerated again at the end. The speed is kept constant at block transitions if no speed change is programmed.

The power-on condition is defined for each kinematic via machine parameter P120.

Programming of the slope mode is explained below in several BAPS program examples and its effect on the movement sequence is shown in the following diagrams.

Example 1

```
PROGRAM ex_1                ;(1)

BEGIN                       ;(2)

   ;;INT=LINEAR             ;(3)

   V=800,A=1000             ;(4)

   MOVE VIA beg_point       ;(5)Program slope not active, i.e. the robot is
                            ;accelerated in each block with A=1000 mm/s2 and is
                            ;again decelerated at the end of the block.

   MOVE VIA point_center    ;(6)

   MOVE TO end_point        ;(7)

   PROGR_SLOPE              ;(8)Program slope is switched on

   MOVE VIA beg_point       ;(9)

   MOVE VIA point_center    ;(10)

   MOVE TO end_point        ;(11)

   MOVE VIA RAMP_1          ;(12)

   MOVE VIA RAMP_2          ;(13)

   MOVE VIA beg_point       ;(14)

   MOVE VIA point_center    ;(15)

   MOVE TO end_point        ;(16)

   BLOCK_SLOPE              ;(17)

   HALT                     ;(18)

PROGRAM_END                 ;(19)
```

Movement statements

Program slope mode is not active in the first part of the program, blocks 5 to 7, i. e. the robot is accelerated in each block with A=1000 mm/s^2 and is decelerated again at the end of the block, see figure 1.

Program slope is switched on in block 8. As a result, the robot is accelerated with A=1000 mm/s^2 in block 9. The block transitions from 9 to 10 and from 10 to 11 are executed at constant speed.

The robot is decelerated with the programmed acceleration of 1000 mm/s^2 at the end of block 11, see figure 2.

The deceleration operation is already initiated in the previous block (block 10 figure 3) if the traversing distance in the MOVE TO block (block 11) is not sufficient to decelerate the robot by means of the slope function.

The speed is set to zero by way of a jump at the end von block 11 if the sum of the distances from block 10 and block 11 is not sufficient as the deceleration path.

In this case, the following message is issued during the runtime: deceleration distance is too short, block No.: 11.

The acceleration phase may take place over any number of blocks, e. g. block 12 to 16, figure 3.

Figure 1

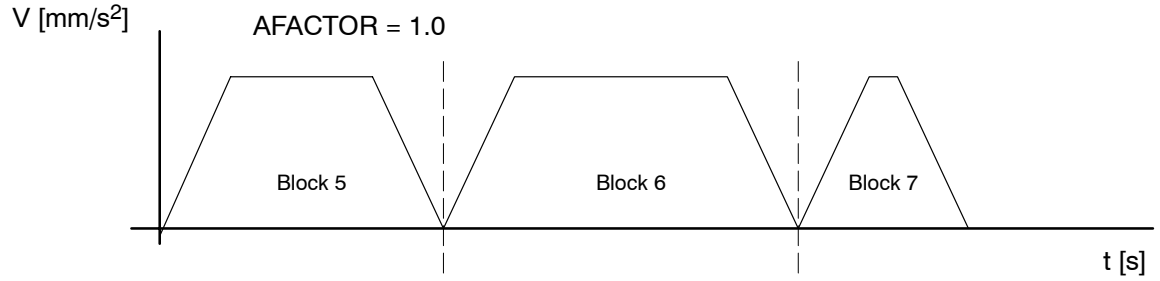
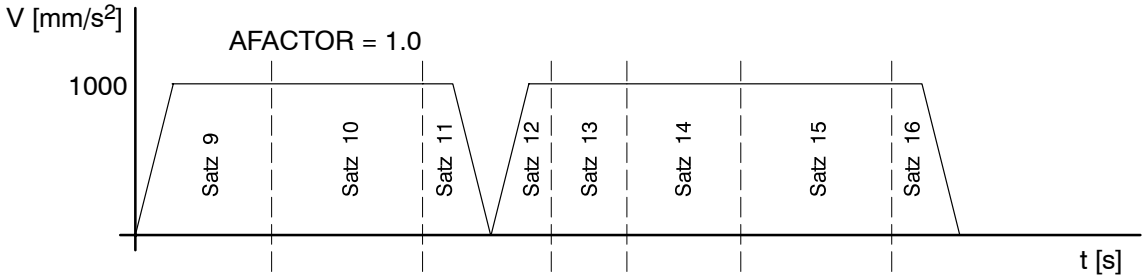
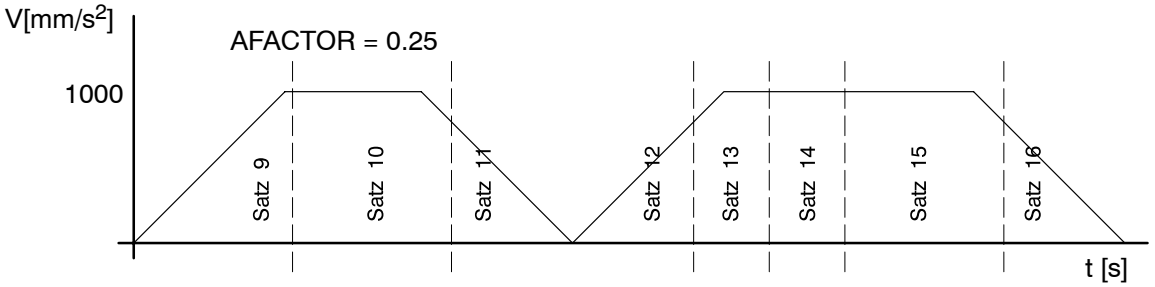


Figure 2



Movement statements

Figure 3



Changing acceleration and speed

Acceleration change (A, AFACTOR)

As before, a change in the acceleration value is effective only at the time of block preparation; this is also true for any change of the AFACTOR with the PHG2000.

Also refer to the rho4 PHG2000 software manual.

Speed changes (V, VFACTOR)

Speed changes programmed in BAPS, such as V = ... or MOVE WITH V = ... act at the block transition.

☞ **Changes of VFACTOR with the PHG2000, mode 11.4, become active immediately.**

All speed changes are implemented in accordance with the slope function if the required speed is higher than the slope point defined in machine parameter P105 or P106. All speed changes are performed as jump below the slope point. If the programmed speed is not reached within a MOVE TO block, acceleration takes place only up to the max. possible speed and is then followed by immediate deceleration again, see example 2, block 14.

Example 2

```
PROGRAM ex_2                ;(1)

BEGIN                       ;(2)

   ;;INT=LINEAR             ;(3)

   PROGR_SLOPE              ;(4)

   A=1000                   ;(5)

   MOVE WITH V=200 VIA p1   ;(6)

   MOVE WITH V=350 VIA p2   ;(7)
```

Movement statements

```
   MOVE WITH V=500 VIA p3    ;(8)

   V=600                     ;(9)

   MOVE VIA p4,p5            ;(10)Program slope is switched on

   MOVE VIA p6               ;(11)

   V=400                     ;(12)

   MOVE VIA p7               ;(13)

   MOVE WITH V=1000 TO p8    ;(14)

   HALT                      ;(15)

PROGRAM_END                  ;(16)
```

Change in deceleration (DFACTOR)

It is possible to influence the deceleration in the BAPS program by assigning a corresponding value to the standard variable DFACTOR. Like the AFACTOR, the DFACTOR is a percentage which refers to the current deceleration of the respective block. A change in the deceleration acts like a change in acceleration at the time of block preparation. The DFACTOR can also be changed by means of PHG2000.

Abort conditions

Abort by external influence

An abort of a travel movement by external influence, e. g. reset, feed halt or abort with MOVE UNTIL instruction takes place as before, subject to the following restriction.

☞ **If the remaining travel distance in the currently active block is not sufficient to decelerate the robot by way of the slope function, the speed V = 0 is defined as a jump function at the end point. Immediate deceleration without slope function takes place in the event of an abort by Emergency stop.**

Abort of a coherent movement in the BAPS program

A movement sequence (activated program slope) is interrupted by the following BAPS instructions:

- WAIT
- PAUSE
- HALT
- BLOCK_SLOPE
- IF ... THEN ... ELSE
- REF_PNT
- WRITE

- READ
- if several outputs with strobe take place (INTEGER–outputs) at first the control waits the preset strobe time (machine parameter P8) until the next output can take place.
- also with alternate programming of travel blocks and e. g.INTEGER–outputs, travel interruptions may occur, if the travel blocks are shorter than the preset strobe time.
- several blocks without travel information, e. g. calculations, variable assignments, setting output signals.

The number of possible blocks depends on the length of the preceding travel blocks and the nature of the assignments or calculations performed.

Example 3

```
PROGRAM ex_3

;;INT=LINEAR

BEGIN

   V=800,A=1000

   PROGR_SLOPE                          ;Program slope is switched on

   MOVE VIA beg_point

   WAIT 1

   MOVE VIA point_center

   MOVE TO end_point

   WAIT 1.5

   MOVE VIA beg_point

   MOVE VIA point_center TO end_point

   BLOCK_SLOPE                          ;Block slope is switched on

   HALT

PROGRAM_END
```

☞ **If program slope is active, the movement must be ended in a defined manner before the above mentioned statements by insertion of a MOVE TO block. This initiates a controlled deceleration operation.**

Movement statements

☞ **The speed is set to 0 in a jump function in the event of an interruption after a MOVE VIA block.**

Interpolation mode change-over

The global acceleration and deceleration behavior can be activated independently of the interpolation mode. Block transitions without any change in the interpolation mode are performed as described in section 8.3.2.

Change-over between linear and circular interpolation

Block transitions are performed as described in section 8.3.2 for changes from linear to circular interpolation and vice versa.

Example 4

```
PROGRAM ex_4

;;INT=LINEAR

BEGIN

   V=800,A=1000

   PROGR_SLOPE

   MOVE VIA beg_point

   MOVE CIRCULAR VIA (int_point1,circular_end1)

   MOVE CIRCULAR TO (int_point2,circular_end2)

   MOVE TO pnt_center

   MOVE CIRCULAR TO (int_point3, circular_end3)

   HALT

PROGRAM_END
```

Change-over between path and PTP modes

The movement sequence must be ended by a MOVE TO block before a change-over from path mode to PTP mode and vice versa so that a controlled transition can be realized. If the change-over takes place during a coherent movement, the speed is changed in a jump function. No controlled acceleration takes place in the first block in the new interpolation mode (i. e. jump to programmed speed). It is thus possible to generate a transition without or with only a slight change of the axis speeds by clever programming.

Example 5

```
PROGRAM ex_5
```

Movement statements

```
BEGIN

   PROGR_SLOPE

   V_PTP=1

   MOVE PTP TO beg_point

   MOVE LINEAR TO p1

   MOVE PTP VIA p2

   MOVE LINEAR VIA end_pnt

   V_PTP=0.5

   MOVE PTP TO end_pnt_1

   HALT

PROGRAM_END
```

Calling external subroutines

The transition to an external subroutine can take place within a coherent movement without speed dip.

A precondition is that program slope is activated at the start of of the external subroutine with the BAPS statement PROGR_SLOPE before the first travel block or that program slope is preset by machine parameter P120.

Slope mode and exact-position signal output

The special functions 1 and 2 can be used fully for both program slope and block slope modes.

Transgression of axis limit values

Transgressions of limit values of individual machine axes in path mode cannot be excluded as a result of coordinate transformation. Only monitoring is possible during the program run.

This monitoring function triggers one of the two following error messages in the event of an error:

- interpolator stop, axis X
- Ax–Velocity exceeded, axis X

X is here the number of the corresponding machine axis

Movement statements

The maximum permitted axis acceleration values are defined as 1.5 times the value of the machine parameter P103. The programmer is thus made to change the program at the corresponding places. Automatic speed adaptation is not possible, since this would contradict the demand for constant path speed.

Test system

Since interrupt points can be set in the test system, only BLOCK_SLOPE is active here, irrespective of the programmed slope mode.

Slope mode and machine parameters

The slope behavior is determined by the following machine parameters:

- slope acceleration PTP
- slope point, path mode
- power-on condition, slope mode
- SLOPE form

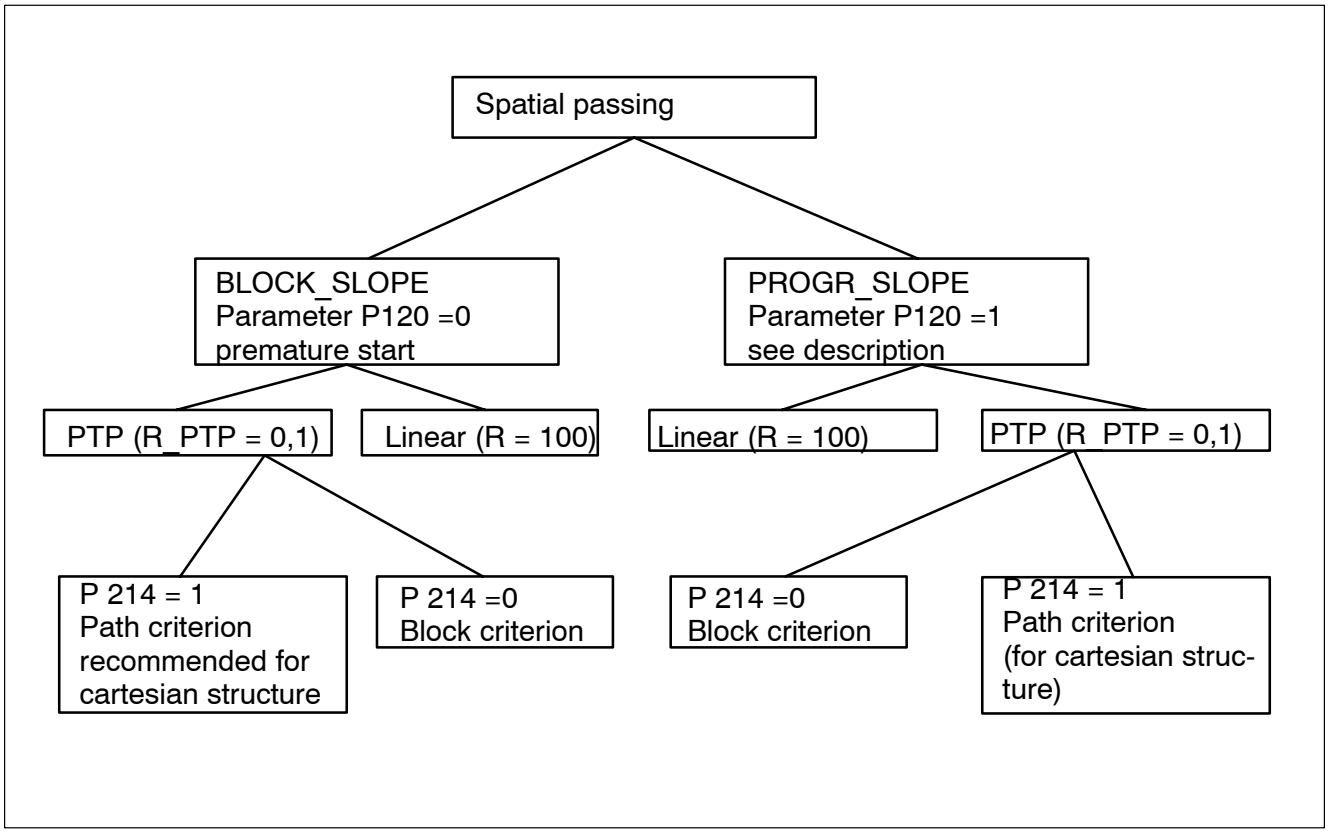
See rho4 machine parameters, parameter group P100.

Movement statements

8.3.3 Spatial passing

Programmed destination points can also be passed to generate a continuous and harmonious movement at block transitions. It is thus possible to achieve an accelerated movement sequence and while treating the robot mechanics with care.

Passing means to ensure a uninterrupted speed sequence with block transitions by a defined deviation from the programmed path.



In general, the following applies to block slope

- The spatial deviation in the passing range is speed-dependent.
- The individual travel blocks are started by the set passing distances too early.

In general, the following applies to program slope

- In a coherent movement sequence it is attempted to keep the speed constant.

Movement statements

Spatial passing with program slope

PTP passing with program slope

A distance in degrees or mm, from which passing takes place, can be programmed for each axis referred to its end position. By means of these distances a passing range around the destination point is defined.

The robot travels on a path which is generated by the control and ensures that the individual axis speeds do not change in a jump function if the working point enters this passing range during a travel movement. After having left the passing range, the robot behaves in the same way as in travel movements without programmed passing.

The distances belonging to each axis are stored in the machine parameter data block. These can be changed in the user program by means of a factor (R_PTP). The function is analog to a V_PTP change. When the factor is programmed, it remains active until it is overwritten. All distances stored for the individual axes are influenced equally so that the passing ranges will be compressed or extended. An own passing range, which can optimally be adapted to the respective requirements, can thus be defined around each destination position.

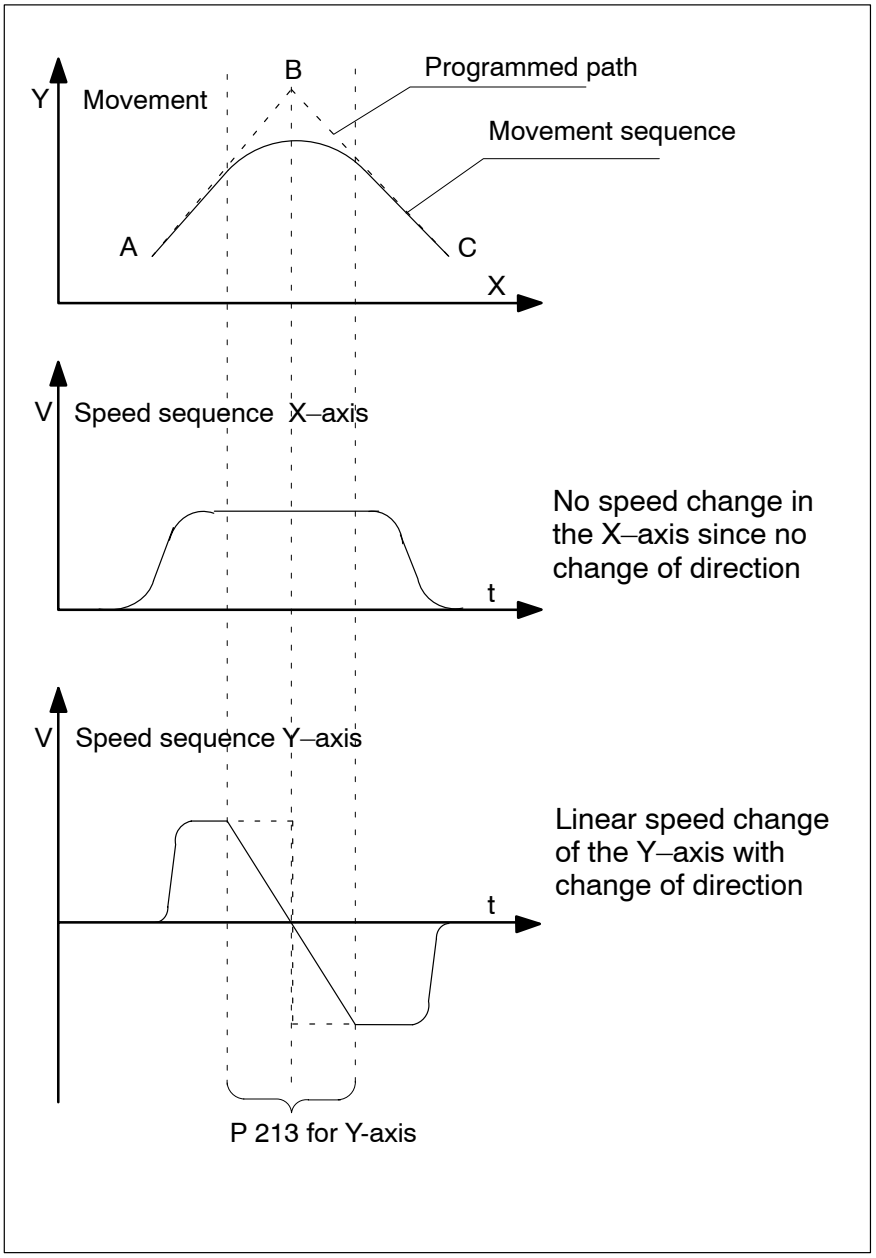
The selected distances are decisive for the time in which the speed change has to be executed at the block transitions. If a relatively small passing range is defined, the differential speed of the successive movements must be passed in a short time. The deviation from the initially programmed path, also without passing, is limited to a small range. If large passing ranges are selected, the speed transition will be smoother, but the deviation will take on a larger distance.

The transitions between programmed path and passing path are tangential.

The following is to clearly show the continuous speed change with one block transition. It is based on a Carthesian system with two axes, one in X- and one in Y-direction. The slope mode is program slope. The passing range in X- and Y-direction is defined via machine parameter P213. If travelling is made without passing, a speed jump from +V to –V will take place at the block transition for the Y-axis. When the passing is switched on, the speed within the passing ranges will be changed continuously from +V to –V. Soft block transitions will be the result.

Movement statements

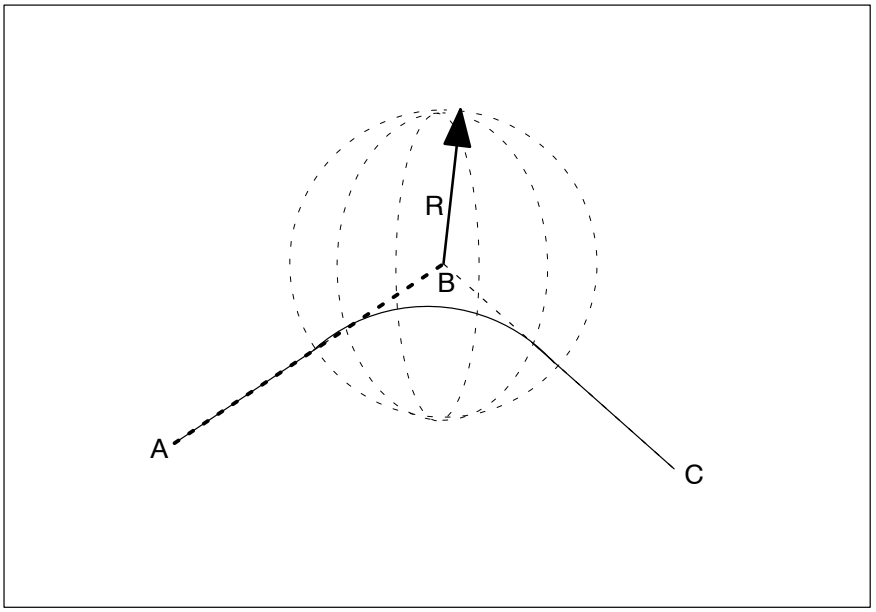
☞ **The generated passing paths depend on the speed.**



Movement statements

Path passing with program slope

In case of path passing, a space sphere is defined around each point to be passed. The radius of this sphere is defined in the BAPS-user program with the standard variable R. It is thus possible with the path passing to provide for each programmed space point its own passing range. The original path is left between entry and exit from this sphere for the sake of a harmonious speed sequence. The robot moves outside this space sphere on the path it would have travelled without passing. For the adjustment of the passing ranges optimal for the user, the same conditions apply as for the PTP-passing.

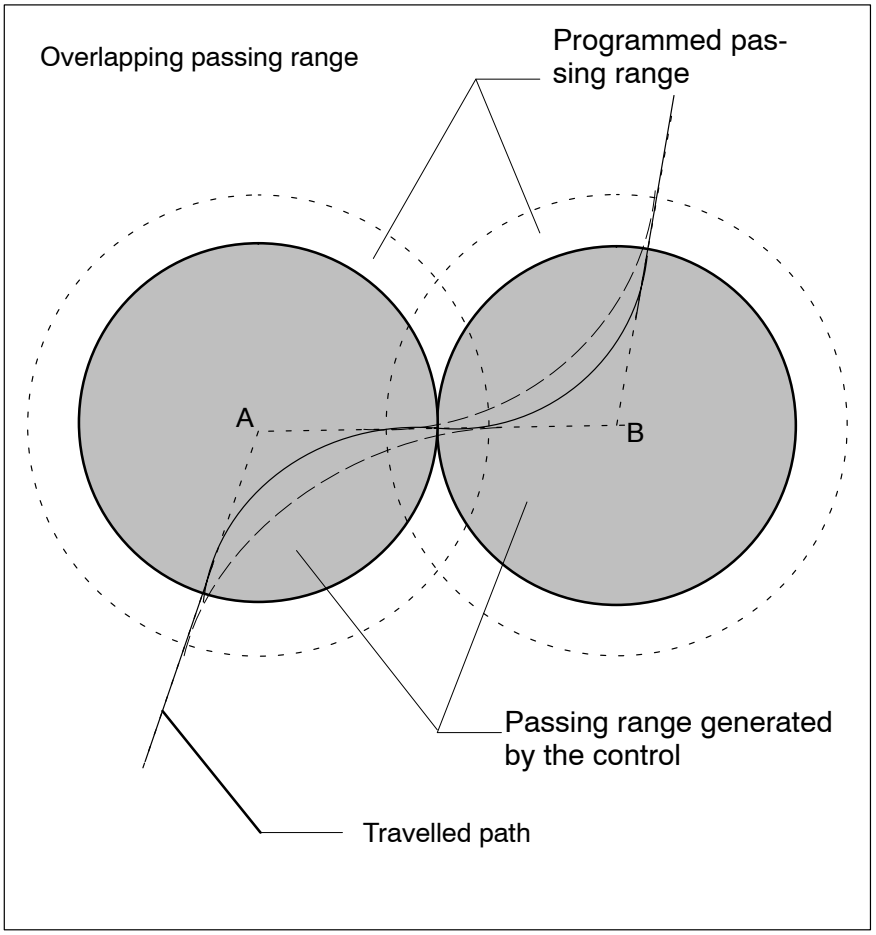


Passing with change of interpolation mode

The passing with active program slope is also effective when changing the interpolation modes. During handling and assembly works it may be reasonable to pass points approached in PTP and from which on a linear travel should take place, or vice versa. Before entering and after having left the passing envelope, there will be no deviation from the initially programmed path. In this case it is not decisive whether travelling takes place from PTP to linear or vice versa.

Special cases

If the distances between the points to be passed are not sufficient to meet the programmed passing criteria, the control will automatically reduce to the maximum possible passing range. No adaptations will have to be made by the programmer for this purpose.



Explanation of the figure: Overlapping passing ranges are automatically reduced by the control to achive the maximum possible passing range in this case. It is shown by the continuous line.

Movement statements

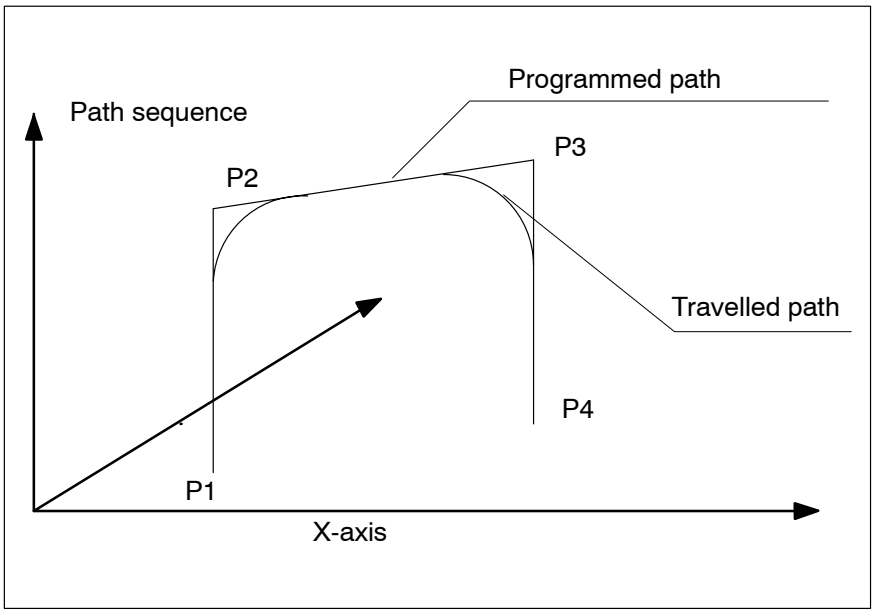
Passing with block slope

PTP passing with block slope

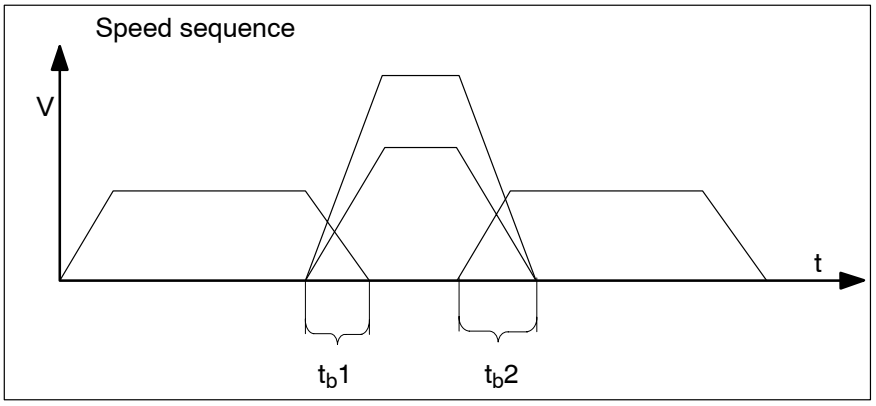
Contrary to the program slope mode, in which a continuous speed sequence with equal time ratios is achieved when passing, the next movement block is started earlier in the block slope mode, whereby time is saved. From the distances specified in degrees or mm for each axis with respect to their end position, a common time is determined at which the subsequent movement block is started. Both blocks are then executed parallel and the results are added. It is thus ensured that the work point of the robot, after having left the passing range, reaches again the path travelled initially without passing. To avoid a simultaneous activation of more than two movement blocks, the earliest start point is automatically limited by the control to the center of the previous movement. A started movement block is ended until the center of the subsequent block.

The following example is to clearly show the travel behavior.

Basis is a Cartesian system with 3 axes, whereby at first the 3rd axis is to travel from the working range upward, then the axes 1 and 2 travel to then emerge again with axis 3 into the working range. Saved time: t_{b1} + $t_{b2.}$



Movement statements



Path passing with block slope

Path passing with the block slope mode acts, referred to the path, in principle as PTP-passing with block slope.

Passing with change of interpolation mode

When the bock slope is active, no passing will take place when changing the interpolation modes.

BAPS language elements

Programmed destination points can also be passed to generate a continuous and harmonious movement with block transitions. Passing means to enxure a continuous speed sequence by a defined deviation from the programmed path.

The spatial passing is influenced in the user program with the standard variables R and R_PTP. The standard variables act kinematic-related. For the linear interpolation an R as radius is used analog to V-programming. Absolute values are programmed for R . The unit of R is defined by the measuring system conversion factor. R_PTP acts as factor with PTP interpolation analog to V_PTP. This factor can be programmed in real terms or as percentual value. It is offset with the machine parameter P213.

Example

MOVE-statement with R, resp. R_PTP

```
;;INT=LINEAR                    ;Default of interpolation mode

R=50                           ;Global effect

MOVE LINEAR WITH R=20 VIA p1   ;Block-related

;;INT=PTP                       ;Default of interpolation mode

R_PTP=1.4                      ;Global effect
```

Movement statements

```
MOVE WITH R_PTP=1.4 VIA p2     ;Block-related
```

☞ **If R = 0 or R_PTP = 0 is programmed, the spatial passing is switched off.**

Machine parameters

The function 'spatial passing' is to be activated or deactivated via machine parameter options. This function is preset at the time of shipment of the control and after a machine parameter backup.

At the program start the passing distances and factors set in parameter P212 are used. With the BAPS standard variables described before, the values stored in the user program can be overwritten. If passing is switched off, 0 is used in each case.

The parameter P212 can be used to define a passing range, without having to program it explicitely in the BAPS program.

☞ **We recommend to use 0 as default for parameter P212.**

In parameter P213, the axis-specific passing distances or path distances are stored, dependent on parameter P214. The factor R_PTP refers to these distances.

The path criterion is suitable above all for Cartesian structured kinematics since the value specified here represents with PTP interpolation a space distance. This path criterion acts with both the program and block slopes.

Restrictions:

Points approached in circular interpolation, or from which a travel with circular interpolation takes place, are not passed spatially since the path part in the interpolation clock changes already due to interpolation. This applies to both the program and block slopes.

Movement statements

Notes:

9 Write/read functions

The BAPS statements WRITE and READ are available for communication.

Syntax:

```
WRITE device names, variable [,variable]

READ device names, variable [,variable]
```

Communication takes place via the serial interfaces available on the control.

The output device is addressed by the device name. This is assigned to the hardware interface via machine parameters or via mode 9.1 with PHG2000 via a device number. The assignment of device number and interface connection is shown in the following table.

Assignment: Device No., device names, interfaces

Device No.	**0**	**1**	**2**	**3**	**4**
Device name	V24_1 to V24_4 TTY SER_1 to SER_4 WIN_1 to WIN_4	V24_1 to V24_4 TTY SER_1 to SER_4 WIN_1 to WIN_4	PHG	V24_1 to V24_4 TTY SER_1 to SER_4 WIN_1 to WIN_4	V24_1 to V24_4 TTY SER_1 to SER_4 WIN_1 to WIN_4
rho4.1	X31	X32	X35	X33	X34

☞ **X31 and X35 cannot be used simultaneously. X33 and X34 are optional.**

Write/read functions

9.1 Protocol selection for communication functions

Different communication protocols are available for communication.These can be selected via machine parameter setting or via mode 9.1 with PHG2000.

Protocol No.	**Protocol structure**	**read echo**
1a **1b**	< DATA > followed by < CR > < LF > < DATA > followed by < CR > or < LF>	yes
2	< DATA >	yes
3	< SOH > < STX > < DATA > < ETX > followed by < SOH > < STX > < CR > < LF > < ETX >	no
4	< SOH > < STX > < DATA > < ETX >	no
5	< DATA >	no
6	PHG protocol	yes
7	rho1/2 compatible according to protocol No. 3	no
8	Data link layer acc. to Siemens protocol 3964/R	no

1a = data input, 1b = data output

Write/read functions

9.2 BAPS instruction WRITE

The instruction WRITE is used to put out data from the control via the specified interface.

As soon as the WRITE instruction is reached in the program run, the desired variables, texts or other data are put out via the selected interface.

The output of the data takes place as ASCII character string, i. e. conversion from internal format to ASCII format is performed.

9.2.1 Protocol 3964/R

As an option, the function READ / WRITE device is to pack or unpack data of a BAPS program also into the protocol 3964/R backed-up in block form.

The protocol 3964/R is set as protocol 8, analog to the other seven protocols, under MODE 9.1. or under MODE 7.8.3 in the interface-dependent subpoint 3, 4, 5 or 6.

The operating system then realizes the link layer for the data to be transferred in the protocol 3964/R, with establishment of the connection, block backup, time monitorings, block repetitions etc.

Contrary to the protocols used so far, data are in any case transferred in both directions when transferring data by the protocol. The data are furthermore not converted for the transfer, since they are transferred in binary form. For this reason, some particularities have to be taken into account in this respect.

Realization of the protocol driver

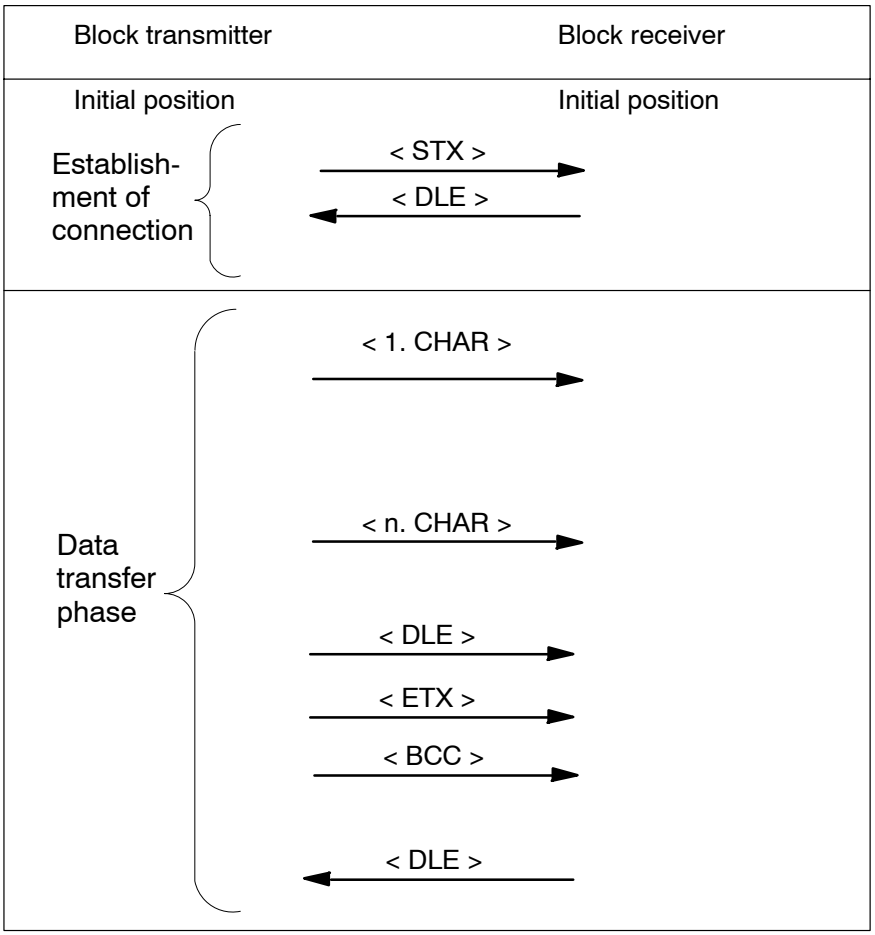
For the processing of protocol 3964/R, establishment of the connection, data backup, time monitorings, possibly block repetitions, a driver task is activated with this protocol for every BAPS interface during the runup of the control.

It receives and acknowledges max. one data block, even if READ is still active. It furthermore synchronizes the sending and receiving of data blocks.

Contrary to READ / WRITE with the protocols used so far, which occupy the interface only temporarily, the interface is definitely occupied by the driver task in the case of protocol 3964/R. In this case it cannot be used for other functions.

Write/read functions

If no errors occur, the protocol 3964/R looks as follows:



With the positive acknowledgement on the block checksum <BCC>, the connection is considered offline so that each data block requires an connection establishment.

If the block receiver replies to the attempt of the connection establishment with another character than <DLE> or if it does not reply within the acknowledgement delay time (QVZ) of 550 ms, the block transmitter repeats the characters <STX> for the establishment of the connection max. 5 times (= 6 attempts).

If the connection has been established, the block transmitter sends the data, whereby a <DLE> contained in the data will be transferred twice. The data block must not have more than a maximum of 138 characters, without taking the doubling into account. The data are followed by the characters <DLE> <ETX> as block end identifier and then by the block check sum <BCC> as XOR link via all characters of the telegram, except for the start control character <STX>.

During the transfer of the data, the block receiver expects further characters within the character delay time (ZVZ) of 220 ms until the reception of <BCC>. Otherwise it will abort the reception, send the characters <NAK> and switch then into the initial position.

Write/read functions

After having sent <BCC>, the block transmitter expects an answer from the block receiver within the acknowledgement delay time. If the block receiver answers with another character than <DLE>, especially <NAK> or not within the acknowledgement delay time (QVZ), the block transmitter will repeat the transfer of the data blocks max. 5 times (= 6 attempts).

If both sides want to establish a connection at the same time, the side without priority will be subordinated. The side with the priority starts with the transfer of the data as described. Since the timeout times for this protocol are fix, see delay times, the priority for the case of conflict is set via the output Timeout time. If the value –1 is set there, the driver operates without priority. The sending priority is part of the interface parameters.

Integration into the BAPS program

Permitted is the reading and writing of the BAPS data types

- BINARY,
- INTEGER,
- REAL
- POINT,
- JC_POINT,
- CHAR,
- TEXT,
- one-dimensional arrays of the type BINARY, INTEGER, REAL, POINT, JC_POINT and CHAR

The use of the type TEXT for the transfer of binary data is problematic for some BAPS operations because of the special position of the NIL character as end character, especially as the application protocol starts in case of a 3964/R connection normally with two NIL characters. A 3964/R useful data block is furthermore up to 128 characters (>80) long, plus a maximum of 10 bytes for the telegram head of the application layer.

As only the link layer of protocol 3964/R is realized, the realization of the application layer of the protocol has to take place in the BAPS program. For the compilation of the BAPS-specific presentation of data into application-specific presentations and vice versa it is recommended to write corresponding subroutines, e. g. for the conversion of an INTEGER value into a word presentation as common in the PLC and vice versa.

Write/read functions

Type	Number (bytes)	Presentation, value range
BINARY	1	0 or 1
CHAR	1	0 to 255
INTEGER	4	Byte order adjustable via options. Normal: Byte of lowest value first.
REAL	4	IEEE format, byte order as for INTEGER
POINT	4 * number of coordinates	Each coordinate as REAL, without separation sign
JC_POINT	4 * number of axes	Each axis as REAL, without separation sign
TEXT	80	Each byte as CHAR
Arrays of the type BINARY, CHAR, INTEGER, REAL, POINT, JC_POINT	Array size	Each element as data type

In case of data of the types POINT and JC_POINT, belt coordinates are also counted with the number.

WRITE device

With WRITE, the data are written into the output block in binary form without conversion. The number of bytes depends on the BAPS data type.

If several expressions separated by a comma are programmed in a WRITE instruction, the data are written succesively in an output block and then the whole block backed-up with the protocol 3964/R is transferred.

READ device

With READ, the same number of bytes is read type-dependent from the reception block, just as with writing. If not all bytes have been read from the reception block in one single READ cycle, the data are continuously read with each further READ. Reading from a new reception will only start when the reception block has been read completely. No block limit must, however, be within a data type to be read. This would lead to a runtime error 'READ protocol error'. Especially READ / WRITE device with protocol 3964/R V1.2. Since the data are transferred binary, a check of the read data values for values of 0 and 1 can only take place with variables of the type BINARY. In case of variables of the type REAL, or components of the type POINT or JC_POINT, a FPU trap system error can only be caught and converted into a runtime error 'prot.err while READ'. It should therefore be checked within the BAPS program whether the read data values are of a reasonable order.

Write/read functions

Special cases and restrictions

The overall execution time, e. g. for reading a data module of a PLC, will mainly be determined for an error-free transfer at 9600 Baud by the execution time for the formation and especially interpretation of the 3964/R application layer in the BAPS program.

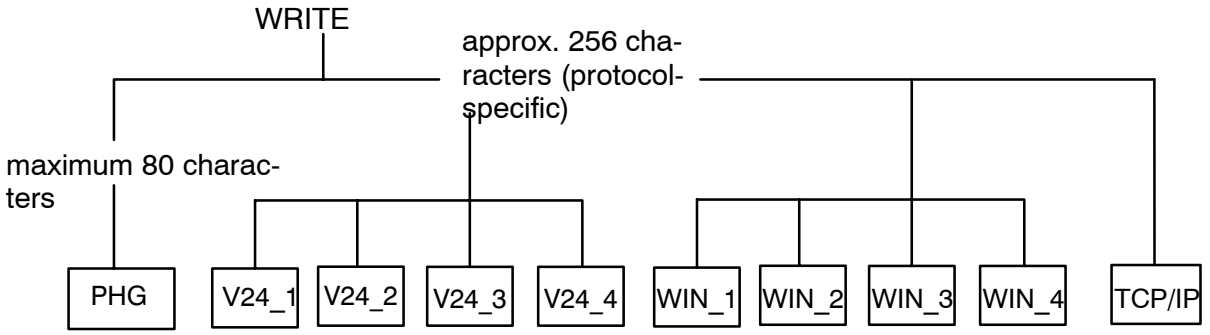
If a program is aborted, which has already read at least 1 byte of a reception block, the rest of the block will be deleted to permit a new positioning. If the physical interface is not available for the runup or if it is occupied, e. g. with the function 'Coupling to programming device', the driver task will not be initialized.

Write/read functions

9.3 Interfaces

The data can be put out via the following interfaces:

- V24_1 to V24_4
- WIN_1 to WIN_4
- PHG
- TCP/IP



In order to identify the interfaces, you must enter their names in the program after the WRITE instruction.

If no interface is specified, the control puts out the desired data to the PHG2000.

Example

```
WRITE PHG, g        ;The variable g is displayed at the PHG

WRITE V24_2, te     ;The variable te is put out via interface V24_2
                    ;e. g. at a printer

WRITE '3'           ;The number 3 is displayed on the standard output device
```

☞ **A comma must be entered after input of an interface name.**

9.3.1 Transferred data

Constants and variables of the type

- BINARY
- INTEGER
- REAL
- POINT
- JC_POINT
- CHAR

Write/read functions

- TEXT and
- one-dimensional arrays of the type BINARY, INTEGER, REAL, POINT, JC_POINT, CHAR and TEXT can be transferred

When writing to the PHG2000, you can transfer a maximum of 80 characters per WRITE instruction. Transfer of a maximum of 120 characters per WRITE instruction is possible for the other interfaces.

A few special restrictions apply to the individual data types as regards the scope of transferability:

BINARY	Only values 0 and 1
INTEGER	Integral numbers with a maximum of 10 digits in the range between -2147483648 and +2147483647 can be transferred.
REAL	Decimal numbers in the range between –999999 and +999999 can be transferred, minimum resolution ±0.00001. Transfer takes place as a floating-point number after REAL-ASCII conversion with sign or blank, 6 digits and a decimal point, whereby the position of the later depends on the value.
POINT, JC_POINT	The individual coordinate values of a position of the type POINT or JC_POINT must lie within the limits of the type REAL.
CHAR	All ASCII characters in accordance with DIN 66003 are transferred.
TEXT	All ASCII characters except 0 can be transferred. However, no more than 80 characters may be transferred in a WRITE instruction. In the case of text constants, the text to be transferred must be placed in inverted commas and be in one line.

Example

```
k=2

d=0.123

WRITE PHG,k,'.VALUE =',d ;The folloowing display appears on the PHG: 2. value =
                         ;0.12300
```

☞ **If several variables or constants are to be transferred within a WRITE instruction, these must be separated from each other by a comma.**

☞ **The WRITE instruction generates additional outputs, depending on the set protocol, see above.**

Write/read functions

9.4 BAPS instruction READ

The READ instruction is used to request the control to read variables from an interface.

As soon as the READ instruction is reached in the program run, the control stops the movement sequence and waits until the data is present at the desired interface.

The read-in variables can thus be used in the rest of the program.

9.4.1 Interfaces

The control can read in the variables via the following interfaces:

- PHG2000
- V24_1 to V24_4
- WIN_1 to WIN_4
- PLC

In order to identify the interface, the interface name must be entered in the program after the READ.

If no interface is specified, the control expects data input from the PHG2000.

Example

```
READ V24_1,g ;The interface V24_1 must provide the variable g

READ k       ;The value of the variable k must be entered at the standard input
             ;device
```

☞ **A comma must be entered after the interface name is put in.**

☞ **The waiting time until the program aborts with the error message 'interface error' as a result of missing data can be set or deactivated by means of the interface presetting.**

Write/read functions

Multiple use of serial interfaces

The operating system of the control rho4.1 provides more functions to write data onto serial interfaces or to read them, than interfaces are provided on the hardware side.

These functions are:

- coupling to programming device, also Online functions
- print file
- WRITE/READ V24_1 to V24_4

Under mode 9.1 resp. mode 7.8.3, an assignment between the function and the serial interface is made.

If for two or more functions the same serial interface is selected there, a situation of conflict will occur if these functions use the interface simultaneously.

It has to be taken into account in this respect that some functions attempt permanently to occupy an interface. These are the function 'coupling to programming device' and the WRITE/READ V24_x functions if the protocol 3964/R (8) has been set there.

Automatic reactions in a case of conflict

If any function (B) wants to use an interface which is already occupied by another function (A), the robot operating software (RBS) lets wait function (B) until function (A) is finished.

Finish in case of WRITE V24_x means that a coherent WRITE statement has been put out and in case of a READ that the reading of exactly one variable is ended. This behavior also applies to the access to this function if several processes attempt to use one and the same function. If several processes wait for the assignment, the priority of the processes will decide on the assignment by the priority order.

Special cases

The functions coupling and WRITE/READ V24_x with the protocol 3964/R do not enable the interfaces. A function (B) would wait endlessly for the end of such a function (A).

WRITE/READ PHG, resp. V24_x with PHG protocol stops the operating surface of the operating system, uses the interface for the PHG2000 and makes sure at its end that the operating surface of the operating system is continued.

Write/read functions

Controlled avoiding of conflicts

Disable input signals are provided at the internal interface to specifically avoid cases of conflicts:

- 28.6 :DIS_COUPL_RCI
- 28.7 :DIS_PRI_RCI
- 29.0 :DIS_SER_1_RCI
- 29.1 :DIS_SER_2_RCI
- 29.2 :DIS_SER_3_RCI
- 29.3 :DIS_SER_4_RCI

An abort takes place if these signals change from 0 to 1 when the function is active. CONDITION = interface error.

If these signals are on 1 when a function is actived, they will not be executed. CONDITION = interface occupied.

When these signals change from 1 to 0, the functions coupling and READ V24_x with protocol 3964/R can be continued directly, the others only when being called again. If the interface changes its transfer rate, it will be reinitialized. If there are still characters in the read-in FIFO of the interface, they will be deleted.

If these signals change to 0, the method described before will be implemented.

The normal condition of these signals is 0. The disable signals need only be served if cases of conflicts can occur and another conflict strategy than the automatic one is desired or if conflicts cannot be solved automatically.

Repositioning or switching over the interface

When repositioning or switching over the interface, an invalid character, which can with the function READ device lead protocol-dependently to the runtime error ’prot.err while READ’ or to the CONDITION ’Framing error’, is normally read by the interface.

This can be avoided by the following sequence:

- disable function (A)
- reposition or switch over
- enable function (B)

Condition display

The message ’Disable coupling PG’ appears in the info function if the function coupling is deactivated via its input signal.

Write/read functions

Example

The functions coupling and READ / WRITE V24_1 with protocol 3964/R are to be operated on the same interface. It is assumed that a digital input COUPL_IN_DI exists, e. g. a signal made available by the interface change-over switch.

The PLC program then looks as follows:

```
DIS_COUPL_RCI=NOT COUPL_IN_DI

DIS_SER_1_RCI=COUPL_IN_DI
```

For some BAPS program parts it may he helpful to have a switch-over signal available too, e. g.

PLC program:

```
OON_1_RCI=NOT COUPL_IN_DI
```

BAPS program:

```
INPUT BINARY: 1=V24_1_free

WAIT UNTIL V24_1_free=1

OR cond_V24_1=CONDITION(V24_1)

IF (cond_V24_1=-7)            ;interface error

OR (cond_V24_1=-2)            ;interface occupied

THEN WAIT UNTIL V24_1_free=1
```

Variant:

If required, the BAPS program switches On the interface via an OUTPUT.

BAPS program:

```
OUTPUT BINARY: 1=now_V24_1

INTEGER: cond_V24_1

now_V24_1=1

READ V24_1, variable

WRITE V24_1, variable

cond_V24_1=CONDITION(V24_1)   ;for synchronization

now_V24_1=0
```

Write/read functions

PLC program:

```
DIS_COUPL_RCI=OOFF_1_RCO

DIS_SER_1_RCI=NOT OOFF_1_RCO
```

For a controllable interface change-over switch, this signal will also be put out at a digital output.

Hints for a realization

The switch-over via a PLC-automated switch-over presents the difficulty that the PLC cannot recognize whether at the moment data, e. g. from a coupling, are transferred on the interface. With the READ device, a synchronization can be made via user signals, with the WRITE device it is necessary to wait for the end of the transfer with the instruction CONDITION device.

There is no return message that the switch-over procedure is finished.

The assignment of the functions coupling and READ / WRITE device for the physical interface is an indirect machine parameter. The adjustment can neither be read from the BAPS program nor from the PLC program, so that it is necessary to definitely take the consideration of a case of conflict in the programs into account.

When the function coupling is switched off, no Online status functions, no Online test and no data transfer are possible. A coupling call on the PG side (ROPS4) leads there to a 'Time-Out error'.

9.4.2 Transferred data

Variables of the type

- BINARY,
- INTEGER,
- REAL
- POINT,
- JC_POINT,
- CHAR,
- TEXT und
- one-dimensional arrays of the type BINARY, INTEGER, REAL, POINT, JC_POINT, CHAR and TEXT can be read

Approx. 256 characters per READ statement can be transferred in this case (protocol-dependent).

Write/read functions

For the variable type INTEGER, the following restriction applies to the transfer: Only integral numbers with a maximum of nine digits in the range between –999999999 and +999999999 can be read.

☞ **For the transfer scope of other variable types, the same restrictions as for the WRITE instruction apply.**

Messages

The control can put out the following messages:

- Interface erro: A WRITE/READ instruction has not been executed within the time that can be set.
- prot.err while READ: The defined transfer format, resp. the computer-internal protocol has not been observed in a READ instruction.
- prot.err while WRITE: A WRITE instruction cannot be not executed since the defined transfer format, resp. the computer-internal protocol has not been observed.

Write/read functions

9.5 Example READ/WRITE

You want to communicate the gripper position POSITION to your control at a specific place in the program which is then to be approached.

For control reasons you want to present the current gripper position at first at the PHG2000, block 38, and document it by means of your printer, block 39. Your printer is connected to the interface V24_2.

Example:

```
MOVE_REL dis12,dis14                      ;(35)

MOVE_REL CIRCULAR(cp7,cp8)                ;(36)

act_pos=POS                               ;(37)

WRITE 'Position end of circle =',act_pos  ;(38)

WRITE V24_2,'Pos.end of circle=',act_pos  ;(39)

WRITE ' '                                 ;(40)

WRITE 'Enter the coordinates of the'      ;(41)

WRITE 'gripper position-position'         ;(42)

READ position                             ;(43)

MOVE LINEAR EXACT position                ;(44)

WRITE V24_2, 'Coordinates of position'    ;(45)

WRITE V24_2, position                     ;(46)
```

With the program blocks 40, 41, 42 you write the request for the coordinate output onto the standard output device.

In block 43, the control expects the input of the coordinate values of the point POSITION via the keyboard of the standard output device.

You enter, for example, the position (200, 0, 120, –20, 40) for a 5-axis robot.

☞ **The input must be terminated with <Enter>.**

The read-in point variable can now be approached, block 44, and be put out to the printer, blocks 45, 46.

Write/read functions

Example READ PLC, WRITE PLC

Apart from the rho4 interface, the rho4 has an own data channel which is able to transfer larger data amounts from or to the PLC or software PLC. This data channel is addressed via the standard channel PLC.

The implicit declaration of the PLC channel of the type BNR_FILE enables to transfer variables and expressions of any type. The data are sent as binaries, i.e. unformated. There is no ASCII conversion as it is usual for DAT files or channels. For the realization of the data buffer to the PLC, it is recommended to represent it through a record type variable.

☞ **Since the name PLC is an implicitely declared BAPS standard variable, there is no need to define it by the user. It can simply be used, such as e. g. V24_1.**
As it is the case for all BAPS standard variables, the user can create a variable with the name PLC. It covers the standard variable PLC. It can no longer be addressed.

For the communication with the PLC, the following must be observed:

- only one channel is supported for the communication (standard channel PLC).
- The number of the data module must be contained in the data buffer to the PLC.
- A function module that transfers the data in the corresponding module (depending on the number of the module in the data buffer) is made available. If this function module is to be used, the length of the buffer must be entered in the first component of the data buffer and the number of the desired data component in the second component (both entries as INTEGER).

Write/read functions

Example: communication with the PLC

```
PROGRAM couplPLC

TYPE:                                   ;Type arrangement of the data buffer to
                                        ;the PLC

tPLCdata=RECORD

             INTEGER: length             ;Length of the data buffer

             INTEGER: PLC_dmno           ;Number of the data module to the PLC

             INTEGER: PLCcode            ;Function code from the PLC

             REAL: beltpos               ;Current belt position

             POINT: actpos               ;Current position of the robot in WC

          RECORD_END

tPLCdata: PLCdata                        ;Declaration of the data buffer

BELT: 501=belt_1

BEGIN

  PLCdata.length=sizeof(tPLCdata)       ;Determine extent of the data buffer by
                                        ;means of the new standard function Sizeof

   PLCdata.PLC_dmno=1                    ;Number of data module

   PLCdata.PLCcode=0                     ;Initialize function code

   PLCdata.beltpos=belt_1                ;Current belt position

   PLCdata.actpos=POS                    ;Current actual position

   WRITE PLC,PLCdata                     ;Transfer to the PLC

                                         ;any instructions

   READ PLC, PLCdata                     ;get modified data from the PLC

   IF PLCdata.PLCcode=0                  ;Check function code

   THEN WRITE 'Everything OK!'

   ELSE

   BEGIN

      WRITE 'Error!'

      HALT

   END

PROGRAM_END
```

Write/read functions

9.6 File operations

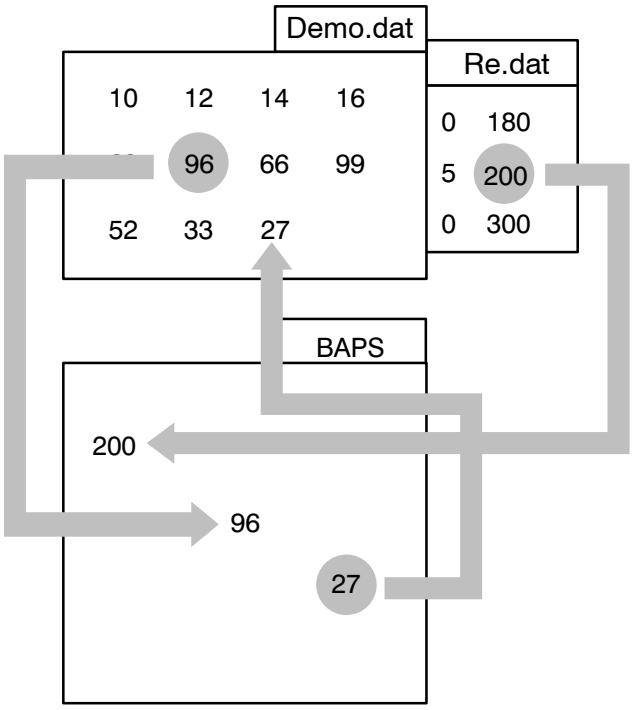
The following BAPS statements are available for the file operation:

- READ_BEGIN <file name>,[line number]
- READ <file name>, variable list
- WRITE <file name>, variable list
- END_OF_FILE <file name>
- WRITE_BEGIN <file name>
- WRITE_END <file name>
- CLOSE <file name>

File operations permit access to files of the type dat during the program run.

The control reads values of these dat files and includes them in the program run.

It is also possible to write arbitrary values from a BAPS program into a dat file.



Write/read functions

9.6.1 dat file

Numeric values for all variables valid in BAPS can be stored in files of the type dat.

The dat file thus represents a value reservoir for program variables.

Creation of a .dat file

Like files of the type qll, dat files are also created and edited with an editor.

Example:

You wish to create a dat file to store values which you can subsequently allocate to variables in a flexible way in the subsequent program run.

You thus create a dat file by inclusion of variables and comments.

Comments may be located at the line start or line end, but must always begin with a semicolon ';'.

If a comment is located at the start of a line, this means that the line is a pure comment line – and no values may be written into this line.

```
NAME: values.dat
DATE: 29. 2. 88
10      20      30      40                                      ;HEIGHT
100     200     300     400     500     600     700     800     ;LENGTH
12.15   21.2    1.5     -160.7  -90     0                       ;R_POS
```

Rules for .dat files

- Different data types may be included in the file in any order, e. g. INTEGER, REAL, JC_POINT.
- The following characters are permitted for the presentation of numbers: 0 1 2 3 4 5 6 7 8 9. + – decimal numbers (REAL) are presented as 6-digit floating-point numbers. For the presentation of CHAR and TEXT the characters ' '(space) to 'z' are permitted.
- The decimal point symbol '.' is permitted only for numeric values of the type REAL and thus also for the types POINT and JC_POINT.
- At least one space must always be placed between values to separate them. Any number of spaces is possible.
- An automatic switching to the next line takes place at the line end. For this reason, it is not necessary that there is a space after the last value in a line.
- Line numbers are visible within the dat file only in Edit mode, i. e. no line information for Print or Write.

Write/read functions

Access to a .dat file

If you wish to access one or more dat files, these must be declared as variables of the type file.

9.6.2 .dat file declaration

Syntax:

```
FILE :  file name [,file name]
```

Example:

```
FILE: values,erg,distances

INTEGER: number,i

TEXT: display,font
```

The file declaration must be contained in the declaration part of the program.

The control can read or write values from several values of the type dat within a BAPS program. Simultaneous reading out a file opened for writing is not possible.

9.6.3 File read statement

Syntax:

```
READ file name,variable[{,variable}]
```

The control is requested to read in values from a file of the type dat by the instruction READ.

The declared file name of the dat file must be entered in the program after the READ instruction so that the control knows from where it is to obtain the desired data.

This is followed, separated by a comma, by specification of the program variable to which a value is to be assigned from the dat file by the READ instruction.

The read instruction can be extended by allocation of a second or further program variable from the same dat file.

Example:

```
READ values,number ;The control reads-in an integral value for the variable number
                   ;from the file values.dat
```

Write/read functions

```
READ erg,i,number  ;The control reads-in a value from the file erg.dat both for
                   ;the variable i and for number
```

9.6.4 Selection of a value within the dat file

The position of the invisible READ pointer is decisive in determining which value is read within the dat file as a result of a READ instruction.

This pointer is used by the control so that it knows at which point in the file it was last active as a result of a READ operation.

Whith the next READ instruction, the control automatically jumps to the following value, reads this value and then postions the invisible READ pointer.

In this way, the control reads from value to value and from line to line.

Example

Name: value.dat

Date: 29. 2. 88

10	20	30	40					;HEIGHT
100	200	300	400	500	600	**700**	800	;LENGTH
12.15	21.2	1.5	–160.7	-90	0			;R_POS

☞ **If the variable type from the program does not agree with the read value in the dat file, the control puts out an error message. Leading blanks and comments are ignored when reading variables of the type BINARY, INTEGER, REAL, POINT and JC_POINT. When reading variables of the type CHAR or TEXT, the characters are read directly from the position of the read pointer. If variables of the type CHAR or TEXT are to be read from the beginning of a line, it is possible instead to read space characters included by an editor at the end of the previous line. In this case, you should position the read pointer by using the READ_BEGIN statement.**

Write/read functions

9.6.5 READ_BEGIN selection of a specific line

Syntax:

```
READ_BEGIN file name [,line number]
```

The invisible READ pointer jumps before the start of a desired line as a result of the BAPS instruction READ_BEGIN.

The next READ instruction has the effect that in the first value of this desired line is read and assigned to a specific variable.

Example

```
READ_BEGIN values,7      ;The invisible pointer jumps to before line 7 in the file
                         ;values.dat

READ_BEGIN values,(v+n)  ;The invisible pointer jumps to a certain line in the file
                         ;values.dat, which is got by the expression V+N

READ values,number       ;The first value of the line (v+n) is read-in for the
                         ;program variable number
```

☞ **The BAPS instruction READ_BEGIN is a positioning instruction for the invisible READ pointer. It does not result in reading a value.**

☞ **If no line number is specified, the control interprets this as a positioning instruction to the start of the file, and therefore positions the invisible READ pointer before the start of line 1.**

9.6.6 BAPS standard function END_OF_FILE

Syntax:

```
END_OF_FILE (file name)
```

This function permits interrogation of whether the file end has been reached when reading a dat file, i. e. whether the invisible READ pointer is pointing to the last value of the file.

Interrogation can take place by means of the BAPS instruction 'IF THEN'.

Write/read functions

Example:

```
IF END_OF_FILE (values)

THEN JUMP finished ;As soon as the last value in the file value.dat has been
                   ;read, the control jumps to the jump label finished
                   ;in the main program
```

☞ **The dat file name must be placed in brackets.**

9.6.7 BAPS instruction WRITE

Syntax:

```
WRITE file name,variable[{,variable}]
```

It is possible to write one or more values into a dat file by using the instruction WRITE and specifying a declared dat file.

If you wish to write several values in one line, this must be done with a WRITE instruction. Each WRITE instruction opens a new line.

Example

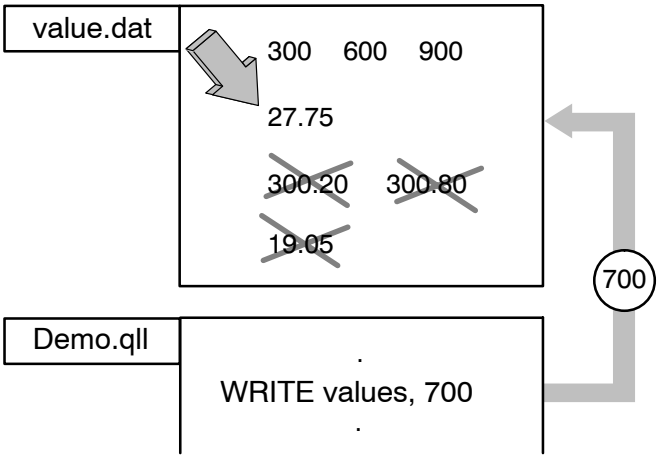
```
FILE: values,erg

INTEGER: w

WRITE values,700  ;The value 700 is written in the file values.dat

WRITE erg,v,w-10  ;The value which the variable v has at the time of the WRITE
                  ;instruction, is written in the file erg.dat along with the
                  ;value yielded by the expression w-10
```

☞ **A file opened for writing can be read only after a CLOSE instruction.**

The WRITE instruction writes the desired value in the dat file in the line which follows the current position of the invisible WRITE pointer.

Write/read functions

☞ **The file is overwritten as from this line! The previous content of this and the following lines is thus deleted!**



The invisible WRITE pointer always identifies the position in the dat file at which the program last executed a WRITE instruction.

A WRITE_BEGIN or WRITE_END instruction must be programmed once before the first WRITE instruction.

9.6.8 WRITE_BEGIN selection of a specific line

Syntax:

```
WRITE_BEGIN file name[,line number]
```

The instruction WRITE_BEGIN results in a jump of the invisible WRITE pointer to the start of a certain line and the file is opened for writing with this statement.

As a result of this, the next WRITE instruction writes values from the BAPS program into the desired line of the dat file. The previous content of this line and all following ones is deleted.

Example:

```
WRITE_BEGIN values,(i+r) ;In the file values.dat, the invisible WRITE pointer
                         ;jumps before the line whose line number is calculated
                         ;from the expression I+R

WRITE values,f,100-r     ;The values for the program variable f and the expression
                         ;100-r  are written into the line (i+r) of the file
                         ;values.dat
```

☞ **The file is deleted (overwritten) as from the line number specified in the WRITE_BEGIN instruction.**

A WRITE_BEGIN instruction must be programmed once before the first WRITE instruction. This opens the file and positions the WRITE pointer to the start of the file or to an arbitrary line number.

After this, no further WRITE_BEGIN instruction is normally necessary, unless the dat file is to be deleted again from a certain line number.

☞ **A new dat file is automatically created if no dat file with the file name specified in the WRITE_BEGIN instruction exists yet.**

9.6.9 BAPS instruction WRITE_END

Syntax:

```
WRITE_END file name
```

The BAPS instruction WRITE_END results in a jump by the invisible WRITE pointer an the end of the dat file.

☞ **If no dat-file with the file name indicated in the WRITE_END instruction exists, a new dat-file is then automatically created**.

So it is not possible for a dat line to be overwritten with the next WRITE instruction. The desired values are then placed at the end of the file.

It is the purpose of this instruction to complement arealdy existing dat files.

Example:

```
WRITE_END, values    ;The invisible WRITE pointer jumps to the end of the
                     ;dat file

WRITE values, 700    ;The value 700 is written in a new line at the file end
```

Write/read functions

9.6.10 BAPS instruction CLOSE

Syntax:

```
CLOSE file name
```

An open file is closed with the close instruction.

Example:

```
CLOSE values ;The file is closed
```

9.6.11 Write, read BINARY files

The BAPS syntax is compared with the already realized ASCII file I/O expanded by the keyword BNR_FILE. The read and write operations differ with respect to the syntax not from the instructions of the ASCII file I/O. The disctinction between the ASCII file I/O used so far and the BINARY files I/O is made in the file declaration. BINARY files have the file extension .bnr.

ASCII files, characterised by the file extension .dat, are declared with the keyword FILE and without type specification. Implicitely, the type CHAR is assumed. A declaration with an explicite specification of the type CHAR leads to the same result, e. g. FILE: dat_1 or FILE CHAR: dat_1.

There are two possibilities for the declaration of BINARY files. The first one uses the new BAPS keyword BNR_FILE. A file declared in this way may contain different BAPS data types in the BINARY format.

The second possibility requires only the already known BAPS keyword FILE and additionally the data type of the data stored in the file. This kind of declaration operates with files of one type only. This means that a file may only contain data of one single BAPS data type. At the time of compilation, all file operations are checked for type compatibility. File operations with bnr-files are executed faster than those with dat files since there is no conversion of the data from ASCII to BINARY and vice versa. The data are during reading furthermore accessed directly from the bnr file and not via the block-oriented file management, which makes it, however, necessary, that the file is available in the user memory in a sequential form. The sequentialization of the file is made in the READ_BEGIN instruction which opens the file at the same time.

Write/read functions

BAPS instructions of the binary file I/O

```
BNR_FILE: file_1,file_2          ;Declaration A, mixed bnr files

FILE JC_POINT: p_jc              ;Declaration B, bnr files of one type only

FILE POINT: p_wc

FILE REAL: real_var

READ_BEGIN file_1                ;Read operations, positioning of the read pointer

READ_BEGIN file_1,byte_offset

READ file_1,@p1,@p2

WRITE_BEGIN file_1               ;Write operations, positioning of the write pointer

WRITE_BEGIN p_wc,byte_offset

WRITE_END file_1

WRITE file_1,@p1,@p2

CLOSE file_1                     ;Close instruction

END_OF_FILE (file_1)             ;End-of-file function
```

Read operations

bnr files are opened with the instruction READ_BEGIN and sequentialized in the user memory to ensure quick access. If a sequentialization is for memory space reasons not possible, the process in which the READ_BEGIN instruction is programmed will be aborted with the runtime error message 'file sequ. failed'. One byte offset can be specified as an option in the READ_BEGIN instruction, by means of which the internal read pointer can be positioned to an arbitrary byte address, e. g. READ_BEGIN file_1, 80.

The first 80 bytes of the file are skipped in the example. The read operation starts with byte 81. If a byte offset is programmed, it must be ensured that a reasonable value is at the corresponding place of the file which is compatible with the following read instruction.

The next example shows the reading of point values from a .bnr file. The file to be read is assumed to have a head of variable length followed by a variably long data section. The head length is assumed to be stored as an INTEGER value in the first 4 bytes of the file.

Write/read functions

Example

```
PROGRAM l_bnr

BNR_FILE: f_pnt                    ;Mixed bnr file

INTEGER: head_length

POINT: p1

BEGIN

   READ_BEGIN f_pnt                ;Read file head length from bnr file

   READ f_pnt,head_length

   READ_BEGIN f_pnt, head_length   ;Position read pointer to the start of the data
                                   ;section

   label_1:                        ;Read loop

      READ f_pnt,p1                ;Read points from bnr file and approach all
                                   ;points until the end of file

      MOVE LINEAR VIA p1

      IF not END_OF_FILE(f_pnt)

      THEN JUMP label_1

      CLOSE f_pnt

PROGRAM_END
```

Write operations

bnr files can be generated or overwritten in the rho4. The new file is available for further processing after having been closed by the BAPS instruction CLOSE. The file is not necessarily sequentialized at this time. The sequentialization of all files is made with the interface signal 'reset'. Similar to ASCII file I/O, files can be recreated or overwritten if they already exist. With the instruction WRITE_END it is possible to add further data to the already existing ones. The WRITE_BEGIN instruction offers the possibility to specify a byte offset. In the following WRITE instructions, the number of bytes specified in the byte offset will be maintained. The remaining bytes of the file are overwritten with new data.

Data format

INTEGER and BINARY variables are stored as 4-byte integral values in the complement of two. Logic 1 is presented as INTEGER value 1 and logic 0 as INTEGER value 0. REAL values also occupy 4 bytes and are stored in the 32-bit IEEE floating-point format. The types POINT and JC_POINT consist of REAL values. The number depends on the axis number (point components). Other BAPS types are not permitted. bnr files do not contain a separation sign.

Write/read functions

Restrictions

A file may not be opened simultaneously for writing by several processes (subprocesses). The file can be opened again after having been closed by a process.

☞ **Files opened for writing cannot be opened for reading.**

In a subprocess, e. g. PARALLEL or ALSO branch, it is not permitted to simultaneously open a file which has already been opened for reading by the associated main process or another subprocess. If a file has for example simultaneously to be accessed for reading in two subprocesses, it has to be copied first. Subprocess 1 then works with bnr file 1 and subprocess 2 with bnr file 2. The copy operation can take place on the BAPS level by using special function 4 (COMMAND).

CLOSE instructions only act in the process (subprocess) in which also the READ_BEGIN instruction has been programmed. When the process is ended, the files opened by this process will be closed automatically if no CLOSE instruction has been programmed.

9.7 Write/read in PLC and Windows applications

This chapter describes options which do not require any syntactical expansions of the BAPS language scope. These are for example expansions concerning the communication with a PLC or with other Windows applications.

Expansion of the standard channels

In addition to the available standard channels, e. g. V24_1 to V24_4, additional standard channels are defined for a coupling with the PLC resp. for the communication with Windows.

Communication with the PLC

Apart from rho4 interface, an own data channel is set up on the rho4.1 which is in a position to transfer larger data volumes from or to the PLC resp. software PCL. This data channel is addressed via the standard channel PLC which is in BAPS defined as follows:

Syntax:

```
PLC_channel=BNR_FILE:PLC.
```

The following applies to the above syntax:

PLC Name of the standard channel to the PLC. This name is no reserved word.

The implicite declaration of the PLC channel of the type BNR_FILE makes it possible to transfer variables and expressions of any type. The data are transmitted binary, i. e. unformatted; above all no ASCII conversion takes place, as this is normally the case with dat files or channels. For the realization of the data buffer to the PLC it is reasonable to present the same by a record variable.

☞ **Since the name PLC is an implicitely declared BAPS standard variable, there is no need to define it by the user. It can simply be used, such as e. g. V24_1. As it is the case with all BAPS standard variables, the user can in this case create, too, a variable with the name PLC. This variable then covers the standard variable PLC. It can thus no longer be addressed.**

The following has to be observed concerning the communication with the PLC:

- Only one channel is supported for communication, standard channel PLC.
- The number of the data module must be contained in the data buffer to the PLC.

Write/read functions

- A function module is made available which transfers the data into the corresponding data module, depending on the module number in the data buffer. If this function module is to be used, the length of the buffer itself must be entered into the first component of the data buffer and the number of the desired data module into the second component. Both specifications are of the type INTEGER.

Example communication with the PLC

```
PROGRAM couplPLC

TYPE:                                   ;Type declaration of the data buffer to
                                        ;the PLC

tPLCdata=RECORD

            INTEGER:  length            ;Length of the data buffer

            INTEGER:  PLC_dmno          ;Number of the data module on the PLC

            INTEGER:  PLCcode           ;function code of the PLC

            REAL:     beltpos           ;Current belt position

            POINT:    aktpos            ;Current position of the robot in WC

         RECORD_END

tPLCdata: PLCdata                       ;Declaration of the data buffer

BELT: 501=belt_1

BEGIN

   PLCdata.length=sizeof (tPLCdata)     ;Determine the size of the data buffer
                                        ;by means of the new standard function
                                        ;sizeof

   PLCdata.PLC_dmno=1                   ;Number of the data module

   PLCdata.PLCcode=0                    ;Initialization of function code

   PLCdata.beltpos=belt_1               ;Current belt position

   PLCdata.actpos=POS                   ;Current actual position

   WRITE PLC,PLCdata                    ;Transfer to PLC

                                        ;Arbitrary statements

   READ PLC, PLCdata                    ;Get changed data from the PLC

   IF PLCdata.PLCcode=0                 ;Check function code

   THEN WRITE 'Everything OK!'

   ELSE
```

Write/read functions

```
   BEGIN

      WRITE 'Error!'

      HALT

   END

PROGRAM_END
```

Communication with Windows applications

The communication between the rho4 real time core and Windows applications takes place via a TCP/IP connection. It does not matter in this respect whether Windows applications and rho4 real time core run on different computers or on a common hardware.

Standard channels are available in BAPS for this data exchange. The user can make a communication with Windows application, using the READ and WRITE statements of the BAPS language scope.

The following standard channels are available for this:

- Win_1
- Win_2
- Win_3
- Win_4

The implicite declaration of the Win channel of the type BNR_FILE makes it possible to transfer variables and expressions of any type. The data are transmitted binary, i. e. unformatted; above all no ASCII conversion takes place, as this is normally the case with dat files or channels. For the realization of the data buffer to the PLC it is reasonable to present the same by a record variable.

Example communication with Windows

```
PROGRAM couplWIN

TYPE:                                   ;Type declaration of the data buffer

tWINdata=RECORD

            INTEGER:  length            ;Length of the data buffer

            INTEGER:  WINcode           ;External function code for
                                        ;Windows application

            REAL:   beltpos             ;Current belt position

            POINT: actpos               ;Current position of the robot in WC

         RECORD_END

tWINdata: Windata                       ;Declaration of the data buffer
```

Write/read functions

```
BELT: 501=belt_1

BEGIN

   WINdata.length=sizeof(WINdata)      ;Determine the size of the data buffer by
                                       ;means of the new standard function
                                       ;sizeof

   WINdata.WINcode=0                   ;Initialize function code

   WINdata.beltpos=belt_1              ;Current belt position

   Windata.actpos=POS                  ;Current actual position

   WRITE Win_1, WINdata                ;Transfer to the Windows application

                                       ;Arbitrary statements

   READ Win_1, WINdata                 ;Get changed data from Windows application

   IF WINdata.WINcode=0                ;Check function code

   THEN WRITE 'Everything OK!'

   ELSE

   BEGIN

      WRITE 'Error!'

      HALT

   END

PROGRAM_END
```

10 BAPS3 keywords

☞ **Hereafter, all currently reserved language symbols for the BAPS3 are listed. The listed language symbols must not be used as variables, file names or sub-program names in a BAPS3 program.**

German	English
@	@
ALLE	EVERY
ANFANG	BEGIN
ANSONSTEN	DEFAULT
AUSGANG	OUTPUT
BAND	BELT
BINAER	BINARY
BIS	UNTIL
CIRCA	APPROX
DANN	THEN
DATEI	FILE
DEF	DEF
DEZ	REAL
EINGANG	INPUT
ENDE	END
EXAKT	EXACT
EXKLUSIV_ENDE	EXCLUSIVE_END
EXKLUSIV	EXCLUSIVE
EXTERN	EXTERNAL
FAHRE	MOVE
FALLS	CASE
FALLS_ENDE	CASE_END
FEHLER	ERROR
FELD	ARRAY
GANZ	INTEGER
GLEICH	EQUAL
GLOBAL	PUBLIC
GRENZE_AUS	LIMIT_OFF
HALT	HALT
KONSTANTE	CONST

BAPS3 keywords

German	**English**
KREIS	CIRCULAR
LESE	READ
LESE_ANFANG	READ_BEGIN
LINEAR	LINEAR
MAL	TIMES
MAX_ZEIT	MAX_TIME
MIT	WITH
MK_PUNKT	JC_POINT
MOD	MOD
NACH	TO
NICHT	NOT
ODER	OR
PARALLEL	PARALLEL
PARALLEL_ENDE	PARALLEL_END
PAUSE	PAUSE
PERMANENT	PERMANENT
PRIO	PRIO
PROGR_SLOPE	PROGR_SLOPE
PROGRAMM_ENDE	PROGRAM_END
PROGRAMM	PROGRAM
PTP	PTP
PUNKT	POINT
REF_PKT	REF_PNT
RHO_FKT	RHO_FCT
RK_RAHMEN	WC_FRAME
RSPRUNG	RETURN
SATZ_SLOPE	BLOCK_SLOPE
SCHLIESSE	CLOSE
SCHREIBE	WRITE
SCHREIBE_ANF	WRITE_BEGIN
SCHREIBE_ENDE	WRITE_END
SEMAPHOR	SEMAPHORE
SONST	ELSE
SOWIE	ALSO
SPRUNG	JUMP
SPZ_FKT	SPC_FCT

BAPS3 keywords

German	English
START	START
STOP	STOP
SYNC	SYNC
SYNCHRON	SYNCHRON
SYNCHRON_ENDE	SYNCHRON_END
TEXT	TEXT
TYP	TYPE
UEBER	VIA
UND	AND
UNTERBRECHE	BREAK
UP	SUBROUTINE
UP_ENDE	SUB_END
VAR	VAR
VERBUND	RECORD
VERBUND_ENDE	RECORD_END
VERSCHIEBE	MOVE_REL
WARTE	WAIT
WDH	REPEAT
WDH_ENDE	REPEAT_END
WENN	IF
WERKZEUG	TOOL
WERT	VALUE
ZEICHEN	CHAR
ZUORDNE	ASSIGN

BAPS3 keywords

BAPS3 Translator statements

German	English
ACHSNAMEN	JC_NAMES
DATEI_FEHLER	FILE_ERROR
EINFUEGE	INCLUDE
FEHLER	ERROR
INT	INT
KINEMATIK	KINEMATICS
KOORDINATEN	WC_NAMES
PROZESS_ART	PROCESS_KIND
SER_EA_STOP	SER_IO_STOP
STEUERUNG	CONTROL
TESTINFO	DEBUGINFO
WARNUNG	WARNING
WERK_KOORD	POSE

BAPS3 keywords

BAPS3 standard variables

German	English
@IPOS	@POS
@MPOS	@MPOS
A	A
AFAKTOR	AFACTOR
AFEST	AFIX
DFAKTOR	DFACTOR
GRENZE_MAX	LIMIT_MAX
GRENZE_MIN	LIMIT_MIN
HBG	MCP
IPOS	POS
PHG	PHG
R	R
R_PTP	R_PTP
RK_SYSTEM	WC_SYSTEM
SER_1	SER_1
SER_2	SER_2
SER_3	SER_3
SER_4	SER_4
SPS	PLC
T	T
TFEST	TFIX
TTY	TTY
V	V
V_PTP	V_PTP
V24_1	V24_1
V24_2	V24_2
V24_3	V24_3
V24_4	V24_4
VFAKTOR	VFACTOR
VFEST	VFIX
WIN_1	WIN_1
WIN_2	WIN_2
WIN_3	WIN_3
WIN_4	WIN_4

BAPS3 keywords

BAPS3 standard functions

German	English
ABS	ABS
ATAN	ATAN
BNR_DATEI	BNR_FILE
CHR	CHR
COS	COS
DATEI_ENDE	END_OF_FILE
GANZ_ZFELD	INT_ASC
GANZTEIL	TRUNC
GROESSE_VON	SIZEOF
MK	JC
ORD	ORD
RK	WC
RK_RECHNUNG	WC_CALCULATION
RUNDUNG	ROUND
SIN	SIN
SPS_PROZESS	PLC_PROCESS
SPS_ZEIT	PLC_TIME
UNTERBRECHE	BREAK
WURZEL	SQRT
ZFELD_GANZ	ASC_INT
ZUSTAND	CONDITION

BAPS3 standard constants

German	English
CLS	CLS
RK_UR	WC_UR
VERSION	VERSION

BAPS3 keywords

General BAPS3 keyword list

German	English
@	@
@IPOS	@POS
@MPOS	@MPOS
A	A
ABS	ABS
ACHSNAMEN	JC_NAMES
AFAKTOR	AFACTOR
AFEST	AFIX
ALLE	EVERY
ANFANG	BEGIN
ANSONSTEN	DEFAULT
ATAN	ATAN
AUSGANG	OUTPUT
BAND	BELT
BINAER	BINARY
BIS	UNTIL
BNR_DATEI	BNR_FILE
CHR	CHR
CIRCA	APPROX
CLS	CLS
COS	COS
DANN	THEN
DATEI	FILE
DATEI_FEHLER	FILE_ERROR
DATEI_ENDE	END_OF_FILE
DEF	DEF
DEZ	REAL
DFAKTOR	DFACTOR
EINFUEGE	INCLUDE
EINGANG	INPUT
ENDE	END
EXAKT	EXACT
EXKLUSIV_ENDE	EXCLUSIVE_END
EXKLUSIV	EXCLUSIVE
EXTERN	EXTERNAL

BAPS3 keywords

German	**English**
FAHRE	MOVE
FALLS	CASE
FALLS_ENDE	CASE_END
FEHLER	ERROR
FELD	ARRAY
GANZ	INTEGER
GANZ_ZFELD	INT_ASC
GANZTEIL	TRUNC
GLEICH	EQUAL
GLOBAL	PUBLIC
GRENZE_AUS	LIMIT_OFF
GRENZE_MAX	LIMIT_MAX
GRENZE_MIN	LIMIT_MIN
GROESSE_VON	SIZEOF
HALT	HALT
HBG	MCP
INT	INT
IPOS	POS
KINEMATIK	KINEMATICS
KONSTANTE	CONST
KOORDINATEN	WC_NAMES
KREIS	CIRCULAR
LESE	READ
LESE_ANFANG	READ_BEGIN
LINEAR	LINEAR
MAL	TIMES
MAX_ZEIT	MAX_TIME
MIT	WITH
MK	JC
MK_PUNKT	JC_POINT
MOD	MOD
NACH	TO
NICHT	NOT
ODER	OR
ORD	ORD
PARALLEL	PARALLEL

BAPS3 keywords

German	English
PARALLEL_ENDE	PARALLEL_END
PAUSE	PAUSE
PERMANENT	PERMANENT
PHG	PHG
PRIO	PRIO
PROGR_SLOPE	PROGR_SLOPE
PROGRAMM_ENDE	PROGRAM_END
PROGRAMM	PROGRAM
PROZESS_ART	PROCESS_KIND
PTP	PTP
PUNKT	POINT
R	R
R_PTP	R_PTP
REF_PKT	REF_PNT
RHO_FKT	RHO_FCT
RK	WC
RK_RAHMEN	WC_FRAME
RK_RECHNUNG	WC_CALCULATION
RK_SYSTEM	WC_SYSTEM
RK_UR	WC_UR
RSPRUNG	RETURN
RUNDUNG	ROUND
SATZ_SLOPE	BLOCK_SLOPE
SCHLIESSE	CLOSE
SCHREIBE	WRITE
SCHREIBE_ANF	WRITE_BEGIN
SCHREIBE_ENDE	WRITE_END
SEMAPHOR	SEMAPHORE
SER_1	SER_1
SER_2	SER_2
SER_3	SER_3
SER_4	SER_4
SER_EA_STOP	SER_IO_STOP
SIN	SIN
SONST	ELSE
SOWIE	ALSO

BAPS3 keywords

German	English
SPRUNG	JUMP
SPS	PLC
SPS_PROZESS	PLC_PROCESS
SPS_ZEIT	PLC_TIME
SPZ_FKT	SPC_FCT
START	START
STOP	STOP
STEUERUNG	CONTROL
SYNC	SYNC
SYNCHRON	SYNCHRON
SYNCHRON_ENDE	SYNCHRON_END
T	T
TESTINFO	DEBUGINFO
TEXT	TEXT
TFEST	TFIX
TTY	TTY
TYP	TYPE
UEBER	VIA
UND	AND
UNTERBRECHE	BREAK
UP	SUBROUTINE
UP_ENDE	SUB_END
V	V
V_PTP	V_PTP
V24_1	V24_1
V24_2	V24_2
V24_3	V24_3
V24_4	V24_4
VAR	VAR
VERBUND	RECORD
VERBUND_ENDE	RECORD_END
VERSCHIEBE	MOVE_REL
VERSION	VERSION
VFAKTOR	VFACTOR
VFEST	VFIX
WARNUNG	WARNING

BAPS3 keywords

German	English
WARTE	WAIT
WDH	REPEAT
WDH_ENDE	REPEAT_END
WENN	IF
WERK_KOORD	POSE
WERKZEUG	TOOL
WERT	VALUE
WIN_1	WIN_1
WIN_2	WIN_2
WIN_3	WIN_3
WIN_4	WIN_4
WURZEL	SQRT
ZEICHEN	CHAR
ZFELD_GANZ	ASC_INT
ZUORDNE	ASSIGN
ZUSTAND	CONDITION

BAPS3 keywords

Notes:

Appendix

A Appendix

A.1 Abbreviations

Abbreviation	Meaning
BAPS3	Programming language; Bewegungs- und Ablaufprogrammiersprache, Version 3; programming language
C:	Hard disk drive
CAN	Controler Area Network
DAC	Digital-analog converter
EEPROM	Electronically erasable programmable read-only memory
EGB	Elektrostatic sensitive components
ESD	Electrostatic discharge
LF	Line feed
MPP	Machine parameter program
MSD	Machine state display
PCL	Memory-programmable control
PE	Protective earth
PHG	Hand-held programming unit
POS	Actual position
PTP	Point to point
RC	Robot control
ROD	Incremental encoder
RPM	Rounds per minute
ROPS4	Robot programming system for rho4
TCP	Tool center point
WC	World coordinates

Appendix

A.2 Index

Appendix

Appendix

Rexroth
Bosch Group

Bosch Rexroth AG
Electric Drives and Controls
P.O. Box 13 57
97803 Lohr, Germany
Bgm.-Dr.-Nebel-Str. 2
97816 Lohr, Germany
Phone +49 93 52-40-50 60
Fax +49 93 52-40-49 41
service.svc@boschrexroth.de
www.boschrexroth.com

Printed in Germany
DOK-RHO*4*-BAPSI*SOFTH-PR06-EN-P