rho 3

BAPS2
Programming Instructions

Version 101

BOSCH
Automationstechnik

rho 3

BAPS2
Programming Instructions

1070 073 033-101 (92.06) GB

Reg. Nr. 16149-03

Discretionary charge 45.– DM

1. BAPS2 Programming Instructions

This manual describes the robot programming language **BAPS2**.

It is directed at all those who use or are responsible for planning use of a Bosch rho3 control.

These programming instructions assume basic knowledge of programming languages as well as knowledge of how the rho3 control functions. This knowledge can be gained, for example, by attending the training courses offered by Bosch.

The relevant safety regulations must be observed when realizing the work task in question.

1. 1. General

BAPS2 is a task−oriented higher−level programming language for programming the **rho 3** control family. **BAPS2** is the further development of the programming language **BAPS** and stands for motion and sequence programming language.

As a task−oriented programming language for robot and handling systems, **BAPS2** is an extensive but easy−to−learn language. It allows quick and maintenance−friendly realization of user tasks.

The language commands can currently be written in either German or English.

The general syntax of each statement is given before every detailed statement description in this document.

The following symbols are used for the purpose of description:

Bold	means part of the language element and must be written,
|	alternative,
{ }	may be optionally specified several times,
[ ]	may be optionally specified once,

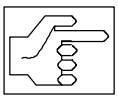
1. 2. Mode of compiler operation

The **BAPS2** compiler is integrated both in the operating system of the **rho 3** control and in the offline programming system **ROPS3**/**IQpro**
.

The **BAPS2** compiler generates the following files from the statements in the source file(**QLL**):

IRD file

This file contains the program code which the rho 3 control executes 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 space 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.

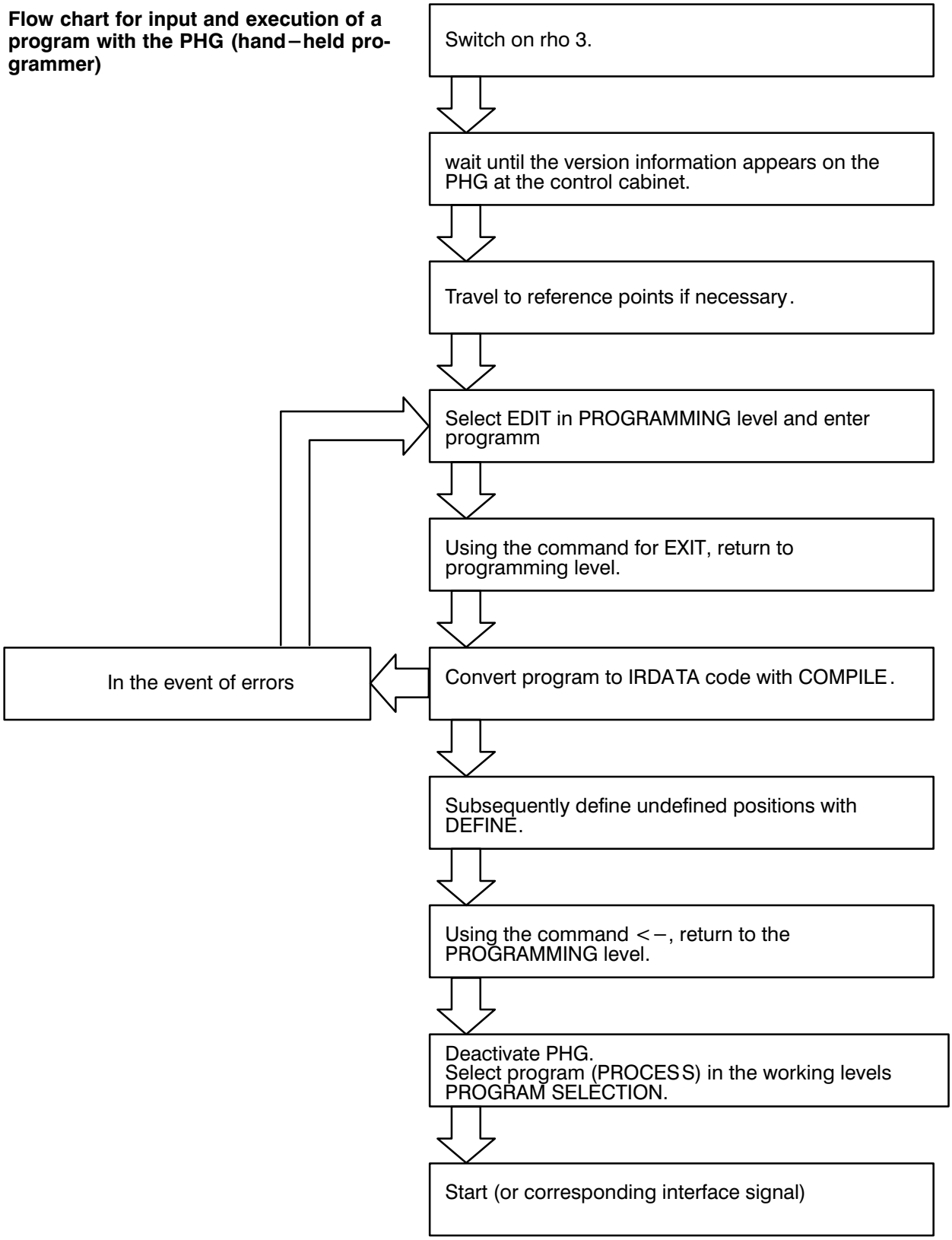
SYM file

This file contains information on the variable names used in the program and is always required for testing BAPS2 programs.

ERR file

This contains the errors detected during compilation of the BAPS2 program in plain text.

Flow chart for input and execution of a program with the PHG (hand−held programmer)



2. Program structuring

Programs are stored in files which are stored in the main memory of the control or on a data medium of your programming system. The program files are identified with names to permit 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.

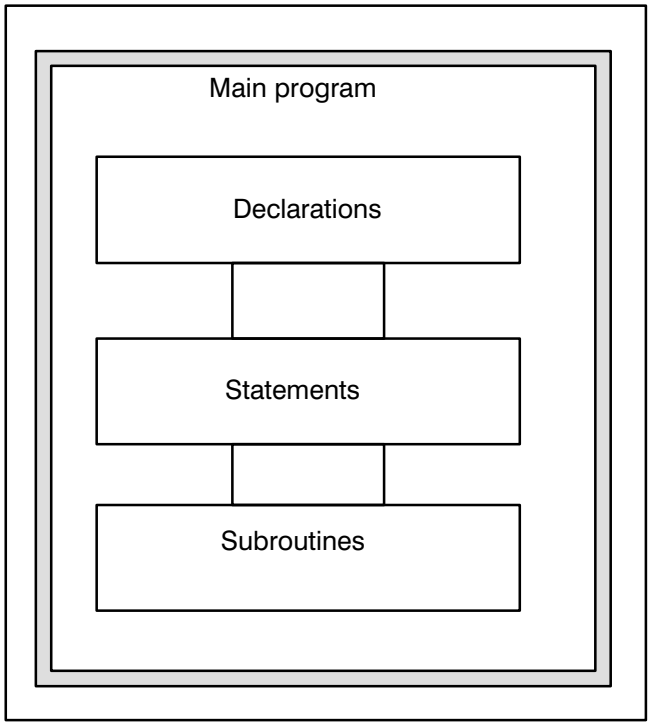
In the **rho 3** control, a distinction is made between main programs and subroutines. Main programs are programs which exist as files and which can be started as a **rho 3 BAPS user process**. It is possible to call other main programs which exist in the control's main memory from within a main program. We then speak of **external subroutines**; these must be declared correspondingly in the declaration part. Also refer to Chapter 2.3 External main program and subroutine declaration.

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

2. 1. Main program structure

Each main program consists of:

- Declaration part,
- Statement part and optionally
- Subroutine declaration(s).



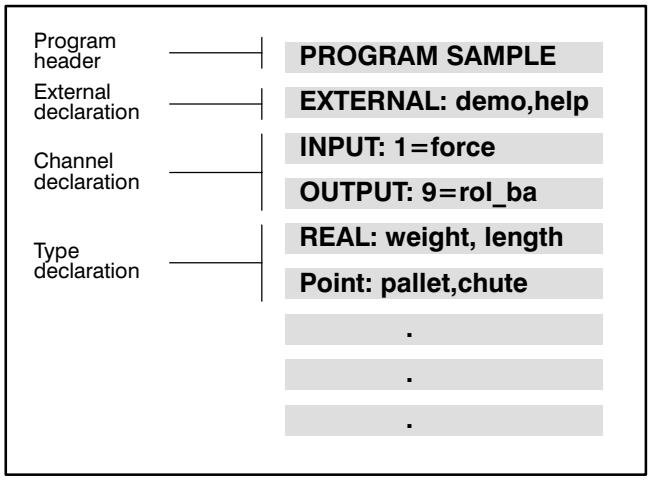
2. 1. 1. Declaration part

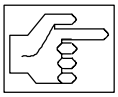
The declaration part is located at the start of the main program. The names which occur in the main program are stated in the declaration part.

This relates to the following:

- **Program header**, 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,
- **Type declaration**, the names of the variables which occur in the program.

The declaration part must be separated from the statement part by the keyword "**BEGIN**".



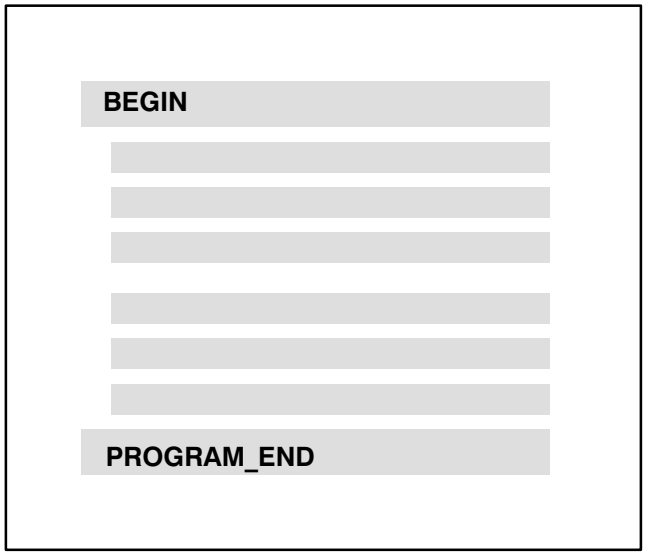


Undeclared variabls are assumed to be variables of the type POINT or JC_POINT.

2. 1. 2. Statement part

The statements which are to be executed are programmed in the statement part. These include, for example:

- Motion statements,
- Deceleration values and halt,
- Main program calls,
- Subroutine calls,
- Program part repetitions,
- Program jumps,
- Arithmetic operations,
- .
- .
- .



The statement part is located between the keywords:

"**BEGIN**" and "**PROGRAM_END**".

2. 1. 3. Subroutine declaration

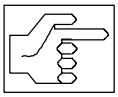
Any subroutines are listed at the end of the main program.

2. 2. 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 sub-routines are listed.



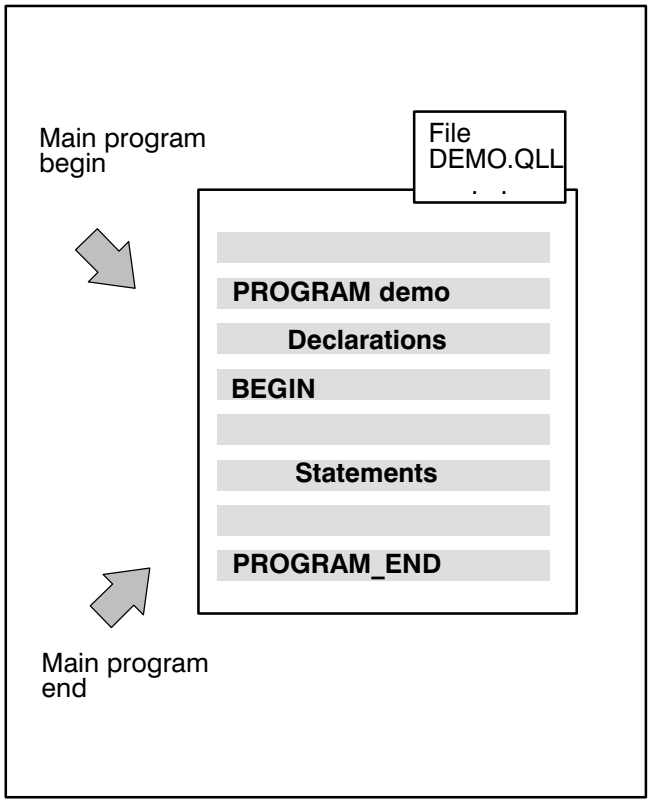
The program name and file name must be identical!

Example: It is wished to give a program the name “demo”:

```
PROGRAM demo
```

End of program demo:

```
PROGRAM_END
```



2. 3. Main program call in the main program

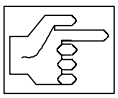
(External declaration)

A main program can consist of several external individual main programs. The external main programs must be declared after the start of the declaration part by **EXTERNAL** and must be present in the control as an **IRD** file when called.

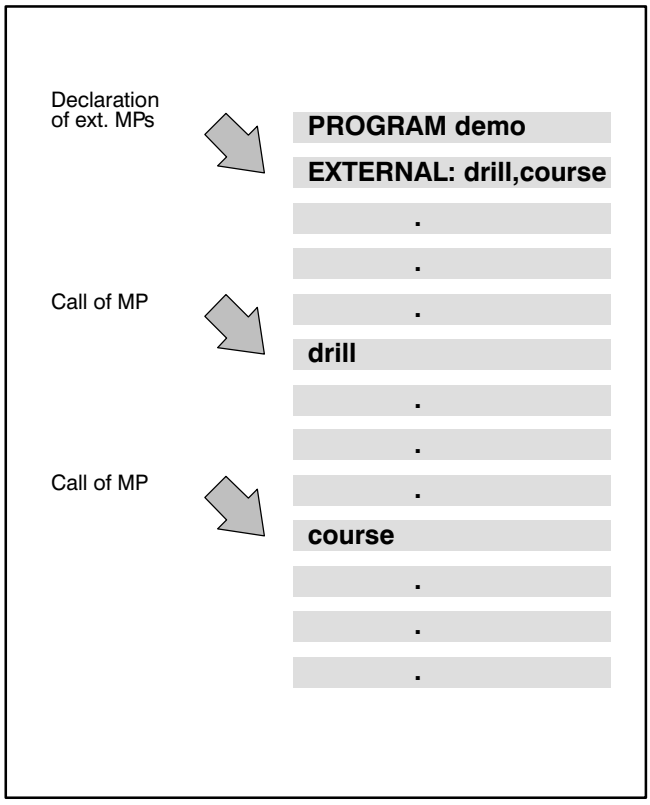
External main programs can be optionally provided with transfer parameters. The number, order and data types must agree with the declaration upon parameter transfer. All variables are permitted as parameters except for variables of the types **array** and **channels** .

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 the declared program name in order to call external programs in the active main program.



The main program and the called external program 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 with the statement **EXTERNAL:**e.g.

```
EXTERNAL: drill, course
```

External declaration with parameter transfer:

```
EXTERNAL: WITHPAR (VALUE INTEGER: I)
```

The main program must be declared correspondingly

PROGRAM WITHPAR (**VALUE INTEGER :** I)

Main program call in the main program

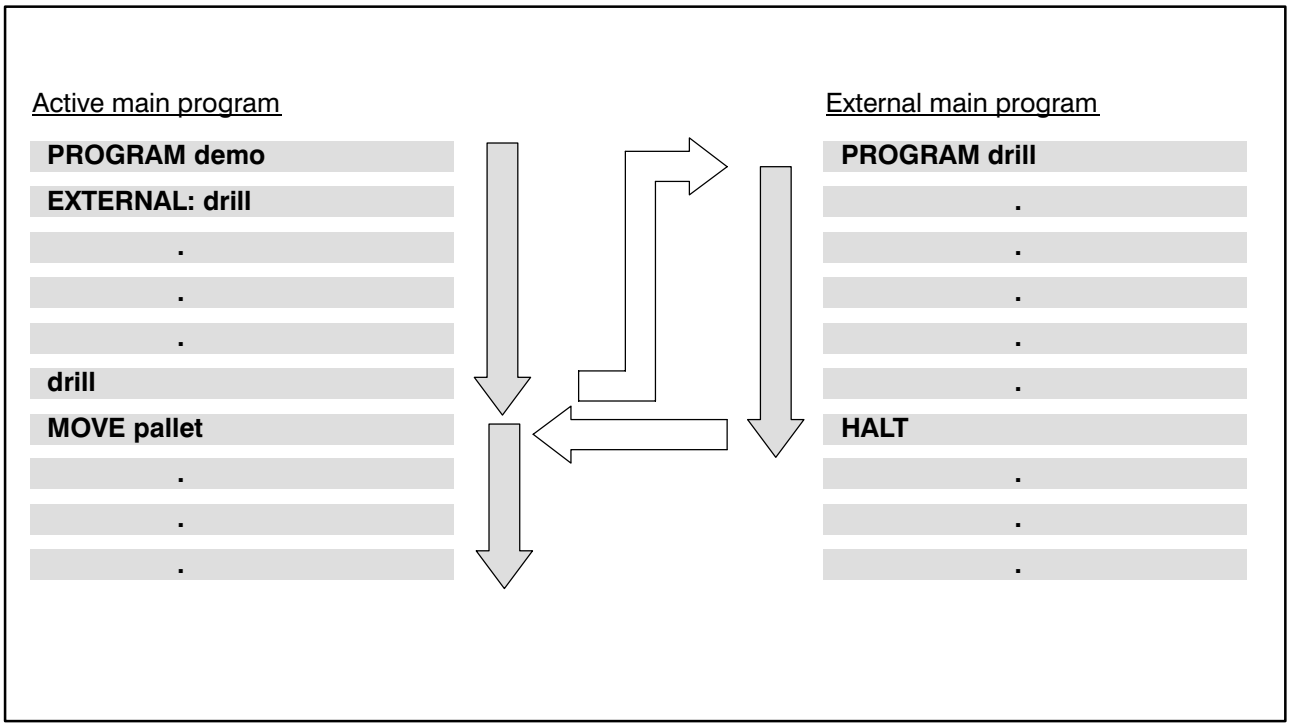
Program run:
The control executes the active main program up to the external program call, here “drill”.

This is 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.

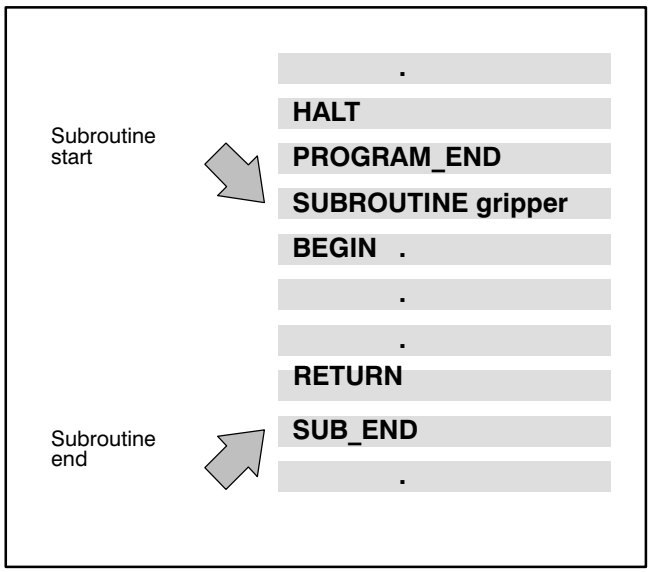


2. 4. Subroutine declaration

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

2. 4. 1. 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 12 characters. Letters and digits are permitted. The first character must be a letter. Upper−case and lower−case letters are deemed to be equivalent.
The subroutine is ended by the BAPS keyword **SUB_END**. Return to the calling program takes place with the BAPS command **RETURN.**

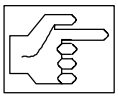


Programming:
A subroutine contains statements for the gripper and is to be given the name “gripper”:

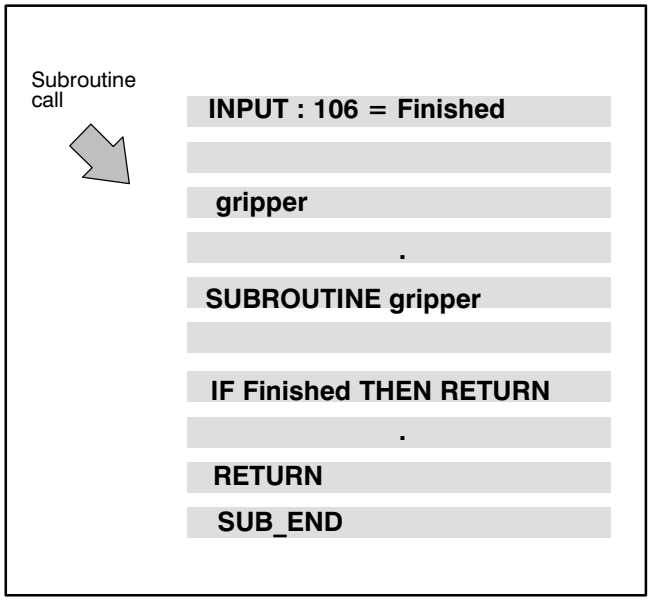
```
SUBROUTINE gripper
```

End of subroutine:

```
RETURN
SUB_END
```



RETURN can be used several times within a subroutine, e.g. with program jumps and conditional statements (see “Program jump” and “Conditional statement”).



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

2. 4. 2. Subroutine call

It is sufficient to specify the declared subroutine name for the subroutine call, e.g.

```
gripper
```

2. 5. Program run

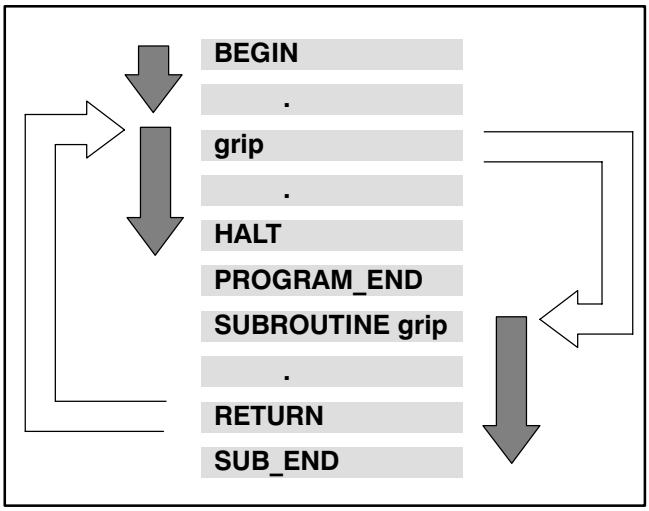
The control executes the main program up to a subroutine call, here

“gripper”

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

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.

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.

Example:

```
Gripper (1.5,6.0)
SUBROUTINE grip (VALUE REAL: force1, force2
```

The variable force1 is assigned the value 1.5 and the variable force2 the value 6.0 in the subroutine call.

If the declaration is proceeded by VALUE, the calling program transfers information (input) 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.

```
 1  PROGRAM DEMO
 2
 3  ;*************************************************************************
 4
 5  ;TEST PROGRAM FOR SUBROUTINE WITH PARAMETER TRANSFER
 6
 7  ;THE SUBROUTINE "ANIX" PROCESSES THE VARIABLES IN THE SUBROUTINE,
 8  ;THE VARIABLE IS UNCHANGED IN THE CALLING
 9  ;PROGRAM AFTER LEAVING THE SUBROUTINE.
10
11  ; THE SUBROUTINE "AWAS" OFFSETS THE VARIABLES IN THE
12  ;SUBROUTINE AND THEN RETURNS THEM TO THE MAIN PROGRAM
13  AFTER COMPUTATION.
14  ;THE SUBROUTINE OUTPUTS THE FOLLOWING NUMBER SERIES:
15  ;                5
16  ;               10
17  ;                5
18  ;               10
19  ;               10
20
21  ;*************************************************************************
22
23  REAL:X
24  BEGIN
25  X=5
26
27  WRITE X
28  ANIX (X)
29  WRITE X
30  AWAS (X)
31  WRITE X
31
33  HOLD
34  PROGRAM_END
35
36  SUBROUTINE AWAS (REAL:AW)
37  BEGIN
38              AW=AW*2
39          WRITE AW
40          RETURN
41  SUB_END

42  SUBROUTINE ANIX (VALUE REAL:AW)
43          BEGIN
44                      AW=AW*2
45                  WRITE AW
46  RETURN
46
47  END
```

2. 5. 1. 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

Program run: A call of the program in the “stacker” is programmed in the main program.

A further subroutine call is programmed in the subroutine “stacker”:

“gripper”

Program run: The control executes the main program up to the call “stacker”.

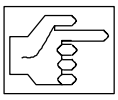
A jump then takes place to the subroutine “stacker”.

The control executes the subroutine “stacker” up to the call “gripper”.

This is followed by a jump to the next subroutine “gripper”.

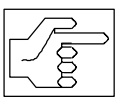
The subroutine “gripper” is executed completely in the example shown here.

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.

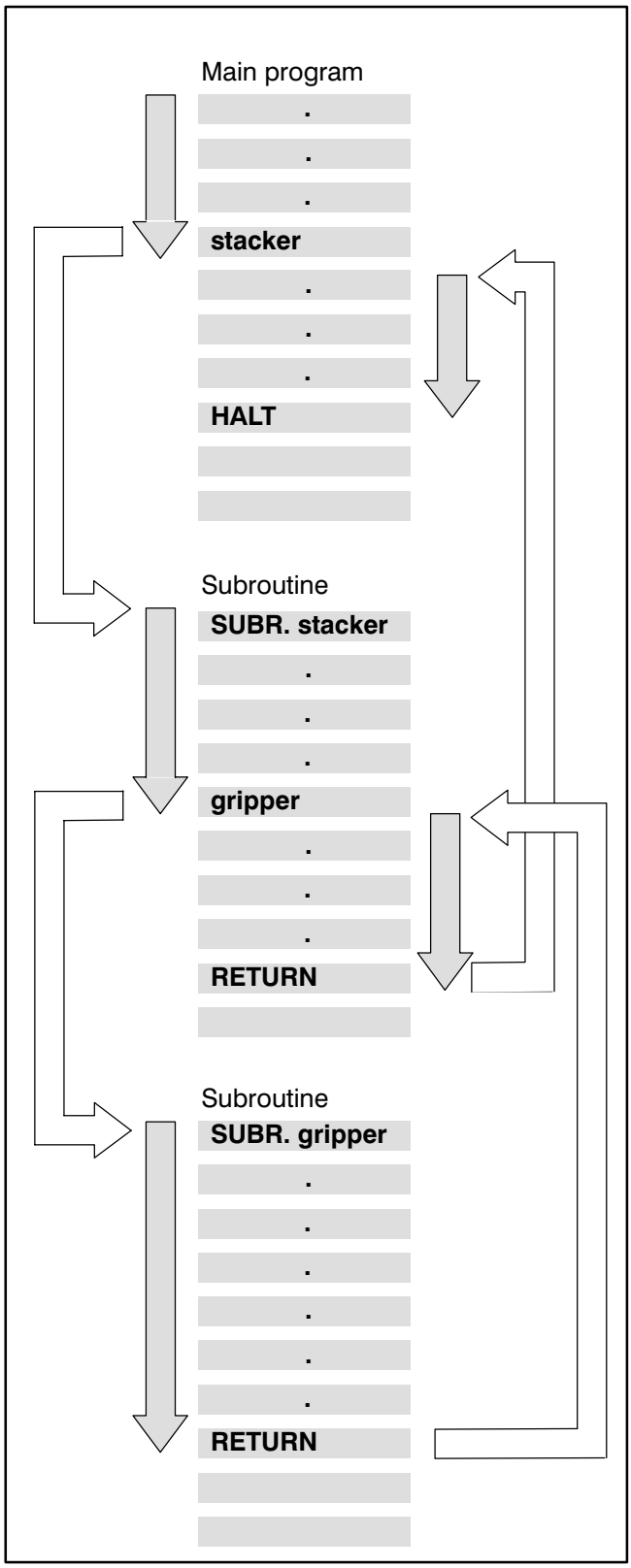


When programs and subroutines are nested, it must be ensured that no endless loops are created!

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



The memory size can be defined by a machine parameter (see rho3 – Description of machine parameters P16)



Program example for nesting program part repetitions

Program part repetitions can also be nested;

A second repetition is programmed within a program part repetition.

Program run: The control executes the program part once up to the start of the second program part.

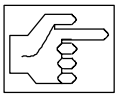
The second program part is repeated three times.

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.

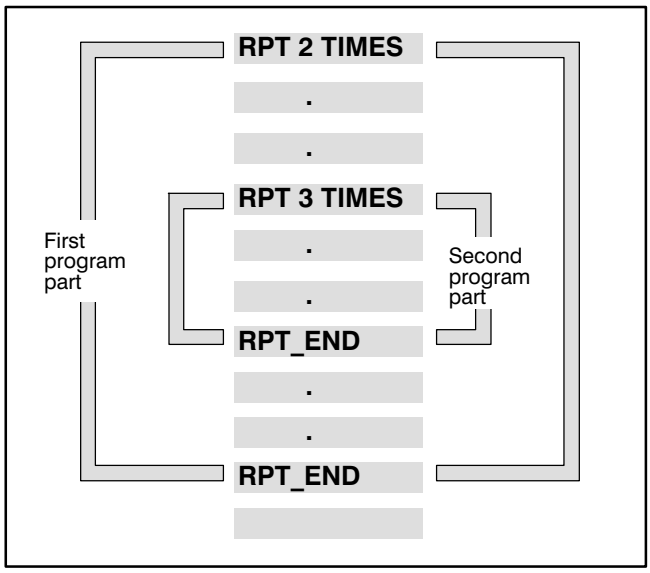
The control then jumps back to the start of the first program part for the second run.

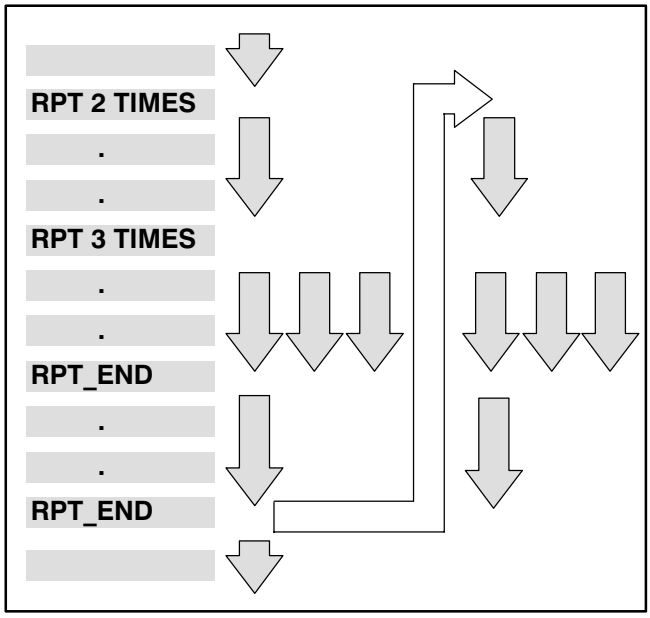
The whole sequence is repeated a second time.

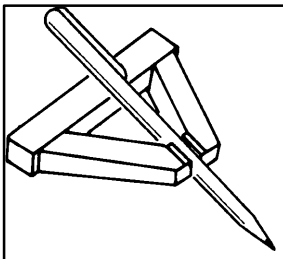
Any nesting depth is possible. This is restricted only by the size of the memory.



The memory size can be defined by a machine parameter (see rho3 – Description of machine parameters P16)







3. 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 rho 3, a distinction is made between direct movement statements and movement statements which influence movement.

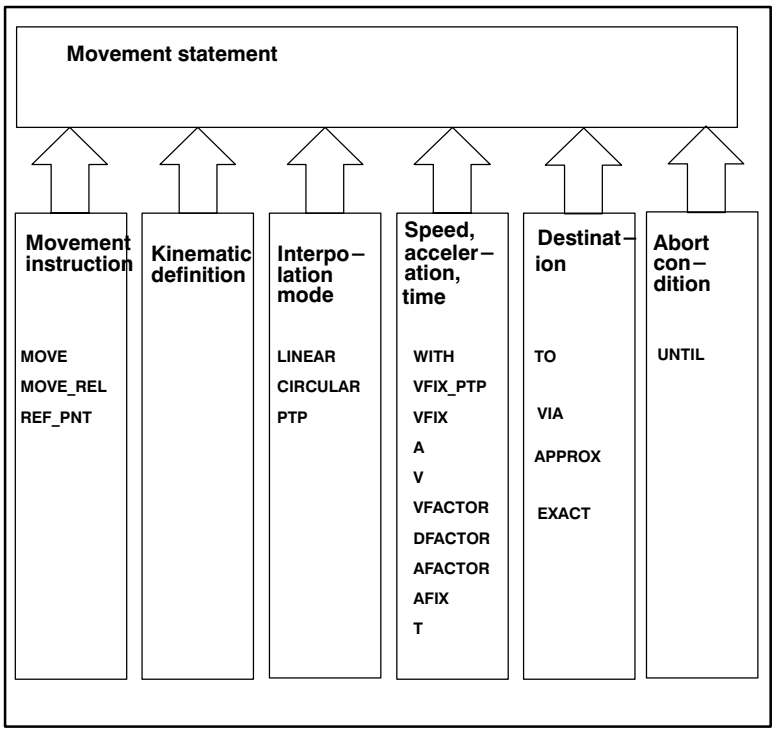
3. 1. Direct movement statements

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.



3. 1. 1. Movement instructions

The control knows the following movement instructions:

- **MOVE**
- **MOVE_REL**
- **REF_PNT**

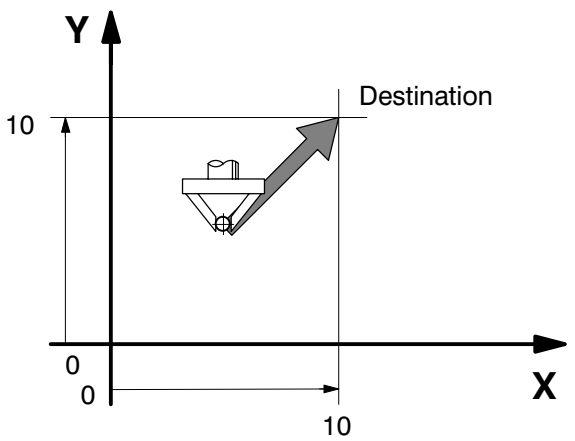
3. 1. 1. 1. MOVE

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.

Destination = (10,10)

MOVE destination

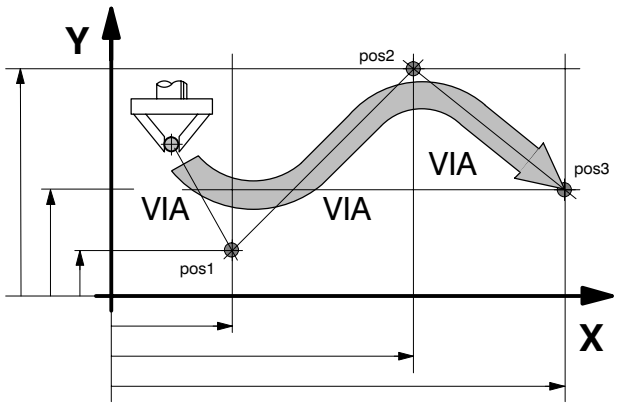


Within the movement instruction, additional information decides whether the robot approaches the programmed points exactly – i.e. within the defined tolerance – or whether it only travels past the points – without halt –. This information consists of **VIA** and **TO** for movements in absolute dimensions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

MOVE VIA...(pass over)

The robot travels past the positions without an intermediate halt.

MOVE VIA pos1, pos2, pos3

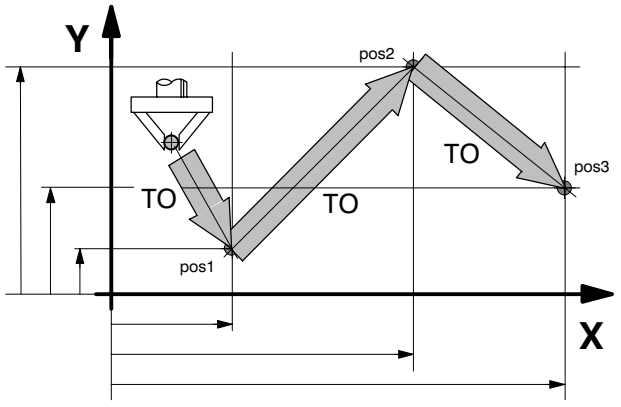


MOVE TO...

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.

MOVE TO pos1, pos2, pos3

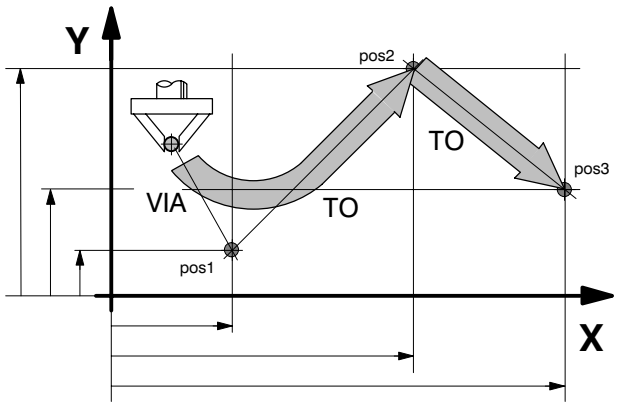


It is possible to link **VIA** and **TO** within a movement instruction:

MOVE VIA...**TO**...

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

MOVE VIA pos1 TO pos2, pos3



3. 1. 1. 2. MOVE_REL

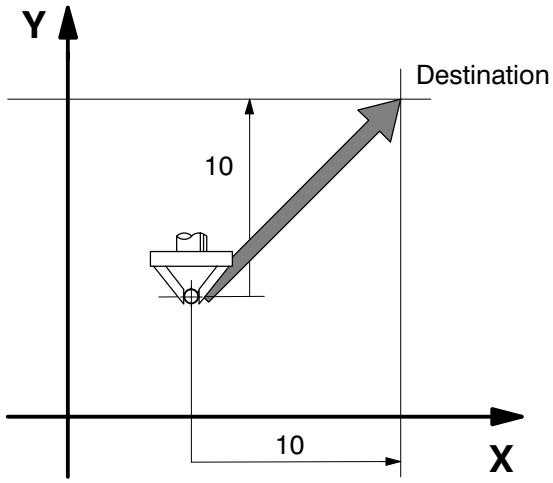
Syntax :

"**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.

Destination = (10,10)

MOVE_REL destination

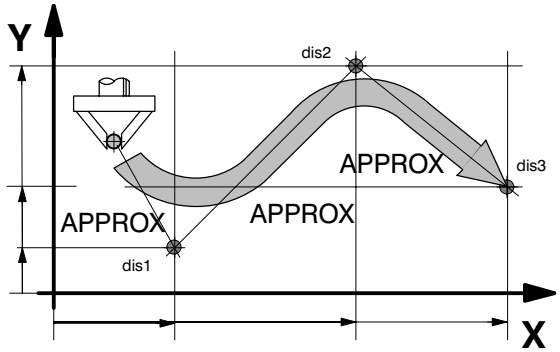


For movements in incremental dimensions with**MOVE_REL**, the words **APPROX** and **EXACT** decide with respect to exact positioning or travel past:

MOVE_REL APPROX...

The robot travels past the positions defined in incremental dimensions without an intermediate halt.

MOVE_REL APPROX dis1, dis2, dis3

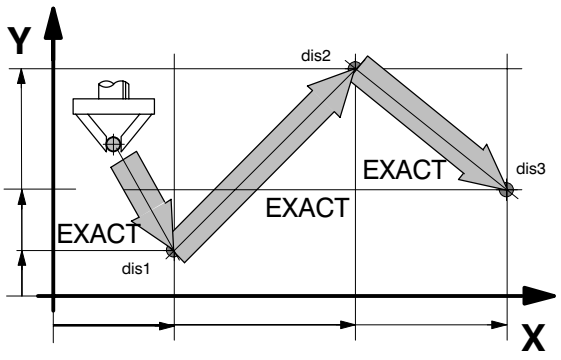


M**OVE_REL EXACT...**

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 then generates the instruction automatically.

MOVE_REL EXACT dis1, dis2, dis3

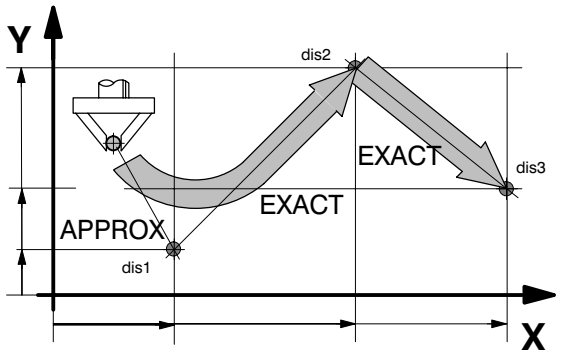


It is possible to link **APPROX** and **EXACT** within the movement instruction:

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.

MOVE_REL APPROX dis1 EXACT dis2, dis3



3. 1. 1. 3. 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 manually 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 brackets 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.

```
REF_PNT  KIN_1(1,2,3)

;;KINEMATIC = KIN_2

REF_PNT  (4,5)

REF_PNT  KIN_3(4,5)
```

3. 1. 1. 4. Destination point designations

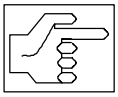
The designations of the points (position and orientation) can be freely selected (see "Point variables"). It is thus possible to assign the names pallet1, pallet2 ... to the pallet points, for example.

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.

3. 1. 2. Kinematic definition

If the program controls more than one kinematic, it is necessary to specify which kinematic is to be moved in the movement statement.



Only one kinematic definition must be made in a movement statement.

```
MOVE sr60 TO corner
MOVE robot_2 VIA prepos TO home
```

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.

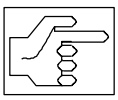
Kinematic preselection is performed using the **compiler statement:**

```
;; KINEMATIC = Kinematic name
```

The kinematic defined in this way is then valid for all subsequent movement statements without kinematic definition.

It continues to be valid (in ascending line order) until it is overwritten by another kinematic preselection.

```
;; KINEMATIC = Robot_2
MOVE VIA prepos TO home
```



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 Chapter: Defining kinematics).

3. 1. 3. Interpolation mode

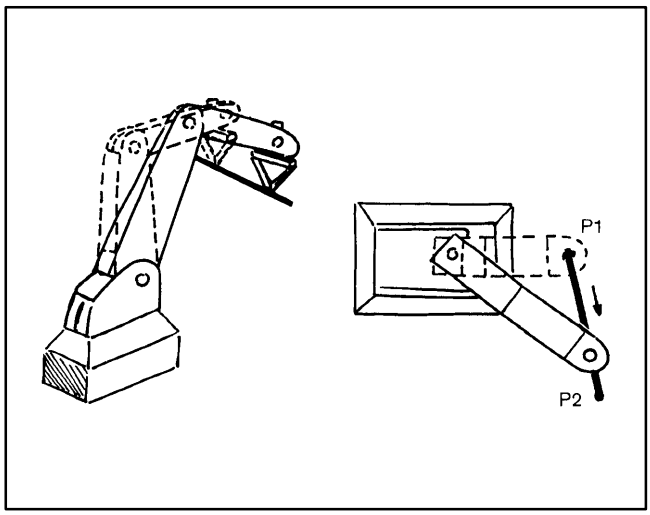
The control must know on which path the robot must approach the next position. In order to define this path, there are three

interpolation modes:

- **LINEAR** (= Straight line in space)
- **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)

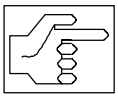
3. 1. 3. 1. LINEAR interpolation mode

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.

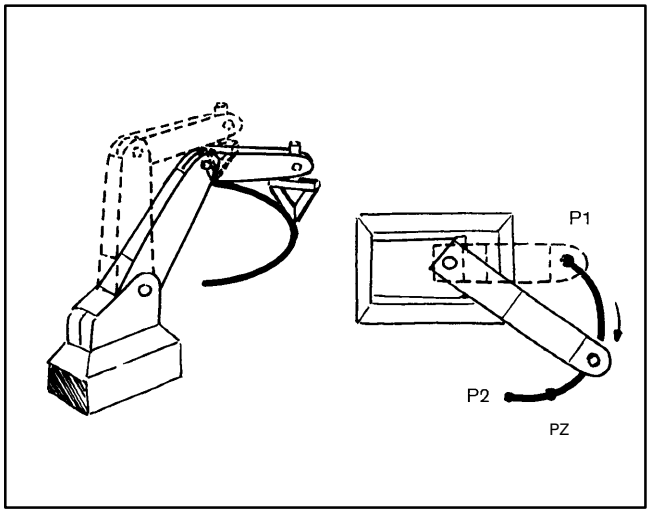


3. 1. 3. 2. CIRCULAR interpolation mode

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 is known to the control. The intermediate point P2 is a point on the arc which is travelled by the robot.



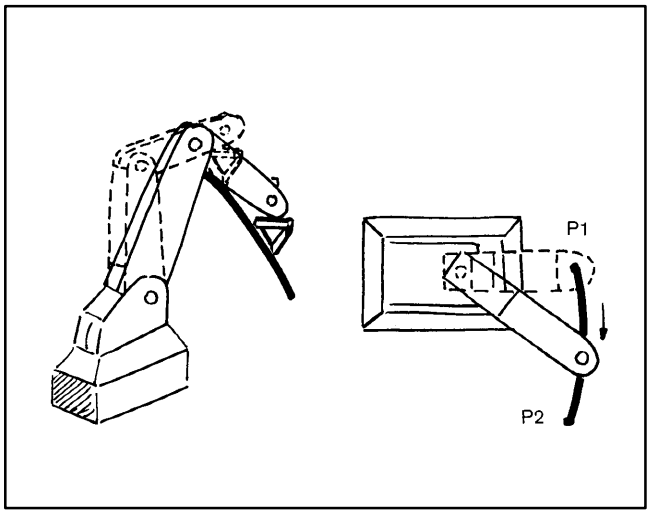
The orientation of the intermediate point (PZ) does not have any influence on the movement sequence.



3. 1. 3. 3. PTP interpolation mode

The control calculates the movements 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 defined dependent on the lever ratio of the robot arms and points P1 and P2.

It is sufficient to specify a destination point for the PTP interpolation method.



3. 1. 3. 4. 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.

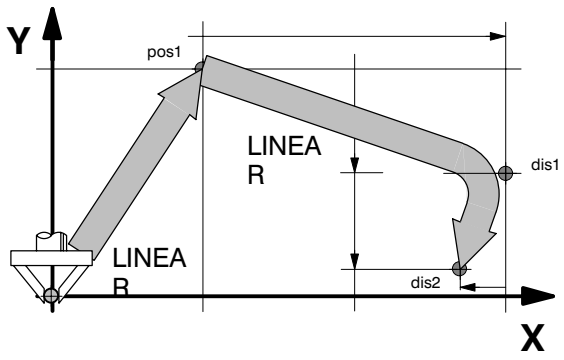
The interpolation mode is contained in the movement instruction, directly following MOVE or MOVE_REL.

There must be only one interpolation mode within a movement function.

Example **LINEAR**:

MOVE LINEAR TO pos1

MOVE_REL LINEAR APPROX dis1 EXACT dis2

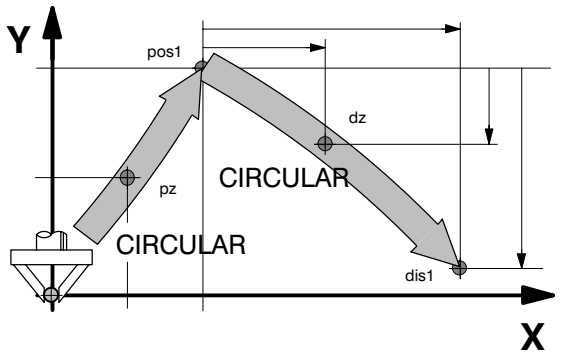


Example **CIRCULAR**:

The point information (intermediate point pz or dz, end position pos1 or dis1) required for CIRCULAR interpolation must be written in brackets and separated by a comma.

MOVE CIRCULAR TO (pz,pos1)

MOVE_REL CIRCULAR EXACT (dz,dis1)

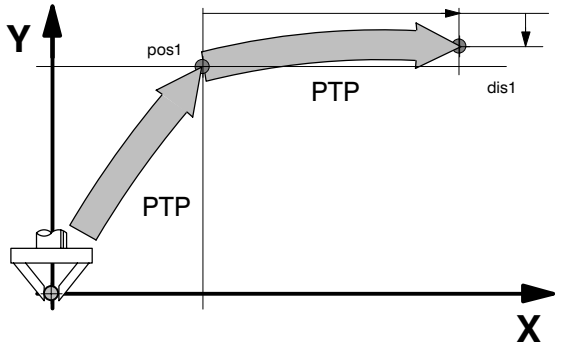


Example **PTP**:

The control automatically selects **PTP** if no interpolation mode is specified and if no global interpolation mode has been specified.

MOVE PTP TO pos1

MOVE_REL PTP EXACT dis1



3. 1. 3. 5. 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 compiler statement

;;INT = Interpolation mode.

The interpolation mode defined in this way then applies to all subsequent movement statements which do not contain any specific interpolation definitions.

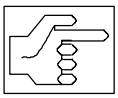
The robot travels to the position pos1 and the point defined via dis1 on a straight line in each case. Travel to point pos2, on the other hand, takes place on a circular path.

```
;; INT = LINEAR
MOVE pos1
MOVE_REL dis1
MOVE CIRCULAR (pz,pos2)
```

The global interpolation mode remains valid until it is replaced by another interpolation mode.

The robot travels to the positions pos1 and pos2 in a straight line in each case.

The positions pos3 and pos4 and all other positions are approached on a circular path after definition of CIRCULAR interpolation mode.

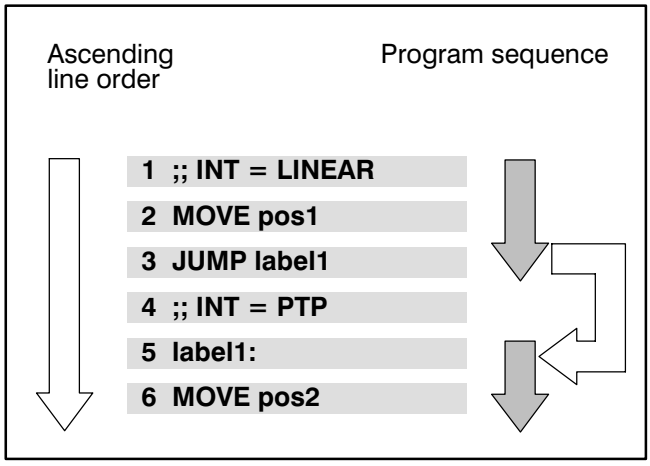


In the case of global definition of **CIRCULAR**−interpolation, the point values within the movement statement for which the interpolation definition is to be valid must be **point pairs**.

Example

```
;; INT = LINEAR
MOVE pos1
MOVE pos2
;; INT = CIRCULAR
MOVE (pz1,pos3)
MOVE (pz2,pos4)
      .
      .
```

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.).



Example:

The position pos2 is approached with **PTP**, although the compiler statement

;;INT = PTP

was skipped in the program run!

3. 1. 4. Destinations

The robot always travels from its current position to the programmed destination point. The path for this must be defined firstly by the **interpolation mode**, so that the control knows how it is to calculate the path to the destination point.

Secondly, 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.

3. 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 it 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.

3. 1. 5. 1. Speed

The speed has different designations, units and input ranges, depending on the path to be travelled by the robot.

If you do not program any values for the speed, the robot travels with speed values which are internally stored in the control (25 mm/s for path interpolation or 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 interpolation:

Designation:	V(Override active) VFIX (Override not active)
Unit of measure:	mm/s
Input range:	1...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..9.9999 (0.01%..999.99%) dependent 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.

Programming:

V=Decimal expression

All subsequent movement functions with the interpolation modes LINEAR and CIRCULAR have the following speed value:

V = 750 mm/s.

Example:

```
V = 750
MOVE LINEAR pos1
MOVE CIRCULAR (pz,pe)
        .
```

The speed value remains valid until it is changed by a further global speed input:

The speed for linear approach to the points pos1 and pos2 is **V** = 750 mm/s. The points place3 and place4 and all other points are approached at a speed **V** = 300 mm/s.

Example:

```
V = 750
MOVE LINEAR pos1
MOVE LINEAR pos2
V = 300
MOVE LINEAR place3
MOVE LINEAR place4
        .
        .
        .
```

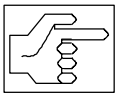
Statement−specific speed definition

If the path speed is to be valid for only one movement statement, the speed is correspondingly defined as a statement−specific speed, i.e. with the movement statement.

Programming:

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**.



The global speed inputs do not have any influence on statement−specific inputs!

Example: LINEAR

```
MOVE_REL LINEAR WITH V=500 EXACT dis
```

Example: PTP

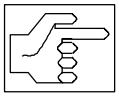
```
MOVE PTP WITH V_PTP=70% TO pos
```

3. 1. 5. 2. Speed override

Speed and time values can be changed with the speed override function **VFACTOR**.

The speed override factor is a factor by which the control automatically multiplies all speed inputs.

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.

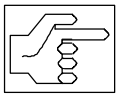
VFACTOR = 180%

is equivalent to

VFACTOR = 1.8

Speed override

Designation:	VFACTOR
Unit of measure:	%
Input range:	0.01...999.99% *
	* depending on machine parameter 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, while the local **VFACTOR** (P119) acts only on the respective kinematic.

Programming:

The designation and value assignment form a separate statement.

The **VFACTOR** of 180% (=factor 1.8) acts on the path speed to the positions 1, 2, 3 and 4. Pos.1 is approached at 180% of Vmax (power−on condition 10%, factor1.8);

pos2 is approached at 360 mm/s;

pos3 is approached at 180 mm/s;

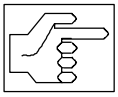
pos4 is approached in a time of 5.6 seconds ($\frac{10}{1.8} = 5.6$).

Example

```
VFACTOR = 180%
MOVE PTP TO pos1
V = 200
MOVE LINEAR TO pos2
MOVE LINEAR WITH V = 100 TO pos3
MOVE LINEAR WITH T = 10 TO pos4
```

3. 1. 5. 3. 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 adjacent table.

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**.

Acceleration for LINEAR and circular interpolation	
Designation:	A(override possible) AFIX (override not possible)
Unit of measure:	mm/s^2
Input range:	$0.001...32000 mm/s^2$
Power−on condition:	$10 mm/s^2$

Programming possibilities

In the case of **LINEAR** and **CIRCULAR** interpolation, acceleration input is possible as a

- global definition,
- statement−specific definition,
- acceleration override.

The acceleration override factor is also active for PTP interpolation.

Global acceleration definition

(Interpolation mode: LINEAR, CIRCULAR)

If the acceleration value remains the same for the whole program or a large section of it, it sensible to program a global acceleration value.

Programming: The designation and value assignment 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:

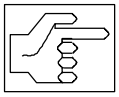
All subsequent movement functions with the interpolation modes LINEAR and CIRCULAR have the value A = 30 mm/s^2 defined for acceleration.

The acceleration until the path speed V = 100 is reached is as follows for travel to the points pos1 to pos3

A = 30 mm/s^2

The positions pos4, pos6 and all other positions with LINEAR and CIRCULAR interpolation have an acceleration value of

A = 15 mm/s^2



pos5 is approached with PTP interpolation.
The acceleration value A=15mm/s^2 is not valid here!

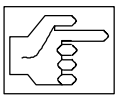
```
A = 30
V = 100
MOVE CIRCULAR (pz,pos1)
MOVE LINEAR pos2
```

Example

```
A = 30
V = 100
MOVE CIRCULAR (pz,pos1)
MOVE LINEAR pos2
MOVE LINEAR pos3
A = 15
MOVE LINEAR pos4
MOVE PTP pos5
MOVE LINEAR pos6
      .
      .
      .
```

Statement−specific acceleration definition

The acceleration is defined as a local value if a path acceleration value is to be active only within one movement instruction.

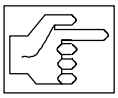


Local acceleration values are possible only for LINEAR and CIRCULAR interpolation!

Acceleration definition

Designation:	A
Unit of measure:	mm/s^2
Input range:	0.001..32000
Power−on condition:	10 mm/s^2

Programming: The designation and value assignment are part of the movement function for which the acceleration value is to be valid. Programming takes place following the interpolation mode with the key word **WITH**.



The global acceleration values do not have any influence on statement−specific values!

Example: LINEAR

```
MOVE LINEAR WITH A=30 TO pos
```

ExampleCIRCULAR

```
MOVE_REL CIRCULAR WITH AFIX=25 EXACT
                                (dz,dis)
```

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)
```

3. 1. 5. 4. Acceleration override

Acceleration values can be influenced once more with the acceleration override function (**AFACTOR**). The acceleration override factor is a factor by which the control automatically multiplies all acceleration inputs. The values calculated in this way then apply to all subsequent movement functions.

Example:

Programmed A = 300.00 mm/s^2 and AFACTOR = 90% results in an active acceleration factor of

A_{active} = A * AFACTOR => 270 mm/s^2

The acceleration override function is not active for programming with **AFIX**.

The adjacent table contains designations, units of measure and input ranges.

Acceleration override	
Designation:	AFACTOR
Unit of measure:	decimal factor
Input range:	0.01...999.99% *
	* depending on the machine parameter P22 and P118
Power−on condition:	100%

The decimal factor refers to the predefined acceleration values.

The factor can also be specified in %, e.g.

AFACTOR = 60%

is equivalent to

AFACTOR = 0.6

Programming: The designation and value assignment form a complete BAPS2 statement.

The **AFACTOR** of 200% (= factor 2) acts on the path acceleration for the movement to the points pos2 and pos3. The point pos4 is approached with the fixed acceleration value **AFIX** = 10 [mm/s^2]. The acceleration override factor does not have any influence here.

Example

```
AFACTOR = 200%
V = 200
MOVE PTP TO pos1
MOVE CIRCULAR TO (pz,pos2)
MOVE WITH A=80 TO pos3
MOVE WITH V=100 ,AFIX=10 TO pos4
```

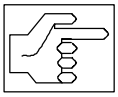
In the same way as the **AFACTOR** acts on the acceleration phase and can be programmed, the **DFACTOR** is used analogously for the deceleration phase of a movement.

Example

```
DFACTOR = 200%
V = 200
MOVE PTP TO pos1
MOVE CIRCULAR TO (pz,pos2)
MOVE WITH A=80 TO pos3
MOVE WITH V=100, DFIX=10 TO pos4
```

3. 1. 5. 5. Time input, indirect speed programming

If the robot must approach the next position within a certain time, it is possible to define a time instead of the speed.



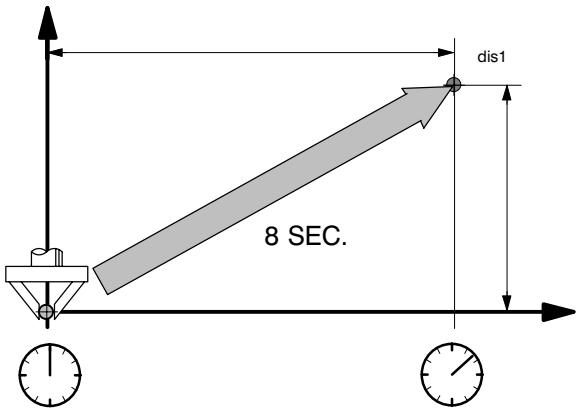
Time inputs are possible only on a statement−specific basis.

Time for LINEAR	
Designation:	T (override active) TFIX (override not active)
Unit of measure:	Second
Input range:	0.5...32000s

Programming: The time value is part of the movement function for which it is to be valid.

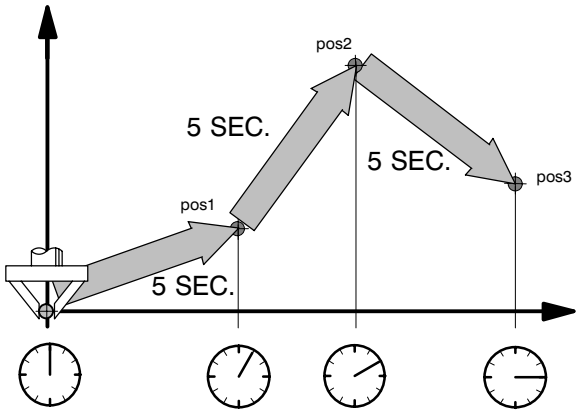
The robot travels in a straight line in incremental dimensions. It has been allocated a time of 8 seconds for the distance to be covered dis1.

MOVE_REL LINEAR WITH T=8 EXACT dis1



The robot successively travels to the positions pos1, pos2 and pos3 in a straight line with intermediate halt in each case. It is allocated a time of 5 seconds in each case to travel from position to position.

MOVE LINEAR WITH T=5 TO pos1,pos2,pos3



3. 2. 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:

1 . Synchronization statements

SYNC, **SYNCHRON, SYNCHRON_END**

2. Acceleration, deceleration between movement statements (block transitions)

BLOCK_SLOPE ,**PROGR_SLOPE**

3. 2. 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. The belt must be a ”straight line”. This line may be arbitrarily positioned in space (see rho3 Description of machine parameters P500). The belt movement is registered by a position measuring system.

The statements SYNC, SYNCHRON and SYNCHRON_END are available in BAPS2 for function programming.

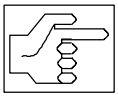
Syntax:

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 (also see Chapter Channel declaration) and must be assigned to a kinematic.



Up to eight belts can be declared for a kinematic. The belt names must be different. Equally, several kinematics can use the same belt if the same channel number is assigned to a belt variable several times.

The component names and axis names must be specified correspondingly for the assigned belt (see Compiler statements, kinematic definition, and rho3 Description of machine parameters P300).

Example

```
; Declaration of belt variables
; several belts for one kinematic
SR400.BELT : 501 = BELT1,502=BELT2
; same belt (501) for different kinematic
SR401.BELT : 501 = BELT_1
```

3. 2. 1. 1. Programming belt synchronization

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 (=> and =<).

The belt variable can be used in the program with **WAIT UNTIL** and **SYNC**.

Example

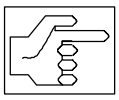
```
;Belt variable in WAIT UNTIL
WAIT UNTIL BELT_1 => 60
```

The instruction **SYNC** sets the belt variable to zero. Zeroing can take place dependent on a condition. The adjacent example shows the possibilities of how the **SYNC** instruction can be used.

Example

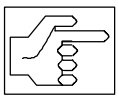
```
;Belt variable and SYNC statement
;Zeroing takes place if BELT_1 => 300
SYNC BELT_1 => 300

;Zeroing takes place dependent on a
;condition
SYNC BELT_1 ,LI_BAR = 1
```

Belt synchronization is switched on with the instruction **SYNCHRON**. From now on, all programmed movements are synchronized with the belt with respect to position and orientation.



The instruction **SYNCHRON** should therefore directly follow the statement SYNC. If this is not the case, a required value may result owing to a belt movement which the control then attempts to compensate for the with the SYNCHRON statement. This may in turn result in a jerky movement of the robot.

Synchronization is switched off with **SYNCHRON_END**.



Only LINEAR and CIRCULAR interpolation must be programmed in belt synchronization mode.

Example

```
;Belt variable and SYNCHRON statement

SYNCHRON SR400 BELT_1
     MOVE LINEAR TO pos_1


SYNCHRON_END
```

Program example:

```
;;      CONTROL = rho3
;;      KINEMATIC : (1 = ROBI_1, 2 = ROBI_2 )      ; Definition of 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 BELT_SYN

        INPUT          : 1=E1,2=E2
        ROBI_2.POINT   : START_POS
        ROBI_1.BELT    : 501=BELT1
        ROBI_2.BELT    : 502=BELT2

BEGIN

; Synchronization of kinematic ROBI_1 with BELT1

        SYNC BELT1, E1=1
        SYNCHRON ROBI_1 BELT1
        MOVE ROBI_1 LINEAR TO ROBI_1.POS
        WAIT UNTIL BELT1 >= 1000
        SYNCHRON_END ROBI_1 BELT1

; Synchronization of kinematic ROBI_2 with BELT2

        SYNC BELT2 >= 200
        SYNCHRON ROBI_2 BELT2
        MOVE ROBI_2 LINEAR TO START_POS
        WAIT UNTIL E2=1
        SYNCHRON_END ROBI_2 BELT2

PROGRAM_END
```

3. 2. 2. Block transitions (SLOPE mode)

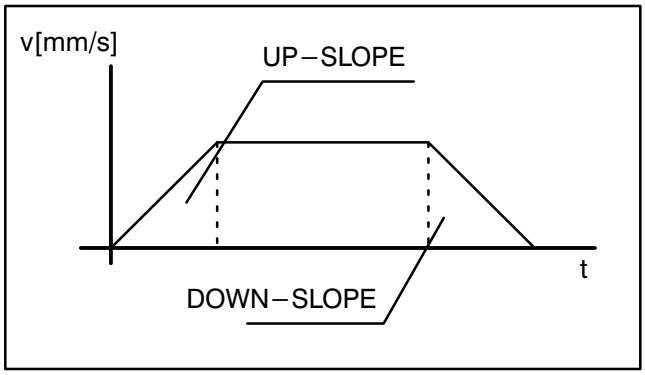
Syntax:

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 the Down− slope.

The slope mode can be activated on a kinematic−specific basis.



3. 2. 2. 1. General

If it is wished to execute several MOVE statements coherently without changing the speed to V = 0 [mm/s] and accelerating to the programmed speed (acceleration and deceleration phase), this can be done by using the BAPS2 statement 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.

3. 2. 2. 2. SLOPE mode activation

The slope mode can be switched in the BAPS program. The two BAPS2 standard functions

BLOCK_SLOPE

and

PROGR_SLOPE

are defined for this purpose.

Program Slope mode is switched off after activation of Block Slope mode by the BAPS2 statement

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 (jump to slope point, then start with defined acceleration) and is then decelerated again correspondingly at the end.

Also refer to ”rho3 Description of machine parameters” P120..P124 .

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 **SLOPE mode** is explained below in several BAPS2 program examples and its effect on the movement sequence shown in the following diagrams.

Example 1 :

Program slope mode is not active in the first part of the program (blocks 7..9), i.e. the robot is accelerated in each block with a=1000 mm/s2 and is decelerated again at the end of the block (Figure 1).

Program slope mode is switched on in block 11. As a result, the robot is accelerated with a=1000 mm/s2 in block 13. The block transitions from 13 to 14 and 14 to 15 are executed at constant speed.

The robot is decelerated with the programmed acceleration (1000 mm/s2) at the end of block 15 (Figure 2).

The deceleration operation is already initiated in the previous block (block 14, Figure 3) if the traversing distance in the MOVE TO block (block 15) 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 of block 15 if the sum of the distances from block 14 and block 15 is not sufficient as the deceleration distance.

In this case, the following message is issued during the run time ”Decel. distance is too short, block No.: 15”.

The acceleration phase may take place over any number of blocks (example), blocks 17..21, Figure 3).

Example 1:

```
 1 PROGRAM BSP_1
 2 BEGIN
 3 ;;INT = LINEAR
 4
 5 V=800, A=1000
 6
 7 MOVE VIA START_POINT
 8 MOVE VIA POINT_CENTER
 9 MOVE TO END_POINT
10
11 PROGR_SLOPE
12
13    MOVE VIA START_POINT
14    MOVE VIA POINT_CENTER
15    MOVE TO END_POINT
16
17    MOVE VIA RAMP_1
18    MOVE VIA RAMP_2
19    MOVE VIA START_POINT
20    MOVE VIA POINT_CENTER
21    MOVE TO END_POINT
22
23 BLOCK_SLOPE
24
25 HALT
26 PROGRAM_END
```

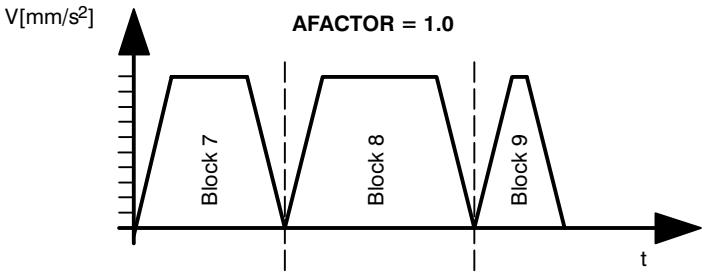


Figure 1.

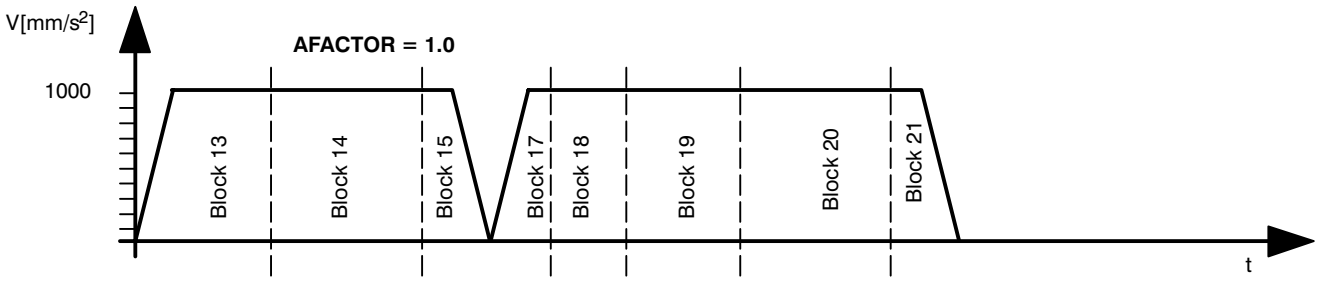


Figure 2.

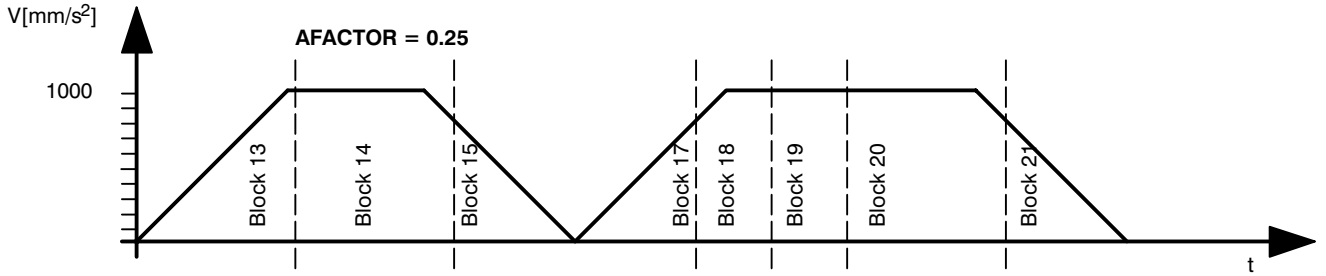


Figure 3.

3. 2. 2. 3. Changing acceleration and speed

Acceleration change (A, AFACTOR)

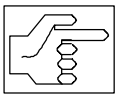
As before, a change in the acceleration value is effective only at the time of block preparation; this is also true for any change of the AFACTOR with the PHG.
Also refer to ”rho 3 PHG operation”

Speed changes (V, VFACTOR)

Speed changes programmed in BAPS2 such as

V = ... or **MOVE WITH V =** ...

act at the block transition.



Changes of VFACTOR with the PHG (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 by way of machine parameter P105 or P106. All speed changes are performed as jump functions below the slope point. If the programmed speed is not reached in 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 15).

Example 2:

```
 1 PROGRAM BSP_2
 2 BEGIN
 3  ;;INT = LINEAR
 4 PROGR_SLOPE
 5      A = 1000
 6
 7      MOVE WITH V = 200 VIA P1
 8      MOVE WITH V = 350 VIA P2
 9      MOVE WITH V = 500 VIA P3
10      V = 600
11      MOVE VIA P4,P5
12      MOVE VIA P6
13      V = 400
14      MOVE VIA P7
15      MOVE WITH V = 1000 TO P8
16
17 HALT
18 PROGRAM_END
```

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 PHG.

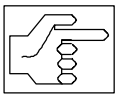
Example 3.3:

```
 1  PROGRAM BSP_33
 2
 3 ;; INT = LINEAR
 4      BEGIN
 5      PROGR_SLOPE
 6
 7      A=1000
 8
 9      MOVE WITH V=200 VIA P1
10      MOVE WITH V=350 TO P2
11      DFACTOR = 0.5
12      MOVE WITH V=350 VIA P3
13      MOVE WITH V=200 TO P4
14      HALT
15      PROGRAM_END
```

3. 2. 2. 4. Abort conditions

Abort by external influences

An abort of a travel movement by external influences (e.g.: Reset, Feed hold, 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:

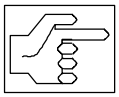
- **WAIT**
- **PAUSE**
- **HALT**
- **BLOCK_SLOPE**
- **IF ... THEN ... ELSE**
- **REF_PNT**
- **WRITE**
- **READ**
- **Several blocks without travel information**

  (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:

```
 1  PROGRAM BSP_3
 2  BEGIN
 3  ;;INT = LINEAR
 4
 5  V=800, A=1000
 6
 7  PROGR_SLOPE
 8
 9   MOVE VIA START_POINT
10  WAIT 1
11  MOVE VIA POINT_CENTER
12  MOVE TO END_POINT
13  WAIT 1.5
14  MOVE VIA START_POINT
15  MOVE VIA POINT_CENTER TO END_ POINT
16
17  BLOCK_SLOPE
18
19  HALT
20  PROGRAM_END
```



If Program slope mode 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.

The speed is set to 0 in a jump function in the event of an interruption after a MOVE VIA block.

In this case, the following error message is additionally output:

"**Speed jump, block No. XX, <dana.IRD>**"

(XX = Line number of the defective block, <dana.IRD> = Name of the current program).

3. 2. 2. 5. Interpolation mode change−over

The global acceleration and deceleration behavior can be activated **independently** of the interpolation mode (linear interpolation, circular interpolation and PTP mode). Block transitions without any change in the interpolation mode are performed as described in Item 2.

Change−over between linear and circular interpolation

Block transitions are peformed as described in Point 2 for changes from linear to circular interpolation.

Example 4:

```
 1 PROGRAM EXP_4
 2   BEGIN
 3 ;;INT = LINEAR
 4
 5 V=800, A=1000
 6
 7 PROGR_SLOPE
 8
 9   MOVE VIA START_POINT
10   MOVE CIRCULAR VIA (INT_POINT1,
     CIRCULAR_END1)
11   MOVE CIRCULAR TO  (INT_POINT2,
     CIRCULAR_END2)
12   MOVE TO PNT_CENTER
13   MOVE CIRCULAR TO (INT_POINT3,
     CIRCULAR_END3)
14
15   HALT
16   PROGRAM_END
```

Change−over between path and PTP modes

The movement sequence must be ended by a MOVE TO block before 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 and the warning ”Speed jump, block No.: XX, <dana.IRD>” is output. No controlled acceleration takes place in the first block in the new interpolation mode (i.e.: jump to program 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:

```
 1 PROGRAM EXP_5
 2 BEGIN
 3 PROGR_SLOPE
 4
 5 V_PTP = 1
 6 MOVE PTP TO START_PNT
 7 MOVE LINEAR TO P1
 8 MOVE PTP VIA P2
 9 MOVE LINEAR VIA END_PNT
10 V_PTP = 0.5
11 MOVE PTP TO END_PNT_1
12 HALT
13 PROGRAM_END
```

3. 2. 2. 6. Calling external subroutines

The transition to an external subroutine can take place within a coherent movement without a speed dip.

A precondition is that program slope mode is activated at the start of the external subroutine with the BAPS2 statement PROGR_SLOPE before the first travel block or that program slope mode is preset by machine parameter P120.

3. 2. 2. 7. Slope mode and exact−position signal output

The special functions 1 and 2 (see chapter ”Special functions) can be used fully for both program slope and block slope modes.

3. 2. 2. 8. Transgression of axis limit values

Transgressions of the 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, block No.: XX, <dana.IRD>

– Max. acceleration exceeded, axis X, block No: XX, <dana.IRD>

X = Number of corresponding machine axis

XX = Line number of defective block

<dana.IRD> = Name of active program

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 adaption is not possible, since this would contradict the demand for constant path speed.

3. 2. 2. 9. Test system

Since interrupt points can be set in the test system, only BLOCK_SLOPE is active here, irrespective of the programmed slope mode.

3. 2. 2. 10. Slope mode and machine parameters

The slope behavior is determined by the following machine parameters:

Slope acceleration PTP

Slope point, path mode

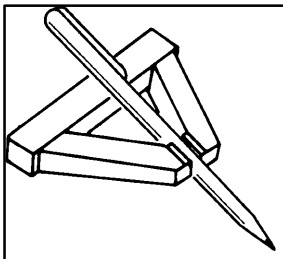
Power−on condition, SLOPE mode

SLOPE form

Also refer to ”rho 3 Description of machine parameters” parameter group P100.

3. 2. 2. 11. Portability of BAPS2 programs

BAPS2 programs are portable with respect to different rho3 hardware configurations if the machine parameters relevant for the functions PROGR_SLOPE and BLOCK_SLOPE are set identically for the different configurations. The SLOPE mode in power−on condition in particular is important for program portability.



4. Program flow statements

4. 1. Wait statement

Syntax:

WAIT Expression

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

4. 1. 1. Dwell time

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

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

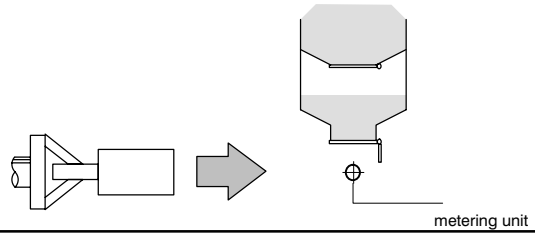
```
WAIT 8.5
```

Dwell time for WAIT

Unit of measure:	Second
Input range:	0.01...32000s

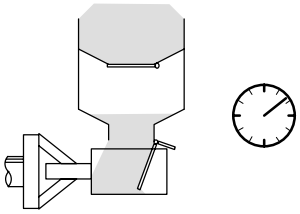
Program example: The robot transports a vessel to the metering unit to have it filled.

MOVE TO met_unit



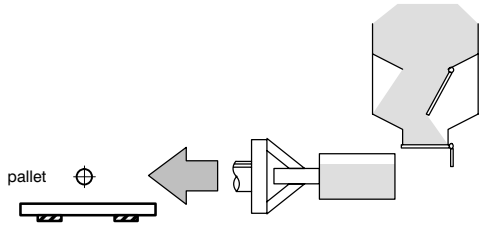
The filling time is approx. 8.5 seconds.

WAIT 8.5



It then transports the vessel to a pallet.

MOVE TO pallet



4. 2. Waiting for a condition to occur

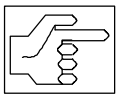
Syntax:

WAIT UNTIL Variable Rel_Operator Expression [**MAX_TIME** = Expression [**ERROR** Statement]]

Rel_Operator = "=" | "<>" | "<=" | ">=" | "<" | ">"

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

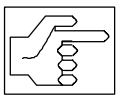
Program execution will then be interrupted until the condition is satisfied.



The conditions can be set only by means of **input channel** variables (also see "Channel assignment").

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

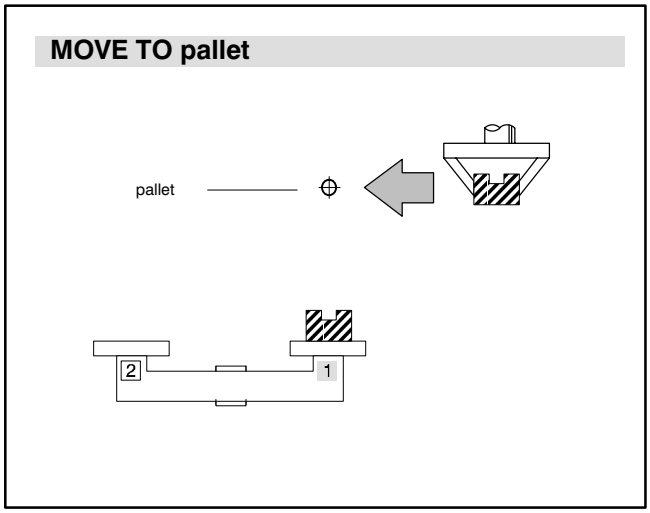
```
WAIT UNTIL pal_empty = 1
```



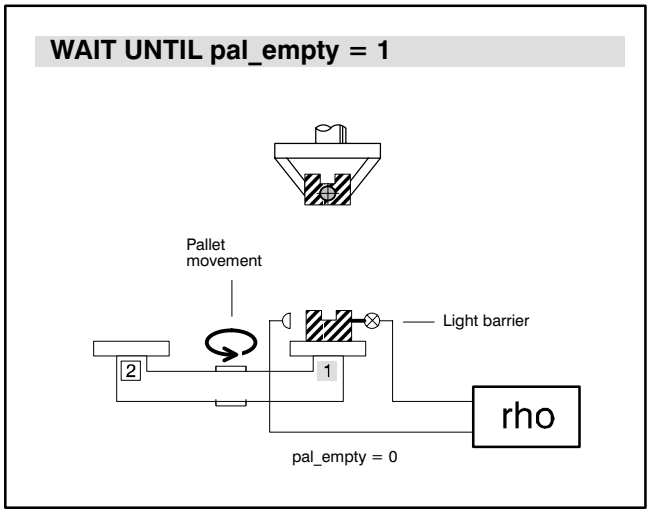
If the condition consists of several input channels, the WAIT statement must be divided up into several steps, e.g.

```
.
WAIT UNTIL signal = 1
WAIT UNTIL light = 1
MOVE TO pallet
.
```

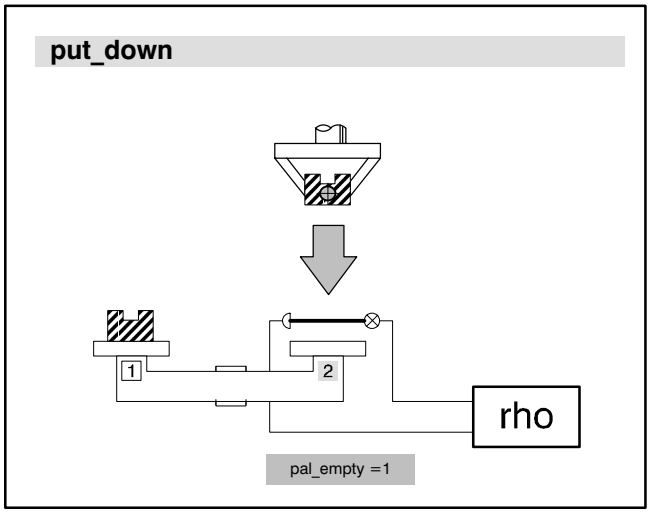
Program example: The robot transports a workpiece to a pallet changer.



It waits until the pallet is empty. The pallet changer then outputs the signal pal_empty = 1 when an empty pallet has arrived at the change position.



The deposit operation is programmed in a subroutine put_down.



4. 2. 1. Maximum wait time

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, e.g.

```
WAIT UNTIL sig=1 MAX_TIME= 60
```

Program example:

An error statement is programmed after MAX_TIME, e.g.

```
WAIT UNTIL sig=1 MAX_TIME =60 ERROR P AUSE
```

Maximum wait time

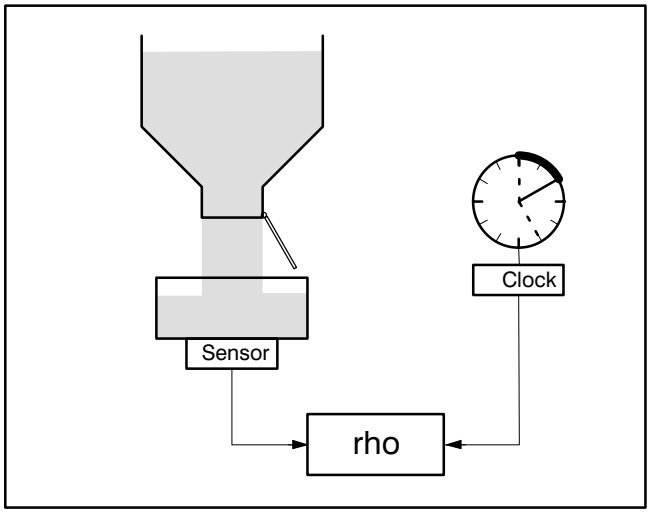
Designation:	MAX_TIME
Unit of measure:	Second
Input range:	0.5...32000

Sequence example

A container is filled with a liquid. A sensor measures the weight of the container. The sensor outputs the signal weight = 1 when the container is full. 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 seconds in order to ensure that the production sequence is not put at risk.

```
          .
          .
WAIT UNTIL weight=1
MAX_TIME=45 ERROR JUMP F_END
          .
; Error processing
F_END :
```



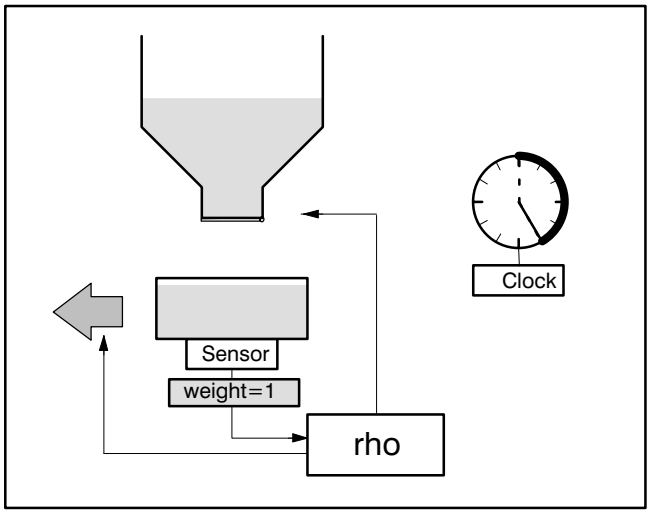
Case 1

The container is completely filled. The sensor issues the following signal to the rho:

Weight = 1

The robot transports the container away.

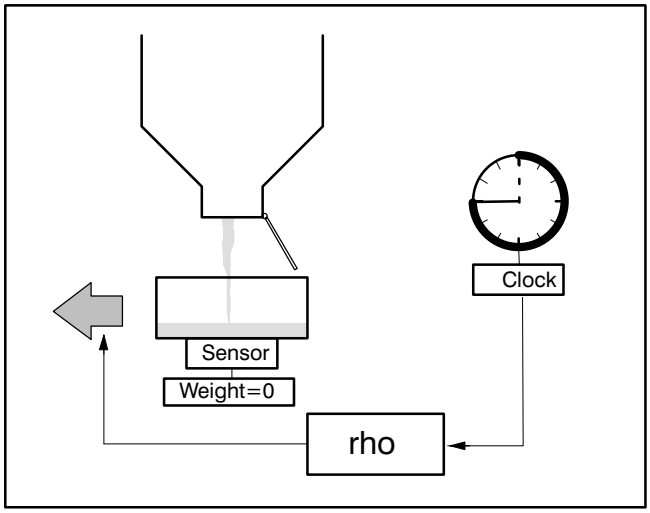
The MAX_TIME was not reached.



Case 2

The filling system is empty and the container is not completely filled.

The MAX_TIME is reached. The program is continued at F_END, e.g. the robot isolates the partially filled container.



4. 3. Pause statement

Program execution can be stopped with the pause statement

It is then necessary to issue the external start signal again in order to continue the program run (see ”rho 3 signal description”).

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

```
PAUSE
```

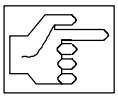
It is recommended to program a text output or to set an output before the PAUSE instruction in order to inform the operator about the program flow.

```
.
.
.
PAUSE
.
.
.
```

4. 4. HALT statement

The HALT statement ends execution of a 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 the case of called external subroutines, HALT results in a return to the calling active main program.

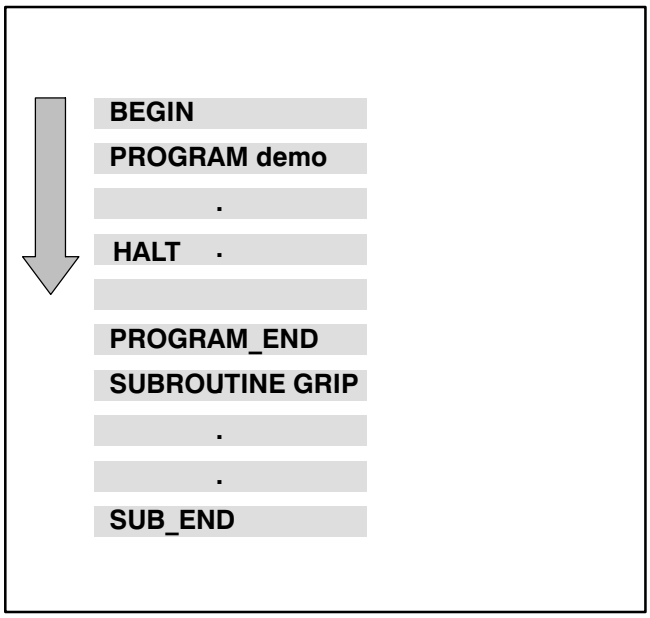
Programming: The HALT instruction is entered before the subroutine declaration or before the program end or is entered several times within a program in the case of branched programs:

```
HALT
```

If the program halt is evident from the program structure, e.g. in the case of the BAPS2 statement

PROGRAM_END,

the control then generates the HALT instruction automatically during compilation.



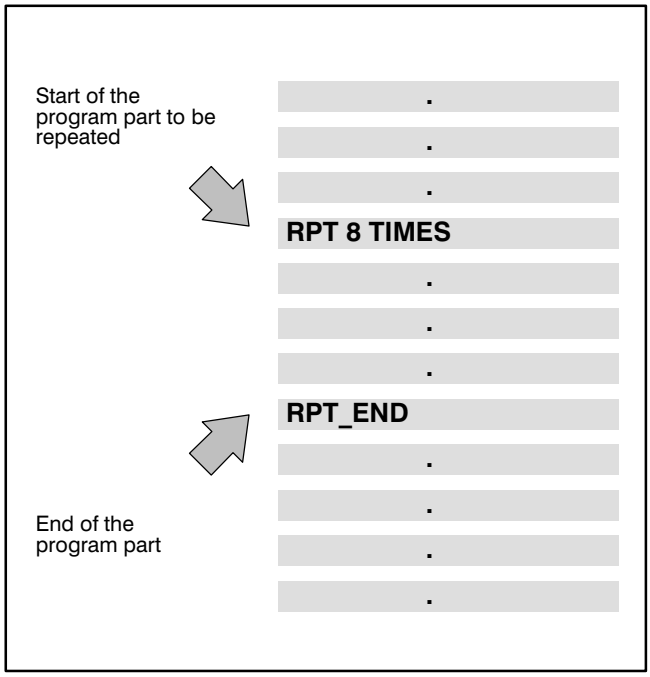
4. 5. Program part repetition

Syntax:

RPT" [Expression "**TIMES**"] Statement string "**RPT_END**" .

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

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



Identification

The program part is identified at the start

by the Repeat statement

RPT number TIMES.

Numbers, variables or expressions of the type INTEGER (see Data types) can be entered for the number of repetitions.

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

The program part is ended by the BAPS instruction RPT_END.

Programming: A certain program part is to be executed 8 times.

```
RPT 8 TIMES
```

End of the program part:

```
RPT_END
```

4. 6. 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.

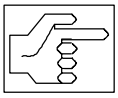
Identification

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 12 characters.

Letters and digits and the underline symbol are permitted. The first character must be a letter. Upper−case and lower−case letters are equivalent.

Setting of a jump label must be identified with a colon (:) in order to distinguish this from subroutine calls.



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

Programming:

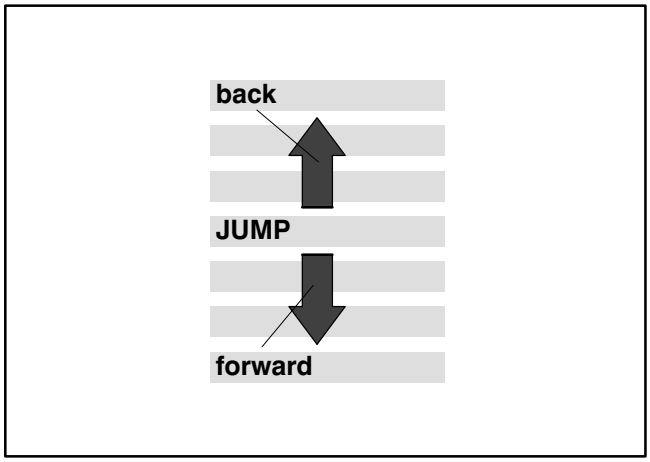
A jump to the jump label “Table” (forward) is to take place in a program.

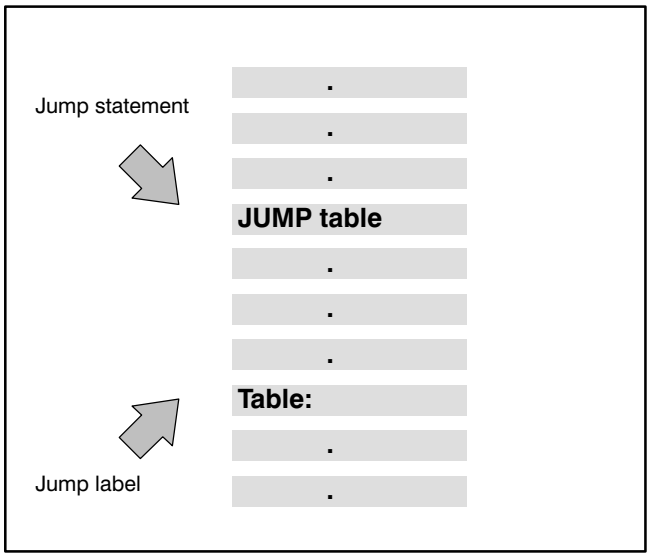
Jump statement for jump to jump label “Table”

```
JUMP TABLE
```

Setting of the jump label with the name “Table”

```
Table:
```



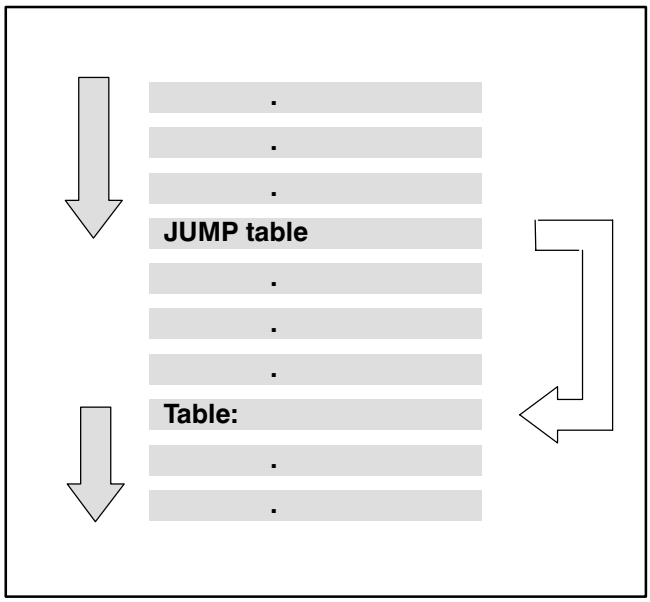


Program run

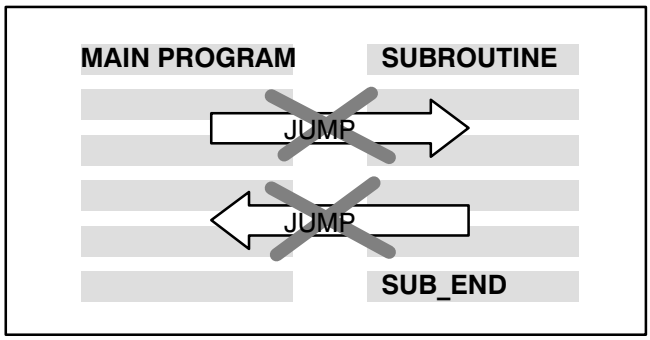
The control executes the program up to the jump statement, here“JUMP table”.

This is followed by a jump to the jump label “Table”.

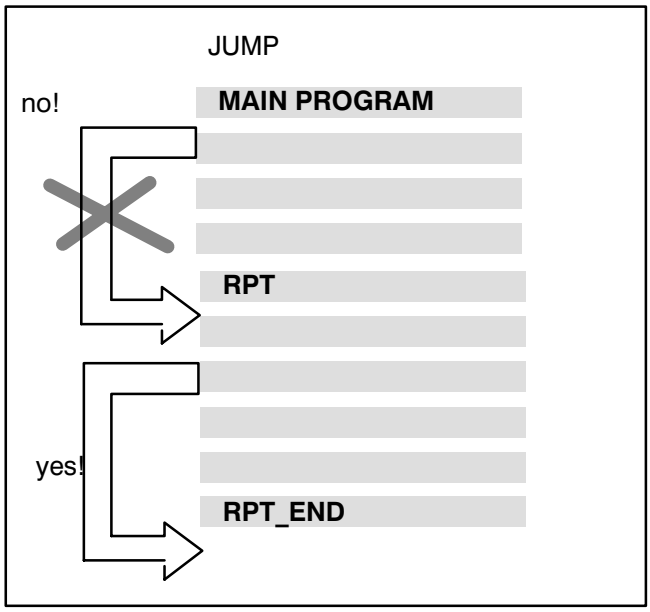
The control then continues the program from this label.



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



Jumps to program part repetitions are also not permitted.



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

4. 7. Conditional 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 a conditional statement.

Condition: 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 possesses 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

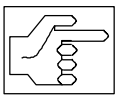
i = 15

sig1 AND sig2

word = j

force > 115.0

torque >= threshold value



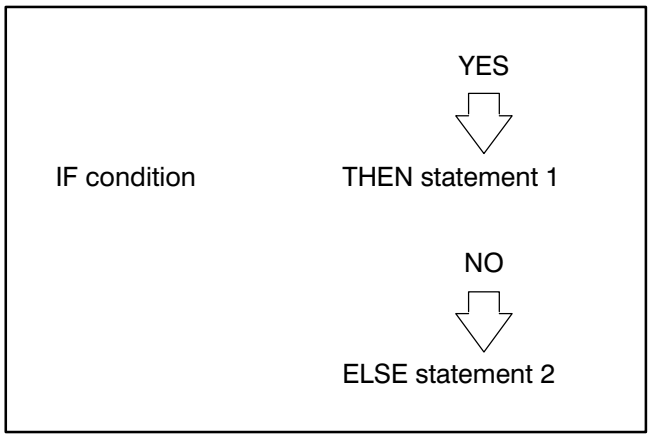
Conditions must be put in brackets if necessary when combined with AND, OR, NOT (see “Logic operations”) in order to obtain the desired priorities of the condition operation.

Programming

A conditional statement has the following form:

IF condition **THEN** statement1 **ELSE** statement2

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 (RPT)
- Movement instructions
  (MOVE, MOVE_REL)
- Conditional statements
  (IF...THEN...ELSE)
- .
- .
- .

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.

Program 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 of the weight of the pallet.

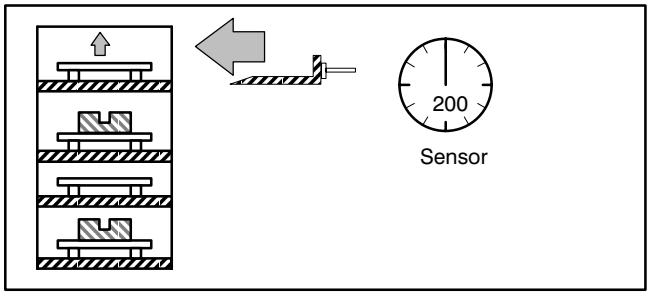
When it has found the workpiece, it is to transport it to the machine.

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

```
next:
In      ;subroutine travel in and lift up
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
```

Call of the subroutine “In”;
(travel in, lift up pallet).

```
Next:
In
```



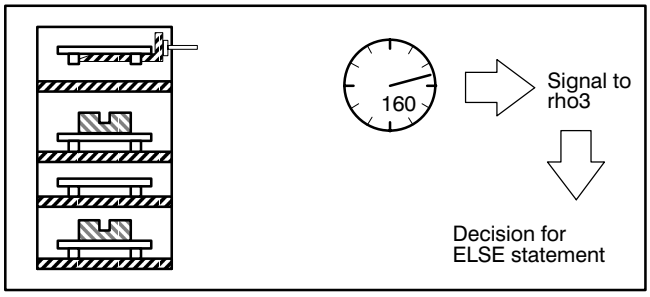
Conditional statement:

Evaluation of the signal < 180.0 kg;

(e.g only empty pallet weight)

Control decides on ELSE statement.

```
IF weight >=180.0
THEN MOVE TO machine
ELSE JUMP search
```



Program jump to the jump label

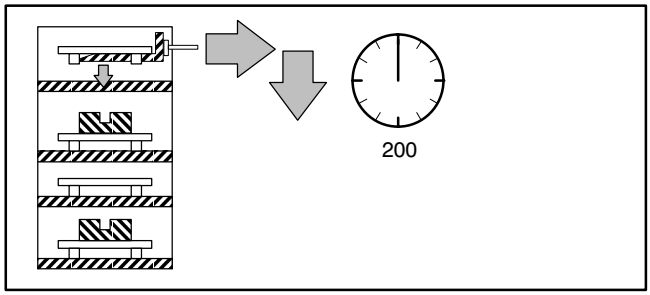
“Search”;

Call up the subroutine “Down”;
(deposit of pallet, travel out, travel down).

JUMP search

Search:

Down



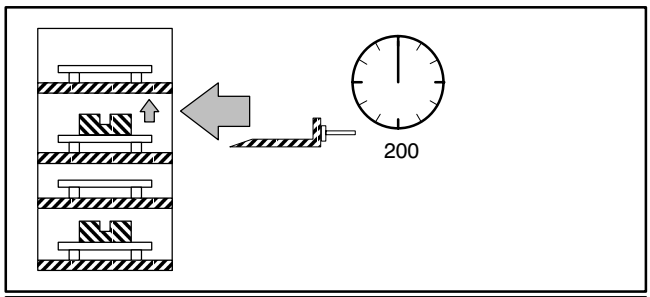
Program jump to the jump label “Next:”

Call of the subroutine “In”; (travel in, lift up pallet).

JUMP next

Next:

In



Conditional statement:

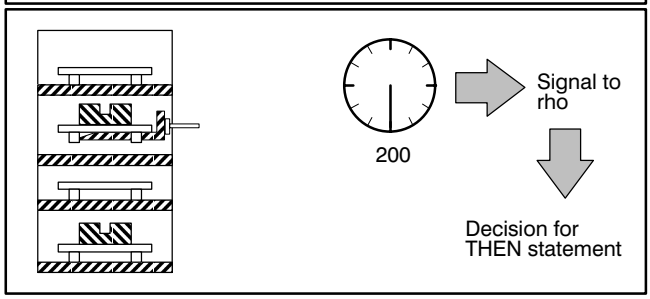
Evaluation of signal 200 kg;

Control decides on THEN statement.

IF WEIGHT > 180.0

THEN MOVE TO machine

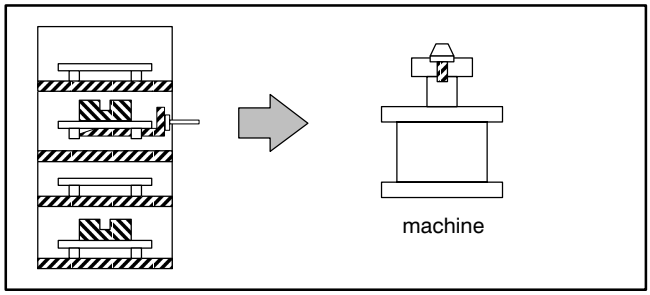
ELSE JUMP search



Movement instruction:

Travel to position ”Machine”.

MOVE TO machine

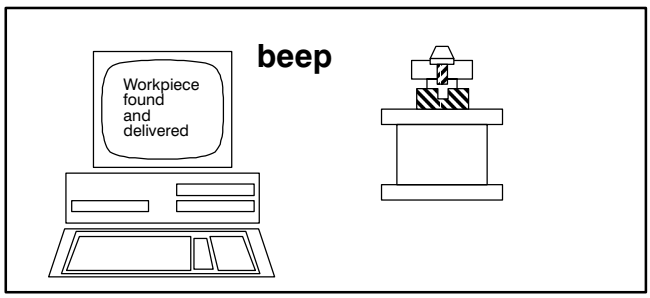


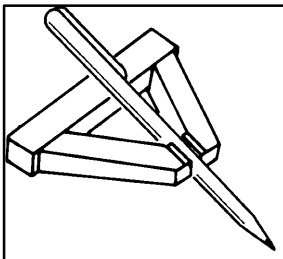
Program run interruption:

Output of text:
”Workpiece found and delivered”
on monitor.

WRITE ’Workpiece found and delivered’

PAUSE





5. Variable declaration

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 in time.

5. 1. Variable names

Every variable has a name. Different variables must have different names. The names should be chosen so that it is also 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”,

e.g. arithmetic variables:

instead of “I”, it is better to use “Counter”

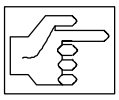
etc.

Name:

The name consists of a maximum of 12 characters.

It is permitted to use letters, numbers and underlines.

The first character must be a letter (also see following page):



Blanks are not permitted!

The first character of a name must be a letter.

It does not matter whether upper−case or lower−case notation is used.

Exception:

Point variables of the data type JC_POINT.

These variables must start with the character @ (commercial a).

5. 2. 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 structured data types.

Simple data types:

- **BINARY**
- **INTEGER**
- **REAL**
- **CHAR**

Structured data types:

- **ARRAY**
- **POINT**
- **JC_POINT**
- **BELT**
- **TEXT**
- **WC_FRAME**
- **FILE**
- **DEF**
- Channel

DEF only for point variables (**POINT,JC_POINT**)

- **SEMAPHORE**

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

5. 2. 1. Simple data types

5. 2. 1. 1. INTEGER

Only whole−number values (numbers without decimal point, positive or negative) must be assigned to variables of the type INTEGER.

Value range:

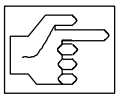
$-2^{31}...+(2^{31}-1)$

5. 2. 1. 2. REAL

Real numbers (numbers with decimal point, positive or negative) may be assigned to variables of the type REAL.

Value range:

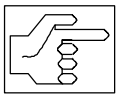
approx. $-10^{37}...-10^{-38}$, real zero, $10^{-38}...10^{37}$



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

5. 2. 1. 3. BINARY

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



No whole−number 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 name	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, for example.

5. 2. 1. 4. CHAR

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

5. 2. 2. Structured data types

5. 2. 2. 1. 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

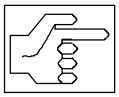
;; KINEMATICS =

e.g **;; KINEMATICS** = ROBOT_1

5. 2. 2. 2. JC_POINT

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

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 type 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

5. 2. 2. 3. 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.

Example:

```
; Declaration of text variables
TEXT : char_string
;Assignment of individual components
char_string [1] = 'A'
char_string [2] = 'B'
; Assignment of a text
char_string = 'THIS IS ALL ASCII TEXT'
```

5. 2. 2. 4. 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] Type**:**

Example :

ARRAY [−10..10] INTEGER : Variable name

5. 2. 2. 5. SEMAPHORES

SEMAPHORE : 'SEMA_NAME'

Variables of the type SEMAPHORE are used as parameters in the

EXCLUSIVE statement (see chapter "Parallel processes").

5. 2. 2. 6. FILE

FILE : 'Cad_dat'

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

5. 3. Declaration of variables

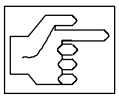
Syntax:

[**DEF**] Type **:** [Channel No. =] Name { **,**[Channel No. =] Name **}**

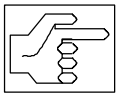
DEF is possible only for point variables(**POINT** and **JC_POINT**) and channel number only for the types **INPUT**,**OUTPUT** or **BELT**.

The control must know which values or characters a variable may possess before execution of a 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, e.g. the compiler interprets undeclared variables as point variables and reserves memory space for these in the point file PNT.



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

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
```

6. Value assignment

Syntax:

Variable = Expression

Value assignments are used to assign values to variables, also standard variables, for speed and acceleration during the course of a program. 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 “Point variables”.

Programming:

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 + may be omitted.

The sign − is positioned directly before the variable or constant. The negative expression must be put in brackets if two operators follow each other directly.

All components must be multiplied by −1 if it is wished 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.

Examples:

Counter = 1

Variable
Numeric value

xvalue = yvalue + 2.5

Variable
Arithmetic expression

Distance = SIN(alpha)

Variable
Standard function

Value = −value

Variable of the type REAL
Negation

d = d * (−2.0)

Variable
Negative expression

Corner = (−1.0)*corner

Variable of the type Point
Negation

7. Arithmetic expressions

Arithmetic expressions are combinations of

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

The type of combination is defined by the operator.
The rho 3 system knows five arithmetic operators:

Addition
e.g. $k = 1 + 5$

Subtraction
e.g. *value = weight −1*

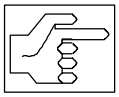
Multiplication
e.g. *length = width x 2*

Division
e.g. *height_new = height/2.0*

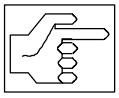
MOD Modulo calculation

only for data type INTEGER

e.g. *rest = number MOD divisor*



The characters + and − can also be used as signs for variables and numeric values.



Variables 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, 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 “Point variables”).

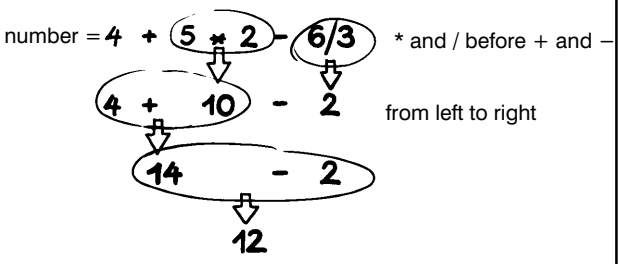
Arithmetic expressions / Modulo function MOD

The operators are executed in the following order:

* and / and MOD
before
+ and −

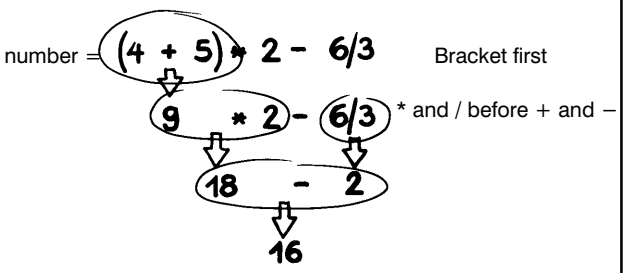
Calculation takes place from left to right within these classes.

Example:



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

Example:



Modulo function

Modulo function: value1 MOD value2

The modulo function, data type INTEGER, calculates the whole−number remainder from division of “value1” by “value2”.“Value1” and “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.

Programming:

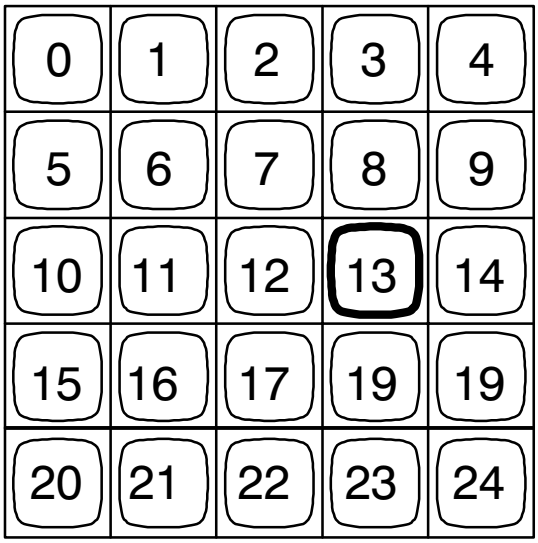
The position number must be substituted for “value1” and the column number for “value2”:

column = 13 MOD 5

Calculation: 13/5 = 2 Remainder 3, i.e. column 3 is the sought answer.

row = 13 MOD 5

Calculation: 13/5 = 2 Remainder 3, i.e. row 3 is the sought answer.

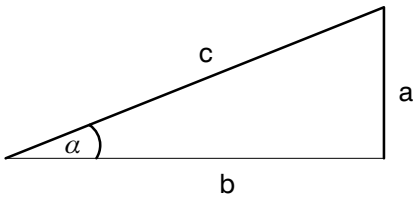


The pallet has 5 columns; (0, 1, 2, 3, 4)
which column and which row does position 13 occupy?

8. Standard functions

8. 1. Sine function: SIN (rad)

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



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

Programming:

The angle α must be specified in radian measure “rad” (type REAL)

Formula for “rad”:

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

The radian is specified after SIN in brackets, e.g.

```
avalue = c * sin(alpha)
```

The designation ”avalue” was chosen instead of only ”a” because ”a” has already been allocated as a reserved name for acceleration.

8. 2. Cosine function

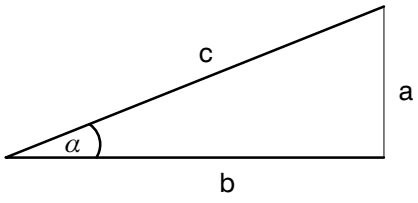
COS (rad**)**

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

Programming:

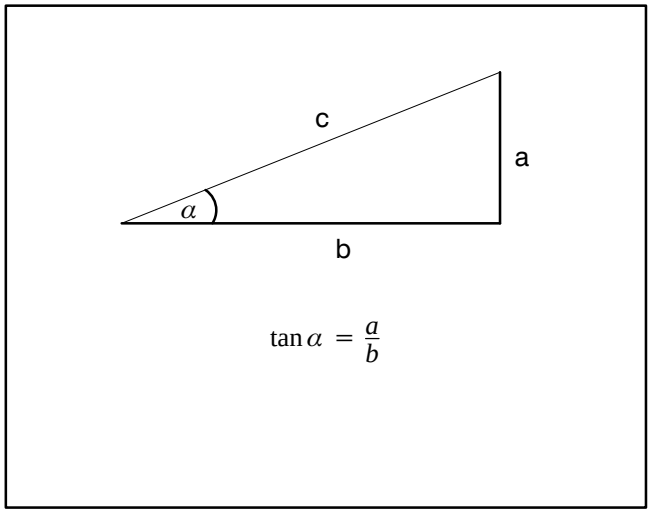
See Sine function, e.g.

```
b = c * cos(alpha)
```



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

8. 3. Arc tangent function



ATAN(expression**)**

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 tan(α) and is defined as follows:

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

The inverse function of the tangent function is then:

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

The angle is available as a radian value. Conversion:

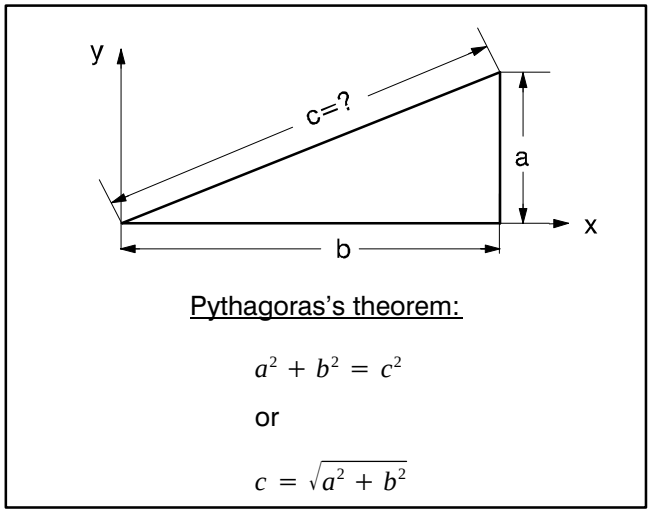
$$(\alpha) = rad * \frac{180^\circ}{\pi}$$

The result from ATAN (expression) must be substituted for rad.

Programming: (Example)

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

8. 4. Root function



SQRT(expression**)**

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

Example: Length of a side c in a right−angle triangle

Programming: The variable designation a is already occupied by the standard variable A for acceleration. For this reason:

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

8. 5. Coordinate transformation

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. POS=WC(@POS).

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).

(Also see “Mixed operations with point variables”)

8. 6. Absolute value

ABS(argument)

The result supplied by the function is the absolute value of the 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)

8. 7. TRUNC

TRUNC (argument**)**

The function transforms the argument of the type **REAL** into a value of the type **INTEGER** by truncation. In the case of a positive argument, the result obtained is the largest whole number less than or equal to the argument.

In the case of a negative argument, the result obtained is the smallest whole number greater or equal to the argument.

8. 8. ORD

ORD (CHARACTER_VARIABLE**)**

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

Example :

```
CHAR_NUM = ORD (ASC_CHAR)
CHAR_NUM = ORD( 'i' )
```

8. 9. CHR

CHR (INTEGER**)**

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

Example:

```
ASC_CHAR = CHR (34)
```

8. 10. ROUND

ROUND (argument**)**

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

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

Example:

Value_3 = **ROUND (**7.81**)**

Value_4 = **ROUND (**−5.43)

8 is assigned to value_3 and −5 to value_4.

8. 11. End of file

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 yields 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:

Use of the function in a conditional statement

IF END_OF_FILE(DAT_VALUES**) THEN** ....

ELSE .....

Use of the function in an assignment

EOF = **END_OF_FILE (**DAT_VALUES**)**

9. Point variables

Point variables are combined (structured) data types and consist of components.

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

In addition to the **complete value assignment**, it is also possible to assign values to the point variables **component−by−component**, e.g.:

CORNER.Z_K = HEIGHT

The component designation is specified with the compiler statements:

;;JC_NAMES = Axis name, ...

and

;;Kinematic name.JC_NAMES= Axis name, ...

as well

;; WC_NAMES = Coordinate name, ...

and

;;Kinematic name.WC_NAMES = Coordinate, ...

Example:
Declaration of point variables

```
DEF POINT : Corner
    ARRAY [ 1..4 ] POINT : Point_array
    JC_POINT :  @Interm_pnt
DEF ARRAY [1..8] JC_POINT: @JC_Pnt_array
SR800.POINT : Start_point
DEF ROBOT1.JC_POINT : @Depot
```

9. 1. Identification of point variables

The names of point variables of the type **POINT** start with a letter.

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

Permitted operations with point variables

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

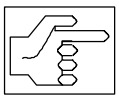
9. 1. 1. Points and point file PNT

While all other variables have to be declared, it is not necessary to expressly declare point variables.

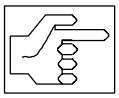
The BAPS2 compiler interprets all undeclared variables as POINT or JC_POINT and reserves the corresponding space for this in the point file.

All points which are declared with **DEF** and all points to which no value is assigned anywhere in the program are stored in a point file with the extension **PNT**.

Values can be assigned directly to these points with the function DEFINE (see rho 3 PHG operation) by TEACH−IN or value input.



Points from the PNT file can be assigned values during the program run, i.e. the content of the point file is overwritten, only if they have been declared with **DEF**.



Points which are not declared and to which values are 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.

9. 1. 2. Complete value assignment

Values are assigned to all coordinates of the point variables in the case of complete value assignments.

Only point variables of one data type must be contained in an assignment.
Exception: Mixed operations with the standard functions JC and WC.

9. 2. Assignment of numeric values

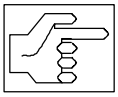
Example:

```
position = (50,0,100,0,15,10)
@edge = @(45,5.8,70,10,5.8,0)
```

9. 3. Assignment of variables for individual components

Example:

```
hole = (xvalue,yvalue,zvalue,0,0,0)
@seat = @(50,95.8,height,40,38,0)
hole.Z_K = height
```



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

9. 3. 1. Assignment by addition and subtraction

The point variable “shelf” has been assigned the sum of the point variable “position” and constant[1].

The individual components are added or subtracted respectively for addition and subtraction.

1) Constants consist of the bracketed string of their coordinates. In contrast to variables, constants do not change during the program run.

Example:

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

9. 3. 2. Assignment with multiplication and division

It is possible to change all coordinate values of point variables or constants by multiplication and division with numeric values or variables of the type REAL.

Example:Multiplication

```
pos_1 = (10,10,10,0,0,0)
pos_2 = pos_1*2
```

Each individual component of pos_1 is doubled by multiplication with 2 and is assigned to the new point variable pos_2.

pos_2 then has the coordinates:

pos_2 = (20,20,20,0,0,0)

Example:Division

```
div = 4.0
pos_3 = pos_2/div
```

pos_3 then has the coordinates:

pos_3 = (5,5,5,0,0,0)

9. 3. 3. Mixed operation with point variables

It is possible to perform mixed operations with point variables of the type **POINT** and **JC POINT** by way of the standard functions **JC** and **WC**.

The calculation must be performed in world coordinates if it is wished to assign the result of the arithmetic operation to a point variable of the type **POINT**.

The calculation must be performed in joint coordinates if it is wished to assign the result of the arithmetic operations to a point variable of the type **JC_POINT**.

Example:

```
@P3 = @(0,100,0,0)
@PI = JC(P2)+@P3
```

The control converts the world coordinates of the point P2 into joint coordinates
and adds them to the coordinates of @P3.
The result is assigned to the point
variable @P1.

9. 3. 4. Reading the actual position POS

The current actual position of the robot can be assigned to point variables during the program run with the standard point variables **POS**(world coordinates) and **@POS** (joint coordinates).

Programming: The standard point variables are on the right side of the assignment.

Component−by−component assignment is also possible.

```
ACT POS = POS
@MACH_POS = @POS
```

```
POS.K_3 = IPOS.K_3
xvalue = POS.K_1
```

9. 3. 5. Component−by−component assignment

New values can be assigned to individual coordinates in the case of point variables.

Conversely, it is possible to assign coordinate values of point variables to variables of the type **REAL**.

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

In the following examples, the coordinates of a position in world coordinates are identified by C1, C2, C3...(Coordinates), while the coordinates of a position in joint coordinates are identified by A1, A2, A3...(Axis).

Programming:

The coordinate designation is appended to the name of the point variable by a full stop, e.g.

position.C3 = 100

The value 100 is assigned to the coordinate C3.

zvalue = armature.C3

The third coordinate value of the point variable ”armature” is assigned to the variable zvalue.

@pal_pos.A4 = radius

The value of the variable ”radius” (type REAL) is assigned to coordinate A4, the position described in joint coordinates.

10. Text variable

Texts can be assigned to text variables within a program.

Variable declaration

Example:

```
TEXT: message,instruction
```

The variables ”message” and ”instruction” are of the type TEXT.

10. 1. Text assignment

The text to be assigned must be in inverted commas (’) and must have a maximum of 80 characters.

The text must be within one line.

Example:

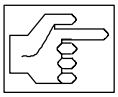
```
message = 'Gripper is faulty'
instruction = 'Change pallet'
```

10. 2. Variable use

Text variables can be output to an output channel (PHG in the example) in the program with the BAPS commands and can be read in with READ.

Example:

```
WRITE PHG, instruction
READ PHG,input_text
```



The variable itself must not be in inverted commas, e.g. WRITE ’instruction’. In this case, the control will output the word “instruction” instead of the agreed text.

11. Arrays

Variables of the same type can be combined in arrays.

Arrays consist of a freely selectable number of array locations which are designated by ascending numbers. A variable can be assigned to each array location by specifying a number (index).

The variables in an array all have the same name and differ only with respect to the index. The index agrees with the number of the assigned array position.

11. 1. Array declaration

The array declaration consists of

- Declaration instruction ARRAY
- Array limits (position numbers)
- Declaration of array variables

The array limits are specified in square brackets and are determined on the basis of the first position number (first index = lower limit) and last position number (last index = upper limit).

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

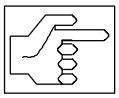
The upper limit of the array must not be lower than the lower limit.

The index is of the type INTEGER.

Example:

Declaration of an array with 5 positions for the point variable“hole”, the first index is to be 1.

ARRAY [1**..**5**] POINT:** hole



All variables can be stored in arrays.

ARRAY[0**..**24**] POINT:** Pallet_Pos

ARRAY[−3**..**17**] REAL:** measured_val

Examples:

ARRAY [1..9] POINT : depot_pos

ARRAY [0..10] TEXT : message

ARRAY [−10..10] INTEGER : number_pos

11. 2. Value assignment for ARRAY variables

Each array variable is addressed by its name and the index in the square brackets.

The variable “number” at array position −5 has the value 1, the variable “number” at array position −4 the value 2, etc.

It is also possible to use a variable or expression of the type INTEGER for the index. Value assignment is then possible via the RPT statement or via the other program loops.

Example:

```
ARRAY[−5..0]INTEGER:number
number[−5] = 1
number[−4] = 2
number[−3] = 4
number[−2] = 8
number[−1] =16
number[ 0] = 32
```

Example: RPT statement

```
ARRAY[−5..0]INTEGER:number
                        ;array with 6 positions
INTEGER:i,k             ;index i and value k
i = −5                  ;first index
K = 1                   ;first value
RPT 6 TIMES
number[i] = k           ;value assignment
i = i+1                 ;increment index
k = k*2                 ;change value
                        (any assignment)
RPT_END                 ;
```

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
Label:                  ;jump label
number[i] = k           ;value assignment
i = i+1                 ;increment index
k = k*2                 ;change value
                        (any assignment)
IF i <= 0               ;conditional statement
THEN JUMP label         ;for program loop
```

Example:
Determination of pallet positions

Declaration:

The pallet has four rows and three columns, i.e. 12 positions.

The distance between the positions is specified by dx and dy as an incremental dimension.

Point assignment:

The following applies to the individual pallet positions palpos k:

palpos [k] = pos + s * dx + z * dy
(Line 23)

The position pos is a teach−in point, i.e. it is defined by travelling to this point and storing it; it is the first position palpos[1].

The entry in the initialization part is thus:

s = 0

z = 0

Incrementation of number of columns:

The next position palpos[2] is located in the neighboring column. The column number s must therefore be incremented (line 27). At the same time, the column number s must not exceed the total number of columns (line 28).

If s is less than column, the control jumps to the jump label “label_1” (line 21), increments the index k by 1 (line 22) and assigns the value to the position palpos[2].

$$pos + 1 \times dx + 0 \times dy$$

If s is higher than column, the control assigns the value zero to the variable s (line 28) and increments the row number (line 33).

Incrementation of row number: Incrementation of the row number takes place analogously to incrementation of the column number.

The ELSE statement is missing in the IF−THEN statement; if the condition z < row is not satisfied, the control continues with the travel instruction (line 36).

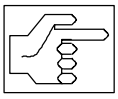
```
 1 PROGRAMM palpos
 2
 3 ;Determination of pallet positions
 4
 5 ; Declarations
 6
 7 INTEGER:row,column,s,z,k
 8 ARRAY[1..12]POINT:palpos
 9 BEGIN
10 dx=(30,0,0,0,0)
11 dy=(0,20,0,0,0)
12 row=4
13 column=3
14 s=0
15 z=0
16 k=0
17
18 ;Statements
19 ;Point assignment
20
21 label_1:
22 k=k+1
23 palpos[k]=pos+s*dx+z*dy
24
25 ;Increment number of columns
26
27 s=s+1
28 IF s<column      THEN JUMP label_1
29                   ELSE s=0
30
31 ;Increment row
32
33 z=z+1
34 IF z<row   THEN JUMP label_1
35
36 ;Travel instruction
37
38 ;;INT=LINEAR
39 V=1000 AFACTOR=9.999
40 k=0
41 RPT 12 TIMES
42 k=k+1
43 MOVE TO palpos[k]
44 MOVE_REL WITH V=36 EXACT (0,0,-20,0,0)
45 WAIT 2
46 MOVE_REL WITH V=59 EXACT (0,0,+20,0,0)
47 RPT_END
48
49 MOVE_REL CIRCULAR ( (-50,-50,100,0,0),
                       (-100,-100,0,0,0) )
50 HALT
51 PROGRAM_END
```

12. Comparison

The control polls values and states in conditions, e.g. “UNTIL condition” or “IF condition”. This interrogation takes place by means of 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 BINARY, CHARACTER, POINT, JC_POINT and TEXT can be polled only with respect to = (equal to) or <> (not equal to).

13. Logic operations

The control checks conditions (also refer to “Comparisons”, “conditional statements”) with respect to their truth value. Conditions can thus only have one of two “values”:

Value 1 for true

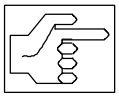
Value 0 for false

This also applies to variables of the type BINARY.

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

13. 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 und NOT.

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

13. 2. Negation of conditions

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

Example 1:

Condition bed_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

Example 2:

The condition cond_2 is false, thus
(cond_2)=0
The following is thus true for the inverse function with NOT:
(NOT cond_2)=1

Notes on programming combinations of several conditions.

Results of comparative operations are always of the data type BINARY. If several conditions are combined with each other, the 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:

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 in order to define the order for processing expressions:

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

14. Channels

BAPS2 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 the rho 3:

Channel number	Type and meaning
1..120	BINARY inputs/outputs
201..224	REAL inputs/outputs
401..408	INTEGER inputs/outputs
501..508	Belt channels

14. 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.

Please refer to the examples on the following page.

Example: Signals of the type BINARY

```
INPUT        : 1 = gate_switch1,
               2 = met_unit,
               5 = li_barrier

OUTPUT       : 7 = alarm
```

14. 2. Data types

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

- BINARY: Interrogation and setting to state 0 or 1.

  Depending on version, the control possesses up to 120 binary inputs and up to 120 binary outputs.

- INTEGER: Interrogation and setting to whole–number numeric values in the range between 0 and 255. The control treats these numeric values internally as data of the type INTEGER.

  Depending on version, the control possesses up to 4 inputs and up to 8 outputs of the type INTEGER; also refer to ”rho3 Description of machine parameters” and ”rho3 Signal description”.

- REAL: Interrogation and setting to analog voltage values. The control treats these voltages as data of the type REAL internally. The number of analog inputs and outputs depends on the hardware configuration and is defined via machine parameters P404 and P406.

- 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 rho 3 can be used as a hardware input. Parameterization takes place via machine parameter 501.

14. 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
```

Example: Signals of the type REAL

```
INPUT REAL       : 201 = torque,
                   206 = force

OUTPUT REAL      : 203 = speed
```

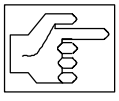
14. 3. 1. Interrogation of channels and signals

Interrogation and evaluation of the channels or their assigned names takes place in conditions, e.g.

```
WAIT UNTIL gate_switch = 1
IF grip_force >= 26 THEN...
WAIT UNTIL meas_height >= 212
MOVE LINEAR UNTIL meas_height >= 200 TO
pos
```

It is not necessary to specify "=1" when interrogating binary signals for 1. The control then automatically interrogates for 1, e.g.

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

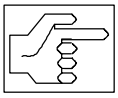


Output signals cannot be interrogated. Interrogation is possible only for input signals.

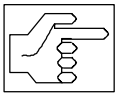
14. 3. 2. Setting signals

Signals are set in statements (see“value assignment”), e.g.

pressure = 75



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



Binary signals can be combined with AND, OR, NOT, e.g.

IF NOT gate_switch1
AND li_barrier
THEN alarm = 1

Exception:

WAIT UNTIL condition and

MOVE UNTIL condition:

No combinations are permitted here.

15. Analog inputs/outputs

The option allows you to receive or output analog signals in the form of voltage values at the interface.

The control thus informs itself about the voltage state of externally connected devices, incorporates the received values in the program run and outputs voltage values itself for control of external devices.

The control processes these input and output voltages internally as decimal values (REAL values).

Analog inputs and outputs are addressed via the channel numbers 201 to 224.

15. 1. Analog inputs

The corresponding inputs must be activated beforehand so that the control can receive analog input data during the program run.

This is done by means of the machine parameter program (see rho 3 Description of machine parameters).

– Activation of inputs

The number of user–accessible analog inputs is defined with machine parameter P406.

Up to 24 analog inputs can be defined, depending on the hardware configuration.

P406 NUMBER OF USER–ACCESSIBLE ANALOG INPUTS

MP SET
P406 ANALOG–INPUTS
1
#

PHG menu

15. 1. 1. Hardware configuration Inputs

Machine parameter P401 serves to define the servo card, connector and input module to be used for reading the input signal.

In the adjacent example, all steps which must be carried out when entering machine parameter P401 for analog inputs are listed in the corresponding order.

The data required here depends on the configuration of your control and robot. Please ask your responsible service department for the interface data valid for you.

P401 CONFIGURATION OF MEASURING SYSTEM CARD

```
ANA-Inp.1       MP SET
P401 CONST.M.S.BOARD
     Servo-B.: 1
#
```

Definition of the servo card

```
ANA-Inp.1       MP SET
P401 CONST.M.S.BOARD
     --- Plug: X
#
```

Input of connector number

```
ANA-Inp.1       MP SET
P401 CONST.M.S.BOARD
     POT Module No: 1
#
```

Input of potentiometer module

```
ANA-Inp. 1      MP SET
P401 CONST.M.S.BOARD
     POT  Input: 1
#
```

Input of potentiometer module input

PHG menu

15. 1. 2. Assignment of input channel numbers

Channel numbers are assigned to the analog inputs at the interface with machine parameter P407.

Permitted values: 201....224. The corresponding analog inputs are addressed in a BAPS2 program by means of this channel number.

P407ASSIGNMENT OF USER-ACCESSIBLE ANALOG INPUTS

```
ANA-Inp.1    MP SET
P407 MEAN. OF A.-IN
Meaning:   REAL:201
#
```

PHG menu

15. 1. 3. Nominal value definition inputs

Machine parameter P401 is used to define a decimal nominal value which corresponds to the decimal input voltage of 10 V. A value range from 0.01 to 9999.99 is available for this purpose.

P401 NOMINAL VALUES FOR ANALOG INPUTS

```
ANA−Inp. 1    MP SET
P401 CONST.M.S.BOARD
Nom.val: 1000.00
#
```

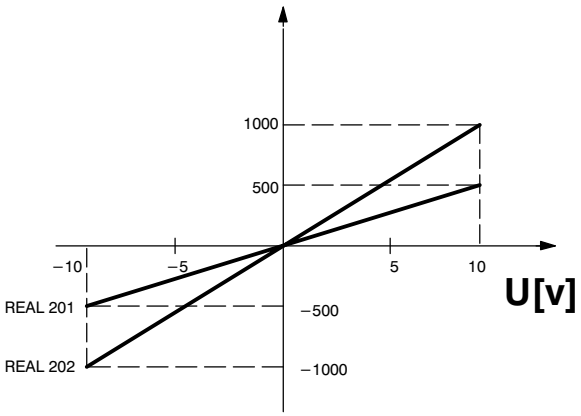
PHG menu

If the control now knows which value it is to assign to a voltage of 10 V, it automatically assigns the proportional REAL value to every other voltage value between −10 V and +10 V.

The value ranges for the analog inputs REAL 201 and REAL 202 are plotted in the adjacent example.

The nominal values have been defined at 500 and 1000.

Example



15. 1. 4. Value ranges: Analog inputs

Permitted input voltage: −10V...+10V

Value range REAL variable:
−(Nominal value)...+(Nominal value)

Formula for REAL value:

$$\text{REAL−value} = \frac{\text{input voltage [v]} \times \text{nominal value}}{\text{10 [v]}}$$

Nominal value = Machine parameter P405

15. 2. Analog outputs

Analogously to analog inputs, analog outputs must also be activated with the machine parameter program before they can be used in the program run.

– Activation of outputs

The **number** of analog outputs is defined with machine parameter P404.

The maximum number of analog outputs is determined by the number of analog outputs available in the hardware configuration, less the axes controlled by the control system (every controlled axis which is not controlled via the CAN interface occupies one analog output for setpoint output).

P404 NUMBER OF USER–ACCESSIBLE ANALOG OUTPUTS

```
          MP SET
P404  ANALOG–OUTPUTS
        0
#
```

It is thus possible to activate a maximum of **three** user–accessible analog outputs with an 8–axis servo and a 5–axis robot (see adjacent allocation).

Analog outputs

1	1st axis
2	2nd axis
3	3rd axis
4	4th axis
5	5th axis
6	Analog output 1
7	Analog output 2
8	Analog output 3

15. 2. 1. Assignment of channel numbers (outputs)

Machine parameter P405 serves to assign channel numbers to the outputs at the interface via which the corresponding output is then addressed in the BAPS program.

Permitted values: 201, 202, 203...max. 224

P405 ASSIGNMENT OF USER−ACCESSIBLE
ANALOG OUTPUTS
DEFINITION OF SERVO CARD AND
BAPS2 CHANNEL NUMBER FOR ANALOG OUTPUTS

```
ANA−Out. 1      MP SET
P405 MEAN. OF A.−OUT
Servo−B.:  1
#
```

PHG menu

```
ANA−Out.  1     MP SET
P405 MEAN. OF A.−OUT
Meaning:   REAL:201
#
```

PHG menu

15. 2. 2. Nominal value definition (outputs)

Machine parameter P405 serves to define a decimal nominal value which corresponds to the maximum output voltage of 13.3 V (10.0 for the narrow 3 to 5−axis servo card).

P405 NOMINAL VALUES FOR ANALOG OUTPUTS

```
ANA−Out. 1      MP SET
P405 MEAN. OF A.−OUT
Nom.value: 1000.00
#
```

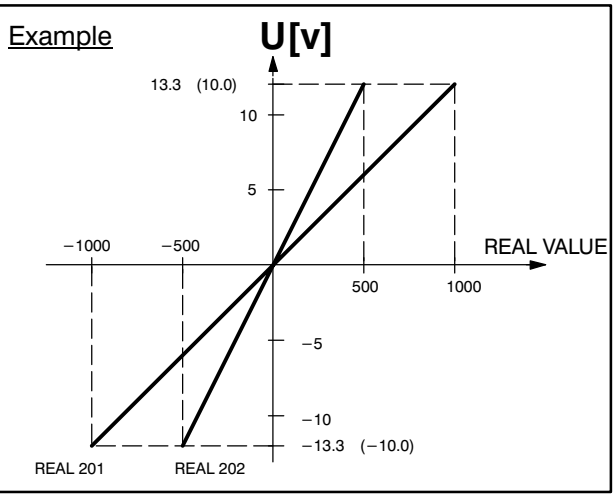
PHG menu

The value range from 0.01 to 9999.99 is available for definition of nominal values.

If the control now knows which REAL value (nominal value) corresponds to the voltage of 13.3 (10.0) V, it then automatically assigns a proportional voltage value between −13.3(10.0) and +13.3(10.0) V to every smaller decimal value.

The adjacent example shows the voltage characteristics of the analog outputs REAL 201 and REAL 202 plotted against the value range.
The nominal values were defined as 500 and 1000 respectively in this case.

Example



15. 2. 3. Fixation of the voltage offset

The voltage offset for the analog output channels is defined with machine parameter P405. The value range for −100 to +100 is available for this purpose

You thus determine the percentage share of the maximum output voltage of 13.3 (10.0) V, which is output during the program run even if the program REAL value for the corresponding output is zero.

The voltage offset results in every output voltage being increased or reduced by this value.

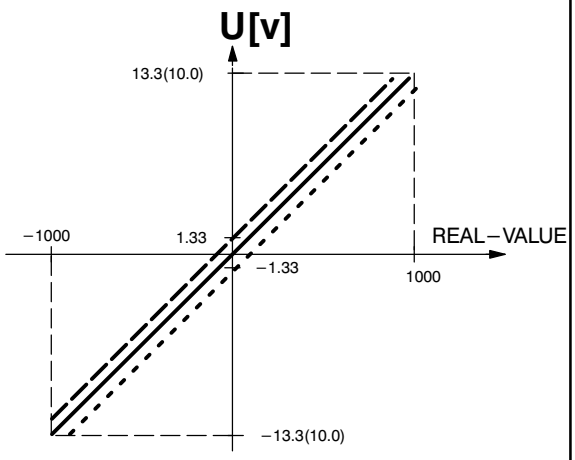
P405 VOLTAGE OFFSET ANALOG OUTPUTS
IN % OF MAXIMUM VOLTAGE

```
ANA−Out. 1     MP SET
P405 MEAN. OF A.−OUT
Volt.off. (%): 10.00
#
```

PHG menu

The adjacent diagram compares the characteristics of three output voltages with different offset factor by means of the relationship of

$$\frac{\text{REAL value}}{\text{nom. value}} \quad 10\ [v]$$



without offset factor

with positive offset factor (+10%)

with negative offset factor (−10%)

15. 2. 4. value range: Analog outputs

Output voltage:

−13.3V...+13.3V (−10.0V..+10.0)

Value range REAL variable:

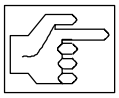
−(Nominal value)...+Nominal value)

Formula for output voltage:

$$\text{output voltage [V]} = \left(\frac{\text{REAL value}}{\text{Nom. value}} + \frac{\text{Offset−factor [\%]}}{\text{100 [\%]}} \right) \text{ x } 13.3\text{V}$$

Nominal value = Machine parameter P405
Offset = Machine parameter P405

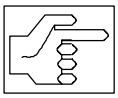
The error message INVALID VALUE is output and the program is aborted if the permitted value range for the REAL variable is exceeded.



The analog outputs are reset in the event of a program abort and program end, i.e. the offset voltage corresponding to machine parameter P405 is present at the analog output.

15. 3. Declaration of analog input and output channels

Analog input and output values are treated internally by the control as data of the type **REAL**.



Analog channels must therefore be declared in the program as **real** inputs and outputs.

Special **reserve** channel numbers must be assigned to the analog channels so that the control can distinguish analog channels from integer digital channels.

The channels 201 to max. 224 had to be allocated when the channels were activated, and you now refer again to these.

The input and output channels must be declared in the declaration part of the program.

The repective input and output may have any name, but the name must not be longer than 12 characters.

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

If you should declare an invalid channel number (e.g. 301), this will lead during the program run to the error message.

”INVALID VALUE”

Example

```
INPUT REAL:     201 = ANA_INPUT_1,
                202 = ANA_INPUT_2,
                203 = ANA_INPUT_3
OUTPUT REAL:    201 = ANA_OUTPUT_1,
                202 = ANA_OUTPUT_2
```

15. 4. Interrogation of inputs – setting outputs

Analog inputs and outputs are treated like internal variables of the type REAL in BAPS programs.

Example:

You have declared two analog inputs and two analog outputs in a BAPS program.

Inside the control, the input variables are constantly allocated values “from outside”.

These input variables allow all arithmetic, logic and comparative operations to be performed which are permitted for REAL variables. (See Chap. 5 “Programming with BAPS variables”).

Output variables can be set directly

- by specifying a REAL values or
- with reference to an input variable or an above–described operation.

15. 5. Restrictions

- No value assignment must take place to an analog input variable within a BAPS program.
- An analog output variable cannot be read.

If one of these conditions is not observed, the following error message appears when the program is compiled.

”INPUT VARIABLE NOT PERMITTED HERE”
or
”OUTPUT VARIABLE NOT PERMITTED HERE”

Example:

```
PROGRAM ANA_IO

; Declarations
REAL: MEAS_VALUE_1, LIMIT_VALUE, MAX_VALUE

INPUT REAL:  201  =  ANA_INPUT_1,
                     202  =  ANA_INPUT_2,
OUTPUT REAL201  =  ANA_OUTPUT_1,
                     202  =  ANA_OUTPUT_2

;Statement part of the program

BEGIN
LIMIT_VALUE =      500
MAX_VALUE = 900

LOOP:

MOVE TO START_POINT

MEAS_VALUE_1 = ANA_INPUT_1 + ANA_INPUT_2

WRITE 'MEASURED VALUE = ', MEAS_VALUE_1

ANA_OUTPUT_1 = MEAS_VALUE 1 / 10

MOVE LINEAR UNTIL ANA_INPUT_1 >= LIMIT_VALUE TO
                                                END_POINT
WAIT UNTIL ANA_INPUT_2 >= LIMIT_VALUE MAX_TIME = 10 ERROR JUMP
TIME_ERROR

IF ANA_INPUT_1 > LIMIT_VALUE THEN
  WRITE 'LIMIT VALUE INPUT 1 EXCEEDED'
ELSE
  ANA_OUTPUT_2 = ANA_INPUT_2

WRITE 'INPUT_2 = ', ANA_INPUT_2

IF ANA_INPUT_2 < MAX_VALUE THEN JUMP LOOP

TIME_ERROR:
WRITE 'Wait time expired'
HALT
PROGRAM_END
```

16. Special functions

Special functions in the control rho 3 for which no BAPS2 language elements have been reserved are made accessible to the BAPS2 programmer here.

Special funtions represent an extension of the BAPS2 language scope. However, they can be called in a program only if they are activated in your control software (as an option) and if they are declared before they are called in a similar way to variables.

The following special functions are currently available in the rho 3:

Fct. No.	Brief function description
1	Exact−position switching of digital outputs on the path
2	Exact−position switching of decimal outputs on the path
23	System time and date
24	System timer
27	Switching WC to main area

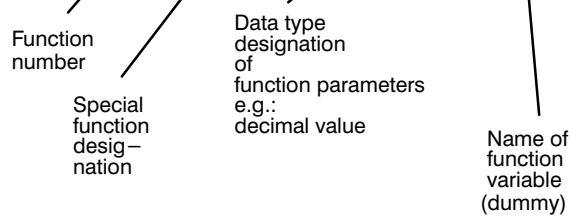
16. 1. Declaration of special functions

The declaration of a special function contains its code number and designation as well as the names and type designations of the function parameter(s) by means of which you define in the special function call when, where and how it is to be active. The declaration must be made in the declaration part of the program.

The designation of the special function and the names of the function parameters can be freely chosen. The data types are predefined by the specification of the respective special function.

Example:

SPC_FCT : X = EXAMPLE (VALUE REAL: WAP)



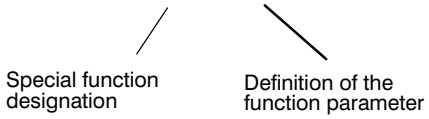
16. 2. Calling special functions

Special functions are called in the statement part of the program by specification of the special function designation and definition of the agreed function parameter(s).

The designation of the special function and the function parameter types used in the call must be as defined in the declaration in the program.
The special functions which are currently available in the rho3 control are described below.

Example:

EXAMPLE (15.5)



16. 3. Exact−position signal output for travel

General

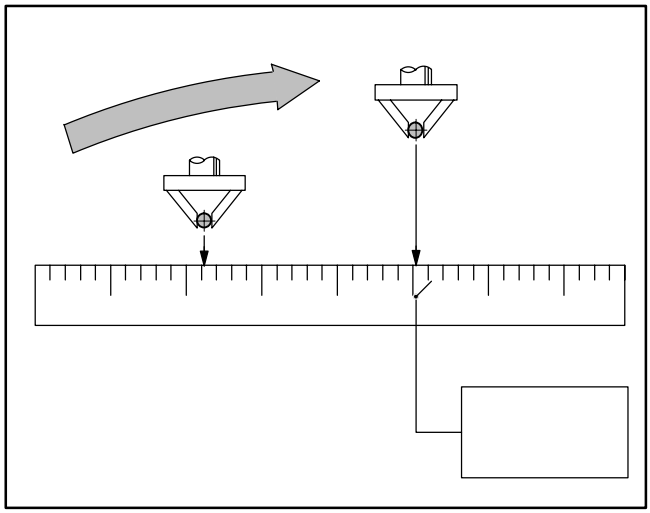
The option allows exact−position control of an external peripheral device with rate time from a BAPS program during a travel movement.

Digital values are output at the interface with special function 1 and whole−number values (process parameters) between 0 and 255 or real values with special function 2 if the output channel is an analog output.

These values can be used to control your technological systems, e.g. the paint quantity for painting applications.

Exact−position output

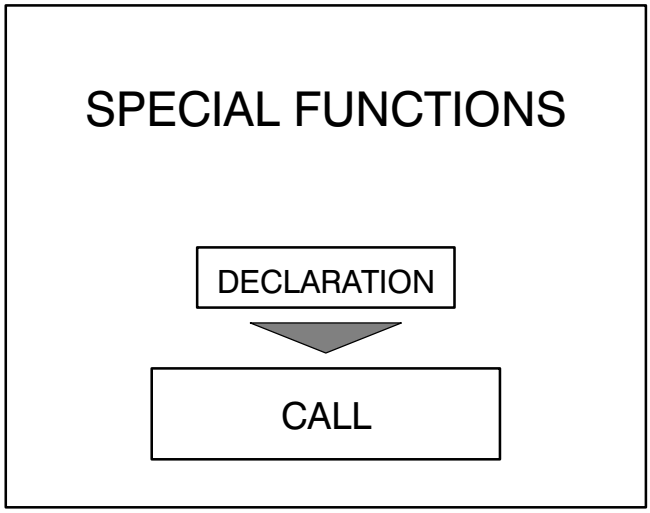
A value is output here in conjunction with an exact position, i.e. you define a certain gripper position in the program at which the external device is to execute the instruction received from the interface.



Programming

The control instruction is issued via the special function.

Special functions must be declared before they are called in the same way as for variables.

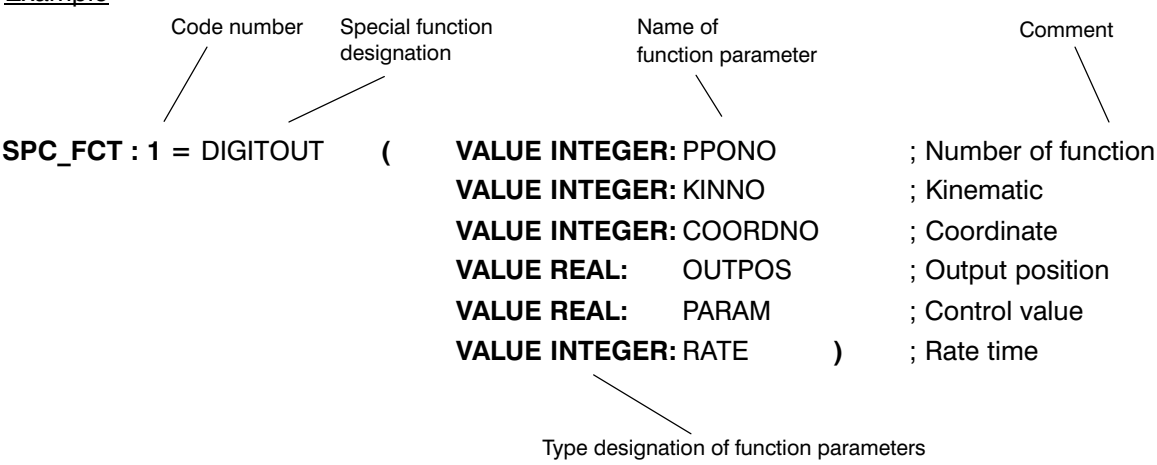


16. 3. 1. Declaration of special function 1

The declaration of this special function includes the code number 1 and the special function designation as well as the names and type designations of the function parameters by means of which you define in the special function call at which position of the corresponding axis which control value is to be output.

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

Example



The control value (PARAM) is defined as a decimal value, whereby a value < 0.5 is output as logical 0 (low) and a value > 0.5 as logical 1 (high).

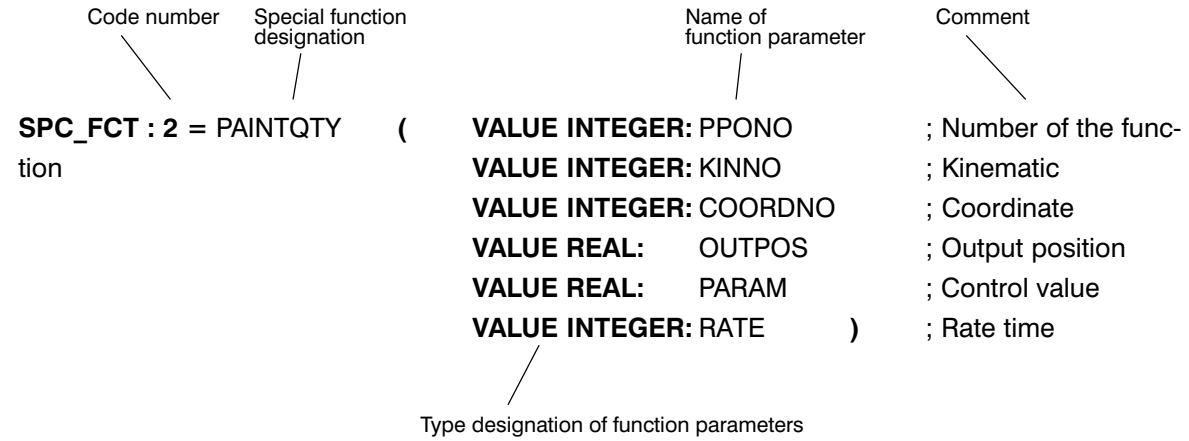
16. 3. 2. Declaration of special function 2

The declaration of this special function includes the code number 2 and special function designation as well as the names and type designations of the function parameters by means of which you define in the call of the special function at which position of the corresponding axis which control value is to be output.

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

The declaration of the special function must be contained in the declaration part of the program.

Example



The control value must be defined as a decimal value. The value to be output is calculated in accordance with the following formula if an INTEGER output is used as the output channel, i.e. PPONO has the values 1..8 (also see machine parameter P18):

PARAM = PARAM MOD (256),

i.e. the parameter to be output has the value range 0..255.

If the output channel is used as a REAL output, i.e. PPONO has the values 201..208, the control value is then weighted with the nominal value entered in machine parameter P406.

16. 3. 3. Function parameters

You declare the following function parameters:

- **Function number**

This specifies the output channel via which the process parameter is to be output; the values 1 to 8 are permitted here (also see rho 3 Description of machine parameters P17 and P18 and P405,P406,P407).

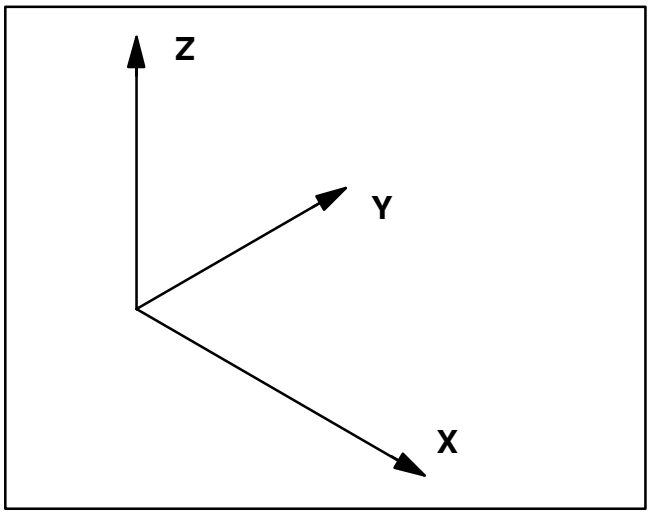
In the case of special function 2, it is possible to choose between digital (numbers 1..8) and analog outputs (numbers 201..208) corresponding to the set machine parameters.

- **Kinematic**

You specify in which kinematic the function is to be used.

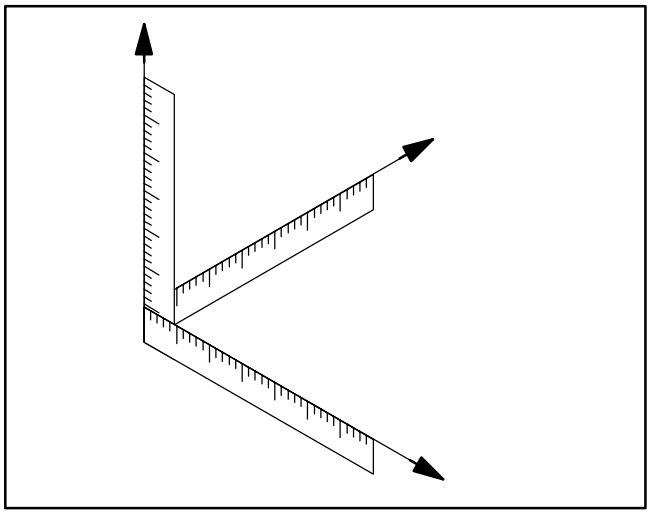
- **Coordinate**

This informs the control that a whole−number value will be input in the special function call which defines the coordinate or axis.



- **Position**

This informs the control that a decimal value will be entered in the special function call which defines the position on the previously stated coordinate axis.

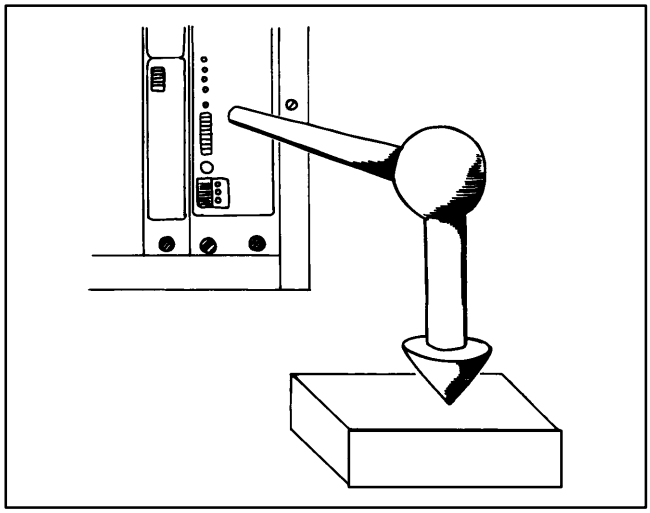


● **Control value (process parameter)**

The control value is used to define a technology−related value.

A digital output is thus switched On (0) or Off (0) with special function 1.

An 8−bit wide digital output or an analog output is set with special function 2, depending on the output channel used.



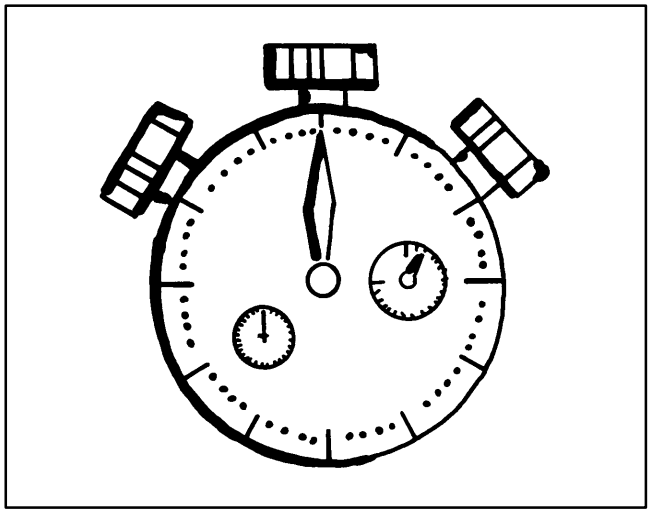
● **Rate time**

This informs the control that a whole−number value will be entered in the special function call which defines a correction time − the rate time.

Rate time

The rate time is the time in milliseconds which elapses between actual output of the control value and attainment of the specified axis position.

The process parameter is thus sent to the external device before the desired position is reached.



The rate time allows the reaction time of your process peripherals to be compensated and thus makes sure that the control signal is output at the ”right” time so that the switching function becomes active at the desired position.

Only positive values are permitted for the rate time.

Example

You wish to inform your metering system of a new paint quantity which is to be become active exactly when the gripper reaches the position POSF.

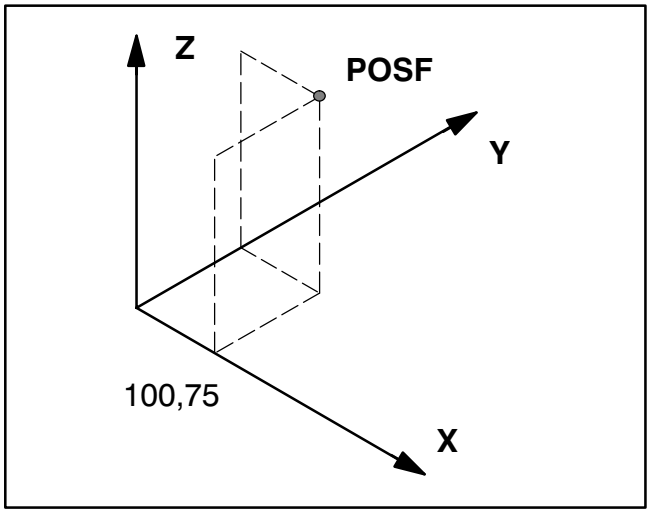
The position POSF has the coordinate value 100.75 on the X−axis.

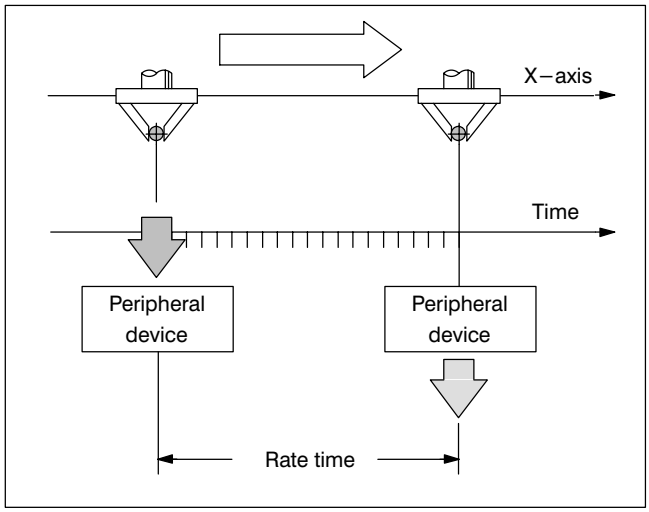
The new paint quantity corresponds to the control value 79.

Since you are less interested in the exact time of output of the control value and more interested in its realization at the time of a certain event (reaching of X−coordinate value 100.75), you should also take into account the reaction time of the power amplifier, i.e. the time which elapses between arrival of the signal and its technical realization.

It is therefore recommended to accept the reaction time of the power amplifier as the rate time since the process parameter change then agrees in time exactly with reaching of the X−coordinate value 100.75 **independently** of the path speed.

Please refer to the technical data sheet of your connected power amplifier to determine its reaction time.

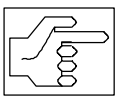




The rho 3 therefore outputs the control signal before the position POSF is reached and thus achieves speed−independent output of the process parameter.

This provides you with the possibility of varying the path speed (in the test run, for example) without influencing process parameter output.

The control internally calculates the respective actual output position in accordance with the valid path speed and defined rate time.



An excessively high path speed moves the actual output position to before the start point of the travel movement. The error message
“PPO:Pos not reached” is then issued.

16. 3. 3. 1. Special function call

Once a special function has been declared, it can be called any number of times in the statement part of your program. To do this, specify the special function designation (e.g. paint quantity) and define the declared function variables.

Example

PAINTQTY(1,1,3,100.75,79,60)

Special function designation

Function parameters

Function variables

The **PPO number** defines the output channel via which the process parameter is output. The output address is defined by way of a machine parameter. Up to 8 channels are possible in the rho3.

The **kinematic number** defines the kinematic for which the process parameter output is to take place.

The **coordinate axis** is defined by input of 1, 2 or 3, this corresponding to the axes X, Y or Z respectively of the world coordinate system.

The **position** is defined by input of a decimal value which expresses the desired axis position in millimeters.

The values from 0 to 255 are available in the case of digital channels for the **control value** (Process parameter).

The **rate time** is defined by whole−number values greater than or equal to zero and expresses the mentioned time in milliseconds.

The above example of a special function call contains the following statement:

The control value 79 is output with a rate time of 60 ms onto the PPO output channel 1 at the coordinate value 100.75 of the 3rd axis of the 1st kinematic.

1	PPO−No. 1..8
1	Kinematic number
3	3rd axis of the kinematic
100.75	Coordinate value: 100.75mm
79	Control value 79
60	Rate time: 60 ms

16. 3. 4. Special function call with variables

Variables can also be used for the function parameters in the special function call, e.g.

PAINTQTY (1,1,COORD, POINT, 207, RATE)

The variables must have been declared and set beforehand.

The call of special function 2 in line 51 of the adjacent example would contain the following statement:

The process parameter 207 is output with a rate time of 70 ms at the 3rd coordinate 98.70.

The advantage of variable programming is that you can change the variables very easily by way of the test system in the optimization phase.

Example

```
8 INTEGER: COORD,RATE
9 REAL  : POINT

48 COORD  = 3
49 RATE   = 70
50 POINT  = 98.70
51 PAINTQTY(1,1,COORD,POINT,207,RATE)
```

16. 3. 5. Effect of the control value

A control value (process parameter) is constantly present at the interface. The last−set value − from an already completed program run − is still present even at the start of the program before the first special function is called.

For this reason, one often speaks of a process parameter change.

16. 3. 6. Preventing a process parameter change

A process parameter change programmed by special function 2 is **not** executed if one of the following conditions is applicable:

- The signal "**feed hold"** is triggered during a travel block in which a process parameter change is to take place. No process parameter change occurs as long as the signal is present.
- The signals "**Traverse enable off all kinematics**" or the signal "**Traverse enable**" of the corresponding kinematic is not set.
- The input signal **"Reset"** or RBS instruction **RESET** is set.
- The signal "**Emergency mode**" is pending.
- **System errors** occur during the program run, e.g. servo errors or interpolator stop.

16. 3. 7. Error messages

If you should not observe the defined function parameter ranges in the special function call, the control will recognize this during the program run and output the error message

“**PPO/IOL:Err.PPO−Prog**”.

If you should not have incorporated the special function call in the program properly, the control will also recognize this during program execution or in the test run and output the error message

“**PPO:Pos not reached**”.

Example

PAINTQTY(1,1,5,123,12,100)

Invalid coordinate programmed.
If the selected kinematic has fewer than 5 axes.

PAINTQTY(1,1,3,123,12,100)

Inadmissible process parameters programmed.
Only values in the range 0..255
are permitted.

PAINTQTY(1,1,3,234,13,−100)

Invalid rate time.
Only times greater than or equal to zero are permitted.

16. 3. 8. Calculation of the actual output position

The parameter 249 is to be output at the Y−coordinate 120 with a rate time of 70 ms(see examp− le).

The gripper moves with a speed of 1000 mm/s from the Y−coordinate 176 to the Y−coordinate 99.

The process parameter must therefore be output at a position from which the Y−position 120 is reached in 70ms at a speed of 1000 mm/s.

Distance = Speed x time

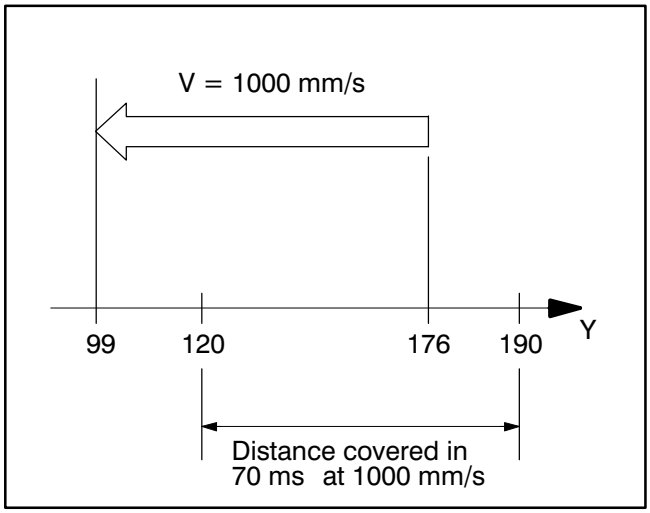
= 1000 mm/s x 0.070 s

= 70 mm

In other words, the actual output position is 70 mm before the specified output position.

(Y−coordinate value 120).

This is thus located at Y−coordinate value 190, and thus leads to the above error message.



Output of the process parameter at the interface

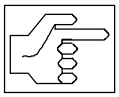
The process parameter is output to the peripheral equipment in accordance with the set machine parameters.

The pending process parameter is output both in the program run and in test operation.

The rate time defines a time before a certain position is reached at which the process parameter is output or the existing one changed.

It therefore does not matter whether you are working with a very high or very low gripper speed.

The rate time allows the process parameter to be output at the ”right” time independently of the programmed speed so that the control value becomes active at the desired position.



An excessively high path speed or excessively high rate time may move the actual output position to before the start point of the travel block. The following error message then appears:

“PPO:Pos not reached”.

16. 3. 9. Preventing process parameter output

A process parameter change programmed by the special functions 1 and 2 is not performed if one of the following conditions is applicable:

- The signal ”**feed hold**” is triggered during a travel block in which a process parameter change is to take place. No process parameter change takes place as long as this signal is present.
- The signals ”**Traverse enable off all kinematics**” or the signal ”**Traverse enable**” of the corresponding kinematic is not set.
- The input signal ”**Reset**” or RBS instruction **RESET** is set.
- The signal ”**Emergency operation**” is present.
- **System errors** occur during the program run, such as servo errors or an interpolation stop.

16. 3. 10. Error messages

It must be ensured that the defined function parameter ranges are observed.

Otherwise, the control will recognize this during the program run and issue the error message

"**PPO/IOL:Err.PPO−Prog**".

In the adjacent example 2, the actual output position of process parameter 1 is inside the travel block range and [1]that of process parameter 2 outside it[2] (block 35/block 38).

The following error message thus appears:

"PPO: Pos not reached".

This error message is also output if the output position is in the acceleration or deceleration phase of the movement.

1) 100.75 + (140 x 0.06) = 109.15

2) 100.75 + (140 x 0.6) = 184.75

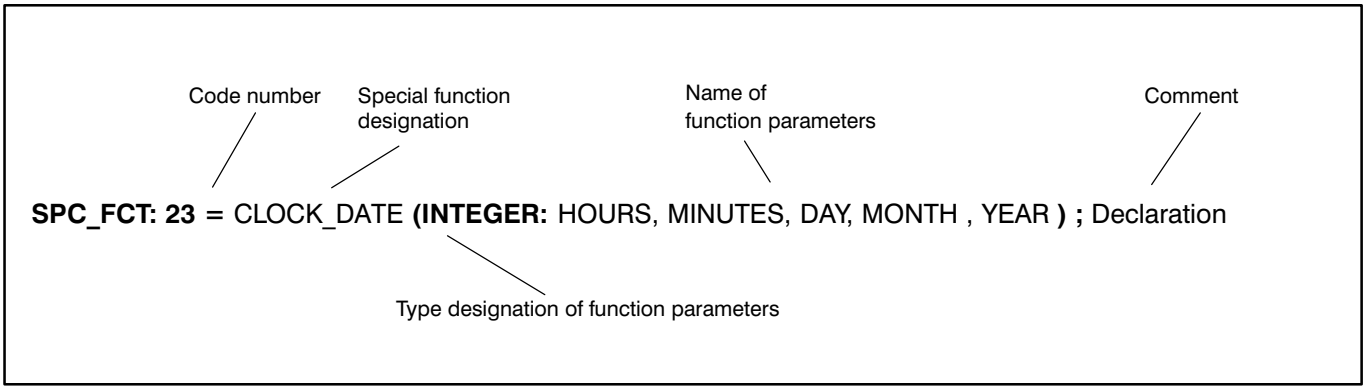
Example 2

```
POS6 = (40,176,70,0,60)
POS7 = (120,99,70,0,−10)

35 MOVE LINEAR VIA POS6
36 PAINTQTY(1,100.75,60)
37 AIRQTY(1,100.75,204,600)
38 MOVE LINEAR WITH V=140 VIA POS7
```

16. 4. Special function 23 System date and time

Special function 23 permits access to the system clock of the rho 3 from a BAPS2 program. The day, month, year, hours and minutes are determined.

Special function declaration:

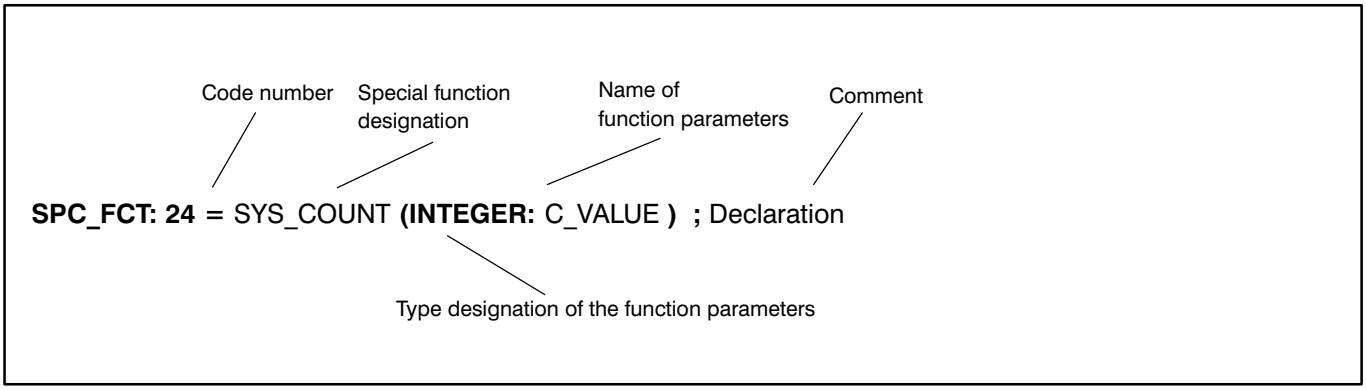


16. 5. Special function 24 System counter

Special function 24 permits access to an internal real time counter of rho3 from BAPS2. The current value of the counter at the run time is determined.

The value of the system counter is written into the variable transferred when the special function was called.

The variable value is in the unit ”milliseconds” [ms]. The counting time base corresponds to the clock time. The real time counter is set to zero again by every ”start−up” of the control.



16. 6. Special function 27

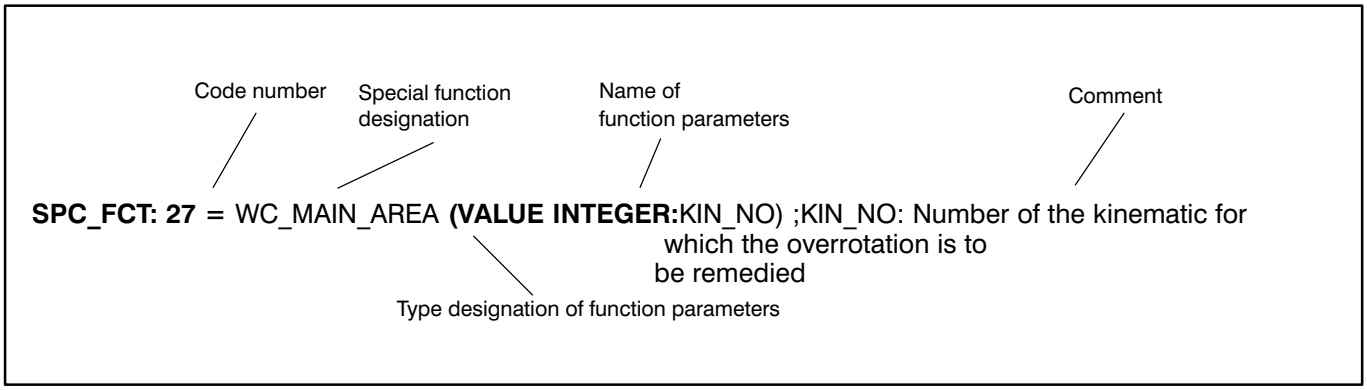
Overrotation of the world orientation angles in WC may occur as a result of coordinate transformation.

The special function 27 serves to eliminate this overrotation.

Preconditions :

The kinematic **m u s t** be travelled into the main area with PTP interpolation before the special function 27 is called.

{ e.g.: MOVE PTP TO @(0,0,...) }



Example:

;;INCLUDE DEFINE

;DEFINE contains compiler statements

PROGRAM SPC27

SPC_FCT: 27=WC_MAIN_AREA**(VALUE INTEGER:**KIN_NO**)**

BEGIN

MOVE ROBI_2 **PTP @(**0,0,0,0,0,0**)**

WC_MAIN_AREA (2)

PROGRAM_END

The above example relates to a 6−axis kinematic with the name 'ROBI_2' which is the second kinematic defined in the control. Travel to the main area takes place in PTP interpolation mode:

{ @(0, 0, 0, 0, 0, 0) }.

After this, the special function 27 with the name 'WC_MAIN_AREA' is called for the second kinematic.

17. Communication functions

General

The option permits communication of your control with other intelligent systems during the program run.

You thus extend your system limits by involving peripheral devices in the control sequence and obtain the possibility of much more flexible programming.

The BAPS2 statements **WRITE** and **READ** are available for communication.

Syntax:

WRITE Device name**,**Variable[**,**Variable]

READ Device name**,**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 a hardware interface via machine parameters or via mode 9.1 with PHG 3 via a device number.

The assignment of device number and interface connection is shown in the adjacent table.

Assignment: Device No., device name, interface					
Device No.	**0**	**1**	**2**	**3**	**4**
Device name:	**V24_1.. V24_4,TTY**		**PHG**	**V24_1.. V24_4,TTY**	
Interface: CP/MEM 4	**X11**	**X12**	**X22**		
AF3	**X11**		**X22**		
CP2.5	**X13**				**X11**
AF5	**X11**	**X12**	**X22**	**X31**	

17. 1. Protocol selection for communication functions

Different communication protocols are available for communication. These can be selected by means of the machine parameter setting or via mode 9.1 with PHG3.

P. 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	rho 1/2 compatible with P. No. 3	no

1a = Data input
1b = Data output

17. 2. The BAPS instruction WRITE

The instruction WRITE is used to output 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 output via the selected interface.

Data output takes place as an ASCII character string, i.e. conversion from internal format to ASCII format occurs.

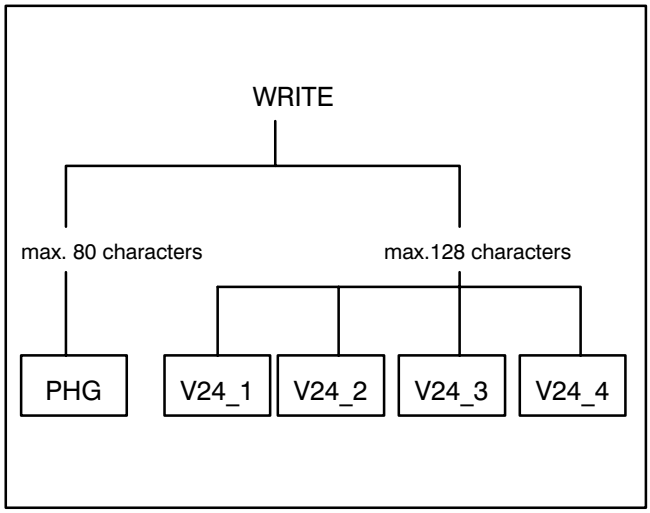
17. 3. Interfaces

The data can be output via the interfaces (device names)

- **V24_1**
- **V24_2**
- **V24_3**
- **V24_4**

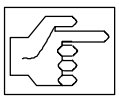
and

- **PHG**.



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 outputs the desired data to the hand−held programming unit **PHG**.



A comma must be entered after input of an interface name.

Example 1

WRITE PHG, G

The variable G is displayed on the PHG.

WRITE V24_2, TE

The variable TE is output via interface V24_2 e.g. to a printer.

WRITE ’3’

The number 3 is displayed on the standard output device.

17. 3. 1. Transferred data

Constants and variables of the type

- **BINARY**
- **INTEGER**
- **REAL**
- **CHAR**
- **TEXT**
- **POINT and**
- **JC_POINT**

can be transferred.

When writing to the PHG, you can transfer a maximum of 80 characters (numbers, letters etc.) per WRITE instruction; transfer of a max. 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 Whole numbers with a maximum of 10 digits in the range from between −2147483648 and +2147483647 can be transferred.

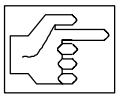
REAL Decimal numbers in the range between −999999, and +999999 can be transferred, minimum resolution ±0.00001. (Transfer takes place as a floating−point number after REAL−ASCII conversion with sign or blank, 6 digits and a decimal point, whereby the position of the latter depends on the value.)

POINT, JC_POINT The individual coordinate values JC_POINT of a position of the type POINT or JC_POINT must lie within the limits of the type REAL.

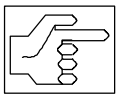
CHAR All ASCII characters in accordance with DIN 66003 can be transferred.

TEXT All ASCII characters except ZERO can be transferred. However, no more than 80 characters may be transferred in one WRITE instruction.

In the case of text constants, the text to be transferred must be placed in quotation marks and be in one line.



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).

Example

```
K = 2
D = 0.123
WRITE PHG,K,'ND VALUE=',D
```

The following display appears on the PHG:
2ND VALUE = 0.12300

17. 4. The 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.

17. 4. 1. Interfaces

The control can read in the variables via the interfaces

- hand−held programming unit PHG
- V24_1
- V24_2, V24_3, V24_4

In order to identify the interface, the interface name must be entered in the program after the READ instruction.

If no interface is specified, the control expects a data input from the PHG.

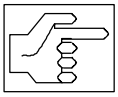
Example

READ V24_1,G

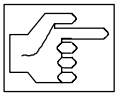
The interface V24_1 must provide the variable G.

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 input.



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.

17. 4. 2. Transferred data

The control can be made to read in variables of the type

- BINARY
- INTEGER
- REAL
- CHAR
- TEXT
- POINT and
- JC_POINT

A maximum of 120 characters (numbers, letters) can be transferred per **READ** statement.

The following restriction applies to transfer in the case of variable type **INTEGER**:

Only whole numbers with a maximum of 9 digits in the range between

−999999999 and +999999999

can be read in.

The same restrictions as for the WRITE instruction apply to the transfer scope of the other variable types.

Error messages

The control may output the following error messages:

- “Interface error” A WRITE/READ instruction has not been executed within the settable time.
- “READ protocol error” The defined transfer format or the internal computer protocol has not been observed for a READ instruction.
- “WRITE protocol error ” A WRITE instruction cannot be executed since the defined transfer format or internal computer protocol has not been observed.

17. 5. Example: READ/WRITE

You wish to inform your control of the gripper position POSITION at a certain point in the program. This is then to be followed by travel to this position.

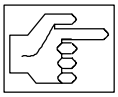
For checking purposes, you wish to display the current gripper position on the PHG beforehand (block 38) and also document this using your printer (block 39). Your printer is connected to the interface V24_2.

The request for coordinate input is written to the standard output device with the program blocks 40, 41, 42.

In block 43, the control expects input of the coordinate values of the point POSITION on the keyboard of the standard output device.

You enter the following position for a 5−axis robot, for example:

200,0,120,−20,40

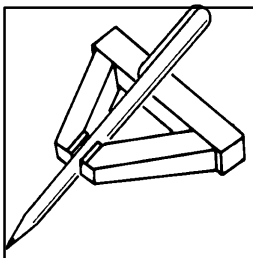


The input must be terminated with RETURN.

The read−in point variable can now be approached (block 44) and output to the printer (block 45, 46).

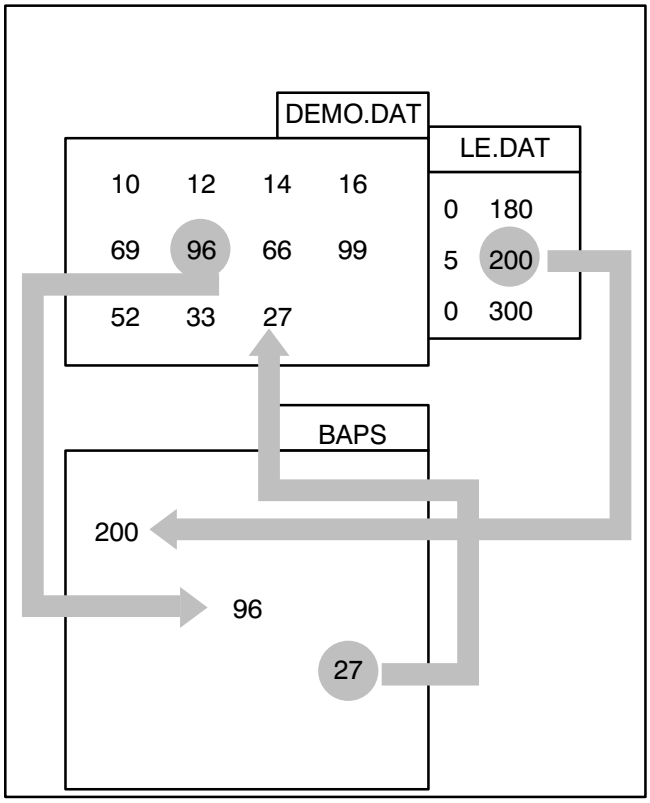
Example

```
35 MOVE_REL DIS12,DIS14
36 MOVE_REL CIRCULAR (KP7,KP8)
37 ACT_POS=IPOS
38 WRITE 'Position circle end=',ACT_POS
39 WRITE V24_2,'Pos. circle end=',ACT_POS
40 WRITE ' '
41 WRITE 'Enter the coordinates of the '
42 WRITE 'gripper position '
43 READ POSITION
44 MOVE LINEAR EXACT POSITION
45 WRITE V24_2, 'Coordinates of POSITION'
46 WRITE V24_2, POSITION
```



18. File operations

18. 1. General



The following **BAPS2** statements are available for 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 out of these **DA**T files and includes them in the program run.

It is also possible to write arbitrary values from a BAPS program into a DAT file.

18. 2. The DAT file

Numeric values for all variable types 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 on the EDIT level.

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 values 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 in this line.

Example

```
;************************
; NAME:          VALUES.DAT
; DATE:          29.2.88
;************************
;
10    20     30     40          ;HEIGHT

100   200    300    400         ;LENGTH

500   600    700    800

12.15        21.2   1.5         ;R_POS

−160.7       −90    0
```

18. 2. 1. 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 representation of numbers:
  0 1 2 3 4 5 6 7 8 9. + −
  Decimal numbers (REAL) are represented as 6−digit floating−point numbers.
  For the representation of CHAR and TEXT the characters ” ”(space) .... ”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 shift to the next line takes place at the line end.
  For this reason, it is not necessary for there to be a space after the last value in a line.
- Line numbers are visible within the DAT file only in EDIT mode (no line information for PRINT or WRITE).

18. 2. 2. Access to a DAT file

If you wish to access one or more DAT files within the scope of a main program, these must be declared as variables of the type FILE.

18. 3. DAT file declaration(s)

Syntax:

FILE : File name[**,**File name]

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.

Example

```
FILE: VALUES,ERG,DISTANCES
INTEGER: NUMBER, I
TEXT:  DISPLAY,FONT
```

18. 4. The 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 variables from the same DAT file.

Example

```
READ VALUES, NUMBER
```

The control reads in a whole−number value for the variable NUMBER from the file VALUES.DAT.

```
READ ERG, I, NUMBER
```

The control reads in a value from the file ERG.DAT both for the variable I and for NUMBER.

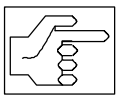
18. 5. Selection of a value within the DAT file

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.

When the next READ instruction occurs, the control automatically jumps to the following value, reads this value and then positions the “invisible READ pointer”.

In this way, the control “reads” from value to value and from line to line.



If the variable type from the program does not agree with the read value in the DAT file, the control outputs 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 and TEXT, all characters from the actuell ”invisible READ pointer” are read.
Note: If you want to read variables of the type CHAR and TEXT from beginning of a line, you should position the ”invisble READ pointer” by using the ”READ_BEGIN” statement.

Example

```
;************************
;  NAME:          VALUES.DAT
;  DATE:          29.2.88
;************************
10   20   30   40          ;HEIGHT
100  200  300  400         ;LENGTH
500  600  700  800
12.15     21.2  1.5         ;R_POS
−160.7    −90   0
```

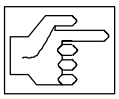
18. 6. READ_BEGIN selection of a certain line

Syntax:

READ_BEGIN File name[**,**Line number]

The “invisible READ pointer” jumps to before the start of a desired line as a result of the BAPS instruction READ_BEGIN.

The next READ instruction thus results in the first value of this desired line being read and assigned to a certain variable.



The BAPS instruction READ_BEGIN is a positioning instruction for the “invisible READ pointer”.
It does **not** result in reading of a value.

Example

READ_BEGIN VALUES, 7

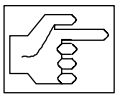
The ”invisible pointer” jumps in the file VALUES.DAT to before line 7.

READ_BEGIN VALUES, (V+N)

The ”invisible pointer” jumps in the file VALUES.DAT to a certain line, which is yielded by the expression V+N.

READ VALUES, NUMBER

The first value of the line (V+N) is read in for the program variable NUMBER.



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.

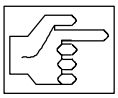
18. 7. The BAPS standard function END_OF_FILE

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”.



The DAT file name must be placed in brackets.

Example

```
IF END_OF_FILE (VALUES)
THEN JUMP FINISHED
```

As soon as the last value in the file V ALUES.DAT has been read, the control jumps in the main program to the jump label FINISHED.

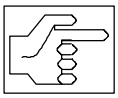
18. 8. The BAPS instruction WRITE

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.



A file opened for writing can be read again only after a CLOSE instruction.

Example

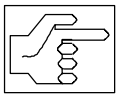
```
FILE: VALUES, ERG
INTEGER: W
WRITE VALUES, 700
```

The value 700 is written in the file V ALUES.DAT

```
WRITE ERG, V, W−10
```

The value which is possessed by the variable V at the time of the WRITE instruction is written in the file ERG.DAT along with the value yielded by the expression W−10.

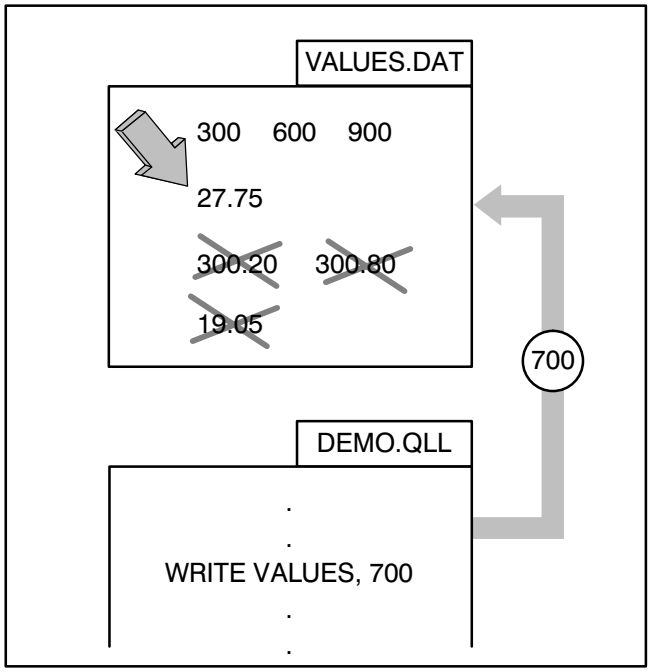
The WRITE instruction writes the desired value in the DAT file in the line which follows the current position of the invisible WRITE pointer.



The file is overwritten as from this line! The previous content of this and all 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 instruction must be programmed once before the first WRITE instruction.



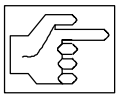
18. 9. WRITE_BEGIN Selection of a certain line

Syntax :

WRITE_BEGIN File name[**,**Line number]

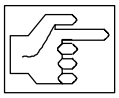
The instruction WRITE_BEGIN results in a jump of the invisible WRITE pointer to the start of a certain line and the file is opened for writing with this instruction.

As a result of this, the next WRITE instruction writes values from the BAPS program in the desired line of the DAT file. The previous content of this line **and all following ones** is deleted.



The DAT 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 as 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.

Example

WRITE_BEGIN VALUES ,(I+R)

The invisible WRITE pointer jumps in the file VALUES.DAT to 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 in the line (I+R) of the file VALUES.DAT

18. 10. The BAPS instruction WRITE_END

Syntax:

WRITE_END File name

The BAPS instruction WRITE_END results in a jump by the invisible WRITE pointer to the end of the DAT file.

This excludes the possibility of a DAT line being over-written for 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 already 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 end of the DAT file

18. 11. The BAPS instruction CLOSE

Syntax:

CLOSE File name

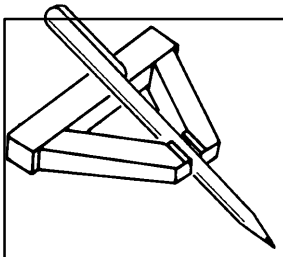
An open file is closed with the CLOSE instruction:

CLOSE File name

Example

CLOSE values

The file VALUES is closed.



19. Compound statements

Syntax:

BEGIN { Statement } END

Several statements can be combined by means of a compound statement.

```
BEGIN
    Statement
    Statement
END
```

A statement string may be included instead of one statement.

```
INPUT : 1 = Valve_1,
        2 = Valve_2


IF ready THEN
   BEGIN
    Valve_1=1

   END
ELSE
  BEGIN
   Valve_2 = 1

   END


PROGRAM_END
```

20. Parallel processes

The control rho 3 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).

20. 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
.
```

20. 1. 1. Starting and stopping external processes

External processes can be started either by

- selection on the PHG3
- program selection via the file EXPROG.DAT and corresponding interface signals
- the BAPS2 statement **START** from a running BAPS2 process.

The **START** statement can be extended by priority information:

```
START temp_control PRIO = 100
```

START process name PRIO = number

The process name is the name of the BAPS2 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.

```
STOP process name
```

The process name is the name of the BAPS2 IRD file and must be declared with **EXTERNAL**.

```
STOP temp
```

20. 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:

```
PARALLEL
        Program statements;
ALSO
        Process statements
ALSO
        Process statements
            .
            .
            .
PARALLEL_END
```

```
            .
            .
            .
PARALLEL
    MOVE sr800 TO corner
ALSO
    MOVE_REL feeder EXACT (0,100)
PARALLEL_END
            .
            .
            .
```

20. 3. Semaphores

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

```
EXCLUSIVE Semaphore name
        Statement
EXCLUSIVE_END
```

```
EXCLUSIVE SEMA_V24_1
        WRITE V24_1, n
EXCLUSIVE_END
```

The semaphore names are declared with the semaphore statement:

```
SEMAPHORE : Semaphore name
```

```
SEMAPHORE: Sema_V24_1
```

21. Compiler statements

BAPS2 contains compiler statements which are used to control the compiler and reduce the writing work. The compiler statement always starts with two successive semicolons.

The following compiler statements exist:

;; INCLUDE Name

;; PROCESS_KIND = PERMANENT

;; [Kinematic_Name **.**] **INT = CIRCULAR** |
PTP | LINEAR

;; CONTROL = RHO3 | IQ140

;;KINEMATICS: ({ INTEGER_Constant =
Kinematic_Variable | **,** } **)**

;; KINEMATICS = Kinematic_Variable

;; Kinematic_Variable **.(JC_NAMES** |
WC_NAMES") = { Name || **,** }

The compiler statement for the axis (JC) names and for the destination control must be located before the first source symbol, i.e. before the program declaration.

21. 1. Kinematic definition

The control can control several kinematics simultaneously.

If more than one kinematic (robot, feeder units etc.) is to be controlled with a BAPS program, these must first be defined once:

```
;; KINEMATICS: (Kinematic NUMBER= Kinematic name)
```

```
;;KINEMATICS: (1=sr800, 2=kin2)
```

It is now possible to distinguish between the kinematics in the program and it is clear to which kinematics the respective instructions are to refer.

The kinematic definition must take place after a control definition (if present) and before the PROGRAM statement.

```
;; CONTROL=RHO3
;;KINEMATICS: (1=sr450, 2=feeder)
PROGRAM main
        .
        .
        .
```

21. 2. Coordinate (WC) name definition

The world coordinate points contain the components for the position and orientation. The component names can be defined:

```
;; WC_NAMES = WC name, ...
```

```
;; WC_NAMES = X_C,Y_C,Z_C,U_C,V_C
```

If several coordinates are controlled by a BAPS program, the kinematic name must be specified first:

```
;;kinematic_name.WC_NAMES = WC_NAMES, ...
```

Example of coordinate declarations :

```
;;automat.WC_NAMES =      X_C,Y_C,Z_C,U_C,V_C

;;sr800.WC_NAMES =      X_C,Y_C,Z_C,A_C
```

The coordinate declaration must be contained in one line and must not be interrupted by a line end.

21. 3. Axis (JC) name definition

The joint coordinate points contain the components for the individual axes. The axis (JC) names can be defined:

;;JC_NAMES = JC name, ...

;; JC_NAMES = A_1,A_2,A_3,A_4

The kinematic name must be specified first if several coordinates are controlled by a BAPS program:

;;kinematic_name.JC_NAMES = JC name, ..

Example of JC name declarations :

;;automat.JC_NAMES = A_1, A_2, A_3, A_4, A_5, A_6, BND

;;sr800.JC_NAMES = A_1, A_2, A_3, A_4

The JC name declaration must be contained in one line and must not be interrupted by a line end.

21. 4. Kinematic−related statements and data

If several kinematics are controlled by a BAPS program, it is necessary to make a distinction in the program as regards the kinematic to which statements or data refer. This is the case for:

- Point variables
- Movement instructions
- Tool statements
- Working area limits

The kinematic name precedes the point variable:

Kinematic name.POINT

or

Kinematic name.JC_POINT

```
sr800.POINT: corner
kin2.JC_POINT: @fetchpos
```

The currently preselected kinematic is assigned if no kinematic information is provided.

The movement instructions may additionally contain the kinematic information or the preselected kinematic is controlled (see Chapter ”Movement instructions”).

The kinematic name must be specified immediately after the REF_PNT key value in the reference point statement.

REF_PNT Kinematic name (axis number)

```
REF_PNT sr800 (1,2,3,4)
TOOL automat gripper
LIMIT_OFF sr800
```

The same applies analogously to the **TOOL** and **LIMIT_OFF**,**LIMIT_MIN** and **LIMIT_MAX** statements:

TOOL Kinematic name Tool name

LIMIT+OFF Kinematic name

LIMIT+MIN Kinematic name (parameter)

LIMIT+MAX Kinematic name (parameter)

21. 5. Inclusion of files

The compiler statement

;; INCLUDE file name

allows parts of source programs to be included in the program; see adjacent example.

The file BAPS.QLL contains the declarations of your inputs, for example. These are defined with respect to

- data type
- channel number
- variable name of the signal

Declarations of your outputs. These are defined with respect to

- data type
- channel number
- variable name of the signal

Example:

```
    .
    .
;; INCLUDE BAPS
    .
    .

;BAPS.QLL

;FILE FOR AUTOMATIC INCLUSION OF

;THE DECLARATION PART IN COMPILATION


;DECLARATION INPUTS:

INPUT REAL:        1=GRIPPER_CONTACT,
                   4=MEAS_HEIGHT

INPUT:             11=GATE_SWITCH1,
                   15=LI_BARRIER


;DECLARATION OUTPUTS

OUTPUT:            7=ALARM

OUTPUT REAL:       1=PRESSURE,
                   2=METER_UNIT
```

21. 6. Process kind

A process can be declared as permanent by means of the compiler statement

```
;;PROCESS_KIND = PERMANENT
```

This means that this process cannot be ended with ”Reset” or Automatic/Manual switch−over.

Permanent processes must **not** contain any movement instructions.

The compiler statement must precede the **PROGRAM**−statement.

```
;;PROCESS_KIND = PERMANENT
PROGRAM temp_control
        .
        .
        .
PROGRAM_END
```

21. 7. Test information

The compiler statement

```
;;TESTINFO –
```

permits generation of test information for the IRDATA code to be switched off for convenient test operation.

This is generally expedient only for completely tested application programs.

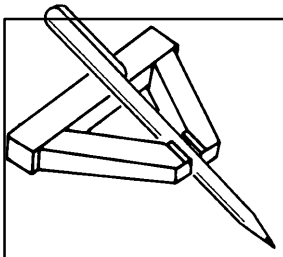
Since information is then missing from the IRDATA code, the IRDATA code is shorter and is executed more quickly. However, a test with the test system is then no longer possible.

Generation of test information is switched back on again with

```
;;TESTINFO +
```

The two test info statements can be used any number of times in the program. The program can then be tested in the corresponding sections.

```
.
.
;;TESTINFO –
.
.
;;TESTINFO +
.
.
.
```



22. Tool change

TOOL.DAT and language symbol TOOL

General

The option 'Tool change' allows the coordinate transformations of the individual kinematics to be adapted

- to the active tool (e.g. for bundle, turret gripper),
- to assembly inaccuracies.

The path control is always referred to the Tool Center Point (TCP) of the active tool (gripper).

Coordinate transformation

Coordinate transformation is divided up into several parts in order to permit work with different tools (grippers) in the world coordinate system:

(1) The first part takes into account the robot kinematic up to the flange defined by machine parameters. This is determined by the robot type (P306), number of axes (P302), arm lengths (P307) and coupling factors (P308). The flange **cannot** be changed from BAPS.

(2) The second part takes into account the kinematic of the tool from the flange to the Tool Center Point. There are 2 possibilities for tool definition, which can also be combined with each other if required:

(2a) One tool can be defined by means of machine parameter P309 (flange coordinates). This is active after the control run−up and cannot be changed from BAPS. The zero point of the flange coordinate system is located in the flange defined by (1).
3 translations

(FL_X,FL_Y,FL_Z)

and 3 rotations

(FL_O1,FL_O2,FL_O3) are entered as flange coordinates.

(2b) Several tools can be defined by means of the file TOOL.DAT. None of these tool coordinate systems is active after the control run−up. They can be activated from BAPS and can be changed on−line during operation.

For general description purposes, 3 translations

(G_X,G_Y,G_Z)

and 3 rotations

(G_O1,G_O2,G_O3) are used again.

The zero point of the tool coordinate systems is defined by (2a). If the flange coordinates (P309) are equal to zero

(i.e. FL_X=FL_Y=FL_Z=FL_O1=FL_O2=FL_O3=0),

this means that the zero point of the tool coordinate system agrees with the flange defined by (1).

Notes

The orientation of the flange or coordinate system and definition of orientations depend on the robot type. Refer to the corresponding transformation documentation for more details.

However, FL_Z and G_Z point in the action direction of the flange or tool for all kinematics.

The orientations are defined as follows for the standard tool:

FL_O1 (G_O1)

Rotation about the axis FL_Z (G_Z)

FL_O2 (G_O2)

Rotation about the resultant axis FL_Y' (G_Y')

FL_O3 (G_O3)

Rotation about the resultant axis FL_X" (G_X")

Which of the orientations

FL_O1, FL_O2, FL_O3, G_O1, G_O2, G_O3

are actually included in the transformation depends on the robot type and specifically on the number of axes and thus the number of degrees of freedom of the robot.

The numeric values act additively if flange coordinates (2a) and tool coordinates (2b) are used simultaneously.

22. 1. Format of the file TOOL.DAT

As mentioned above, all possible tool coordinate systems are stored in the file **TOOL.DAT**.

WERKZG is the reserved German name for the file to be produced by the user himself. In foreign languages, the reserved name must be taken from the corresponding text file (e.g. English: TOOL.DAT).

The individual tools are provided with a name which can be freely selected by the user. The corresponding coordinates are then stored under this name.

The file **TOOL.DAT** is edited as a whole as an ASCII file in the robot operating system or edited off−line. **One** tool name and **all** corresponding coordinate values are entered as follows for each line:

Syntax:

Tool name = G_X G_Y G_Z G_O1 G_O2 G_O3

whereby G_X , ... , G_O3 represent the corresponding numeric values.

The tool name must be located at the start of the line and may have a maximum of 12 characters. It can be freely selected.

The tool name and coordinate values must be separated by the '='.

The order of the individual coordinates: First G_X, then G_Y, ... finally G_O3, must be observed under all circumstances. The individual coordinate values must be separated by spaces (any number is possible). The values are decimal values, whereby the decimal point need not necessarily be entered. Only the inputs '0','1',...,'9','+','−','.' are permitted for the coordinate values.

Zeros must be entered explicitly for missing values. If fewer than 6 coordinates are specified in a line for a tool, the message

"format error in DAT"

is issued at the run time.

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

Blank lines are redundant.

The translations (these are the first three values) are entered in [mm] and the rotations (the last three values) in [degrees].

Example for file
TOOL.DAT:

```
;------------------------------------------------------------
;Tool name= G_X  G_Y  G_Z G_O1 G_O2 G_O3
;;----- IC grippers No. 10 and No. 11
IC_GRIPP_10 =  10  2.5   5   1   2   3
IC_GRIPP_11 = -20   0   120   5   0   6
;----- Bundle gripper No. 5
; front left
BUNDLE5_F_L = -50  -50  200   0   0   0
; front right
BUNDLE5_F_R = -50   50  200  0  0   0
; rear left
BUNDLE5_R_L =  50  -50  200  0  0   0
; rear right
BUNDLE5_R_R =  50   50  200   0   0   0
;----- Dummy gripper for switching off
OFF =   0   0   0   0   0   0
;------------------------------------------------------------
```

22. 2. Tool selection in the movement program

A specific tool is selected in the BAPS program with the instruction

TOOL kin1 grp_name

kin1 = Name of kinematic for which the tool is activated or default kinematic if no name is specified.

grp_name = Name of the tool (gripper) to be activated defined inTOOL.DAT.

EXAMPLE:

Let us assume that there are several bundle grippers of the type as shown in program example 4.1 in a magazine.

Let us assume here also that the individual grippers of bundle gripper No. 5 are defined in the file TOOL.DAT with the following names:

BUNDLE5_F_L,

BUNDLE5_F_R,

BUNDLE5_R_L,

BUNDLE5_R_R.

The rear left gripper of the bundle gripper No. 5 is then selected for the kinematic SCARA_1 by

TOOL SCARA_1 BUNDLE5_R_L

i.e. the values of the tool coordinate system BUNDLE5_R_L are included in the coordinate transformation of SCARA_1.

A tool change is possible only in automatic mode.

A gripper remains active until the next call of TOOL.

The last−programmed tool remains active after the program end.

In the event of program abort (e.g. emergency−stop input, auto−manual change−over, reset etc.), the tool active at the time of the abort remains active.

No tool is active after the control run−up, i.e.: G_X = ... = G_O3 = 0.

There is no direct instruction for switching off the gripper or tool.

However, the desired effect can be realized simply by programming a dummy gripper with the name 'OFF' in TOOL.DAT and defining the corresponding coordinates G_X= ... = G_O3 = 0 (see examples P. 4/6). The BAPS instruction "TOOL OFF" then has the desired effect.

In the case of Teach−in in the mode DEFINE/ TEACH−IN (Mode 4, Mode 2) or jog mode in Manual mode (Mode 2), it is possible to select a specific tool

by means of the function key program which can be realized via the PLC.

This may be done, for example, with the help of the following program:

PROGRAM BUN5FL

BEGIN

TOOL SCARA_1 BUNDLE5_F_L

PROGRAM_END

This instruction string results in internal conversion of the TCP of SCARA_1 and thus of the world coordinates. They now refer to the tip of the front left gripper of bundle gripper No. 5.

This is done without a traversing movement, since the joint coordinates have remained unchanged.

The following additional messages are output in the event of an error at the runtime:

”TOOL.DAT missing”

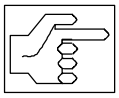
”name of tool missing”

The name and coordinates of the currently active tools (grippers) of the individual kinematics are displayed

under (Mode 7, Mode 1, ’Shift −>’),(Mode 2, ’Shift −>’)

and (Mode 4, Mode 2, ’Shift −>’).

C A U T I O N:



It must be ensured under all circumstances at the program start that the correct tool is activated. Unexpected movements may occur if programs are executed with the wrong tools. The same effect can occur if a WC point in the program is approached and taught−in with different tools.

P309 is zero in the following program examples, i.e. the origin of the tool coordinate system is located in the flange.

Program example:

```
0  TOOL SCARA_1 BUNDLE_F_L
1  MOVE  SCARA_1 LINEAR CORNER_LEFT
2  TOOL  SCARA_1 BUNDLE_F_R
3  MOVE  SCARA_1 LINEAR CORNER_RIGHT
4  TOOL SCARA_1 BUNDLE_R_L
5  MOVE  SCARA_1 LINEAR DEPOT
6  TOOL SCARA_1 OFF
```

Corresponding file TOOL.DAT

```
BUNDLE_F_L= −50 −50 200 0 0 0
BUNDLE_F_R= −50  50 200 0 0 0
BUNDLE_R_L=  50 −50 200 0 0 0
BUNDLE_R_R=  50  50 200 0 0 0
OFF       =   0   0   0   0 0 0
```

Effect of instruction string:

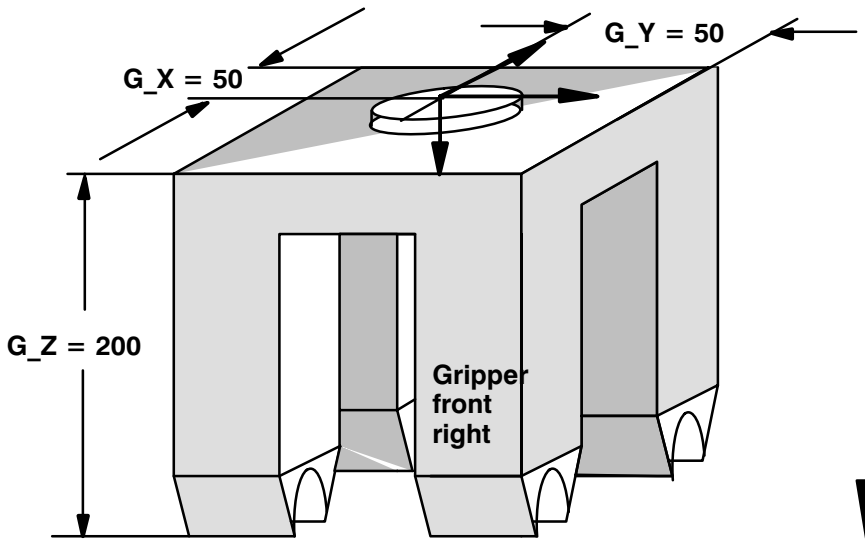
(1) The point CORNER_LEFT is approached by SCARA_1 with the left front gripper.

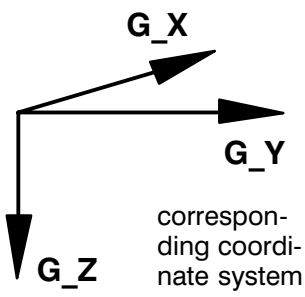
(3) The point CORNER_RIGHT is approached with the right front gripper.

(5) The point DEPOT is approached with the left rear gripper.

(6) The Tool Center Point is now located in the flange

Bundle gripper:





corresponding coordinate system

Program example:

(The action direction of the grippers is always vertically downwards. Gripper No. 2 is rotated downwards in an uncontrolled manner by digital outputs.)

```
(0) ;;KINEMATIC = PORTAL_2
(1) TOOL TURRET1_G1
(2) MOVE LINEAR START_POS
;Switch mechanical system so that gripper 2
;is pointing down (e.g. by setting digital
;output signals)

(3) TOOL TURRET1_G2
(4) MOVE LINEAR DEST_POS
(5) TOOL OFF.
```

Corresponding file TOOL.DAT

```
TURRET1_G1 = 0 0 100 0 0 0
TURRET1_G2 = 0 0  92 0 0 0
OFF        = 0 0   0 0 0 0
```

Effect of the instruction string:

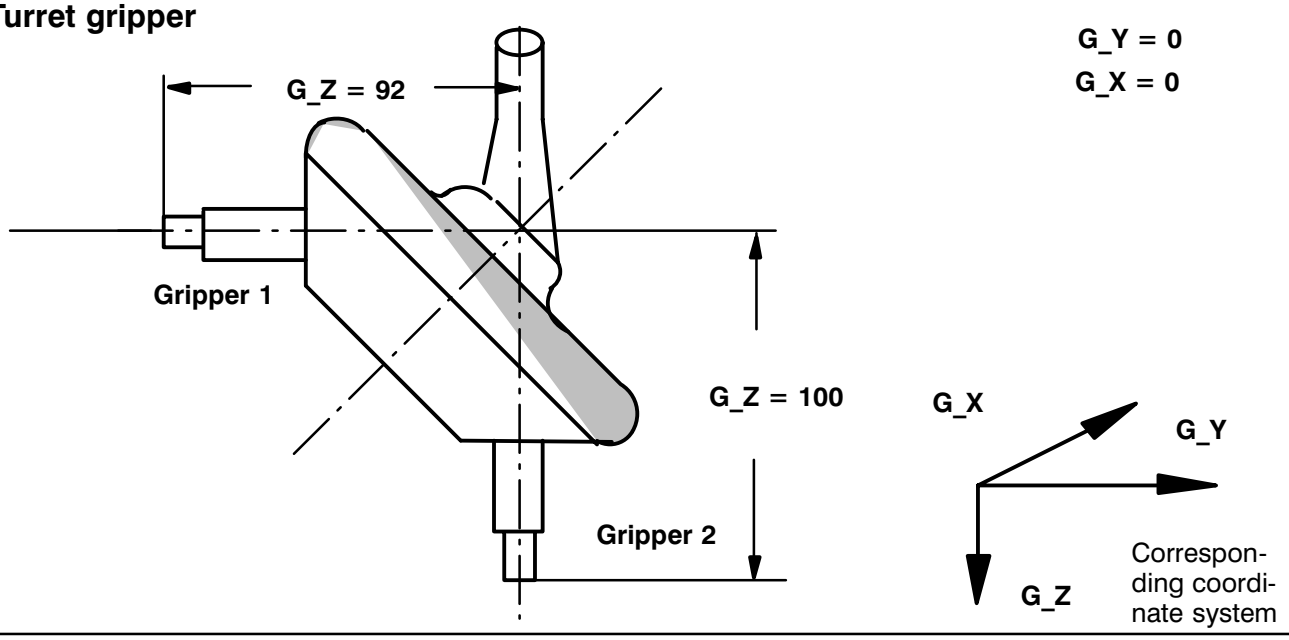
(2) The point START_POS is approached by PORTAL_2 (default kinematic) with gripper 1.

Mechanical system switch−over so that gripper 2 faces down (e.g. by setting digital output signals).

(4) The point DEST_POS is approached with gripper 2.

(5) The Tool Center Point of PORTAL_2 is now located in the flange.

Turret gripper



23. B A P S 2 – K E Y W O R D S

All language symbols which are currently reserved for BAPS2 are listed below. The listed language symbols must not be used as variable, file name or subroutine name in a BAPS2 program.

```
'@'                ;(* '@' *)
'SUBROUTINE'       ;(* 'UP' *)
'UNTIL'            ;(* 'BIS' *)
'REAL'             ;(* 'DEZ' *)
'TIMES'            ;(* 'MAL' *)
'WITH'             ;(* 'MIT' *)
'MOD'              ;(* 'MOD' *)
'PTP'              ;(* 'PTP' *)
'AND'              ;(* 'UND' *)
'REPEAT'           ;(* 'WDH' *)
'THEN'             ;(* 'DANN' *)
'END'              ;(* 'ENDE' *)
'ARRAY'            ;(* 'FELD' *)
'INTEGER'          ;(* 'GANZ' *)
'HALT'             ;(* 'HALT' *)
'READ'             ;(* 'LESE' *)
'TO'               ;(* 'NACH' *)
'OR'               ;(* 'ODER' *)
'SYNC'             ;(* 'SYNC' *)
'TEXT'             ;(* 'TEXT' *)
'IF'               ;(* 'WENN' *)
'VALUE'            ;(* 'WERT' *)
'APPROX'           ;(* 'CIRCA' *)
'FILE'             ;(* 'DATEI' *)
'EXACT'            ;(* 'EXAKT' *)
'MOVE'             ;(* 'FAHRE' *)
'CIRCULAR'         ;(* 'KREIS' *)
'NOT'              ;(* 'NICHT' *)
'PAUSE'            ;(* 'PAUSE' *)
'POINT'            ;(* 'PUNKT' *)
'ELSE'             ;(* 'SONST' *)
'VIA'              ;(* 'UEBER' *)
'WAIT'             ;(* 'WARTE' *)
'BINARY'           ;(* 'BINAER' *)
'EXTERNAL'         ;(* 'EXTERN' *)
'ERROR'            ;(* 'FEHLER' *)
'LINEAR'           ;(* 'LINEAR' *)
'JUMP'             ;(* 'SPRUNG' *)
'OUTPUT'           ;(* 'AUSGANG' *)
'INPUT'            ;(* 'EINGANG' *)
```

```
'REF_PNT'           ;(* 'REF_PKT' *)
'SPC_FCT'           ;(* 'SPZ_FKT' *)
'RETURN'            ;(* 'RSPRUNG' *)
'CHAR'              ;(* 'ZEICHEN' *)
'PROGRAM'           ;(* 'PROGRAMM' *)
'MAX_TIME'          ;(* 'MAX_ZEIT' *)
'JC_POINT'          ;(* 'MK_PUNKT' *)
'WRITE'             ;(* 'SCHREIBE' *)
'SYNCHRON'          ;(* 'SYNCHRON' *)
'REPEAT_END'        ;(* 'WDH_ENDE' *)
'TOOL'              ;(* 'WERKZEUG' *)
'LIMIT_OFF'         ;(* 'GRENZE_AUS' *)
'MOVE_REL'          ;(* 'VERSCHIEBE' *)
'READ_BEGIN'        ;(* 'LESE_ANFANG' *)
'WRITE_BEGIN'       ;(* 'SCHREIBE_ANF' *)
'WRITE_END'         ;(* 'SCHREIBE_END' *)
'SYNCHRON_END'      ;(* 'SYNCHRON_END' *)
'DEF'               ;(* 'DEF' *)
'PROGR_SLOPE'       ;(* 'PROGR_SLOPE' *)
'BLOCK_SLOPE'       ;(* 'SATZ_SLOPE' *)
'WC_FRAME'          ;(* 'RK_RAHMEN' *)
'BEGIN'             ;(* 'ANFANG' *)
'START'             ;(* 'START' *)
'PRIO'              ;(* 'PRIO' *)
'PROGRAM_END'       ;(* 'PROGRAMM_ENDE' *)
'SUB_END'           ;(* 'UP_ENDE' *)
'PERMANENT'         ;(* 'PERMANENT' *)
'ALSO'              ;(* 'SOWIE' *)
'CLOSE'             ;(* 'SCHLIESSE' *)
'PARALLEL'          ;(* 'PARALLEL' *)
'PARALLEL_END'      ;(* 'PARALLEL_END' *)
'EXCLUSIVE'         ;(* 'EXKLUSIV' *)
'EXCLUSIVE_END'     ;(* 'EXKLUSIV_END' *)
'STOP'              ;(* 'STOP' *)
'SEMAPHORE'         ;(* 'SEMAPHOR' *)
'BELT'              ;(* 'BAND' *)
'RPT_END'           ;(* 'WDH_ENDE' *)
'RPT'               ;(* 'WDH' *)
```

23. 1. B A P S – COMPILER STATEMENTS

'WARNING'	;(* 'WARNUNG' *)
'INT'	;(* 'INT' *)
'INCLUDE'	;(* 'EINFUEGE' *)
'JC_NAMES'	;(* 'ACHSNAMEN' *)
'WC_NAMES'	;(* 'KOORDINATEN' *)
'CONTROL'	;(* 'STEUERUNG' *)
'TOOL_COORD'	;(* 'WERK_KOORD' *)
'KINEMATICS'	;(* 'KINEMATIK' *)
	;(* 'ANTRIEBSART' *)
'PROCESS_KIND'	;(* 'PROZESS_ART' *)
'DEBUGINFO'	;(* 'TESTINFO' *)

23. 2. B A P S – STANDARD VARIABLES

```
'V'                 ;(* 'V' *)
'VFIX'              ;(* 'VFEST' *)
'T'                 ;(* 'T' *)
'TFIX'              ;(* 'TFEST' *)
'A'                 ;(* 'A' *)
'AFIX'              ;(* 'AFEST' *)
'V_PTP'             ;(* 'V_PTP' *)
'VFIX_PTP'          ;(* 'VFEST_PTP' *)
'VFACTOR'           ;(* 'VFAKTOR' *)
'AFACTOR'           ;(* 'AFAKTOR' *)
'LIMIT_MIN'         ;(* 'GRENZE_MIN' *)
'LIMIT_MAX'         ;(* 'GRENZE_MAX' *)
'TTY'               ;(* 'TTY' *)
'MCP'               ;(* 'HBG' *)
'V24_1'             ;(* 'V24_1' *)
'V24_2'             ;(* 'V24_2' *)
'PHG'               ;(* 'PHG' *)
'POS'               ;(* 'IPOS' *)
'@POS'              ;(* '@IPOS' *)
'V24_3'             ;(* 'V24_3' *)
'V24_4'             ;(* 'V24_4' *)
'WC_SYSTEM'         ;(* 'RK_SYSTEM' *)
'DFACTOR'           ;(* 'DFAKTOR' *)
'@MPOS'             ;(* '@MPOS' *)
```

23. 3. B A P S – STANDARD FUNCTIONS

```
'WC'                    ;(* 'RK' *)
'JC'                    ;(* 'MK' *)
'SIN'                   ;(* 'SIN' *)
'COS'                   ;(* 'COS' *)
'ATAN'                  ;(* 'ATAN' *)
'SQRT'                  ;(* 'WURZEL' *)
'END_OF_FILE' ;(* 'DATEI_ENDE' *)
'ABS'                   ;(* 'ABS' *)
'ROUND'                 ;(* 'RUNDUNG' *)
'TRUNC'                 ;(* 'GANZTEIL' *)
'WC_COMPUTATION'        ;(* 'RK_RECHNUNG' *)
'ORD'                   ;(* 'ORD' *)
'CHR'                   ;(* 'CHR' *)
'INT_ASC'               ;(* 'GANZ_ZFELD' *)
'ASC_INT'               ;(* 'ZFELD_GANZ' *)
```

23. 4. B A P S – STANDARD CONSTANTS

```
'WC_UR'                    ;(* 'RK_UR' *)
```

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

V

W

Bosch-Automationstechnik

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Industriehydraulik
Postfach 30 02 40
D-70442 Stuttgart
Telefax (07 11) 8 11-18 57

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Fahrzeughydraulik
Postfach 30 02 40
D-70442 Stuttgart
Telefax (07 11) 8 11-17 98

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Pneumatik
Postfach 30 02 40
D-70442 Stuttgart
Telefax (07 11) 8 11-89 17

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Montagetechnik
Postfach 30 02 07
D-70442 Stuttgart
Telefax (07 11) 8 11-77 12

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Antriebs- und Steuerungstechnik
Postfach 11 62
D-64701 Erbach
Telefax (0 60 62) 78-4 28

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Schraub- und Einpreßsysteme
Postfach 11 61
D-71534 Murrhardt
Telefax (0 71 92) 22-1 81

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Entgrattechnik
Postfach 30 02 07
D-70442 Stuttgart
Telefax (07 11) 8 11-34 75

Ihr Ansprechpartner

BOSCH

Robert Bosch GmbH
Geschäftsbereich
Automationstechnik
Antriebs- und Steuerungstechnik
Postfach 11 62
D-64701 Erbach
Telefax (0 60 62) 78-4 28

1070 073 033-101 (92.06) GB · HB RC · AT/VSP · Printed in Germany