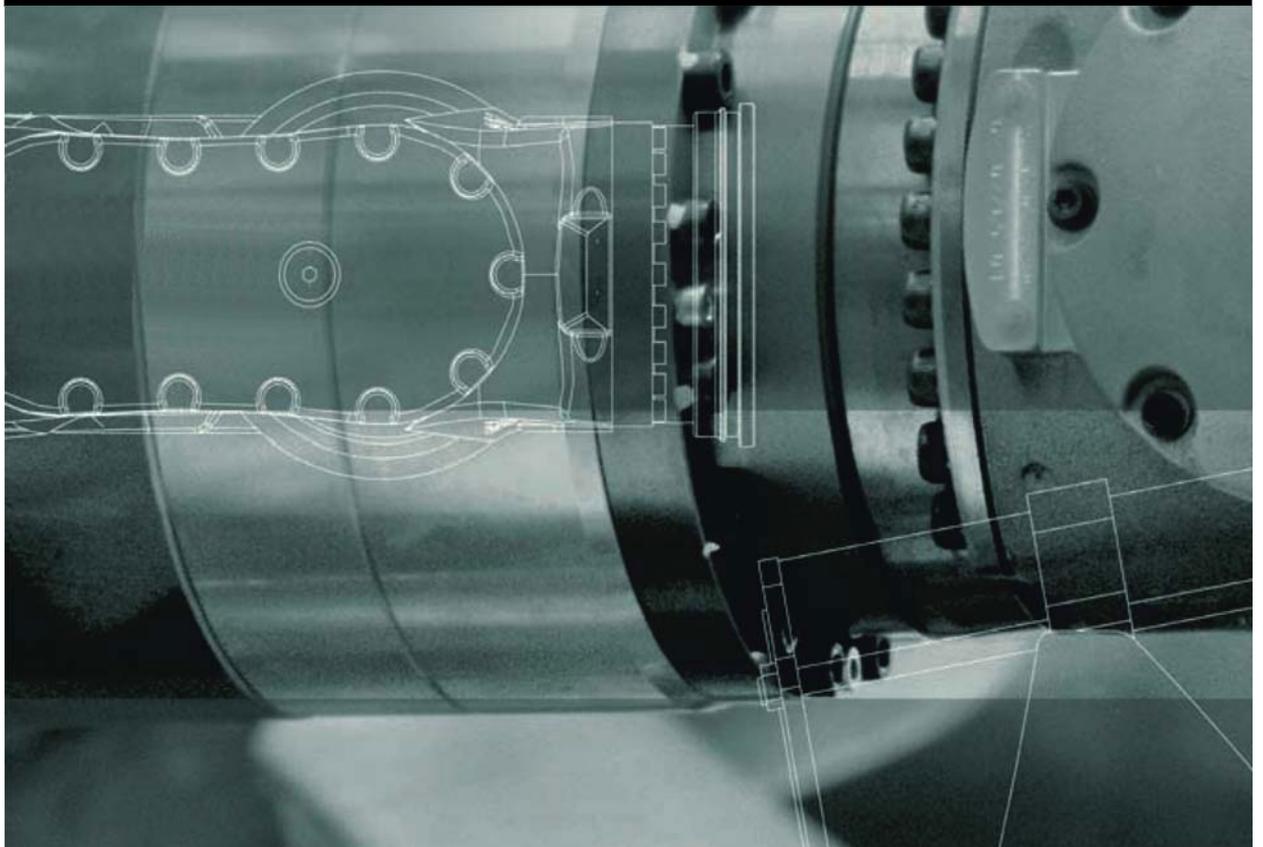


Robot Programming 2

KUKA System Software 8
Training Documentation



Issued: 14.12.2011

Version: P2KSS8 Roboterprogrammierung 2 V1 en



© Copyright 2011

KUKA Roboter GmbH
Zugspitzstraße 140
D-86165 Augsburg
Germany

This documentation or excerpts therefrom may not be reproduced or disclosed to third parties without the express permission of KUKA Roboter GmbH.

Other functions not described in this documentation may be operable in the controller. The user has no claims to these functions, however, in the case of a replacement or service work.

We have checked the content of this documentation for conformity with the hardware and software described. Nevertheless, discrepancies cannot be precluded, for which reason we are not able to guarantee total conformity. The information in this documentation is checked on a regular basis, however, and necessary corrections will be incorporated in the subsequent edition.

Subject to technical alterations without an effect on the function.

Translation of the original documentation

KIM-PS5-DOC

Publication:	Pub COLLEGE P2KSS8 Roboterprogrammierung 2 (PDF-COL) en
Bookstructure:	P2KSS8 Roboterprogrammierung 2 V2.3
Version:	P2KSS8 Roboterprogrammierung 2 V1 en

Contents

1	Structured programming	5
1.1	Objectives for consistent programming methodology	5
1.2	Tools for creating structured robot programs	5
1.3	Creating a program flowchart	8
2	Introduction to Expert level	11
2.1	Using Expert level	11
3	Variables and declarations	15
3.1	Data management in KRL	15
3.2	Working with simple data types	17
3.2.1	Declaration of variables	17
3.2.2	Initialization of variables with simple data types	20
3.2.3	Manipulation of variable values of simple data types with KRL	22
3.3	Arrays with KRL	25
3.4	Structures with KRL	28
3.5	The enumeration data type ENUM	30
4	Subprograms and functions	33
4.1	Working with local subprograms	33
4.2	Working with global subprograms	35
4.3	Transferring parameters to subprograms	37
4.4	Programming functions	40
4.5	Working with standard KUKA functions	42
5	Motion programming with KRL	45
5.1	Programming motions with KRL	45
5.2	Programming relative motions with KRL	50
5.3	Calculating or manipulating robot positions	52
5.4	Deliberate modification of Status and Turn bits	53
6	Working with system variables	59
6.1	Cycle time measurement by means of timers	59
7	Using program execution control functions	61
7.1	Programming conditional statements or branches	61
7.2	Programming a switch statement	62
7.3	Programming loops	65
7.3.1	Programming an endless loop	65
7.3.2	Programming a counting loop	67
7.3.3	Programming a rejecting loop	69
7.3.4	Programming a non-rejecting loop	70
7.4	Programming wait functions	72
7.4.1	Time-dependent wait function	72
7.4.2	Signal-dependent wait function	73
8	Switching functions with KRL	75
8.1	Programming simple switching functions	75
8.2	Programming path-related switching functions with TRIGGER WHEN DISTANCE	78

8.3	Programming path-related switching functions with TRIGGER WHEN PATH	81
9	Programming with WorkVisual	85
9.1	Managing a project with WorkVisual	85
9.1.1	Opening a project with WorkVisual	85
9.1.2	Comparing projects with WorkVisual	88
9.1.3	Transferring a project to the robot controller (installing)	92
9.1.4	Activating a project on the robot controller	96
9.2	Editing KRL programs with WorkVisual	98
9.2.1	File handling	98
9.2.2	Working with the KRL Editor	104
	Index	113

1 Structured programming

1.1 Objectives for consistent programming methodology

Objectives for consistent programming methodology

Consistent programming has the following advantages:

- The rigidly structured program layout allows complex problems to be dealt with more easily.
- It allows a comprehensible presentation of the underlying process (without the need for in-depth programming skills).
- Programs can be maintained, modified and expanded more effectively.

Forward-looking program planning has the following advantages:

- Complex tasks can be broken down into simple subtasks.
- The overall programming time is reduced.
- It enables components with the same performance to be exchanged.
- Components can be developed separately from one another.

The 6 requirements on robot programs:

1. Efficient
2. Free from errors
3. Easy to understand
4. Maintenance-friendly
5. Clearly structured
6. Economical

1.2 Tools for creating structured robot programs

What is the point of a comment?

- Comments about the contents or function of a program
- Contents and benefits can be freely selected.
- Improved program legibility
- Clearer structuring of a program
- The programmer is responsible for ensuring that comments are up to date.
- KUKA uses line comments.
- Comments are not registered as syntax by the controller.

Where and when are comments used?

Information about the entire source text:

```
DEF PICK_CUBE()
;This program fetches the cube from the magazine
;Author: I. M. Sample
;Date created: 09.08.2011
INI
...
END
```

Structure of the source text:

```

DEF PALLETIZE()
;*****
;*This program palletizes 16 cubes on the table*
;*Author: I. M. Sample-----*
;*Date created: 09.08.2011-----*
;*****
INI
...
;-----Calculation of positions-----
...
;-----Palletizing of the 16 cubes-----
...
;-----Depalletizing of the 16 cubes-----
...
END

```

Explanation of an individual line:

```

DEF PICK_CUBE()

INI

PTP HOME Vel=100% DEFAULT

PTP Pre_Pos ; Move to preliminary position for gripping

LIN Grip_Pos ; Move to cube gripping position
...

END

```

Indication of work to be carried out:

```

DEF PICK_CUBE()

INI

;Calculation of the pallet positions must be inserted here!

PTP HOME Vel=100% DEFAULT

PTP Pre_Pos ; Move to preliminary position for gripping

LIN Grip_Pos ; Move to cube gripping position

;Closing of the gripper is still missing here

END

```

Commenting out:

```

DEF Palletize()

INI

PICK_CUBE()

;CUBE_TO_TABLE()

CUBE_TO_MAGAZINE()

END

```

What is the effect of using folds in a robot program?

- Program sections can be hidden in FOLDS.
- The contents of FOLDS are not visible to the user.
- The contents of FOLDS are processed normally during program execution.
- The use of FOLDS can improve the legibility of a program.

What examples are there for the use of folds?

```

DEF Main()
...
INI                ; KUKA FOLD closed

SET_EA            ; FOLD created by user closed

PTP HOME Vel=100% DEFAULT ; KUKA FOLD closed

PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table
...
PTP HOME Vel=100% Default

END

```

```

DEF Main()
...
INI                ; KUKA FOLD closed

SET_EA            ; FOLD created by user opened
$OUT[12]=TRUE
$OUT[102]=FALSE
PART=0
Position=0

PTP HOME Vel=100% DEFAULT ; KUKA FOLD closed
...
PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table

PTP HOME Vel=100% Default

END

```

```

DEF Main()
...
INI                ; KUKA FOLD closed

SET_EA            ; FOLD created by user closed

PTP HOME Vel=100% DEFAULT ; KUKA FOLD opened
$BWDSTART=FALSE
PDAT_ACT=PDEFAULT
FDAT_ACT=FHOME
BAS (#PTP_PARAMS,100)
$H_POS=XHOME
PTP XHOME
...

PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table

PTP HOME Vel=100% Default

END

```

Why are subprograms used?

- Multiple use possible
- Avoidance of code repetition
- Memory savings
- Components can be developed separately from one another.
- Components with the same performance can be exchanged at any time.
- Structuring of the program
- Overall task broken down into subtasks
- Improved ease of maintenance and elimination of programming errors

Using sub-programs

```

DEF MAIN ()

INI

LOOP

    GET_PEN ()
    PAINT_PATH ()
    PEN_BACK ()
    GET_PLATE ()
    GLUE_PLATE ()
    PLATE_BACK ()

    IF $IN[1] THEN
        EXIT
    ENDIF

ENDLOOP

END
    
```

What is achieved by indenting command lines?

```

DEF INSERT ()
INT PART, COUNTER
INI
PTP HOME Vel=100% DEFAULT
LOOP
    FOR COUNTER = 1 TO 20
        PART = PART+1
        ;Inline forms cannot be indented!!!
PTP P1 CONT Vel=100% TOOL[2]:Gripper BASE[2]:Table
        PTP XP5
    ENDFOR
    ...
ENDLOOP
    
```

What is achieved by the meaningful identification of data names?

To enable correct interpretation of the function of data and signals in a robot program, it is advisable to use meaningful terms when assigning names. These include, for example:

- Long text names for input and output signals
- Tool and base names
- Signal declarations for input and output signals
- Point names

1.3 Creating a program flowchart

What is a program flowchart?

- Tool for structuring the sequence of a program
- Program sequence is made more legible.
- Structure errors are detected more easily.
- Simultaneous documentation of the program

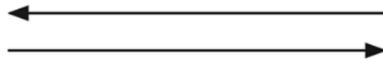
Program flowchart symbols

Start or end of a process or program

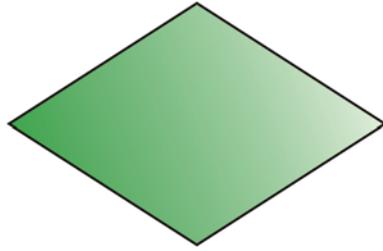


Fig. 1-1

Linking of statements and operations

**Fig. 1-2**

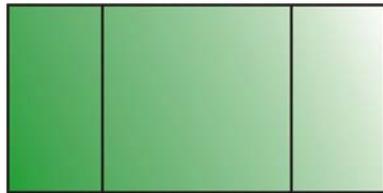
Branch

**Fig. 1-3**

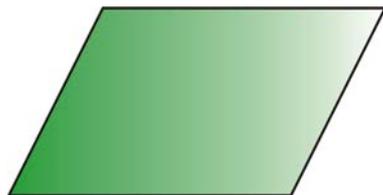
General statements in the program code

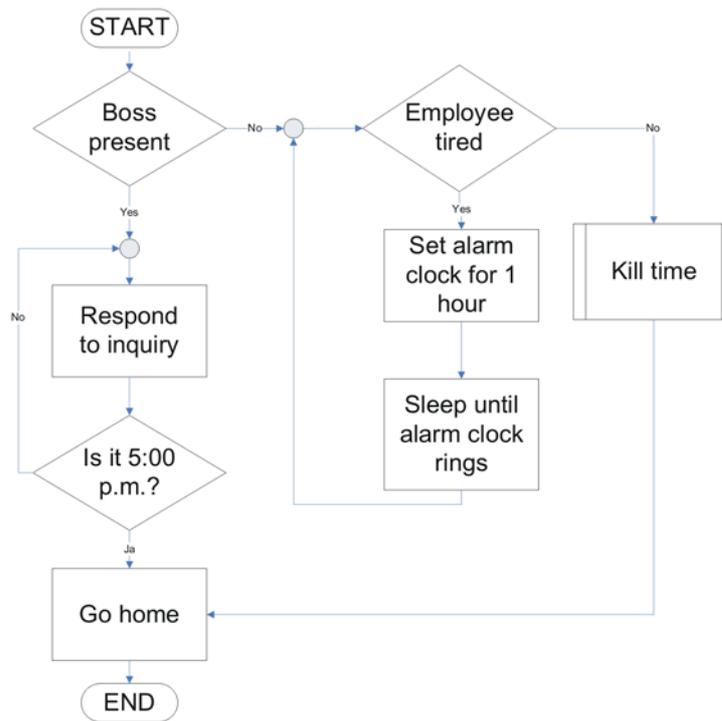
**Fig. 1-4**

Subprogram call

**Fig. 1-5**

Input/output statement

**Fig. 1-6**

**Program
flowchart
example****Fig. 1-7****Creating a
program
flowchart**

1. Rough outline of the overall sequence on approx. 1-2 pages
2. Breakdown of the overall task into small subtasks
3. Rough outline of the subtasks
4. Refinement of the structure of the subtasks
5. Implementation in KRL code

2 Introduction to Expert level

2.1 Using Expert level

Description	<p>The robot controller offers various user groups with different functions. The following user groups can be selected:</p> <ul style="list-style-type: none"> ■ Operator User group for the operator. This is the default user group. ■ User User group for the operator. (By default, the user groups “Operator” and “User” are defined for the same target group.) ■ Expert User group for the programmer. This user group is protected by means of a password. ■ Administrator The range of functions is the same as that for the user group “Expert”. It is additionally possible, in this user group, to integrate plug-ins into the robot controller. This user group is protected by means of a password. ■ Safety Recovery This user group can activate and configure the safety configuration of the robot. This user group is protected by means of a password. ■ Safety Maintenance This user group is only relevant if KUKA.SafeOperation or KUKA.SafeRangeMonitoring is used. The user group is protected by means of a password.
--------------------	---

Advanced functions of the “Expert” user group:

- Password-protected (default: kuka)
- Programming in editor possible using KRL
- Detail view available for modules
- Hide/show DEF line
- Open and close FOLDS
- Display detail view in program
- Predefined templates can be selected for program creation.
- The “Expert” user group is left again automatically:
 - if the operating mode is switched to AUT or AUT EXT.
 - if no action is carried out on the user interface within a specific time (300 s).

Functions	<p>Creating programs with templates</p> <ul style="list-style-type: none"> ■ Cell: Existing Cell program can only be replaced or, if Cell is deleted, recreated. ■ Expert: Module consisting of SRC and DAT file with just the program header and the end of the program. ■ Expert Submit: Additional Submit file (SUB) consisting of the program header and the end of the program. ■ Function: Creation of SRC function consisting of just the function header with a BOOL variable. The end of the function is present, but the return still has to be programmed. ■ Module: Module consisting of SRC and DAT file with the program header, the end of the program and the basic framework (INI and 2x PTP HOME).
------------------	--

- **Submit:** Additional Submit file (SUB) consisting of the program header, the end of the program and the basic framework (DECLARATION, INI, LOOP/ENDLOOP).

The **filter** defines how programs are displayed in the file list. The following **filters** are available:

- **Detail**
Programs are displayed as SRC and DAT files. (Default setting)
- **Modules**
Programs are displayed as modules.

Hide/show DEF line

- By default, the **DEF line** is hidden. Declarations can only be made in a program if the **DEF line** is visible.
- The **DEF line** is displayed and hidden separately for opened and selected programs. If detail view (ASCII mode) is activated, the **DEF line** is visible and does not need to be activated separately.

FOLD open/close

- The **FOLDS** are always closed for the user and can be opened by the Expert.
- The Expert can also program his own **FOLDS**.
- The **syntax** for a FOLD is:


```
;FOLD Name
Statements
;ENDFOLD <Name>
```

Procedure for activating Expert level and eliminating errors

Activating Expert level

1. Select **Configuration > User group** in the main menu.
2. Log on as **Expert**: Press **Login**. Select the user group **Expert** and confirm with **Login**.
3. Enter the password (default: kuka) when prompted and confirm with **Login**.

Eliminating errors in the program

1. Select faulty module in the Navigator.



Fig. 2-1: Program containing errors

2. Select the **Error list** menu.
3. The error display (*program_name.ERR*) is opened.
4. Select error; a detailed description is displayed at the bottom of the error display.
5. In the error display window, press the **Display** key and jump to the faulty program.

6. Eliminate the error.
7. Exit editor and save changes.

3 Variables and declarations

3.1 Data management in KRL

Working with variables with KRL

General information about variables

- In the context of robot programming with KRL, a variable, in the broadest sense, is simply a container for values that arise in the course of a robot process.
- A variable has a specific address assigned to it in the memory of the computer.
- A variable is identified by a name which is not a KUKA keyword.
- Every variable is linked to a specific data type.
- The data type must be declared before use.
- A distinction is made in KRL between local and global variables.

Variable life in KRL

- The variable life is the time in which the variable has reserved the memory location.
- When the program or function is exited, runtime variables re-enable their memory location.
- Variables from a data list receive the current value in their memory location permanently.

Validity of variables in KRL

- Variables declared as local are only available and visible in this program.
- Global variables are created in a central (global) data list.
- Global variables can also be created in a local data list and are assigned the keyword **global** during declaration.

Data types with KRL

- A data type is a grouping together of objects to form a set.
- Predefined standard data types
- User-defined standard data types
- Predefined KUKA data types

Using variables with KRL

Naming convention

- Names in KRL can have a maximum length of 24 characters.
- Names in KRL can consist of letters (A-Z), numbers (0-9) and the signs “_” and “\$”.
- Names in KRL must not begin with a number.
- Names in KRL must not be keywords.
- No distinction is made between uppercase and lowercase letters.

Data types with KRL

- **Predefined standard data types**

Simple data types	Integer	Floating-point number	Logic values	Individual character
Keyword	INT	REAL	BOOL	CHAR

Simple data types	Integer	Floating-point number	Logic values	Individual character
Range of values	$-2^{31} \dots (2^{31}-1)$	$\pm 1.1 \cdot 10^{-38} \dots \pm 3.4 \cdot 10^{+38}$	TRUE / FALSE	ASCII character set
Examples	-199 or 56	-0.0000123 or 3.1415	TRUE or FALSE	"A" or "q" or "7"

■ Array

```
Voltage[10] = 12.75
Voltage[11] = 15.59
```

- Save multiple variables of the same data type by means of an index.
- Initialization or modification of values is carried out by means of an index.
- The maximum array size depends on the amount of memory required by the data type.

■ Enumeration data type

```
color = #red
```

- All values of the enumeration type are defined with a name (in plain text) when created.
- The system also defines a sequence.
- The maximum number of elements depends on the available memory.

■ Composite data type / structure

```
Date = {day 14, month 12, year 1996}
```

- Composite data type consisting of components of various data types
- The components can consist of simple data types or of structures.
- It is possible to access individual components.

Variable life / validity

■ Variables created in the SRC file are called runtime variables and

- cannot always be displayed.
- are only valid in the declared program section.
- re-enable their memory location on reaching the last line of the program (END line).

■ Variables in the local DAT file

- can always be displayed during program execution of the corresponding SRC file.
- are available in the entire SRC file, i.e. also in local subprograms.
- can also be created as global variables.
- receive the current value in the DAT file and start with the saved value when called again.

■ Variables in the system file \$CONFIG.DAT

- are available in all programs (global).
- can always be displayed, even if no program is active.
- receive the current value in \$CONFIG.DAT.

Double declaration of variables

- A double declaration always occurs if the same character string (name) is used.
- There is **no** double declaration if the same name is used in different SRC or DAT files.

- Double declarations in the same SRC and DAT file are not permissible and generate an error message.
- Double declarations in the SRC or DAT file and \$CONFIG.DAT are permissible.
 - During execution of the program routine in which the variable was declared, only the local value is modified, not the value in \$CONFIG.DAT.
 - During execution of an “external” program routine, only the value from \$CONFIG.DAT is accessed and modified.

KUKA system data

- KUKA system data can be of the following types:
 - Enumeration data type, e.g. operating mode (mode_op)
 - Structure, e.g. date/time (date)
- System information is obtained from KUKA system variables.
 - Read the current system information.
 - Modify current system configurations.
 - They are predefined and begin with the “\$” sign.
 - \$DATE (current date and time)
 - \$POS_ACT (current robot position)
 - ...

3.2 Working with simple data types

The creation, initialization and modification of variables is explained below. Only simple data types are used here.

Simple data types with KRL

- Integers (INT)
- Floating-point numbers (REAL)
- Logic values (BOOL)
- Individual character (CHAR)

3.2.1 Declaration of variables

Creating variables

Declaration of variables

- The variables must always be declared before use.
- Every variable must be assigned to a data type.
- The naming convention must be observed when assigning names.
- The keyword for the declaration is **DECL**.
- The keyword DECL can be omitted in the case of the four simple data types.
- Value assignments are carried out with the advance run pointer
- The variable declaration can be carried out in different ways; these determine the variable life and validity of the variable.
 - Declaration in the SRC file
 - Declaration in the local DAT file
 - Declaration in \$CONFIG.DAT
 - Declaration in the local DAT file with the keyword “global”
- Creation of constants
 - Constants are created using the keyword **CONST**.
 - Constants may only be created in data lists.

Principle of variable declaration

Program structure in the SRC file

- Variables must be declared in the declaration section.
- The initialization section begins with the first value assignment, which is usually the “INI” line, however.
- Values are assigned or modified in the instruction section.

```
DEF main( )
; Declaration section
...
; Initialization section
INI
...
; Instruction section
PTP HOME Vel=100% DEFAULT
...
END
```

Changing the standard view

- Displaying the DEF line is only possible in the “Expert” user group.
- Necessary in the case of modules for accessing the declaration section before the “INI” line.
- In order to be able to see the DEF and END line, but also important for variable transfer in subprograms.

Planning variable declaration

- Defining the variable life
 - **SRC** file: runtime variable “dies” at the end of the program routine.
 - **DAT** file: variable is retained on completion of program execution.
- Defining validity/availability
 - locally in the **SRC** file: only available in the program routine in which it was declared. The variable is thus only available between the local DEF and END line (main program **or** local subprogram).
 - locally in the **DAT** file: valid in the entire program, i.e. also in all local subprograms.
 - **\$CONFIG.DAT**: globally available, i.e. read/write access is possible in all program routines.
 - locally in the **DAT** file as a global variable: globally available; read/write access is possible in all program routines as soon as the DAT file is assigned the keyword **PUBLIC** and additionally the keyword **GLOBAL** in the declaration.
- Defining the data type
 - **BOOL**: classic “YES”/“NO” results.
 - **REAL**: results of calculations to avoid rounding errors.
 - **INT**: classic counting variables for counting loops or part counters.
 - **CHAR**: one character only
A string or text can only be implemented as a CHAR array.
- Name assignment and declaration
 - Use **DECL** to ensure simpler legibility of the program.
 - Use meaningful, self-explanatory variable names.
 - Do not use cryptic names or abbreviations.
 - Use sensible name lengths, i.e. do not always use 24 characters.

Procedure for the declaration of a variable with a simple data type

Creating a variable in the SRC file

1. “Expert” user group
2. Display the DEF line.
3. Open the SRC file in the editor.

4. Carry out declaration of the variable.

```

DEF MY_PROG ( )
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
INI
...
END

```

5. Close and save the program.

Creating a variable in the DAT file

1. "Expert" user group
2. Open the DAT file in the editor.
3. Carry out declaration of the variable.

```

DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
...
ENDDAT

```

4. Close and save the data list.

Creating a variable in \$CONFIG.DAT

1. "Expert" user group
2. In the folder SYSTEM, open \$CONFIG.DAT in the editor.

```

DEFDAT $CONFIG
BASISTECH GLOBALS
AUTOEXT GLOBALS
USER GLOBALS
ENDDAT

```

3. Select the fold "USER GLOBALS" and open it with the softkey "Fold open/cls".
4. Carry out declaration of the variable.

```

DEFDAT $CONFIG ( )
...
;=====
; User-defined types
;=====
; User-defined externals
;=====
; User-defined variables
;=====
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
...
ENDDAT

```

5. Close and save the data list.

Creating a global variable in the DAT file

1. "Expert" user group
2. Open the DAT file in the editor.
3. Expand the program header in the data list to include the keyword PUBLIC.

```
DEFDAT MY_PROG PUBLIC
```

4. Carry out declaration of the variable.

```
DEFDAT MY_PROG PUBLIC
EXTERNAL DECLARATIONS
DECL GLOBAL INT counter
DECL GLOBAL REAL price
DECL GLOBAL BOOL error
DECL GLOBAL CHAR symbol
...
ENDDAT
```

5. Close and save the data list.

3.2.2 Initialization of variables with simple data types

Description of initialization with KRL

- After every declaration, a variable only has a memory location reserved; the value is always an invalid value.
- In the SRC file, the declaration and initialization are always carried out in two separate lines.
- In the DAT file, the declaration and initialization are always carried out in one line.
A constant must be initialized immediately in the declaration.
- The initialization section begins with the first value assignment.

Initialization principle

Initialization of integers

- Initialization as a decimal value

```
value = 58
```

- Initialization as a binary number

```
value = 'B111010'
```

Binary	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
Dec	32	16	8	4	2	1

Calculation: $1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 58$

- Hexadecimal initialization

```
value = 'H3A'
```

Hex	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dec	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Calculation: $3 \cdot 16 + 10 = 58$

Procedure for initialization with KRL

Declaration and initialization in the SRC file

1. Open the SRC file in the editor.
2. Declaration has been carried out.
3. Carry out initialization.

```

DEF MY_PROG ( )
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
INI
counter = 10
price = 0.0
error = FALSE
symbol = "X"
...
END

```

4. Close and save the program.

Declaration and initialization in the DAT file

1. Open the DAT file in the editor.
2. Declaration has been carried out.
3. Carry out initialization.

```

DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL INT counter = 10
DECL REAL price = 0.0
DECL BOOL error = FALSE
DECL CHAR symbol = "X"
...
ENDDAT

```

4. Close and save the data list.

Declaration in the DAT file and initialization in the SRC file

1. Open the DAT file in the editor.
2. Carry out the declaration.

```

DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL INT counter
DECL REAL price
DECL BOOL error
DECL CHAR symbol
...
ENDDAT

```

3. Close and save the data list.
4. Open the SRC file in the editor.
5. Carry out initialization.

```

DEF MY_PROG ( )
...
INI
counter = 10
price = 0.0
error = FALSE
symbol = "X"
...
END

```

6. Close and save the program.

Declaration and initialization of a constant

1. Open the DAT file in the editor.
2. Carry out declaration and initialization.

```

DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL CONST INT max_size = 99
DECL CONST REAL PI = 3.1415
...
ENDDAT

```

3. Close and save the data list.

3.2.3 Manipulation of variable values of simple data types with KRL

List of options for modifying variable values with KRL

Modification of the variable values in the program routines (SRC file) varies according to the specific task. The most commonly used methods are described below. Manipulation by means of bit operations and standard functions is also possible, but is not dealt with in detail here.

Data manipulation by means of:

- Basic arithmetic operations
 - **(+)** Addition
 - **(-)** Subtraction
 - **(*)** Multiplication
 - **(/)** Division
- Comparison operations
 - **(==)** identical / equal to
 - **(<>)** not equal to
 - **(>)** greater than
 - **(<)** less than
 - **(>=)** greater than or equal to
 - **(<=)** less than or equal to
- Logic operations
 - **(NOT)** Inversion
 - **(AND)** Logic AND
 - **(OR)** Logic OR
 - **(EXOR)** Exclusive OR
- Bit operations
 - **(B_NOT)** Bit-by-bit inversion
 - **(B_AND)** Bit-by-bit ANDing
 - **(B_OR)** Bit-by-bit ORing
 - **(B_EXOR)** Bit-by-bit exclusive ORing

Standard functions

- Absolute function
- Root function
- Sine and cosine function
- Tangent function
- Arc cosine function
- Arc tangent function
- Multiple functions for string manipulation

Data manipulation relationships

Value modification using the data types REAL and INT

- Rounding up/down

```

; Declaration
DECL INT A,B,C
DECL REAL R,S,T
; Initialization
A = 3      ; A=3
B = 5.5    ; B=6 (> x.5 is rounded up)
C = 2.25   ; C=2 (rounded down)
R = 4      ; R=4.0
S = 6.5    ; S=6.5
T = C      ; T=2.0 (the rounded-down value is taken)

```

■ Results of arithmetic operations (+;-;*)

Operands	INT	REAL
INT	INT	REAL
REAL	REAL	REAL

```

; Declaration
DECL INT D,E
DECL REAL U,V
; Initialization
D = 2
E = 5
U = 0.5
V = 10.6
; Instruction section (data manipulation)
D = D*E ; D = 2 * 5 = 10
E = E+V ; E= 5 + 10.6 = 15.6 -> rounded up to E=16
U = U*V ; U= 0.5 * 10.6 = 5.3
V = E+V ; V= 16 + 10.6 = 26.6

```

■ Results of arithmetic operations (/)

The following must be noted for arithmetic operations with integer values:

- In the case of interim results for operations with integers only, all decimal places are simply cut off.
- In the case of value assignments to an integer variable, the result is rounded up or down in the normal manner.

```

; Declaration
DECL INT F
DECL REAL W
; Initialization
F = 10
W = 10.0
; Instruction section (data manipulation)
; INT / INT -> INT
F = F/2 ; F=5
F = 10/4 ; F=2 (10/4 = 2.5 -> decimal place eliminated)
; REAL / INT -> REAL
F = W/4 ; F=3 (10.0/4=2.5 -> rounded up)
W = W/4 ; W=2.5

```

Comparison operations

Using relational operators, it is possible to form logical expressions. The result of a comparison is always of data type BOOL.

Operator/KRL	Description	Permissible data types
==	identical / equal to	INT, REAL, CHAR, BOOL
<>	not equal to	INT, REAL, CHAR, BOOL
>	greater than	INT, REAL, CHAR
<	less than	INT, REAL, CHAR

Operator/KRL	Description	Permissible data types
>=	greater than or equal to	INT, REAL, CHAR
<=	less than or equal to	INT, REAL, CHAR

```

; Declaration
DECL BOOL G,H
; Initialization / Instruction section
G = 10>10.1 ; G=FALSE
H = 10/3 == 3 ; H=TRUE
G = G<>H ; G=TRUE
    
```

Logic operations

Logic expressions can be formed using logic operations. The result of such an operation is always of data type BOOL.

Operations		NOT A	A AND B	A OR B	A EXOR B
A=TRUE	B=TRUE	FALSE	TRUE	TRUE	FALSE
A=TRUE	B=FALSE	FALSE	FALSE	TRUE	TRUE
A=FALSE	B=TRUE	TRUE	FALSE	TRUE	TRUE
A=FALSE	B=FALSE	TRUE	FALSE	FALSE	FALSE

```

; Declaration
DECL BOOL K,L,M
; Initialization / Instruction section
K = TRUE
L = NOT K ; L=FALSE
M = (K AND L) OR (K EXOR L) ; M=TRUE
L = NOT (NOT K) ; L=TRUE
    
```

Operators are executed in order of priority.

Priority	Operator
1	NOT (B_NOT)
2	Multiplication (*); division (/)
3	Addition (+), subtraction (-)
4	AND (B_AND)
5	EXOR (B_EXOR)
6	OR (B_OR)
7	Any comparison (==, <>; ...)

```

; Declaration
DECL BOOL X, Y
DECL INT Z
; Initialization / Instruction section
X = TRUE
Z = 4
Y = (4*Z+16 <> 32) AND X ; Y=FALSE

```

- Procedure for data manipulation**
1. Define the data type for the variable(s).
 2. Determine the validity and variable life of the variable.
 3. Carry out variable declaration.
 4. Initialize the variable.
 5. Manipulate the variable in the program routines, i.e. always in the SRC file.
 6. Close and save the SRC file.

3.3 Arrays with KRL

Description of arrays with KRL Arrays provide memory for multiple variables of the same data type, differentiated by means of an index.

- The memory for arrays is finite, i.e. the maximum array size depends on the amount of memory required by the data type.
- For declaration, the size of the array and the data type must be known.
- The start index in KRL always begins with 1.
- Initialization can always be carried out individually.
- Initialization in the SRC file can also be carried out using a loop.

Array dimensions

- 1-dimensional array

```
dimension1[4]= TRUE
```

- 2-dimensional array

```
dimension2[2,1]= 3.25
```

- 3-dimensional array

```
dimension1[3,4,1]= 21
```

- 4-dimensional array or higher **not** supported by KRL.

Relationships in the use of arrays The variable life and validity of array variables are the same as for variables of simple data types.

Array declaration

- Creation in the SRC file

```

DEF MY_PROG ( )
DECL BOOL error[10]
DECL REAL value[50,2]
DECL INT parts[10,10,10]
INI
...
END

```

- Creation in the data list (also \$CONFIG.DAT)

```

DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
DECL REAL value[50,2]
DECL INT parts[10,10,10]
...
ENDDAT

```

Declaring and initializing an array in the SRC file

- Call each array individually by means of the index.

```

DECL BOOL error[10]
error[1]=FALSE
error[2]=FALSE
error[3]=FALSE
error[3]=FALSE
error[4]=FALSE
error[5]=FALSE
error[6]=FALSE
error[7]=FALSE
error[8]=FALSE
error[9]=FALSE
error[10]=FALSE

```

- using suitable loops

```

DECL BOOL error[10]
DECL INT x
FOR x = 1 TO 10
error[x]=FALSE
ENDFOR

```



After execution of the loop, x has the value 11.

Initializing an array in the data list

- Call each array individually by means of the index and then display the value in the data list.

```

DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
error[1]=FALSE
error[2]=FALSE
error[3]=FALSE
error[4]=FALSE
error[5]=FALSE
error[6]=FALSE
error[7]=FALSE
error[8]=FALSE
error[9]=FALSE
error[10]=FALSE

```

- Impermissible declaration and initialization in the data list

```

DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
DECL INT size = 32
error[1]=FALSE
error[2]=FALSE
error[3]=FALSE
error[4]=FALSE
error[5]=FALSE
error[6]=FALSE
error[7]=FALSE
error[8]=FALSE
error[9]=FALSE
error[10]=FALSE

```



Generates ten “Initial value block not in initialization section” error messages.

Declaring an array in the data list and initializing it in the SRC file

- If an array is created in the data list in this way, the current values **cannot** be viewed in the data list; the current values can **only** be checked using the variable display.

```
DEFDAT MY_PROG
EXTERNAL DECLARATIONS
DECL BOOL error[10]
```

```
DEF MY_PROG ( )
INI
Fehler[1]=FALSE
Fehler[2]=FALSE
Fehler[3]=FALSE
...
Fehler[10]=FALSE
```

or

```
DEF MY_PROG ( )
INI
FOR x = 1 TO 10
Fehler[x]=FALSE
ENDFOR
```

Initialization by means of loops

- 1-dimensional array

```
DECL INT parts[15]
DECL INT x
FOR x = 1 TO 15
parts[x]= 4
ENDFOR
```

- 2-dimensional array

```
DECL INT parts_table[10,5]
DECL INT x, y
FOR x = 1 TO 10
FOR y = 1 TO 5
parts_table[x, y]= 6
ENDFOR
ENDFOR
```

- 3-dimensional array

```
DECL INT parts_palette[5,4,3]
DECL INT x, y, z
FOR x = 1 TO 5
FOR y = 1 TO 4
FOR z = 1 TO 3
parts_palette[x, y, z]= 12
ENDFOR
ENDFOR
ENDFOR
```

Procedure for using arrays

1. Define data types for the array
2. Determine the validity and variable life of the array.
3. Carry out the array declaration.
4. Initialize the array elements.
5. Manipulate the array in the program routines, i.e. always in the SRC file.

6. Close and save the SRC file.

```

DEF MY_PROG ( )
DECL REAL palette_size[10]
DECL INT counter
INI
; Initialization
FOR counter = 1 TO 10
  palette_size[counter] = counter * 1.5
ENDFOR
...
; Change value individually
palette_size[8] = 13
...
; Comparison of values
IF palette_size[3] > 4.2 THEN
...

```

3.4 Structures with KRL

Variables with several individual items of information

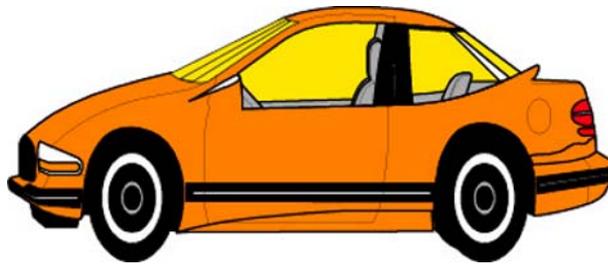


Fig. 3-1

Composite data type: **Structure**

- Arrays can be used to group together variables of the same data type. In the real world, however, variables mostly consist of different data types.
- A car, for example, has an engine power or mileage for which the type “Integer” is used. An obvious choice for the price is the type “Real”. For the presence of an air-conditioning system, on the other hand, the data type “Bool” is more appropriate.
- Together, they all describe a car.
- A structure can be defined with the keyword STRUC.
- A structure is a combination of different data types.

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
```

- A structure must first be defined; it can then be used.

Using a structure

Availability/definition of a structure

- The simple data types INT, REAL, BOOL and CHAR can be used in a structure.

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
```

- CHAR arrays can be integrated into a structure.

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition, CHAR
car_model[15]
```

- Known structures, such as a position POS, can also be used in a structure.

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition, POS
car_pos
```

- Following definition of the structure, a working variable must be declared for it.

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
DECL CAR_TYPE my_car
```

Initializing/modifying a structure

- The initialization can be carried out using brackets.
- In the case of initialization using brackets, only constants (fixed values) may be used.
- The value assignment order is not significant.

```
my_car = {motor 50, price 14999.95, air_condition = TRUE}
```

```
my_car = {price 14999.95, motor 50, air_condition = TRUE}
```

- Not all structure elements have to be specified in a structure.
- A structure is initialized with a structure element.
- Values that have not been initialized are set to "unknown".

```
my_car = {motor 75} ; Price not known
```

- The initialization can also be carried out using a point separator.

```
my_car.price = 9999.0
```

- In the case of initialization using a point separator, variables can also be used.

```
my_car.price = value_car
```

- Structure elements can be modified again at any time individually by means of a point separator.

```
my_car.price = 12000.0
```

Variable life / validity

- Structures created locally are invalid once the END line has been reached.
- Structures that are used in multiple programs must be declared in \$CONFIG.DAT.

Nomenclature

- Keywords may not be used.
- For greater ease of recognition, user-defined structures should end in **TYPE**.

KUKA works a lot with predefined structures that are stored in the system. Examples can be found for positions and in message programming.

Predefined KUKA structures for positions

- **AXIS**: STRUC AXIS REAL A1, A2, A3, A4, A5, A6
- **E6AXIS**: STRUC E6AXIS REAL A1, A2, A3, A4, A5, A6, E1, E2, E3, E4, E5, E6
- **FRAME**: STRUC FRAME REAL X, Y, Z, A, B, C
- **POS**: STRUC POS REAL X, Y, Z, A, B, C INT S,T
- **E6POS**: STRUC E6POS REAL X, Y, Z, A, B, C, E1, E2, E3, E4, E5, E6 INT S,T

Initialization of a structure with a position

- In the case of initialization using brackets, only constants (fixed values) may be used.

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition, POS
car_pos
DECL CAR_TYPE my_car
my_car = {price 14999.95, motor 50, air_condition = TRUE, car_pos {X
1000, Y 500, A 0}}
```

- The initialization can also be carried out using a point separator.

```
my_car.price = 14999.95
my_car.car_pos = {X 1000, Y 500, A 0}}
```

- In the case of initialization using a point separator, variables can also be used.

```
my_car.price = 14999.95
my_car.car_pos.X = x_value
my_car.car_pos.Y = 750
```

Creation of a structure

1. Definition of the structure

```
STRUC CAR_TYPE INT motor, REAL price, BOOL air_condition
```

2. Declaration of the working variable

```
DECL CAR_TYPE my_car
```

3. Initialization of the working variable

```
my_car = {motor 50, price 14999.95, air_condition = TRUE}
```

4. Modification of the values and/or value comparison of the working variable

```
my_car.price = 5000.0
```

```
my_car.price = value_car
```

```
IF my_car.price >= 20000.0 THEN
...
ENDIF
```

3.5 The enumeration data type ENUM

Plain text as variable value

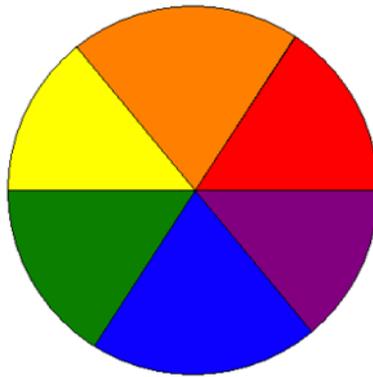


Fig. 3-2

- The enumeration data type consists of a limited number of constants, e.g. green, yellow and blue.

```
ENUM COLOR_TYPE green, blue, red, yellow
```

- The constants are freely selectable names.
- The constants are defined by the programmer.
- An enumeration type must first be defined; it can then be used.
- A working variable, such as crate color, of type COLOR_TYPE, can only ever take on the value of one constant.

- The value assignment of a constant is always carried out using the symbol #.

Using an enumeration data type

Availability/use

- Only known constants can be used.
- An enumeration type can be expanded as often as required.
- An enumeration type can be used on its own.

```
ENUM COLOR_TYPE green, blue, red, yellow
```

- An enumeration type can be integrated into a structure.

```
ENUM COLOR_TYPE green, blue, red, yellow
STRUC CAR_TYPE INT motor, REAL price, COLOR_TYPE car_color
```

Variable life / validity

- Enumeration data types created locally are invalid once the END line has been reached.
- Enumeration data types that are used in multiple programs must be declared in \$CONFIG.DAT.

Nomenclature

- Names of enumeration types and their constants should be self-explanatory.
- Keywords may not be used.
- For greater ease of recognition, user-defined enumeration types should end in **TYPE**.

Creation of an enumeration data type

1. Definition of the enumeration variables and constants

```
ENUM LAND_TYPE de, be, cn, fr, es, br, us, ch
```

2. Declaration of the working variable

```
DECL LAND_TYPE my_land
```

3. Initialization of the working variable

```
my_land = #be
```

4. Value comparison of the working variable

```
IF my_land == #es THEN
...
ENDIF
```


4 Subprograms and functions

4.1 Working with local subprograms

Definition of local subprograms

- Local subprograms are located after the main program and are identified by means of `DEF Name_subprogram()` and `END`

```

DEF MY_PROG( )
; This is the main program
...
END

-----

DEF LOCAL_PROG1( )
; This is local subprogram 1
...
END

-----

DEF LOCAL_PROG2( )
; This is local subprogram 2
...
END

-----

DEF LOCAL_PROG3( )
; This is local subprogram 3
...
END

```

- An SRC file can consist of up to 255 local subprograms.
- Local subprograms can be called repeatedly.
- Local program names require brackets.

Relationships when working with local subprograms

- Once a local subprogram has been executed, the program jumps back to the next command after the subprogram call.

```

DEF MY_PROG( )
; This is the main program
...
LOCAL_PROG1( )
...
END

-----

DEF LOCAL_PROG1( )
...
LOCAL_PROG2( )
...
END

-----

DEF LOCAL_PROG2( )
...
END

```

- The maximum nesting depth for subprograms is 20.
- Point coordinates are saved in the corresponding DAT list and are available for the entire file.

```

DEF MY_PROG( )
; This is the main program
...
PTP P1 Vel=100% PDAT1
...
END

-----

DEF LOCAL_PROG1( )
...
; This is the same position as in the main program
PTP P1 Vel=100% PDAT1
...
END

```

```

DEFDAT MY_PROG( )
...
DECL E6POS XP1={X 100, Z 200, Z 300 ... E6 0.0}
...
ENDDAT

```

- RETURN can be used to terminate a subprogram and jump back to the program module from which it was called.

```

DEF MY_PROG( )
; This is the main program
...
LOCAL_PROG1( )
...
END

-----

DEF LOCAL_PROG1( )
...
IF $IN[12]==FALSE THEN
RETURN ; Jump back to main program
ENDIF
...
END

```

Procedure for creating local subprograms

1. "Expert" user group
2. Display the DEF line.
3. Open the SRC file in the editor.

```

DEF MY_PROG( )
...
END

```

4. Jump to **beneath** the END line with the cursor.
5. Assign a new local program header with DEF, program name and brackets.

```

DEF MY_PROG( )
...
END
DEF PICK_PART( )

```

6. Complete the new subprogram with an END command.

```

DEF MY_PROG( )
...
END
DEF PICK_PART( )
END

```

7. When Return is pressed, a horizontal bar is inserted between the main program and the subprogram.

```
DEF MY_PROG ( )
...
END

DEF PICK_PART ( )
END
```

8. The main program and the subprogram can now be edited further.
9. Close and save the program.

4.2 Working with global subprograms

Definition of global subprograms

- Global subprograms have their own SRC and DAT files.

```
DEF GLOBAL1 ( )
...
END
```

```
DEF GLOBAL2 ( )
...
END
```

Relationships when working with local subprograms

- Global subprograms can be called repeatedly.
- Once a local subprogram has been executed, the program jumps back to the next command after the subprogram call.

```
DEF GLOBAL1 ( )
...
GLOBAL2 ( )
...
END
```

```
DEF GLOBAL2 ( )
...
GLOBAL3 ( )
...
END
```

```
DEF GLOBAL3 ( )
...
END
```

- The maximum nesting depth for subprograms is 20.
- Point coordinates are saved in the corresponding DAT list and are only available for the corresponding program.

```
DEF GLOBAL1 ( )
...
PTP P1 Vel=100% PDAT1
END
```

```
DEFDAT GLOBAL1 ( )
DECL E6POS XP1={X 100, Z 200, Z 300 ... E6 0.0}
ENDDAT
```

Different coordinates for P1 in Global2 ()

```
DEF GLOBAL2 ( )
...
PTP P1 Vel=100% PDAT1
END
```

```
DEFDAT GLOBAL2 ( )
DECL E6POS XP1={X 800, Z 775, Z 999 ... E6 0.0}
ENDDAT
```

- RETURN can be used to terminate a subprogram and jump back to the program module from which it was called.

```
DEF GLOBAL1 ( )
...
GLOBAL2 ( )
...
END
```

```
DEF GLOBAL2 ( )
...
IF $IN[12]==FALSE THEN
RETURN ; Jump back to GLOBAL1 ( )
ENDIF
...
END
```

Procedure for programming with global subprograms

1. "Expert" user group
2. Create a new program.

```
DEF MY_PROG ( )
...
END
```

3. Create the second new program.

```
DEF PICK_PART ( )
...
END
```

4. Open the SRC file of the program MY_PROG in the editor.
5. Program the subprogram call using program name and brackets.

```
DEF MY_PROG ( )
...
PICK_PART ( )
...
END
```

6. Close and save the program.

4.3 Transferring parameters to subprograms

Description of parameter transfer

- Syntax

```
DEF MY_PROG( )
...
CALC (K, L)
...
END
```

```
DEF CALC(R:IN, S:OUT)
...
END
```

Principle of parameter transfer

- There are two possibilities for transferring parameters to subprograms:
 - as IN parameters
 - as OUT parameters
- Parameter transfer can be carried out in local or global subprograms.
- Parameter transfer as IN parameter (*Call by value*):
 - The value of the variable remains unchanged in the main program, i.e. it continues to be processed with the old value from the main program.
 - The subprogram can only read the value of the variable, and not write it.
- Parameter transfer as OUT parameter (*Call by reference*):
 - The value of the variable is changed in the main program, i.e. the value is taken from the subprogram.
 - The subprogram reads the value, modifies it and writes the new value back.
- Parameter transfer to local subprograms

```
DEF MY_PROG( )
DECL REAL r,s
...
CALC_1(r)
...
CALC_2(s)
...
END

-----
DEF CALC_1(num1:IN)
; The value "r" is only transferred to num1 for reading
DECL REAL num1
...
END

-----
DEF CALC_2(num2:OUT)
; The value "s" is transferred to num2, altered and written back
DECL REAL num2
...
END
```

- Parameter transfer to global subprograms

```
DEF MY_PROG ( )
DECL REAL r, s
...
CALC_1(r)
...
CALC_2(s)
...
END
```

```
DEF CALC_1(num1:IN)
; The value "r" is only transferred to num1 for reading
DECL REAL num1
...
END
```

```
DEF CALC_2(num2:OUT)
; The value "s" is transferred to num2, altered and written back
DECL REAL num2
...
END
```

- Value transfer to the same data types is always possible.
- Value transfer to different data types:

```
DEF MY_PROG ( )
DECL DATATYPE1 value
CALC(value)
END

DEF CALC(num:IN)
DECL DATATYPE2 num
...
END
```

DATATYPE 1	DATATYPE 2	Comments
BOOL	INT, REAL, CHAR	ERROR (...parameters not compatible)
INT	REAL	INT value is used as REAL value.
INT	CHAR	Character from the ASCII table is used.
CHAR	INT	INT value from the ASCII table is used.
CHAR	REAL	REAL value from the ASCII table is used.
REAL	INT	REAL values are rounded.
REAL	CHAR	REAL values are rounded; character from the ASCII table is used.

- Transfer of multiple parameters

```

DEF MY_PROG ( )
DECL REAL w
DECL INT a, b
...
CALC(w, b, a)
...
CALC(w, 30, a)
...
END

DEF CALC(ww:OUT, bb:IN, aa:OUT)
;1.) w <-> ww, b -> bb, a <-> aa
;2.) w <-> ww, 30 -> bb, a <-> aa
DECL REAL ww
DECL INT aa, bb
...
END

```



It is also possible to transfer no values if calculation is carried out in the subprogram without these values. Example: CALC(w, , a)

- Parameter transfer with arrays
 - Arrays can only be transferred to a new array in their entirety.
 - Arrays may only be transferred with parameter OUT (Call by reference)

```

DEF MY_PROG ( )
DECL CHAR name[10]
...
name="PETER"
RECHNE(name[])
...
END

DEF RECHNE(my_name[]):OUT
; Only ever create array in subprogram without array size
; The array size adapts itself to the output array
DECL CHAR my_name[]
...
END

```



Transfer of **complete** arrays: ARRAY_1D[] (1-Dimensional), ARRAY_2D[,] (2-Dimensional), ARRAY_3D[, ,] (3-Dimensional)

- Individual array elements can also be transferred.

```

DEF MY_PROG ( )
DECL CHAR name[10]
...
name="PETER"
CALC(name[1])
...
END

DEF RECHNE(symbol:IN)
; Only one character is transferred
DECL CHAR symbol
...
END

```



When transferring individual array elements, the destination can only be a variable and not an array. Here, only the letter "P" is transferred to the subprogram.

Procedure for parameter transfer

Preliminary considerations

1. Define which parameters are required in the subprogram.

2. Determine the type of parameter transfer (IN or OUT parameter).
3. Define the output and target data types (ideally the same data type).
4. Determine the order for parameter transfer.



Note: The first parameter sent is written to the first parameter in the subprogram, the second to the second parameter in the subprogram, etc.

1. Load the main program into the editor.
2. In the main program, declare, initialize and, if necessary, manipulate the variables.
3. Create a subprogram call with a variable call.
4. Close and save the main program.
5. Load the subprogram into the editor.
6. Complete the DEF line with variables and IN/OUT.
7. In the subprogram, declare, initialize and, if necessary, manipulate the variables.
8. Close and save the subprogram.

Complete example:

```

DEF MY_PROG ( )
DECL REAL w
DECL INT a, Anzahl
w = 1.5
a = 3
b = 5
CALC(w, b, a)
; Current values
; w = 3.8
; a = 13
; b = 5
END

DEF CALC(ww:OUT, bb:IN, aa:OUT)
; w <-> ww, b -> bb, a <-> aa

DECL REAL ww
DECL INT aa, bb
ww = ww + 2.3 ; ww = 1.5 + 2.3 =3.8 ->w
bb = bb + 5 ; bb = 5 + 5 = 10
aa = bb + aa ; aa = 10 + 3= 13 -> a
END

```

4.4 Programming functions

Definition of functions with KRL

- A function is a subprogram that returns a certain value to the main program.
- Frequently, certain input values are required in order to be able to calculate the return value.
- The data type to be written back to the main program is determined in the function header.
- The value to be transferred is transferred using the RETURN(return_value) statement.
- There are local and global functions.
- Syntax of a function

```

DEFECT DATATYPE NAME_FUNCTION ( )
...
RETURN (return_value)
ENDEFCT

```

Principle of functions with KRL

- The program name is also the variable name of a certain data type.
- Call of a global function

```
DEF MY_PROG( )
DECL REAL result, value
...
result = CALC(value)
...
END
```

```
DEFECT REAL CALC(num:IN)
DECL REAL return_value, num
...
RETURN(return_value)
ENDEFCT
```

 The statement RETURN(return_value) must come before the statement ENDEFCT.

- Call of a local function

```
DEF MY_PROG( )
DECL REAL result, value
...
result = CALC(value)
...
END

-----

DEFECT REAL CALC(num:IN)
DECL REAL return_value, num
...
RETURN(return_value)
ENDEFCT
```

- Use of IN / OUT parameters for value transfer
 - Value transfer as IN parameters

```
DEF MY_PROG( )
DECL REAL result, value
value = 2.0
result = CALC(value)
; Value = 2.0
; Result = 1000.0
END
```

```
DEFECT REAL CALC(num:IN)
DECL REAL return_value, num
num = num + 8.0
return_value = num * 100.0
RETURN(return_value)
ENDEFCT
```

 The transfer value value is not changed.

- Value transfer as OUT parameters

```

DEF MY_PROG( )
DECL REAL result, value
value = 2.0
result = CALC(value)
; Value = 10.0
; Result = 1000.0
END

```

```

DEFFCT REAL CALC(num:OUT)
DECL REAL return_value, num
num = num + 8.0
return_value = num * 100.0
RETURN(return_value)
ENDFCT

```



The transfer value `value` is changed and returned.

Procedure for programming functions

1. Define the value the function is to supply (return data type).
2. Define which parameters are required in the function (transfer data types).
3. Determine the type of parameter transfer (IN or OUT parameter).
4. Determine whether a local or global function is required.
5. Load the main program into the editor.
6. In the main program, declare, initialize and, if necessary, manipulate the variables.
7. Create a function call.
8. Close and save the main program.
9. Create a function (global or local).
10. Load the function into the editor.
11. Complete the `DEFFCT` line with data type, variables and `IN/OUT`.
12. In the function, declare, initialize and manipulate the variables.
13. Create the `RETURN(return_value)` line.
14. Close and save the function.

4.5 Working with standard KUKA functions

List of standard KUKA functions

Mathematical functions:

Description	KRL function
Absolute value	ABS(x)
Square root	SQRT(x)
Sine	SIN(x)
Cosine	COS(x)
Tangent	TAN(x)
Arc cosine	ACOS(x)
Arc tangent	ATAN2(y,x)

Functions for string variables:

Description	KRL function
Determination of the string length in the declaration	StrDeclLen(x)
String variable length after initialization	StrLen(x)

Description	KRL function
Deleting the contents of a string variable	StrClear(x)
Extending a string variable	StrAdd(x,y)
Comparing the contents of a string variable	StrComp(x,y,z)
Copying a string variable	StrCopy(x,y)

Functions for message generation:

Description	KRL function
Generate message	Set_KrlMsg(a,b,c,d)
Generate dialog	Set_KrIDLg(a,b,c,d)
Check message	Exists_KrlMsg(a)
Check dialog	Exists_KrIDLg(a,b)
Delete message	Clear_KrlMsg(a)
Read message buffer	Get_MsgBuffer(a)

Principle for the use of standard KUKA functions

Each standard function is called with transfer parameters:

- With fixed values

```
result = SQRT(16)
```

- Variables of a simple data type

```
result = SQRT(x)
```

- Variables consisting of arrays

```
result = StrClear(Name[])
```

- Variables consisting of enumeration data types
- Variables consisting of structures
- With several different variables

```
result = Set_KrlMsg(#QUIT, message_parameter, parameter[], option)
```



Message_parameter, parameter[1...3] and option are predefined KUKA structures.

Every function requires a suitable variable in which the result of the function can be stored.

- Mathematical functions return a REAL value.
- String functions return BOOL or INT values.

```
; Deletion of a string
result = StrClear(Name[])
```

- Message functions return BOOL or INT values.

```
; Deletion of a message (BOOL: deleted?)
result = Clear_KrlMsg(Rueckwert)
```


5 Motion programming with KRL

5.1 Programming motions with KRL

Definition of a motion

Specifications required for a motion:

- Motion type – PTP, LIN, CIRC
- End position and auxiliary position if applicable
- Exact positioning or approximate positioning
- Approximation distance if applicable
- Velocity – PTP (%) and CP motion (m/s)
- Acceleration
- Tool – TCP and load
- Working base
- Robot-guided or external tool
- Orientation control with CP motions
- Approximation distance if applicable
- Circular angle in the case of circular motion CIRC

Principle of motion programming

Motion type PTP

- PTP *End point* <C_PTP <CP approximation>>
- The robot moves to a position in the DAT file (the position has been taught beforehand by means of an inline form) and approximates this point P3.

```
PTP XP3 C_PTP
```

- The robot moves to an entered position.
 - Axis-specific (AXIS or E6AXIS)

```
PTP {A1 0, A2 -80, A3 75, A4 30, A5 30, A6 110}
```

- Position in space (with currently active tool and base)

```
PTP {X 100, Y -50, Z 1500, A 0, B 0, C 90, S 3, T3 35}
```

- The robot only moves if one or more aggregates have been entered.

```
PTP {A1 30} ; Only A1 is moved to 30°
```

```
PTP {X 200, A 30} ; Only in X to 200mm and A to 30°
```

Motion type LIN

- LIN *End point* <CP approximation>
- The robot moves to a calculated position and approximates this point ABLAGE[4]

```
LIN ABLAGE[4] C_DIS
```

Motion type CIRC

- CIRC *Auxiliary point, End point*<, CA *Circular angle*> <Approximate positioning>
- The robot moves to the positions in the DAT file (the positions have been taught beforehand by means of inline forms) and executes a circular angle of 190°.

```
CIRC XP3, XP4, CA 190
```

- Circular angle CA



The orientation taught at the programmed end point is accepted at the actual end point.

- **Positive circular angle ($CA > 0$):** the circular path is executed in the programmed direction: Start point - Auxiliary point - End point.

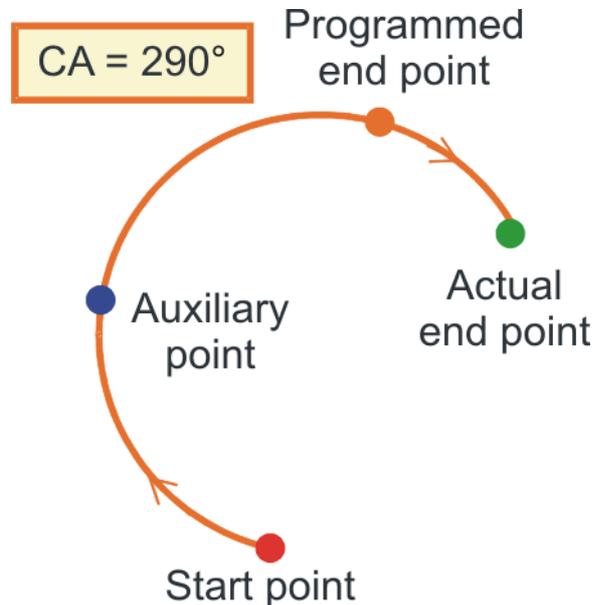


Fig. 5-1: Circular angle $CA = +290^\circ$

- **Negative circular angle ($CA < 0$):** the circular path is executed in the opposite direction from the programmed direction: Start point - End point - Auxiliary point.

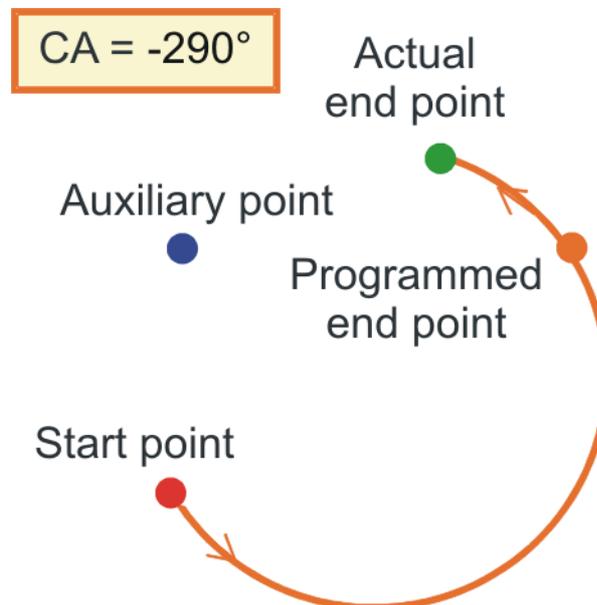


Fig. 5-2: Circular angle $CA = -290^\circ$

Function of the motion parameters

Default settings for motion programming

- Existing settings can be applied:
 - from execution of the `INI` line
 - from the last inline form
 - from the last settings of the relevant system variables
- Modification or initialization of the relevant system variables

System variables of the motion parameters

■ Tool: \$TOOL and \$LOAD

- Activation of the calibrated TCP

```
$TOOL = tool_data[x] ; x = 1...16
```

- Activation of the corresponding load data

```
$LOAD = load_data[x] ; x = 1...16
```

■ Reference base/working base: \$BASE

- Activation of the calibrated base

```
$BASE = base_data[x] ; x = 1...16
```

■ Robot-guided or external tool: \$IPO_MODE

- Robot-guided tool

```
$IPO_MODE = #BASE
```

- External tool

```
$IPO_MODE = #TCP
```

■ Velocity:

- for PTP motion

```
$VEL_AXIS[x] ; x=1...8 for each axis
```

- for CP motions LIN or CIRC

```
$VEL.CP = 2.0 ; [m/s] path velocity
```

```
$VEL.ORI1 = 150 ; [°/s] swivel velocity
```

```
$VEL.ORI2 = 200 ; [°/s] rotational velocity
```



The working direction of the tool is the X axis in most cases. The rotational velocity is the rotation about the X axis with angle C. Swivel velocity is the velocity of the swivel motion about the other two angles (A and B).

■ Acceleration

- for PTP motion

```
$ACC_AXIS[x] ; x=1...8 for each axis
```

- for CP motions LIN or CIRC

```
$ACC.CP = 2.0 ; [m/s] path acceleration
```

```
$ACC.ORI1 = 150 ; [°/s] swivel acceleration
```

```
$ACC.ORI2 = 200 ; [°/s] rotational acceleration
```

■ Approximation distance

- only for PTP motion: **C_PTP**

```
PTP XP3 C_PTP
$APO_CPTP = 50 ; Approximation magnitude in [%] for C_PTP
```

- for CP motions LIN, CIRC and for PTP: **C_DIS**

The distance from the end point must be less than the value \$APO.CDIS

```
PTP XP3 C_DIS
LIN XP4 C_DIS
$APO.CDIS = 250.0 ; [mm] distance
```

- for CP motions LIN, CIRC: **C_ORI**
The dominant orientation angle must be less than the value \$APO.CO-RI

```
LIN XP4 C_ORI
$APO.CORI = 50.0 ; [°] angle
```

- for CP motions LIN, CIRC: **C_VEL**
The velocity in the deceleration phase to the end point must be less than the value \$APO.CVEL

```
LIN XP4 C_VEL
$APO.CVEL = 75.0 ; [%] percent
```

- **Orientation control:** only for LIN and CIRC

- for LIN and CIRC: **\$ORI_TYPE**

```
$ORI_TYPE = #CONSTANT
```

The orientation remains constant during the CP motion. The programmed orientation is disregarded for the end point

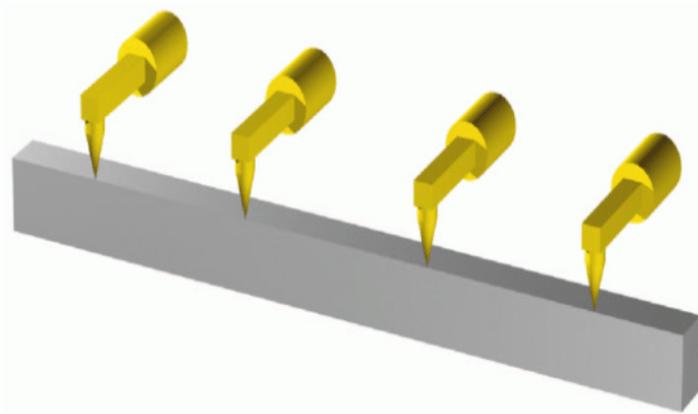


Fig. 5-3: Orientation control - Constant

```
$ORI_TYPE = #VAR
```

During the CP motion the orientation changes continuously to the orientation of the end point.

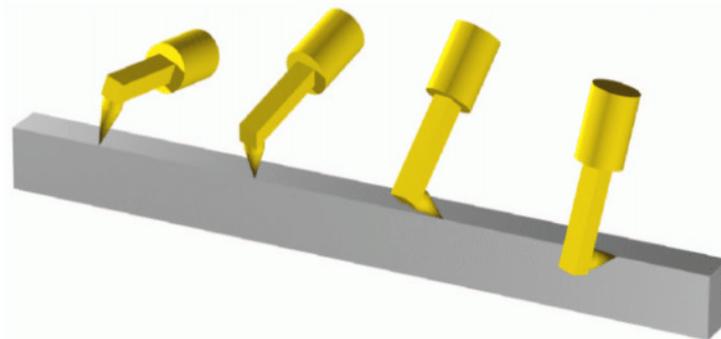


Fig. 5-4: Standard or Wrist PTP

```
$ORI_TYPE = #JOINT
```

During the path motion, the orientation of the tool changes continuously from the start position to the end position. This is done by linear transformation of the wrist axis angles. The problem of the wrist singularity can be avoided using this option as there is no orientation control by rotating and pivoting the tool direction.

- only for CIRC: **\$CIRC_TYPE**

i The variable **\$CIRC_TYPE** is meaningless in the case of a linear transformation of the wrist axis angles with **\$ORI_TYPE = #JOINT**.

```
$CIRC_TYPE = #PATH
```

Path-related orientation control during the circular motion

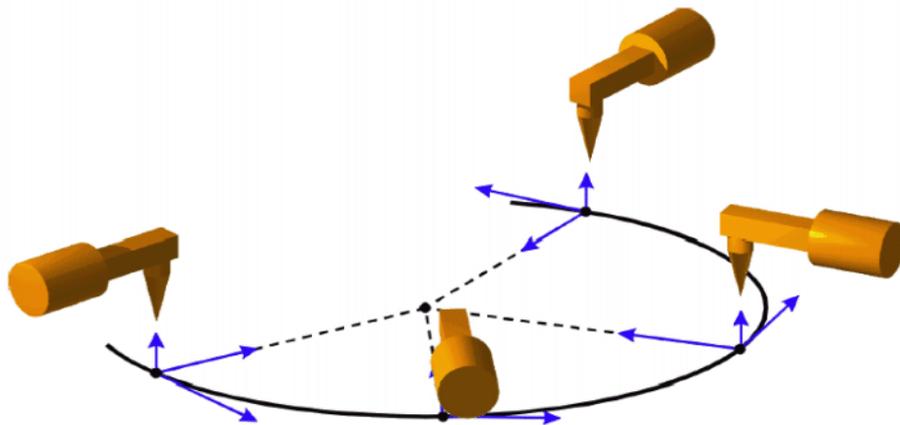


Fig. 5-5: Constant orientation, path-related

```
$CIRC_TYPE = #BASE
```

Space-related orientation control during the circular motion

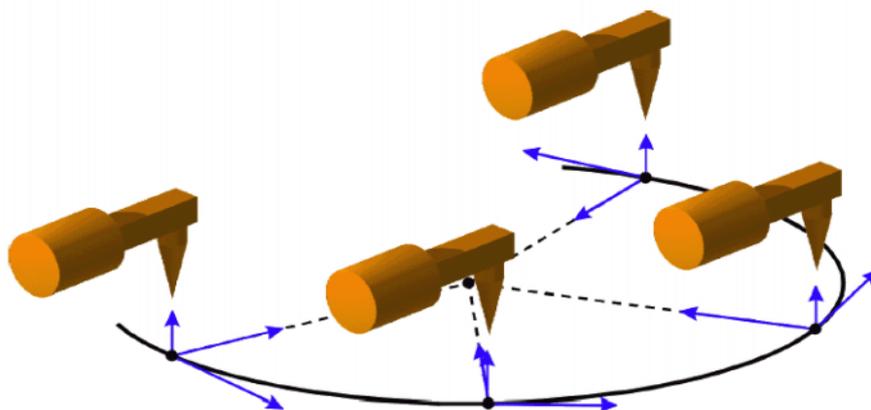


Fig. 5-6: Constant orientation, base-related

Procedure for programming motions with KRL

1. At Expert level, load the program into the editor by pressing the **Open** key.
2. Check the default settings for motion programming and apply them or re-initialize:

- Tool (\$TOOL and \$LOAD)
 - Base settings (\$BASE)
 - Robot-guided or external tool (\$IPO_MODE)
 - Velocity
 - Acceleration
 - Approximation distance if applicable
 - Orientation control if applicable
3. Create motion command consisting of:
 - Motion type (PTP, LIN, CIRC)
 - End point (for CIRC: auxiliary point also)
 - for CIRC: circular angle (CA) if applicable
 - Activate approximate positioning (C_PTP, C_DIS, C_ORI, C_VEL)
 4. For new motion, go back to step 3.
 5. Close editor and save changes.

5.2 Programming relative motions with KRL

Description

- Absolute motion

```
PTP {A3 45}
```

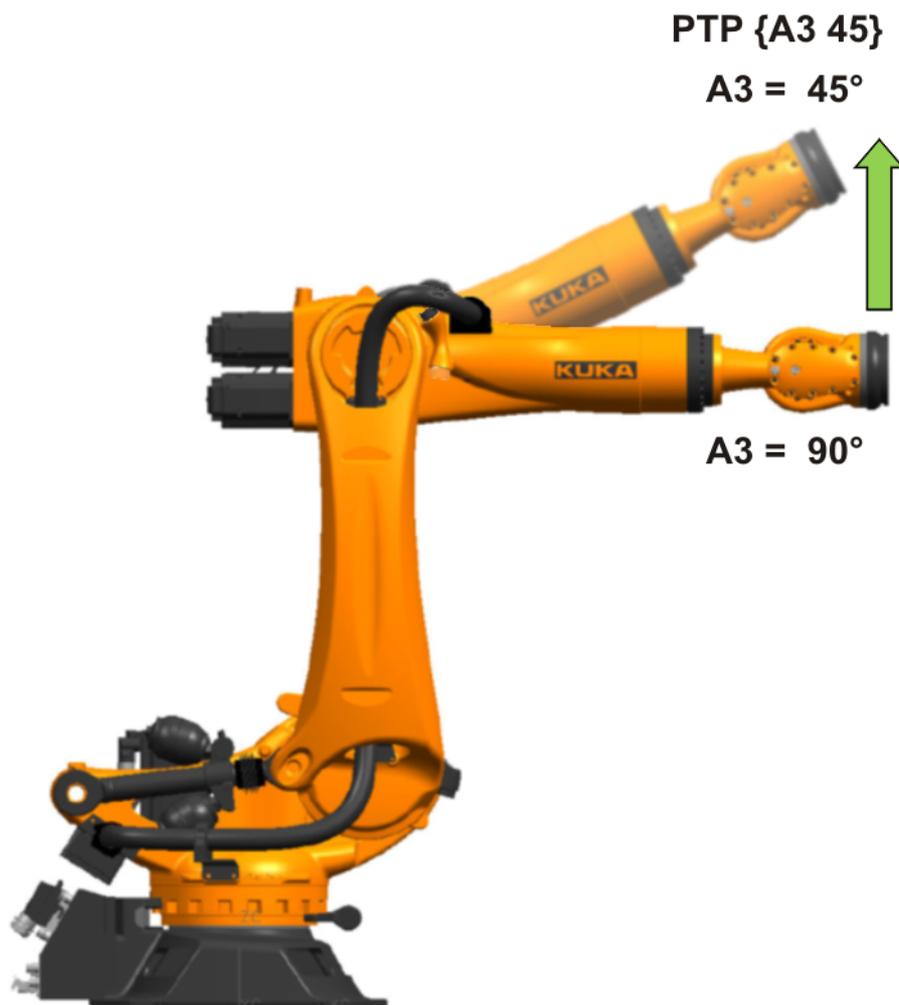


Fig. 5-7: Absolute motion of axis A3

- Relative motion

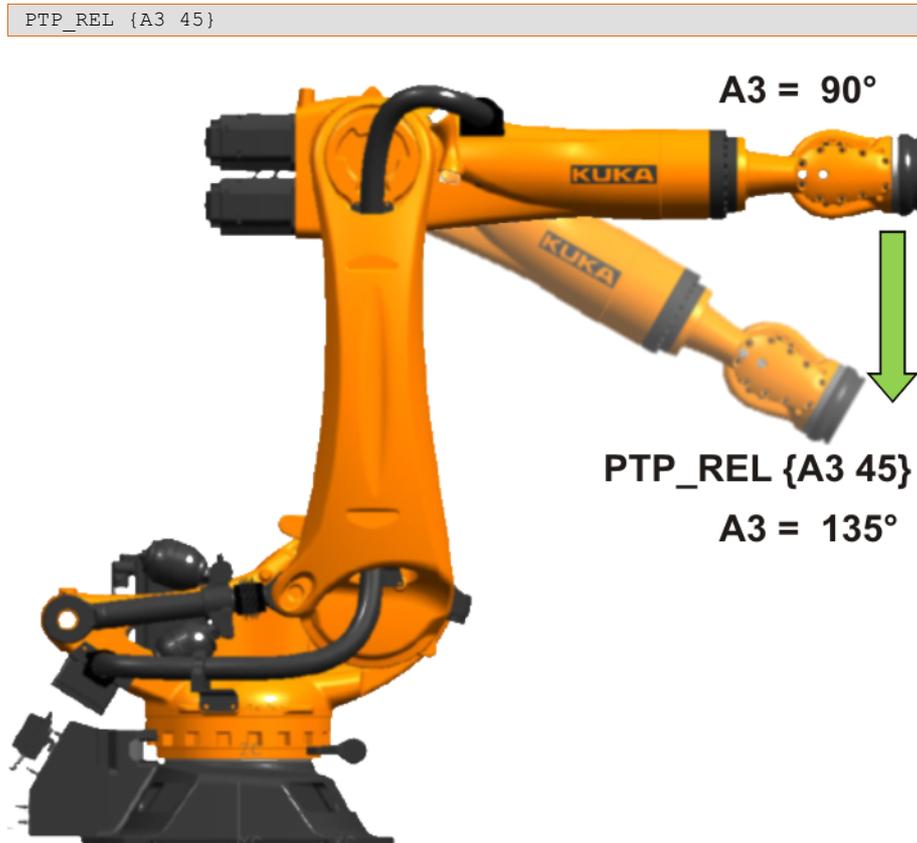


Fig. 5-8: Relative motion of axis A3

Relative motions are available for:

- PTP motion
- LIN motion
- CIRC motion

Principle of a relative motion



A REL statement always refers to the current position of the robot. For this reason, if a REL motion is interrupted, the robot executes the entire REL motion again, starting from the position at which it was interrupted.

Relative motion **PTP_REL**

- `PTP_REL End point <C_PTP <CP approximation>>`
- Axis 2 is moved 30 degrees in a negative direction. None of the other axes moves.

```
PTP_REL {A2 -30}
```

- The robot moves 100 mm in the X direction and 200 mm in the negative Z direction from the current position. Y, A, B, C and S remain constant. T is calculated in relation to the shortest path.

```
PTP_REL {X 100,Z -200}
```

Relative motion **LIN_REL**

- `LIN_REL End point <CP approximation> <#BASE|#TOOL>`
- The TCP moves 100 mm in the X direction and 200 mm in the negative Z direction from the current position in the BASE coordinate system. Y, A, B, C and S remain constant. T is determined by the motion.

```
LIN_REL {X 100,Z -200} ; #BASE is default setting
```

- The TCP moves 100 mm from the current position in the negative X direction in the TOOL coordinate system. Y, Z, A, B, C and S remain constant. T is determined by the motion.

This example is suitable for moving the tool backwards against the tool direction. The precondition is that the tool direction has been calibrated along the X axis.

```
LIN_REL {X -100} #TOOL
```

Relative motion **CIRC_REL**

- **CIRC_REL** Auxiliary point, End point<, CA Circular angle> <CP approximation>
- The end point of the circular motion is defined by a circular angle of 500°. The end point is approximated.

```
CIRC_REL {X 100,Y 30,Z -20},{Y 50},CA 500 C_VEL
```

Procedure for programming motions with KRL

1. At Expert level, load the program into the editor by pressing the **Open** key.
2. Check the default settings for motion programming and apply them or re-initialize:
 - Tool (\$TOOL and \$LOAD)
 - Base settings (\$BASE)
 - Robot-guided or external tool (\$IPO_MODE)
 - Velocity
 - Acceleration
 - Approximation distance if applicable
 - Orientation control if applicable
3. Create motion command consisting of:
 - Motion type (PTP_REL, LIN_REL, CIRC_REL)
 - End point (for CIRC: auxiliary point also)
 - for LIN: select a reference system (#BASE or #TOOL)
 - for CIRC: circular angle (CA) if applicable
 - Activate approximate positioning (C_PTP, C_DIS, C_ORI, C_VEL)
4. For new motion, go back to step 3.
5. Close editor and save changes.

5.3 Calculating or manipulating robot positions

Description

Robot end positions

- are stored in the following structures:
 - **AXIS / E6AXIS** - axis angle (A1...A6 and possibly E1...E6)
 - **POS / E6POS** - position (X, Y, Z), orientation (A, B, C), status and turn (S, T)
 - **FRAME** - only position (X, Y, Z), orientation (A, B, C)
- can manipulate existing positions from the DAT file.
- Individual aggregates of existing positions can be modified using a point separator.

Principle



For the calculation, it is important to observe the correct TOOL and BASE settings and then activate them during motion programming. If this is not observed, unexpected motions and collisions may occur.

Important system variables

- **\$POS_ACT**: Current robot position. The variable (E6POS) defines the set-point position of the TCP in relation to the BASE coordinate system.
- **\$AXIS_ACT**: Current axis-specific robot position (setpoint). The variable (E6AXIS) contains the current axis angles or axis position.

Calculating the absolute end position

- Modify the position from the DAT file once

```
XP1.x = 450 ; New X value 450 mm
XP1.z = 30*distance ; New Z value is calculated
PTP XP1
```

- Modify the position from the DAT file every time it is executed

```
; X value is offset by 450 mm each time
XP2.x = XP2.x + 450
PTP XP2
```

- Position is applied and saved in a variable

```
myposition = XP3
myposition.x = myposition.x + 100 ; 100 mm is added to the X value
myposition.z = 10*distance ; Calculate new Z value
myposition.t = 35 ; Set Turn value
PTP XP3 ; Position was not changed
PTP myposition ; Calculated position
```

Procedure

1. At Expert level, load the program into the editor by pressing the **Open** key.
2. Calculate/manipulate position. If required, temporarily save the newly calculated values to a new variable.
3. Check the default settings for motion programming and apply them or re-initialize:
 - Tool (\$TOOL and \$LOAD)
 - Base settings (\$BASE)
 - Robot-guided or external tool (\$IPO_MODE)
 - Velocity
 - Acceleration
 - Approximation distance if applicable
 - Orientation control if applicable
4. Create motion command consisting of:
 - Motion type (PTP, LIN, CIRC)
 - End point (for CIRC: auxiliary point also)
 - for CIRC: circular angle (CA) if applicable
 - Activate approximate positioning (C_PTP, C_DIS, C_ORI, C_VEL)
5. For new motion, go back to step 3.
6. Close editor and save changes.

5.4 Deliberate modification of Status and Turn bits

Description

- The position (X, Y, Z) and orientation (A, B, C) values of the TCP are not sufficient to define the robot position unambiguously, as different axis positions are possible for the same TCP. Status and Turn serve to define an unambiguous position that can be achieved with different axis positions.

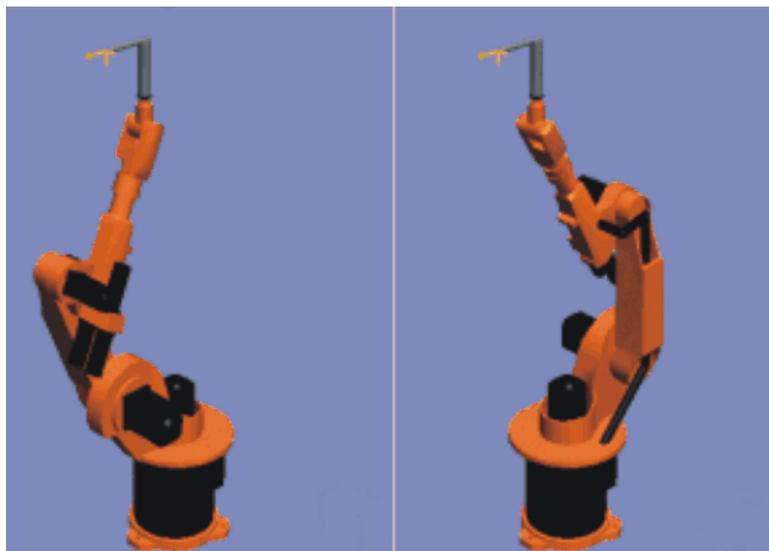


Fig. 5-9: Example: Same TCP position, different axis position

- Status (S) and Turn (T) are integral parts of the data types POS and E6POS:

```
STRUC POS REAL X, Y, Z, A, B, C, INT S, T
```

```
STRUC E6POS REAL X, Y, Z, A, B, C, E1, E2, E3, E4, E5, E6, INT S, T
```

- The robot controller **only** takes the programmed Status and Turn values into consideration **for PTP motions**. They are ignored for CP motions.
- The first motion instruction in a KRL program must therefore be one of the following instructions so that an unambiguous starting position is defined for the robot:
 - A complete PTP instruction of type POS or E6POS
 - Or a complete PTP instruction of type AXIS or E6AXIS

“**Complete**” means that all components of the end point must be specified. The default HOME position is always a complete PTP instruction.
- Status and Turn can be omitted in the subsequent instructions:
 - The robot controller retains the previous Status value.
 - The Turn value is determined by the path in CP motions.
 - In the case of PTP motions, the robot controller selects the Turn value that results in the shortest possible path (i.e. no software limit switches violated and also closest to the start angle).

Function

STATUS

- The Status specification prevents ambiguous axis positions.
- **Bit 0**: specifies the position of the intersection of the wrist axes (A4, A5, A6).

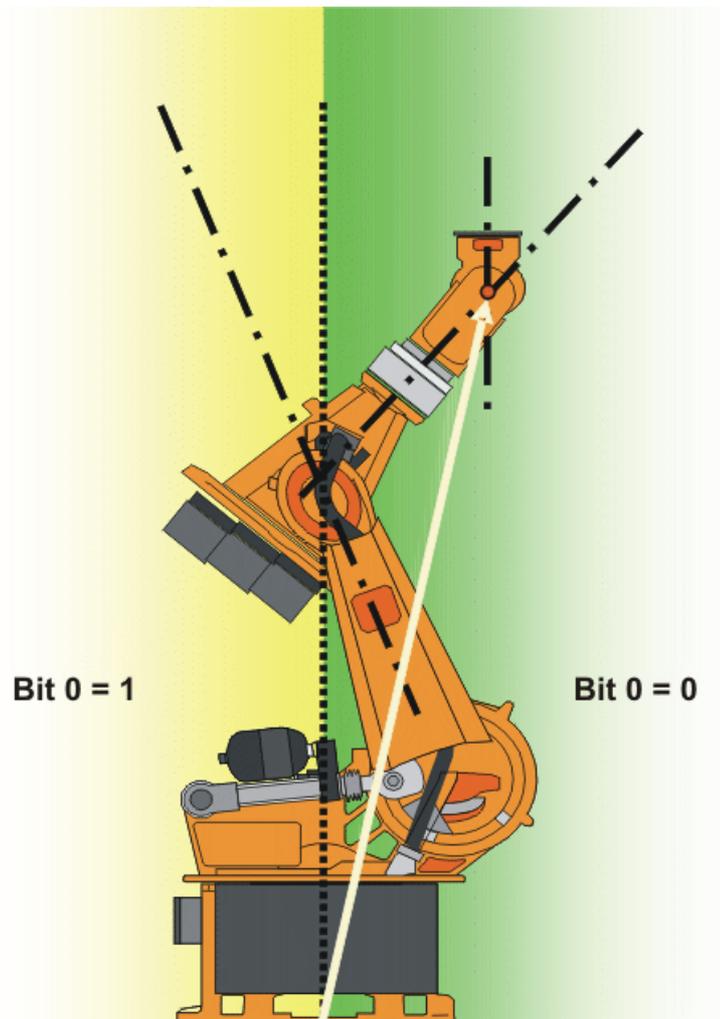


Fig. 5-10: Example: The intersection of the wrist axes (red dot) is in the basic area.

- **Bit 1:** specifies the position of axis 3. The angle at which the value of bit 1 changes depends on the robot type.

For robots whose axes 3 and 4 intersect, the following applies:

Position	Value
$A3 \geq 0^\circ$	Bit 1 = 1
$A3 < 0^\circ$	Bit 1 = 0

For robots with an offset between axis 3 and axis 4, the angle at which the value of bit 1 changes depends on the size of this offset.

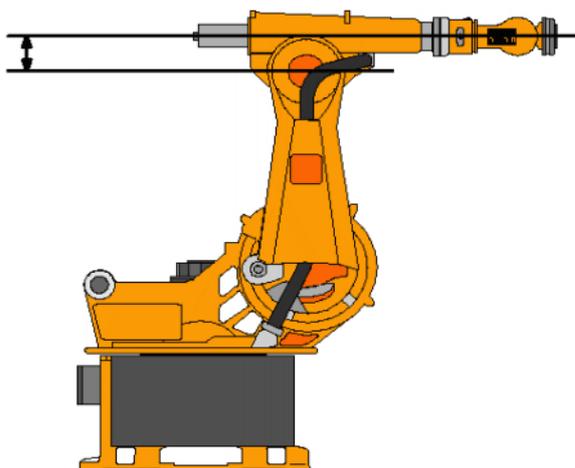


Fig. 5-11: Offset between A3 and A4 – example: KR 30

- Bit 2: specifies the position of axis 5.

Position	Value
A5 > 0	Bit 2 = 1
A5 ≤ 0	Bit 2 = 0

- Bit 3 is **not used** and is always 0.
- Bit 4: specifies whether or not the point was taught using an absolutely accurate robot.

Depending on the value of the bit, the point can be executed by both absolutely accurate robots and non-absolutely-accurate robots. Bit 4 is for information purposes only and has no influence on how the robot calculates the point. This means, therefore, that when a robot is programmed offline, bit 4 can be ignored.

Description	Value
The point was not taught with an absolutely accurate robot.	Bit 4 = 0
The point was taught with an absolutely accurate robot.	Bit 4 = 1

TURN

- The Turn specification makes it possible to move axes through angles greater than +180° or less than -180° without the need for special motion strategies (e.g. auxiliary points). With rotational axes, the individual bits determine the sign before the axis value in the following way:

Bit = 0: angle ≥ 0°

Bit = 1: angle < 0°

- Overview of all axes

Value	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	A6 ≥ 0°	A5 ≥ 0°	A4 ≥ 0°	A3 ≥ 0°	A2 ≥ 0°	A1 ≥ 0°
1	A6 < 0°	A5 < 0°	A4 < 0°	A3 < 0°	A2 < 0°	A1 < 0°

- Example

```
DECL POS XP1 = {X 900, Y 0, Z 800, A 0, B 0, C 0, S 6, T 19}
```

T 19 corresponds to T 'B010011'. This means:

Axis	Angle	Binary
A 1	negative	1
A 2	negative	2
A 3	positive	4

Axis	Angle	Binary
A 4	positive	8
A 5	negative	16
A 6	positive	32

Procedure

1. At Expert level, load the program into the editor by pressing the **Open** key.
2. Manipulate **Status** and **Turn**. If required, temporarily save the newly calculated values to a new variable.
3. Check the default settings for motion programming and apply them or re-initialize:
 - Tool (\$TOOL and \$LOAD)
 - Base settings (\$BASE)
 - Robot-guided or external tool (\$IPO_MODE)
 - Velocity
 - Acceleration
 - Approximation distance if applicable
 - Orientation control if applicable
4. Create motion command consisting of:
 - Motion type (PTP, LIN, CIRC)
 - End point (for CIRC: auxiliary point also)
 - for CIRC: circular angle (CA) if applicable
 - Activate approximate positioning (C_PTP, C_DIS, C_ORI, C_VEL)
5. For new motion, go back to step 3.
6. Close editor and save changes.

6 Working with system variables

6.1 Cycle time measurement by means of timers

Description of cycle time measurement with KUKA system timers



Fig. 6-1

- \$TIMER[1]
- \$TIMER[2]
- ...
- \$TIMER[32]

The system variables \$TIMER[No] serve the purpose of measuring time sequences.



The values of the timer \$TIMER[No] are entered/displayed in milliseconds **ms**.

Starting and stopping a timer by means of KRL

- STARTING: \$TIMER_STOP[No] = FALSE
- STOPPING: \$TIMER_STOP[No] = TRUE



The timer can also be preset, started and stopped manually in the display window.

Principle of cycle time measurement

Presetting a timer

- The factory setting of a timer is 0 ms.
- The timer has its current value.
- A timer can be set forwards or backwards to any freely selected value.

```
;Timer 5 is preset to 0 ms
$TIMER[5] = 0

; Timer 12 is set to 1.5 seconds
$TIMER[12] = 1500

; Timer 4 is reset to -8 seconds
$TIMER[4] = -8000
```

- Resetting and starting a timer

```
; Timer 7 reset to 0 ms
$TIMER[7] = 0
; Start timer 7
$TIMER_STOP[7] = FALSE
```

■ Stopping a timer and subsequent comparison

```
; Timer 7 running
...
; Stop timer 7
$TIMER_STOP[7] = TRUE

; Executed if 10 seconds or greater ...
IF $TIMER[7] >= 10000 THEN
...
```



A timer is always started and stopped by means of the **advance run pointer**.

Procedure for cycle time measurement

1. Select a “free” timer from the 32 timers available.
2. Preset/reset the timer.
3. Start the timer, observing the advance run pointer.
4. Stop the timer, observing the advance run pointer.
5. Save the current cycle time if required, or preset the timer again.

```
DEF MY_TIME ( )
...
INI
$TIMER[1] = 0 ; Reset TIMER 1
PTP HOME Vel=100% DEFAULT

WAIT SEC 0 ; Trigger advance run stop
$TIMER_STOP[1]=FALSE ; Start cycle time measurement

PTP XP1
PTP XP2
LIN XP3
...
PTP X50
PTP HOME Vel=100% DEFAULT

WAIT SEC 0 ; Trigger advance run stop
$TIMER_STOP[1]=TRUE ; Stop cycle time measurement

; Current cycle time is buffered in timer 12
$TIMER[12] = $TIMER[1]
END
```

7 Using program execution control functions

7.1 Programming conditional statements or branches

Description of conditional statements and branches with KRL

- A branch is used to divide a program into several paths.
- The `IF` statement checks a condition that can be `TRUE` or `FALSE`. Accordingly, statements are either executed or not executed.
- Branch

```
IF ..... THEN
...
ELSE
...
ENDIF
```

Using branches

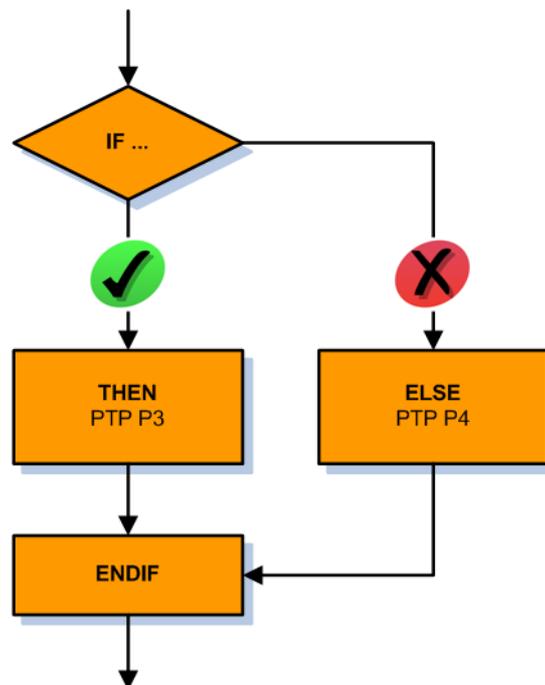


Fig. 7-1: Program flowchart: IF branch

Branch

- with alternative branch

```
IF condition THEN
Anweisung
ELSE
;Statement
ENDIF
```

- without alternative branch (conditional statement)

```
IF condition THEN
;Statement
ENDIF
```

Examples of branches

Branch without alternative branch

```

DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; P21 is only addressed with error_nr 5
IF error_nr == 5 THEN
PTP P21 Vel=100% PDAT21
ENDIF
...
END

```

Branch with alternative branch

```

DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; P21 is only addressed with error_nr 5, otherwise P22
IF error_nr == 5 THEN
PTP P21 Vel=100% PDAT21
ELSE
PTP P22 Vel=100% PDAT22
ENDIF
...
END

```

Branch with complex execution conditions

```

DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; P21 is only addressed with error_nr 1 or 10 or greater than 99
IF ((error_nr == 1) OR (error_nr == 10) OR (error_nr > 99)) THEN
PTP P21 Vel=100% PDAT21
ENDIF
...
END

```

Branch with Boolean expressions

```

DEF MY_PROG( )
DECL BOOL no_error
...
INI
no_error = TRUE
...
; P21 is only addressed if there is no error (no_error)
IF no_error == TRUE THEN
PTP P21 Vel=100% PDAT21
ENDIF
...
END

```



The expression `IF no_error==TRUE THEN` can also be reduced to `IF no_error THEN`. Omission always signifies comparison with `TRUE`.

7.2 Programming a switch statement

Description of the switch statement with KRL

- A “switch case” statement can be used to differentiate between numerous different cases and execute different actions for each case.
- The `switch` statement is used to distinguish between different cases.

- A transferred variable in the `switch` statement is used as the switch and jumps to the predefined `case` statements in the statement block.
- If the `switch` statement finds no predefined `case`, the `default` section is executed.
- Switch statement

```
SWITCH ...
CASE ...
...
CASE ...
...
CASE
...
...
...
DEFAULT
...
ENDSWITCH
```

Using switch statements

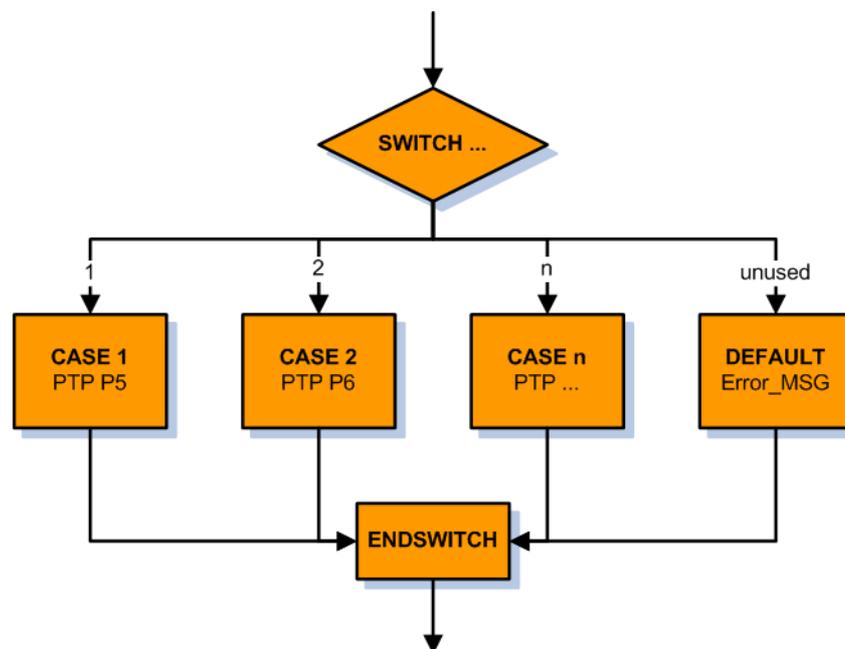


Fig. 7-2: Program flowchart: SWITCH – CASE statements

Switch statement

- A switch statement can be used with the following data types:
 - INT (integer)

```
SWITCH number
CASE 1
...
```

- CHAR (character)

```
SWITCH symbol
CASE "X"
...
```

- ENUM (enumeration data type)

```
SWITCH mode_op
CASE #T1
...
```

- Only with defined switch statements

```
SWITCH number
CASE 1
...
CASE 2
...
CASE 3
...
ENDSWITCH
```



If *number* is not equal to 1 or 2 or 3, the program jumps directly to ENDSWITCH without a statement being executed.

- Only with defined switch statements and an alternative case

```
SWITCH number
CASE 1
...
CASE 2
...
CASE 3
...
DEFAULT
...
ENDSWITCH
```



If *number* is not equal to 1 or 2 or 3, the program jumps to the DEFAULT case in order to execute the statement(s) contained therein.

- With several solutions in a switch statement

```
SWITCH number
CASE 1,2
...
CASE 3,4,5
...
CASE 6
...
DEFAULT
...
ENDSWITCH
```

Examples of switch statements

Switch statement without alternative case

```
DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 4
...
; Motion is possible in a defined case only
SWITCH error_nr
CASE 1
PTP P21 Vel=100% PDAT21
CASE 2
PTP P22 Vel=100% PDAT22
CASE 3
PTP P23 Vel=100% PDAT23
CASE 4
PTP P24 Vel=100% PDAT24
ENDSWITCH
...
```

Switch statement without alternative case

```

DEF MY_PROG( )
DECL INT error_nr
...
INI
error_nr = 99
...
; In a non-defined case, robot moves to HOME
SWITCH error_nr
CASE 1
PTP P21 Vel=100% PDAT21
CASE 2
PTP P22 Vel=100% PDAT22
CASE 3
PTP P23 Vel=100% PDAT23
CASE 4
PTP P24 Vel=100% PDAT24
DEFAULT
PTP HOME Vel=100% DEFAULT
ENDSWITCH
...

```

Switch statement with an enumeration data type

```

DEF MY_PROG( )
ENUM COLOR_TYPE red, yellow, blue, green
DECL COLOR_TYPE my_color
...
INI
my_color = #red
...
SWITCH my_color
CASE #red
PTP P21 Vel=100% PDAT21
CASE #yellow
PTP P22 Vel=100% PDAT22
CASE #green
PTP P23 Vel=100% PDAT23
CASE #blue
PTP P24 Vel=100% PDAT24
ENDSWITCH
...

```

7.3 Programming loops

General information regarding loops

- Loops are used for repeating program statements.
- It is not permissible to jump into a loop from outside.
- Loops can be nested.
- There are various different types of loop:
 - Endless loop
 - Counting loop
 - Conditional loops
 - Rejecting loop
 - Non-rejecting loop

7.3.1 Programming an endless loop

Description of an endless loop

- The endless loop is a loop that is executed again every time execution has been completed.
- Execution can be canceled by external influences.
- Syntax

```

LOOP
; Statement
...
; Statement
ENDLOOP

```

Principle of an endless loop

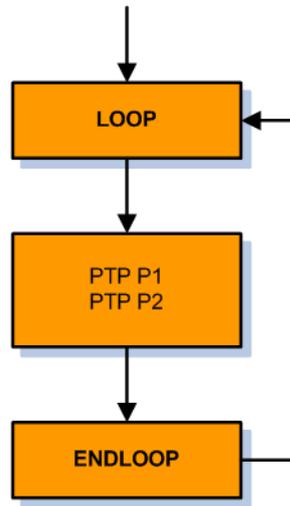


Fig. 7-3: Program flowchart: endless loop

- The endless loop can be exited with `EXIT`.
- When exiting an endless loop with `EXIT`, it must be ensured that there is no risk of a collision.
- If two endless loops are nested in one another, two `EXIT` commands are required in order to exit both loops.

Examples for programming of an endless loop

Endless loop without break

```

DEF MY_PROG( )
INI
PTP HOME Vel=100% DEFAULT

LOOP
PTP P1 Vel=90% PDAT1
PTP P2 Vel=100% PDAT2
PTP P3 Vel=50% PDAT3
PTP P4 Vel=100% PDAT4
ENDLOOP

PTP P5 Vel=30% PDAT5
PTP HOME Vel=100% DEFAULT
END

```



Point P5 is never addressed in the program.

Endless loop with break

```

DEF MY_PROG( )
INI
PTP HOME Vel=100% DEFAULT

LOOP
PTP P1 Vel=90% PDAT1
PTP P2 Vel=100% PDAT2
IF $IN[3]==TRUE THEN ; Condition for break
EXIT
ENDIF
PTP P3 Vel=50% PDAT3
PTP P4 Vel=100% PDAT4
ENDLOOP

PTP P5 Vel=30% PDAT5
PTP HOME Vel=100% DEFAULT
END

```

i Point P5 is addressed as soon as input 1 is active.
Important: The motion between P2 and P5 must be checked to ensure there is no risk of a collision.

7.3.2 Programming a counting loop

Definition of a counting loop

- The FOR loop is a control structure which can be used to execute one or more statements with a defined number of repetitions.
- Syntax with step size +1

```

FOR counter = start TO last
;Statement
ENDFOR

```

- The step size (increment) can also be specified as an integer using the keyword STEP.

```

FOR counter = start TO last STEP increment
;Statement
ENDFOR

```

Principle of a counting loop

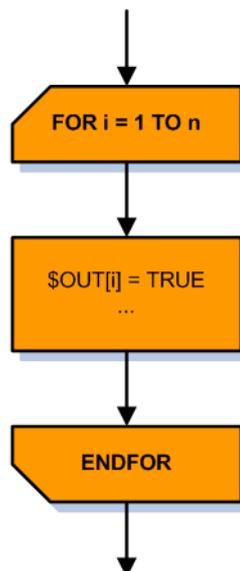


Fig. 7-4: Program flowchart: counting loop

- For a counting loop, a previously declared integer variable is required.
- The counting loop starts with the value `start` and ends, at the latest, with the value `last`.

```
FOR counter = start TO last
;Statement
ENDFOR
```

- The counting loop can be exited immediately with `EXIT`.

How does a counting loop work?

```
DECL INT counter

FOR counter = 1 TO 3 Step 1
;Statement
ENDFOR
```

1. Loop counter is initialized with the start value: `counter = 1`
2. At `ENDFOR`, the loop counter is incremented by the value of `STEP`.
3. Loop begins again at the `FOR` line.
4. The entry condition is checked: counting variable must be less than the specified end value, otherwise the loop is terminated.
5. Depending on the result of the check, either the loop counter is incremented again, or the loop is terminated and the program is resumed after the `ENDFOR` line.

Counting backwards with a counting loop

```
DECL INT counter

FOR counter = 15 TO 1 Step -1
;Statement
ENDFOR
```



The initial value or start value of the loop must be greater than the end value in order to allow the loop to be executed several times.

Programming with a counting loop

Examples for programming of counting loops

- Simple counting loop without specification of step size

```
DECL INT counter

FOR counter = 1 TO 50
$OUT[counter] == FALSE
ENDFOR
```



If no step size is specified with `STEP`, the step size `+1` is used by default.

- Simple counting loop with specification of step size

```
DECL INT counter

FOR counter = 1 TO 4 STEP 2
$OUT[counter] == TRUE
ENDFOR
```



This loop is only executed twice: once with the start value `counter=1` and the second time with `counter=3`. Once the counter value reaches 5, the loop is terminated immediately.

- Double counting loop with specification of step size

```
DECL INT counter1, counter2

FOR counter1 = 1 TO 21 STEP 2
  FOR counter2 = 20 TO 2 STEP -2
    ...
  ENDFOR
ENDFOR
```

i The inner loop is always executed first (here `counter1`), and then the outer one (`counter2`).

7.3.3 Programming a rejecting loop

Description of a rejecting loop

- A rejecting loop is also referred to as a pre-test loop.
- This type of loop repeats operations as long as a specified condition is fulfilled.
- Syntax

```
WHILE condition
; Statement
ENDWHILE
```

- The rejecting loop can be exited immediately with `EXIT`.

Principle of a rejecting loop

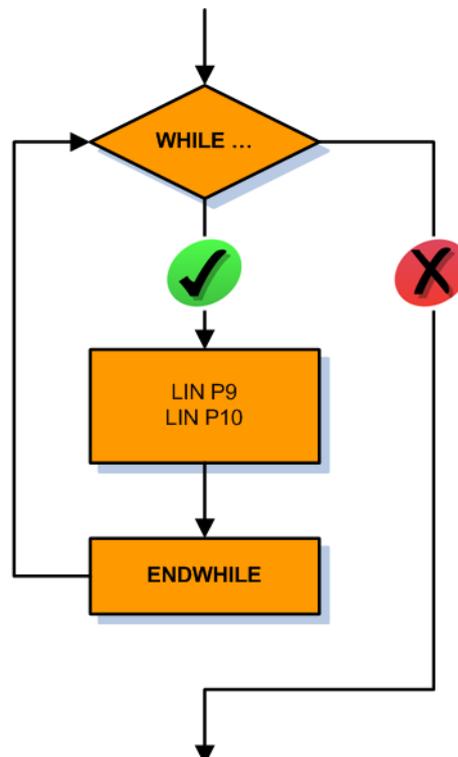


Fig. 7-5: Program flowchart: rejecting loop

- Rejecting loops are used to check first whether a recurring operation is started.
- The execution condition must be met before the loop can be executed.

Programming with a rejecting loop

- If the execution condition is not met, the loop is terminated immediately and the statements after `ENDWHILE` are executed.
- Rejecting loop with simple execution condition

```
...
WHILE IN[41]==TRUE ; Part is ready in magazine
PICK_PART( )
ENDWILE
...
```



The expression `WHILE IN[41]==TRUE` can also be reduced to `WHILE IN[41]`. Omission always signifies comparison with `TRUE`.

- Rejecting loop with simple negated execution condition

```
...
WHILE NOT IN[42]==TRUE ; Input 42: magazine is empty
PICK_PART( )
ENDWILE...
```

or

```
...
WHILE IN[42]==FALSE ; Input 42: magazine is empty
PICK_PART( )
ENDWILE...
```

- Rejecting loop with complex execution condition

```
...
WHILE ((IN[40]==TRUE) AND (IN[41]==FALSE) OR (counter>20))
PALETTE( )
ENDWILE
...
```

7.3.4 Programming a non-rejecting loop

Description of a non-rejecting loop

- A non-rejecting loop is also referred to as a post-test loop.
- This non-rejecting loop first executes the statements and then checks at the end whether a `condition` has been met in order to be able to exit the loop.
- Syntax

```
REPEAT
; Statement
UNTIL condition
```

- The non-rejecting loop can be exited immediately with `EXIT`.

Principle of a non-rejecting loop

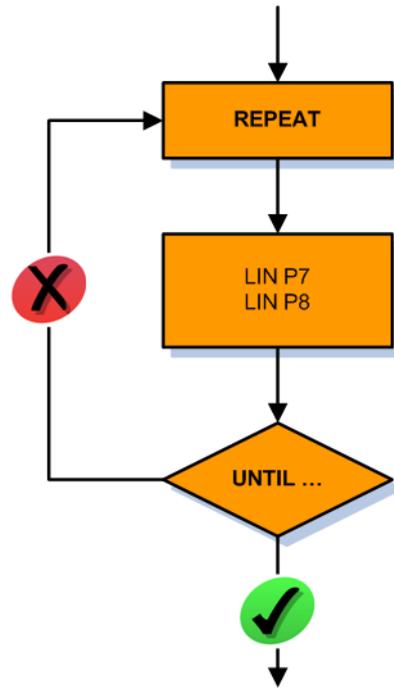


Fig. 7-6: Program flowchart: non-rejecting loop

- If the result of the condition is positive, the loop is exited and the statements after `UNTIL` are executed.
- If the result of the condition is negative, the loop is started again from `REPEAT`.

Programming a non-rejecting loop

- Non-rejecting loop with simple execution condition

```
...
REPEAT
PICK_PART( )
UNTIL IN[42]==TRUE ; Input 42: magazine is empty
...
```

- Non-rejecting loop with complex execution condition

```
...
REPEAT
PALETTE( )
UNTIL ((IN[40]==TRUE) AND (IN[41]==FALSE) OR (counter>20))
...
```

7.4 Programming wait functions

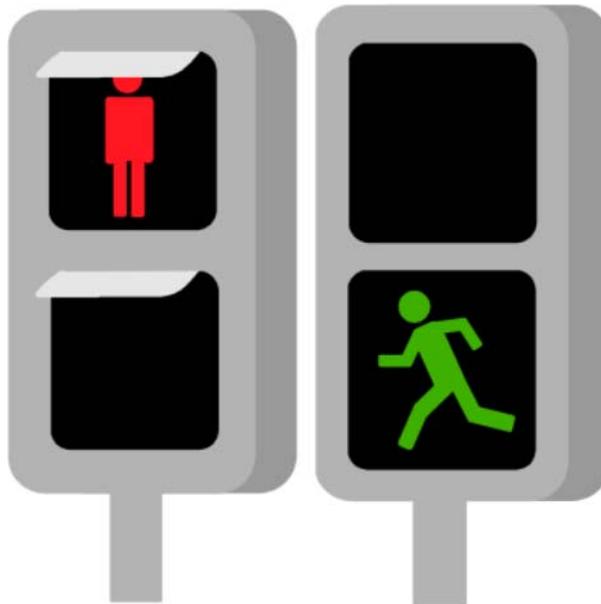


Fig. 7-7

KRL programming of:

- Time-dependent wait function
- Signal-dependent wait function

7.4.1 Time-dependent wait function

Description of a time-dependent wait function with KRL

- The time-dependent wait function waits for the specified `time` before the process can be resumed.
- Syntax

```
WAIT SEC time
```

Principle of the time-dependent wait function

- The time base for a time-dependent wait function is seconds (s).
- The maximum time is 2147484 seconds, i.e. more than 24 days.
- The time value can also be transferred with a suitable variable.
- The smallest meaningful unit of time is 0.012 seconds (interpolation cycle).
- If the specified time is negative, the program does not wait.
- A time-dependent wait function triggers an advance run stop; approximate positioning is thus not possible.
- In order to generate just one advance run stop, the command `WAIT SEC 0` is used.

Programming a time-dependent wait function

- Time-dependent wait function with a fixed time

```
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
WAIT SEC 5.25
PTP P3 Vel=100% PDAT3
```

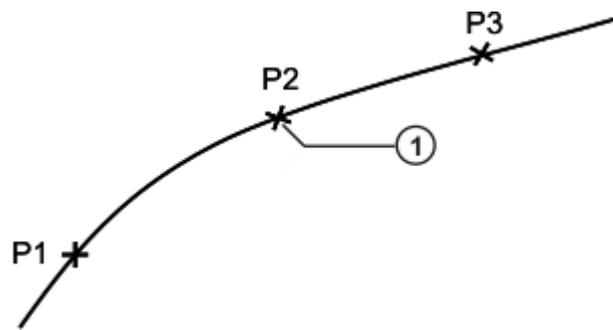


Fig. 7-8: Example motion for logic

- Time-dependent wait function with a calculated time

```
WAIT SEC 3*0.25
```

- Time-dependent wait function with a variable

```
DECL REAL time
time = 12.75
WAIT SEC time
```

7.4.2 Signal-dependent wait function

Description of a signal-dependent wait function

- The signal-dependent wait function switches when the `condition` is met and the process is resumed.
- Syntax

```
WAIT FOR condition
```

Principle of the signal-dependent wait function

- The signal-dependent wait function triggers an advance run stop; approximate positioning is thus not possible.
- Even if the condition has already been met, an advance run stop is still generated.
- An advance run stop is prevented by means of the command `CONTINUE`.

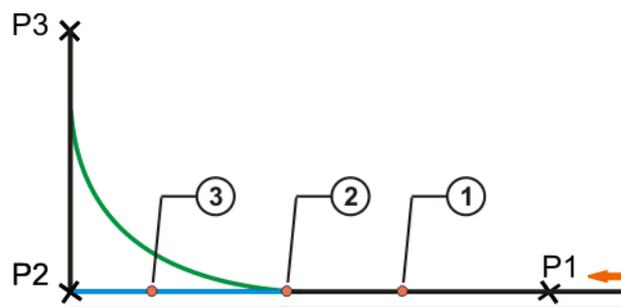


Fig. 7-9: Example motion for logic with advance run

Programming a signal-dependent wait function

- WAIT FOR with advance run stop

```
PTP P1 Vel=100% PDAT1
PTP P2 CONT Vel=100% PDAT2
WAIT FOR $IN[20]
PTP P3 Vel=100% PDAT3
```

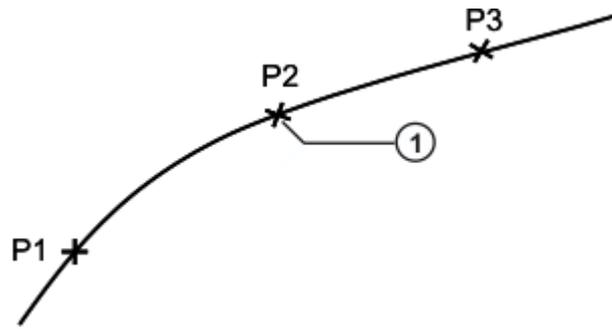


Fig. 7-10: Example motion for logic

- WAIT FOR with processing in the advance run (use of CONTINUE)

```
PTP P1 Vel=100% PDAT1
PTP P2 CONT Vel=100% PDAT2
CONTINUE
WAIT FOR ($IN[10] OR $IN[20])
PTP P3 Vel=100% PDAT3
```

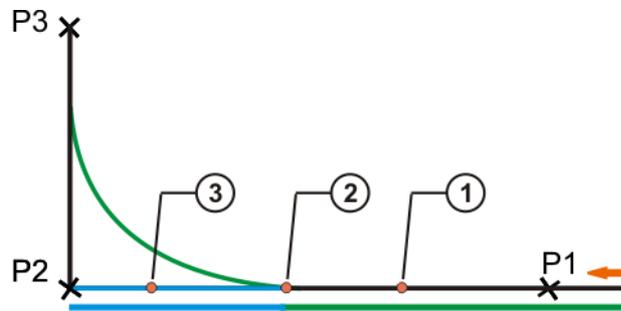


Fig. 7-11: Example motion for logic with advance run

8 Switching functions with KRL

8.1 Programming simple switching functions

Description of simple switching functions

General

- The robot controller can manage up to 4096 digital inputs and 4096 digital outputs.
- The inputs/outputs are implemented using optional field bus systems.
- The configuration is customer-specific.
- Configuration is carried out using WorkVisual.

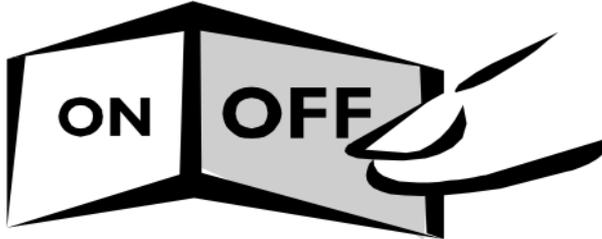


Fig. 8-1

Capabilities of simple switching functions

- Simple activation/deactivation of an output (with advance run/advance run stop)
- Pulsing of an output
- Switching of an output with the main run pointer (without advance run stop)

Function of simple switching functions

Simple activation/deactivation of an output

- Activation of an output

```
$OUT[10]=TRUE
```

- Deactivation of an output

```
$OUT[10]=FALSE
```

- When an output is switched, an advance run stop is generated; the motion can thus not be approximated.

```
...
PTP P20 CONT Vel=100% PDAT20
$OUT[30]=TRUE
PTP P21 CONT Vel=100%PDAT21
```

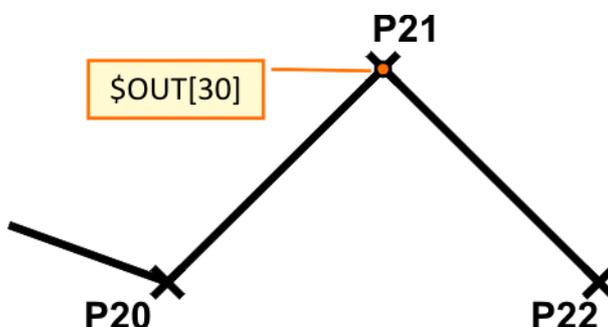


Fig. 8-2: Switching with the advance run stop

- Using the command `CONTINUE` cancels the advance run stop.
- Using the command `CONTINUE` causes switching to be carried out with the advance run stop.
- Approximate positioning is possible with `CONTINUE`.

- CONTINUE **only** refers to the next line (even if this line is empty).

```
...
PTP P20 CONT Vel=100% PDAT20
CONTINUE
$OUT[30]=TRUE
PTP P21 CONT Vel=100%PDAT21
```

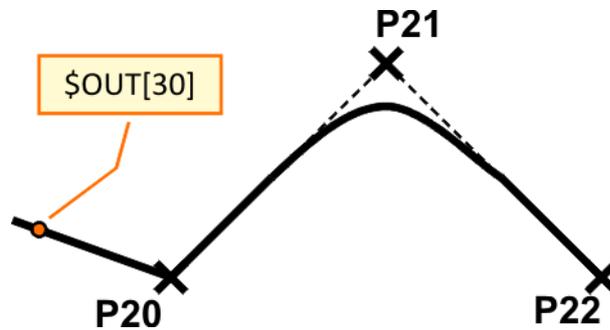


Fig. 8-3: Switching in the advance run

Switching of an output with the main run pointer

- Up to 8 outputs can be switched with reference to the main run and without an advance run stop.
- If exact positioning is programmed, switching is carried out when the end point is reached.
- If approximate positioning is programmed, switching is carried out in the middle of the approximate positioning motion of the end point.

```
...
LIN P20 CONT Vel=100% PDAT20
$OUT_C[30]=TRUE
LIN P21 CONT Vel=100%PDAT21
```

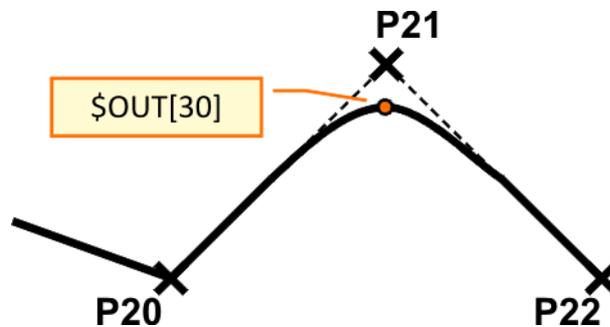


Fig. 8-4: Switching with the main run

Pulsing of an output

- Sets a pulse.
- The output is set to a defined level for a specified duration.
- The output is then reset automatically by the system.
- The PULSE statement triggers an advance run stop.

NOTICE

The pulse is not terminated in the event of an EMERGENCY STOP, an operator stop or an error stop!

- Syntax

PULSE (*Signal, Level, Pulse duration*)

```
PULSE ($OUT[30], TRUE, 20); Positive pulse
```

```
PULSE ($OUT[31], FALSE, 20); Negative pulse
```

- If a pulse is programmed before the END statement, the duration of program execution is increased accordingly.

```
...
PULSE ($OUT[50], TRUE, 2)
END
```

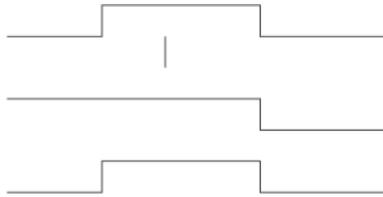


Fig. 8-5: PULSE+END, example

- If program execution is reset (RESET) or aborted (CANCEL) while a pulse is active, the pulse is immediately reset.

```
...
PULSE ($OUT[50], TRUE, 2)
; Program is now reset or deselected
```



Fig. 8-6: PULSE+RESET, example

Programming of simple switching functions

- Switching outputs with an advance run stop

```
...
LIN P20 CONT Vel=100% PDAT20
$OUT[50]=TRUE ; Switch on
LIN P21 CONT Vel=100%PDAT21
$OUT[50]=FALSE ; Switch off
LIN P22 CONT Vel=100%PDAT22
```

- Switching outputs by means of a pulsed function with an advance run stop

```
...
LIN P20 CONT Vel=100% PDAT20
PULSE ($OUT[50], TRUE, 1.5) ; Positive pulse
PULSE ($OUT[51], FALSE, 1.5) ; Negative pulse
LIN P21 CONT Vel=100%PDAT21
```

- Switching outputs in the advance run

```
...
LIN P20 CONT Vel=100% PDAT20
CONTINUE
$OUT[50]=TRUE ; Switch on
LIN P21 CONT Vel=100%PDAT21
CONTINUE
$OUT[50]=FALSE ; Switch off
LIN P22 CONT Vel=100%PDAT22
```

- Switching outputs by means of a pulsed function in the advance run

```
...
LIN P20 CONT Vel=100% PDAT20
CONTINUE
PULSE ($OUT[50], TRUE, 1.5) ; Positive pulse
CONTINUE
PULSE ($OUT[51], FALSE, 1.5) ; Negative pulse
LIN P21 CONT Vel=100%PDAT21
```

- Switching outputs with the main run

```

...
LIN P20 CONT Vel=100% PDAT20
$OUT_C[50]=TRUE
LIN P21 CONT Vel=100%PDAT21

```

8.2 Programming path-related switching functions with TRIGGER WHEN DISTANCE

Description of path-related switching functions with TRIGGER WHEN DISTANCE

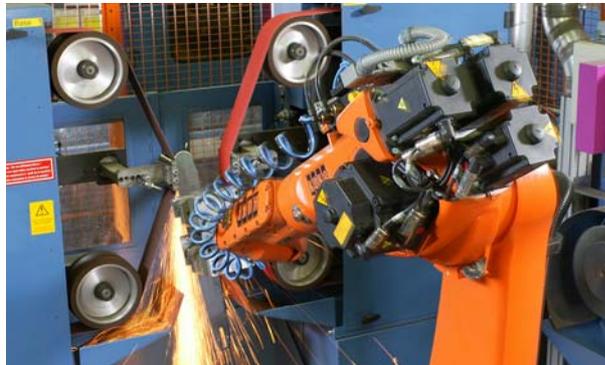


Fig. 8-7: Grinding application

- The path-related switching function `TRIGGER` triggers a defined statement.
- The statement refers to the start point or end point of the motion block.
- The statement is executed parallel to the robot motion.
- The switching point can be shifted in time.

Function of path-related switching functions with TRIGGER WHEN DISTANCE

Syntax

- `TRIGGER WHEN DISTANCE=Position DELAY=Time DO Statement <PRIO=Priority>`
- **Position:** Defines the point at which the statement is triggered. Possible values:
 - **0:** The statement is triggered at the start point of the motion block.
 - **1:** The statement is triggered at the end point. If the end point is approximated, the statement is triggered in the middle of the approximate positioning arc.
- **Time:** This defines a shift time relative to the selected position.
 - Positive and negative values can be used.
 - The time base is milliseconds (ms).
 - Times of up to 10,000,000 ms can be used without problems.
 - If the value set for the time is too great or too small, switching is carried out at the switching limits.
- **Statement:** Options include:
 - Assignment of a value to a variable



The value must not be assigned to a runtime variable.

- OUT statement
- PULSE statement
- Subprogram call. In this case, *Priority* **must** be specified.
- **Priority** (only for a subprogram call):
 - Priorities 1, 2, 4 to 39 and 81 to 128 are available.

- Priorities 40 to 80 are reserved for cases in which the priority is automatically assigned by the system. If the priority is to be assigned automatically by the system, the following is programmed: `PRIO = -1`.

Programming of path-related switching functions

TRIGGER WHEN
DISTANCE

Switching options with `TRIGGER WHEN DISTANCE`

- Start point and end point are exact positioning points

```

LIN XP1
LIN XP2
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3
LIN XP4

```

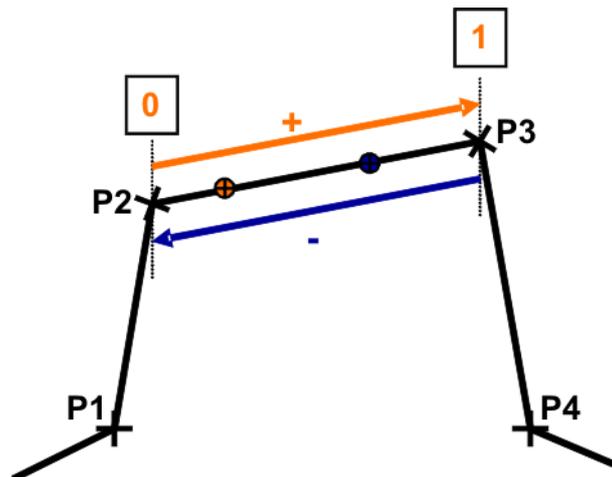


Fig. 8-8: Example of `TRIGGER WHEN DISTANCE` with exact positioning/exact positioning

- Start point is approximate positioning point, end point is exact positioning point

```

LIN XP1
LIN XP2 C_DIS
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3
LIN XP4

```

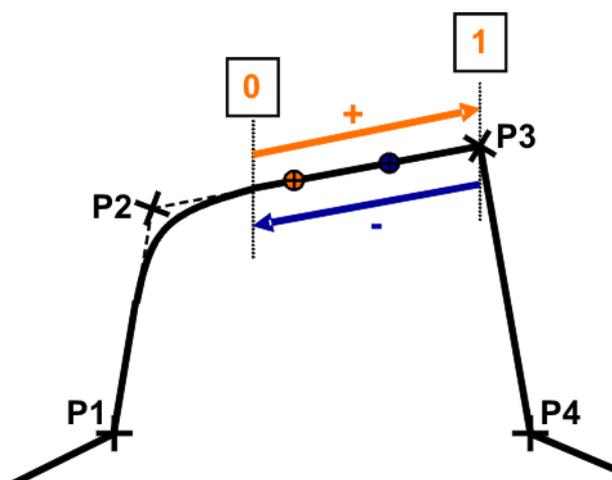


Fig. 8-9: Example of `TRIGGER WHEN DISTANCE` with approximate positioning/exact positioning

- Start point is exact positioning point, end point is approximate positioning point

```

LIN XP1
LIN XP2
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3 C_DIS
LIN XP4
    
```

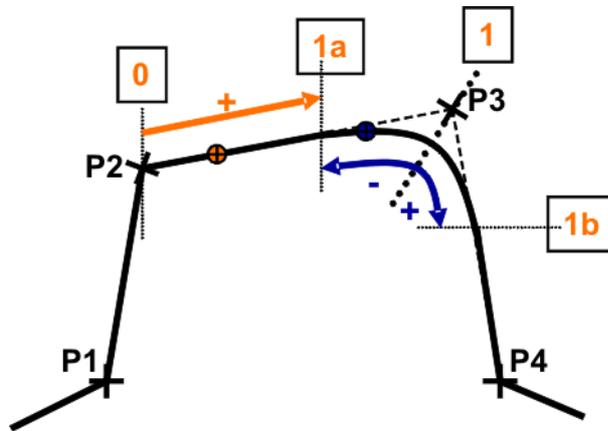


Fig. 8-10: Example of TRIGGER WHEN DISTANCE with exact positioning/approximate positioning

- Start point and end point are approximate positioning points

```

LIN XP1
LIN XP2 C_DIS
TRIGGER WHEN DISTANCE = 0 DELAY = 20 DO duese = TRUE
TRIGGER WHEN DISTANCE = 1 DELAY = -25 DO UP1() PRIO=75
LIN XP3 C_DIS
LIN XP4
    
```

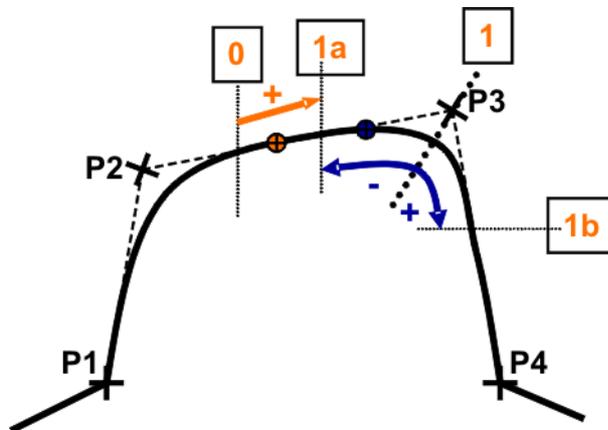


Fig. 8-11: Example of TRIGGER WHEN DISTANCE with approximate positioning/approximate positioning

8.3 Programming path-related switching functions with `TRIGGER WHEN PATH`

Description of path-related switching functions with `TRIGGER WHEN PATH`

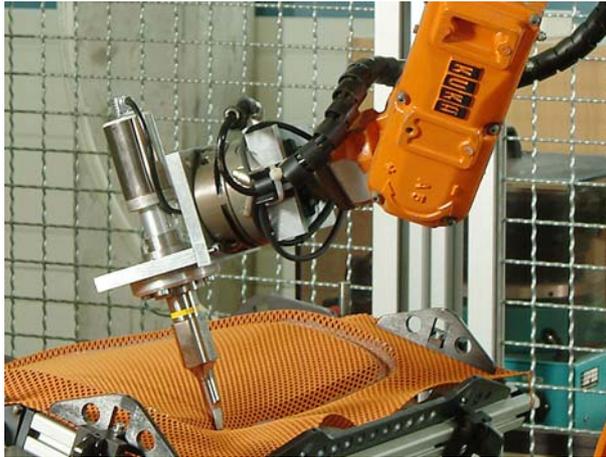


Fig. 8-12: Adhesive bonding application

- The path-related switching function `TRIGGER` triggers a defined statement.
- The statement `PATH` refers to the end point of the motion block.
- The statement is executed parallel to the robot motion.
- The switching point can be shifted in space and/or time.



The end point must be addressed by means of a CP motion (`LIN` or `CIRC`). The motion must not be `PTP`.

Function of path-related switching functions with `TRIGGER WHEN PATH`

Syntax

- `TRIGGER WHEN PATH=Distance DELAY=Time DO Statement <PRIO=Priority>`
- **Distance:** Defines the offset of the end point.
 - Positive value: shifts the statement towards the end of the motion.
 - Negative value: shifts the statement towards the start of the motion.
 - The distance is specified in millimeters (mm).
 - The specified distance can be up to +/- 10,000,000 mm.
 - If the specified value is too great or too small, switching is carried out at the switching limits.
- **Time:** The `PATH` specification defines a shift time relative to the selected position.
 - Positive and negative values can be used.
 - The time base is milliseconds (ms).
 - Times of up to 10,000,000 ms can be used without problems.
 - If the value set for the time is too great or too small, switching is carried out at the switching limits.
- **Statement:**
 - Assignment of a value to a variable



The value must not be assigned to a runtime variable.

- `OUT` statement
- `PULSE` statement
- Subprogram call. In this case, the *Priority* **must** be specified.
- **Priority** (only for a subprogram call):

- Priorities 1, 2, 4 to 39 and 81 to 128 are available.
- Priorities 40 to 80 are reserved for cases in which the priority is automatically assigned by the system. If the priority is to be assigned automatically by the system, the following is programmed: `PRIO = -1`.

Switching range

- Shift towards the end of the motion:

A statement can be shifted, **at most, as far as the next exact positioning point after TRIGGER WHEN PATH** (skipping all approximate positioning points).

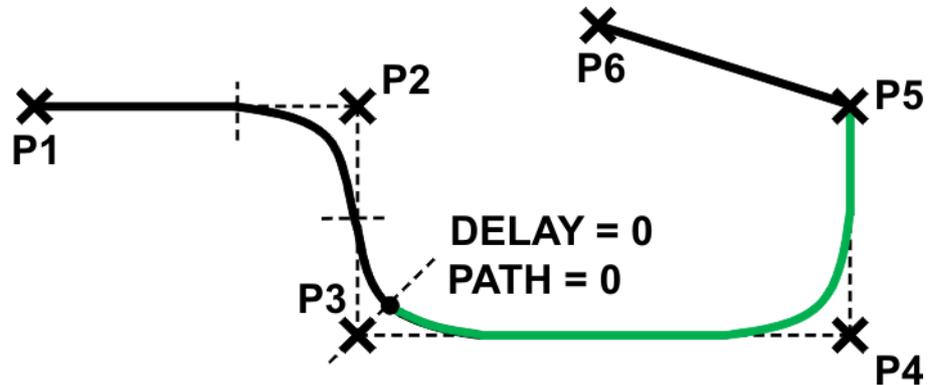


Fig. 8-13: TRIGGER WHEN PATH – switching limit for end of motion



In other words, if the end point is an exact positioning point, the statement cannot be shifted beyond the end point.

- Shift towards the start of the motion:

A statement can be shifted, **at most, as far as the start point of the motion block** (i.e. as far as the last point before TRIGGER WHEN PATH).

- If the start point is an exact positioning point, the statement can be shifted, at most, as far as the start point.

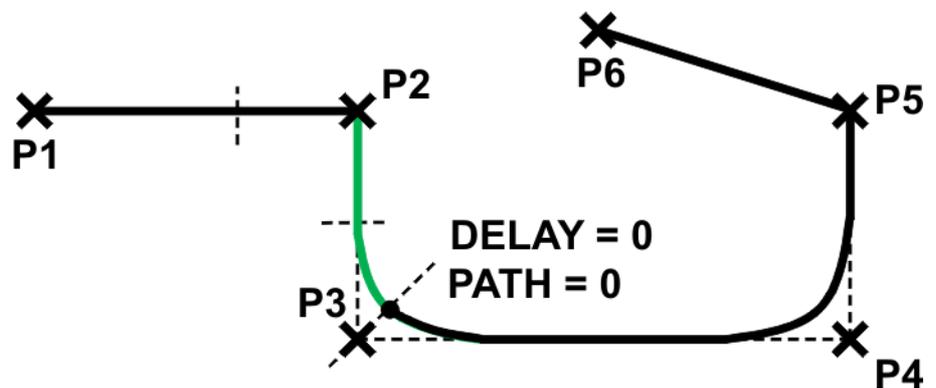


Fig. 8-14: TRIGGER WHEN PATH – switching limit for start point (exact positioning)

- If the start point is an approximated PTP point, the statement can be brought forward, at most, as far as the end of its approximate positioning arc.

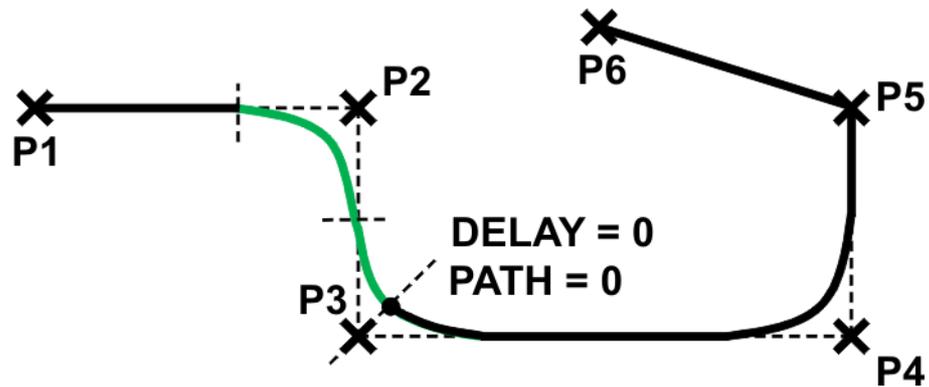


Fig. 8-15: TRIGGER WHEN PATH – switching limit for start point (approximated)

Programming of path-related switching functions

TRIGGER WHEN
PATH

- Switching towards the end of the motion

```

LIN XP2 C_DIS
TRIGGER WHEN PATH = Y DELAY = X DO $OUT[2] = TRUE
LIN XP3 C_DIS
LIN XP4 C_DIS
LIN XP5
LIN XP6
  
```

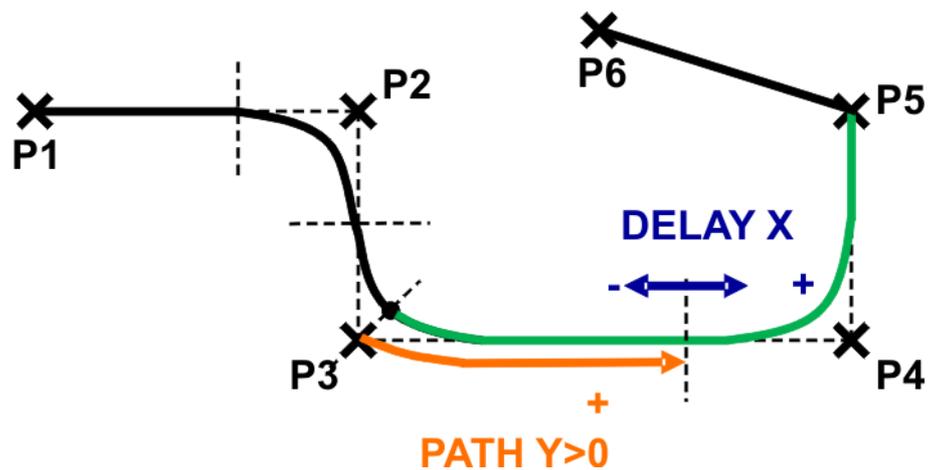


Fig. 8-16: TRIGGER WHEN PATH – switching towards the end of the motion

- Switching towards the start of the motion

```

LIN XP2 C_DIS
TRIGGER WHEN PATH = Y DELAY = X DO $OUT[2] = TRUE
LIN XP3 C_DIS
LIN XP4 C_DIS
LIN XP5
LIN XP6
  
```

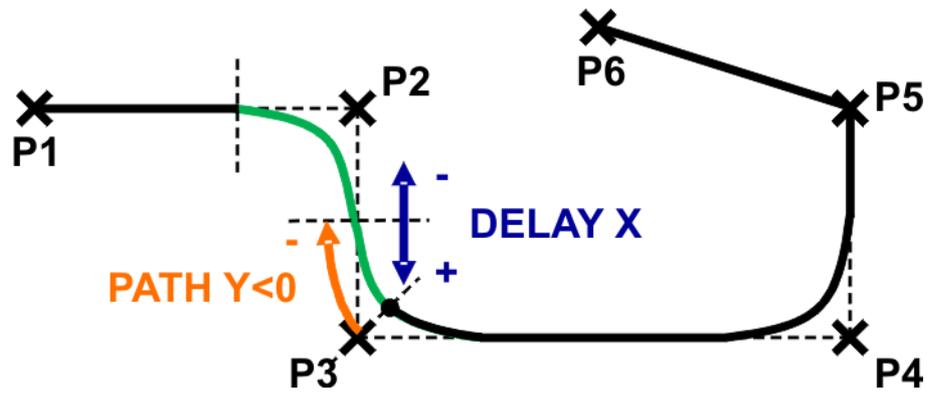


Fig. 8-17: TRIGGER WHEN PATH – switching towards the start of the motion

9 Programming with WorkVisual

9.1 Managing a project with WorkVisual

Project phases

1. Loading a project from the robot controller to WorkVisual
(>>> 9.1.1 "Opening a project with WorkVisual" Page 85)
2. Modifying a project, e.g. KRL programs
(>>> 9.2 "Editing KRL programs with WorkVisual" Page 98)
3. Comparing a project (merging)
(>>> 9.1.2 "Comparing projects with WorkVisual" Page 88)
4. Loading (deploying) a project from WorkVisual to the robot controller
(>>> 9.1.3 "Transferring a project to the robot controller (installing)" Page 92)
5. Project activation (enabling)
(>>> 9.1.4 "Activating a project on the robot controller" Page 96)

9.1.1 Opening a project with WorkVisual

Brief description of WorkVisual

The **WorkVisual** software package is the engineering environment for KR C4 controlled robotic cells. It offers the following functionalities:

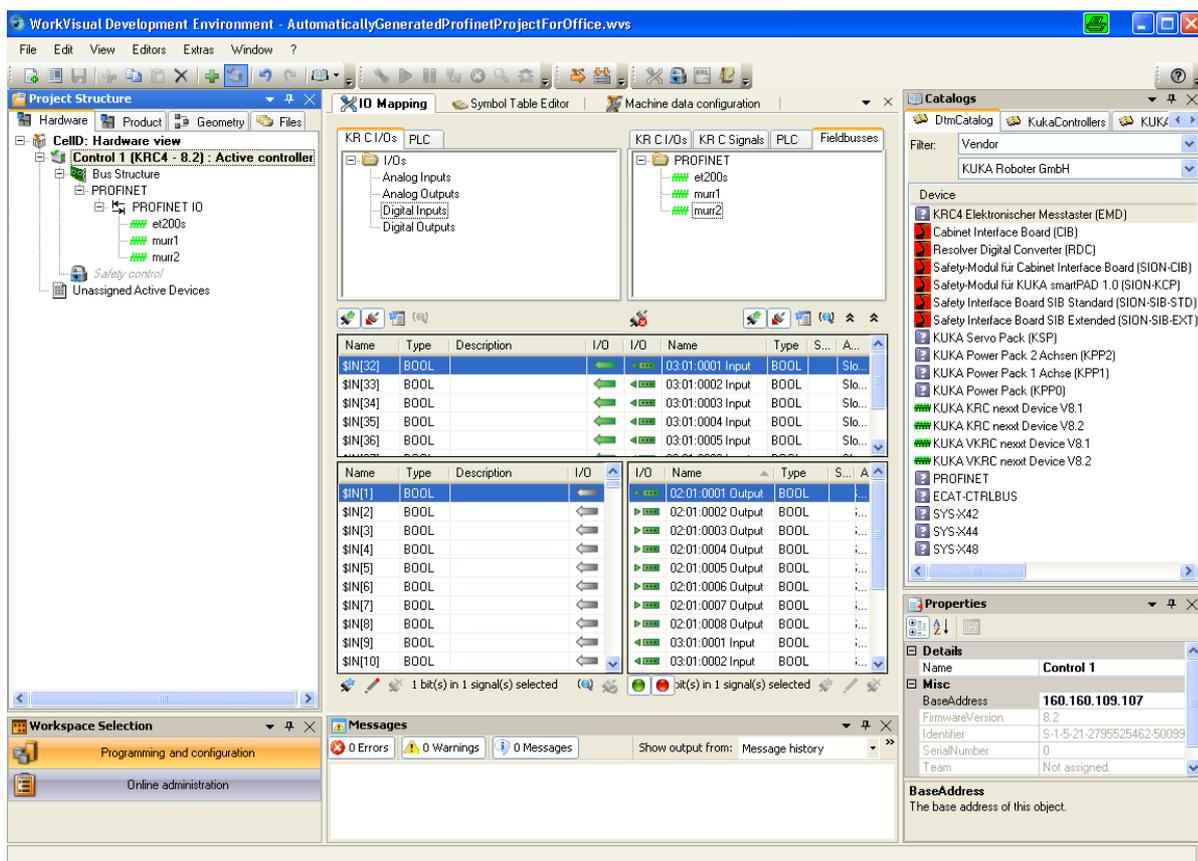


Fig. 9-1: WorkVisual graphical user interface

- Transferring projects from the robot controller to WorkVisual
- Comparing a project with another project and accepting differences where necessary
- Transferring projects to the robot controller

- Configuring and connecting field buses
- Editing the safety configuration
- **Programming robots offline**
- Managing long texts
- Diagnostic functionality
- Online display of system information about the robot controller
- Configuring traces, starting recordings, evaluating traces (with the oscilloscope)

Structure and function of the WorkVisual graphical user interface

Not all elements on the graphical user interface are visible by default, but they can be shown or hidden as required.

There are other windows and editors available in addition to those shown here. These can be displayed via the **Window** and **Editors** menus.

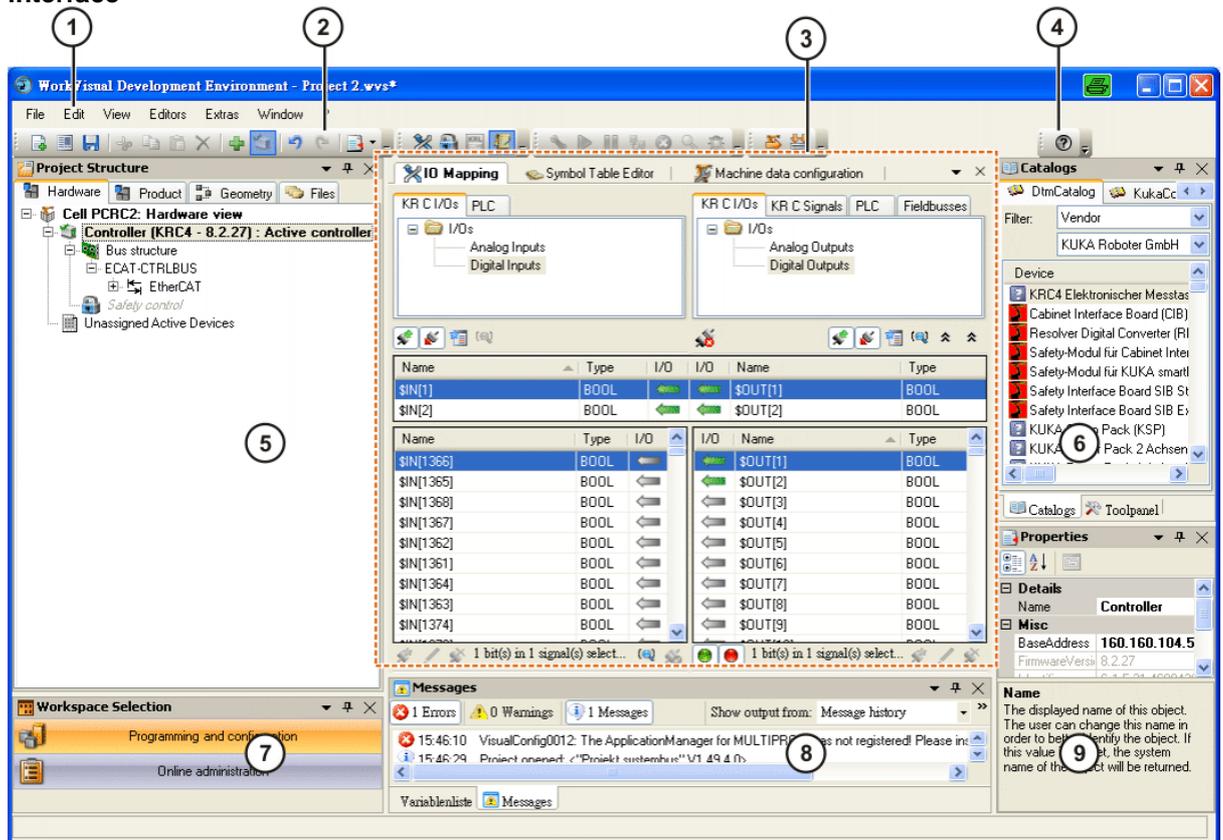


Fig. 9-2: Overview of the graphical user interface

Project Structure window



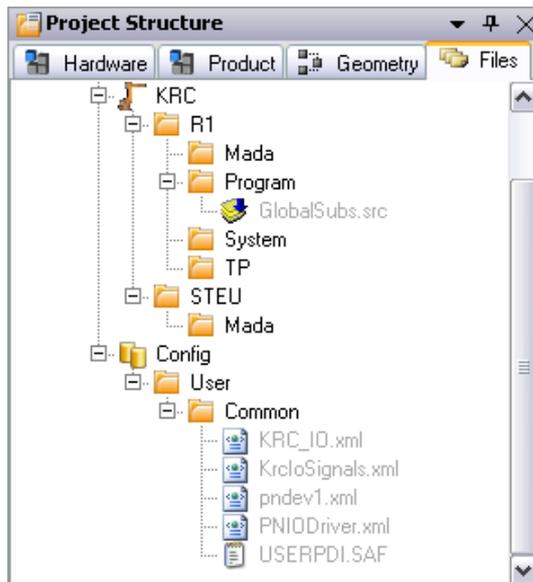


Fig. 9-3: Example: automatically generated files in gray

- **Hardware:**
The **Hardware** tab shows the relationship between the various devices. Here, the individual devices can be assigned to a robot controller.
- **Product:**
The **Product** tab is used mainly in WorkVisual Process and less in WorkVisual. This displays all the tasks required for a product in a tree structure.
- **Geometry:**
The **Geometry** tab is used mainly in WorkVisual Process and less in WorkVisual. This displays all the 3D objects used in the project in a tree structure.
- **Files:**
The **Files** tab contains the program and configuration files belonging to the project.
Coloring of file names:
 - Files generated automatically (with **Generate code** function): Gray
 - Files inserted manually in WorkVisual: Blue
 - Files transferred to WorkVisual from the robot controller: Black

Project Explorer

■

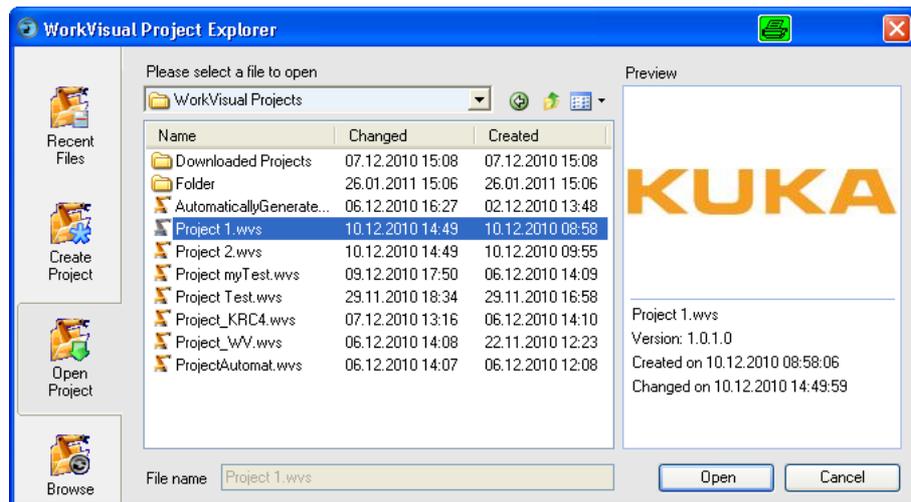


Fig. 9-4: Project Explorer

- **Recent Files** displays the most recently used files
- **Create Project** generates
 - a new, empty project
 - a new project using a template
 - a new project on the basis of an existing project
- **Open Project** is used to open existing projects
- **Browse** is required in order to load a project from the robot controller.

Procedure for loading a project with WorkVisual

On every robot controller **to which a network connection is established**, a project can be selected and transferred to WorkVisual.

This is also possible if this project is not yet present on this PC.

The project is saved in the directory: My Files\WorkVisual Projects\Downloaded Projects.

1. Select the menu sequence: **File > Browse for project**. The **Project Explorer** is opened. On the left, the **Search** tab is selected.
2. In the **Available cells** area, expand the node of the desired cell. All the robot controllers of this cell are displayed.
3. Expand the node of the desired robot controller. All projects are displayed.
4. Select the desired project and click on **Open**. The project is opened in WorkVisual.

9.1.2 Comparing projects with WorkVisual

Description

- A project in WorkVisual can be compared with another project.
- This can be a project on a robot controller or a locally saved project.
- The differences are clearly listed and detailed information can be displayed.
- For each individual difference, the user can decide:
 - whether the state should be left as in the current project
 - or whether the state from the other project should be applied.

Principle of project comparison

Merge projects

-

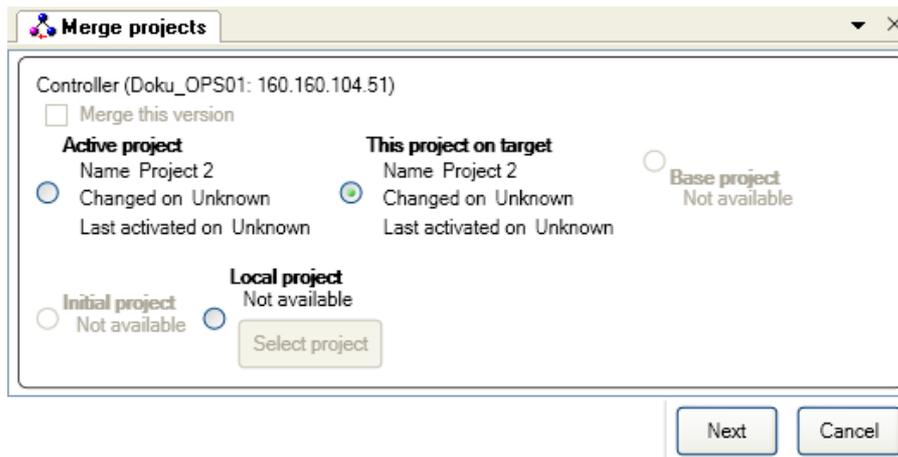


Fig. 9-5: Selecting a project for “Merge”

- Active project
- Project of the same name on the controller (only possible with network connection)
- Base project
- Initial project
- Local project (from notebook)

Comparison

The differences between the projects are displayed in an overview. For each difference, the user can select which state to accept.

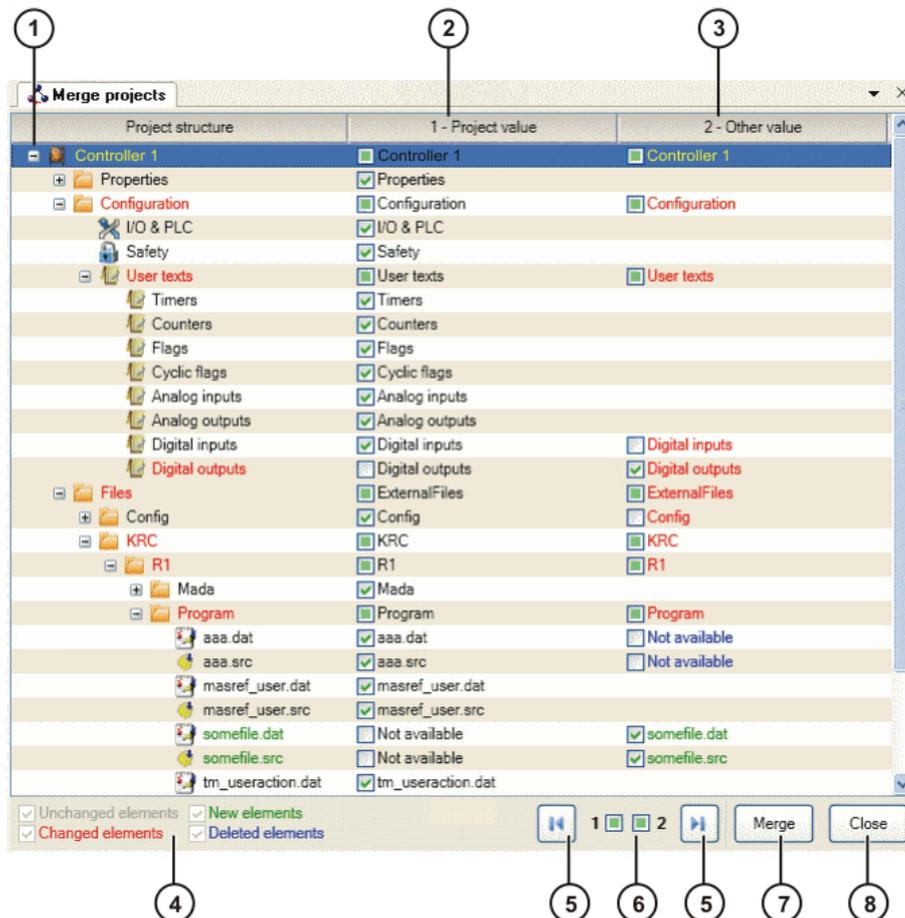


Fig. 9-6: Example: overview of differences

Description of the colors:

Column	Description
Project structure	Each element is displayed in the color that it has in the column in which it is selected.
Current project value (1)	All elements are displayed in black.
Comparison value (2)	<ul style="list-style-type: none"> ■ Green: Elements which are not present in the open project, but in the comparison project. ■ Blue: Elements which are present in the open project, but not in the comparison project. ■ Red: All other elements. These include higher-level elements which contain elements in various colors.

Procedure for project comparison

1. In WorkVisual, select the menu sequence **Extras > Compare projects**. The **Comparing projects** window is opened.

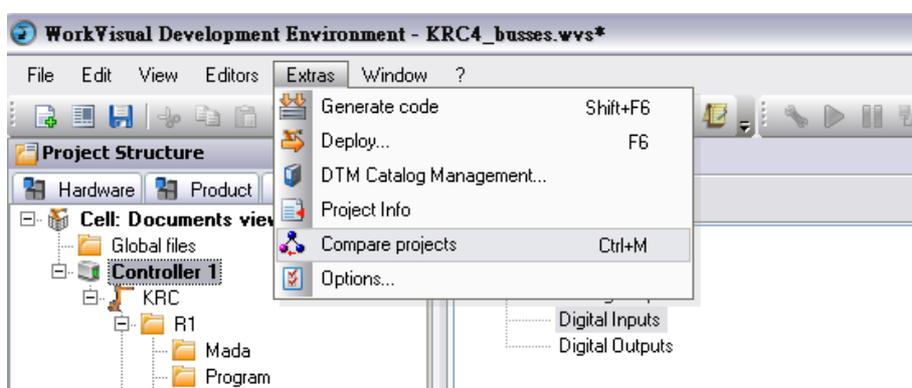


Fig. 9-7

2. Select the project with which the current WorkVisual project should be compared, e.g. the project of the same name on the real robot controller.

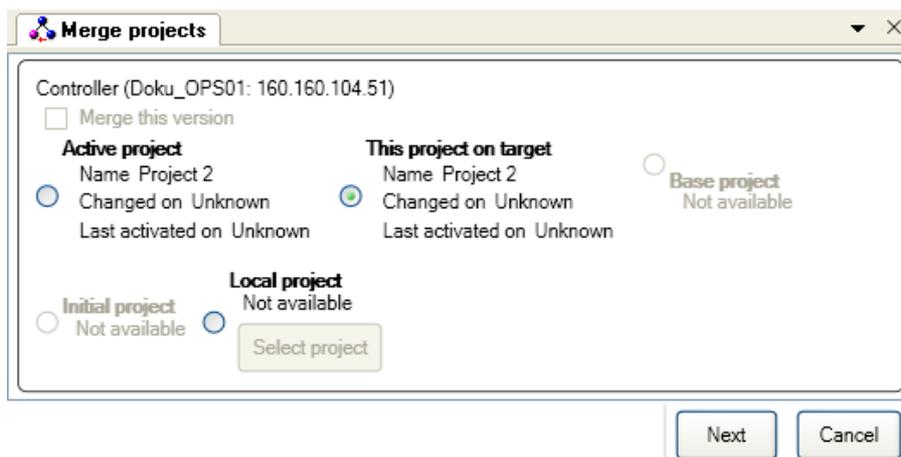


Fig. 9-8: Selecting a project for “Merge”

3. Click on **Next**. A progress bar is displayed. (If the project contains more than one controller, a bar is displayed for each one.)
4. When the progress bar is full and the message **Status: Ready for merge** is displayed: Click on **Show differences**. The differences between the projects are displayed in an overview.
If no differences were determined, this is indicated in the message window. Continue with step 8. After this, no further steps are necessary.

5. For each difference, select which state to accept. This does not have to be done for all the differences at one go.
If suitable, the default selection can also be accepted.
6. Press **Merge** to transfer the changes to WorkVisual.
7. Repeat steps 5 to 6 as many times as required. This makes it possible to work through the different areas bit by bit.

When there are no more differences left, the following message is displayed: **No further differences were detected.**

8. Close the **Comparing projects** window.
9. If parameters of external axes have been changed in the project on the robot controller, these must now be updated in WorkVisual:
 - Open the **Machine data configuration** window for this external axis.
 - In the area **General axis-specific machine data**, click on the button for importing machine data.
The data are updated.
10. Save the project.

Example of a project comparison

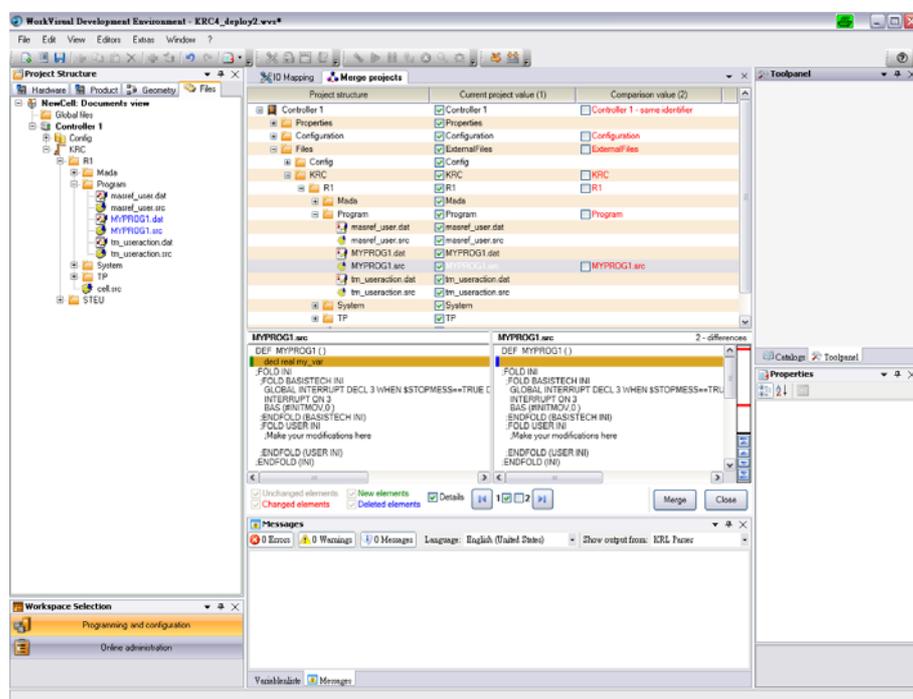


Fig. 9-9: Example: Overview of project comparison

Specifying which state of the file(s) is to be accepted

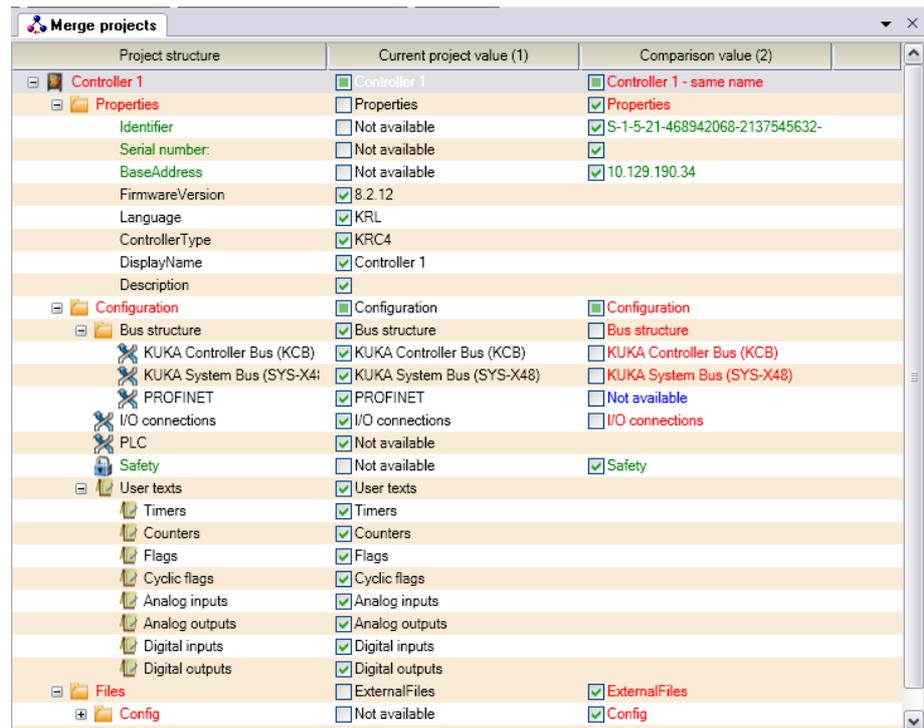


Fig. 9-10: Example: Merge projects

The differences between the files can be displayed by activating the **Details**.

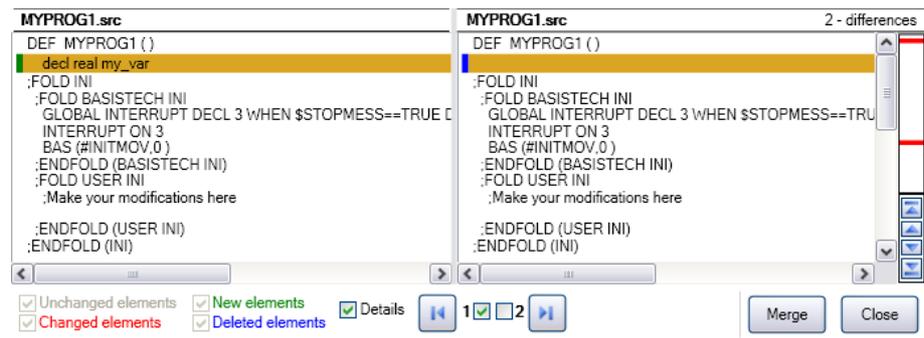


Fig. 9-11: Example: Details activated

9.1.3 Transferring a project to the robot controller (installing)

Description

- After making changes to the project, it must be transferred from WorkVisual to the controller
- This procedure is called **deployment** by KUKA
- When a project is transferred to the robot controller, the code is always generated first
- A network connection to the real robot controller is required for **deployment**



If a project was transferred to the real robot controller at an earlier time and has not yet been activated then this will be overwritten if a further project is transferred.

Transferring and activating a project overwrites a project of the same name that already exists on the real robot controller (after a request for confirmation).

Functions

Generate code

- This procedure can be used to generate the code separately and thus to check in advance whether generation runs without error.
- The function is called via
 - the menu sequence **Extras > Generate code**
 - or the button 
 - The code is displayed on the **Files** tab of the **Project structure** window.

Automatically generated code is displayed in pale gray.

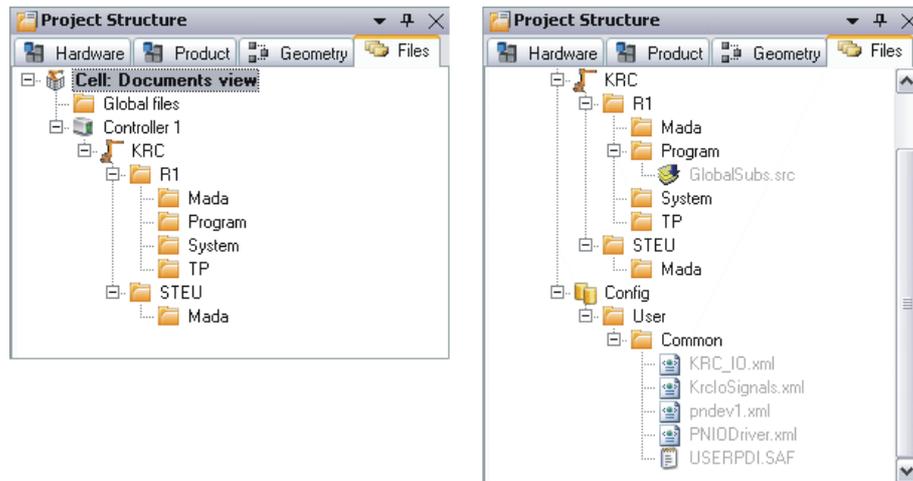


Fig. 9-12: Example of code generation: before – after

- The code is generated. When the process is finished, the following messages are displayed in the message window: **The project <"{0}" V{1}> has been compiled. The results can be seen in the file tree.**

Instructions

1. Click on the **Deploy...** button in the menu bar. The **Project deployment** window is opened.

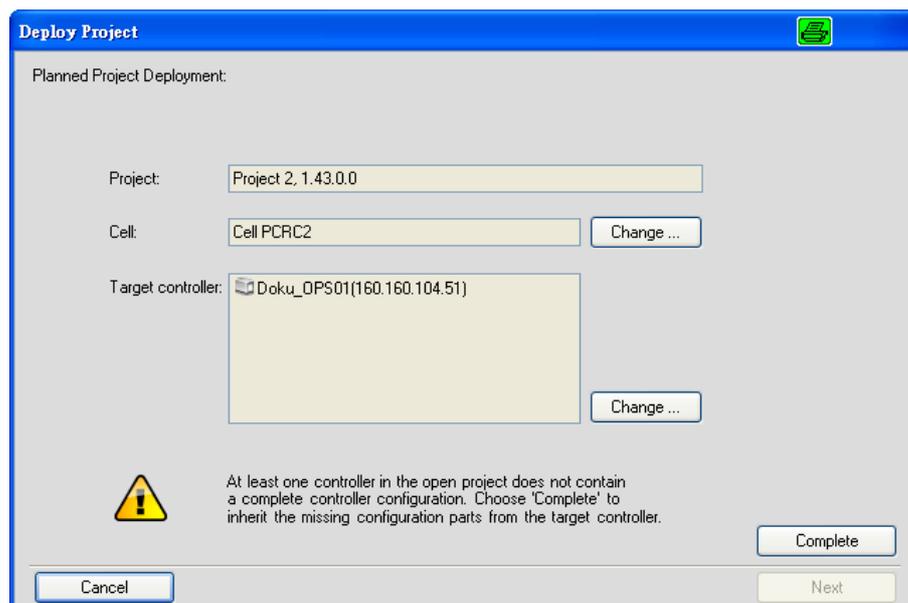


Fig. 9-13: Overview with warning about incomplete configuration

2. If the project has never been transferred back to WorkVisual from a robot controller before, it will not yet contain all the configuration files. This is in-

licated by a message. (The configuration files include machine data files, safety configuration files and many others.)

- If this message is not displayed: Continue with step 13.
 - If this message is displayed: Continue with step 3.
3. Click on **Complete**. The following confirmation prompt is displayed: **The project must be saved and the active controller will be reset! Do you want to continue?**
 4. Answer the query with **Yes**. The **Merge projects** window is opened.

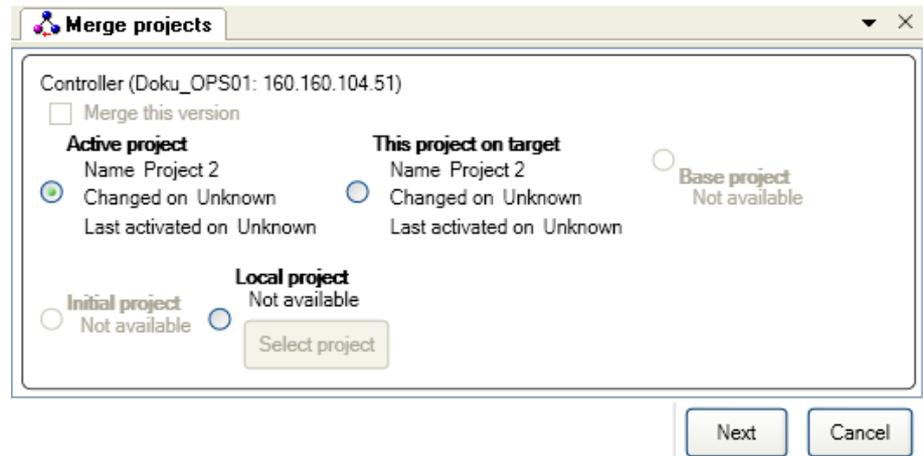


Fig. 9-14: Selecting a project for “Complete”

5. Select a project from which the configuration data are to be transferred, e.g. the active project on the real robot controller.
6. Click on **Next**. A progress bar is displayed. (If the project contains more than one controller, a bar is displayed for each one.)
7. When the progress bar is full and the message **Status: Ready for merge** is displayed: Click on **Show differences**.
The differences between the projects are displayed in an overview.
8. For each difference, select which state to accept. This does not have to be done for all the differences at one go.
If suitable, the default selection can also be accepted.
9. Press **Merge** to transfer the changes.
10. Repeat steps 8 to 9 as many times as required. This makes it possible to work through the different areas bit by bit.
When there are no more differences left, the following message is displayed: **No further differences were detected**.
11. Close the **Comparing projects** window.
12. Click on the button **Deploy...** in the menu bar. The overview of the cell assignment is displayed again. The message about the incomplete configuration is no longer displayed.

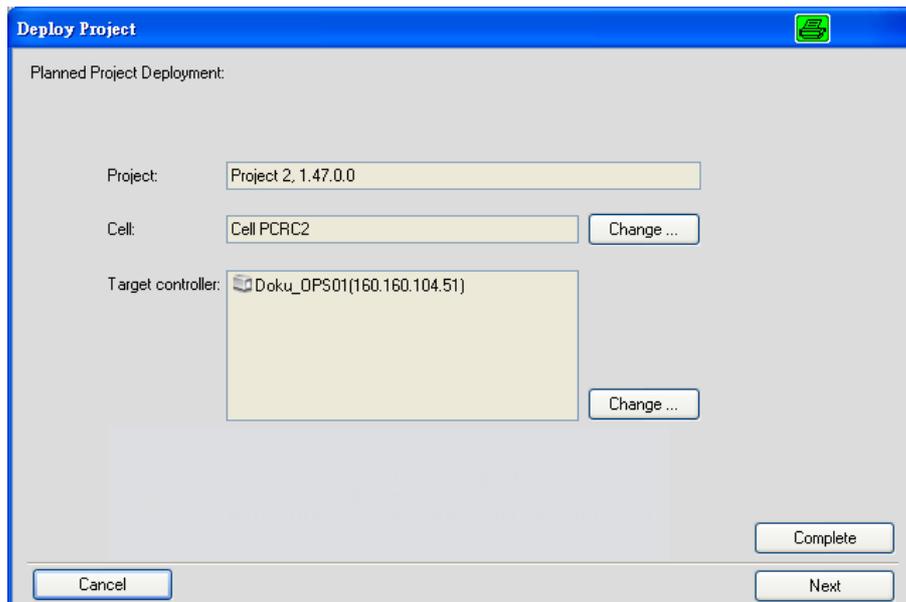


Fig. 9-15: Overview

13. Click on **Next**. Program generation begins. When the progress indicator bar reaches 100%, the program is generated and the project is transferred.
14. Click on **Activate**.

WARNING In the operating modes AUT and AUT EXT, the project is activated without any request for confirmation if there are only program changes.

15. Only in operating modes T1 and T2: The KUKA smartHMI displays the request for confirmation *Do you want to activate the project [...]?*. In addition, a message is displayed as to whether the activation would overwrite a project, and if so, which.
If no relevant project will be overwritten: Confirm with **Yes** within 30 minutes.
16. An overview is displayed of the changes which will be made in comparison to the project that is still active on the robot controller. The check box **Details** can be used to display details about the changes.
17. The overview displays the request for confirmation *Do you want to continue?*. Confirm with **Yes**. The project is activated on the robot controller. A confirmation is displayed in WorkVisual.

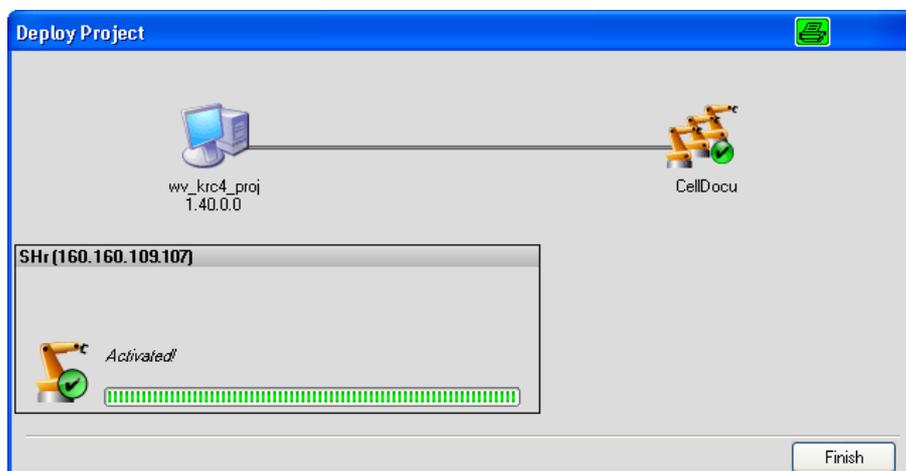


Fig. 9-16: Confirmation in WorkVisual

18. Close the **Project deployment** window by selecting **Finish**.
19. If the request for confirmation on the robot controller is not answered within 30 minutes, the project is still transferred, but is not activated on the robot controller. The project can then be activated separately.

WARNING After activation of a project on the robot controller, the safety configuration must be checked there! If this is not done, the robot will possibly be operated with incorrect data. Death to persons, severe physical injuries or considerable damage to property may result.

WARNING If the activation of a project fails, an error message is displayed in WorkVisual. In this case, one of the following measures must be carried out:

- Either: Activate a project again (the same one or a different one).
- Or: Reboot the robot controller with a cold restart.

9.1.4 Activating a project on the robot controller

Description

- A project can be activated directly on the robot controller.
- A project can also be activated on the robot controller from within WorkVisual.
(not described here; for this, see WorkVisual online documentation)

Project management function

General

- The robot controller offers a function for managing a number of projects on the controller
- All functions are only available in the user group **Expert** or higher.
- The function is called via:
 - the menu sequence **File > Project management**
 - on the user interface via the **WorkVisual symbol** button, followed by the **Open** button

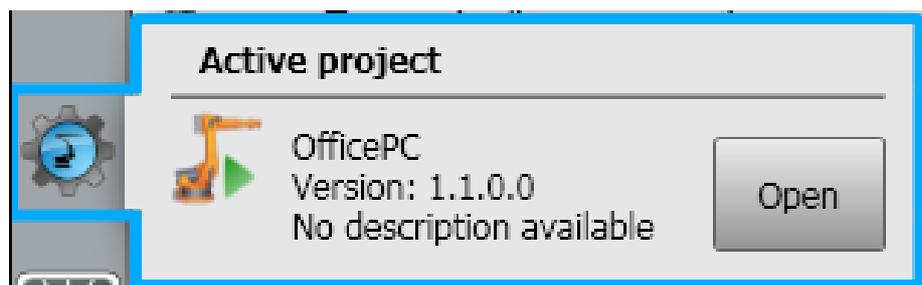


Fig. 9-17: Project display on the user interface

Use / operation

-

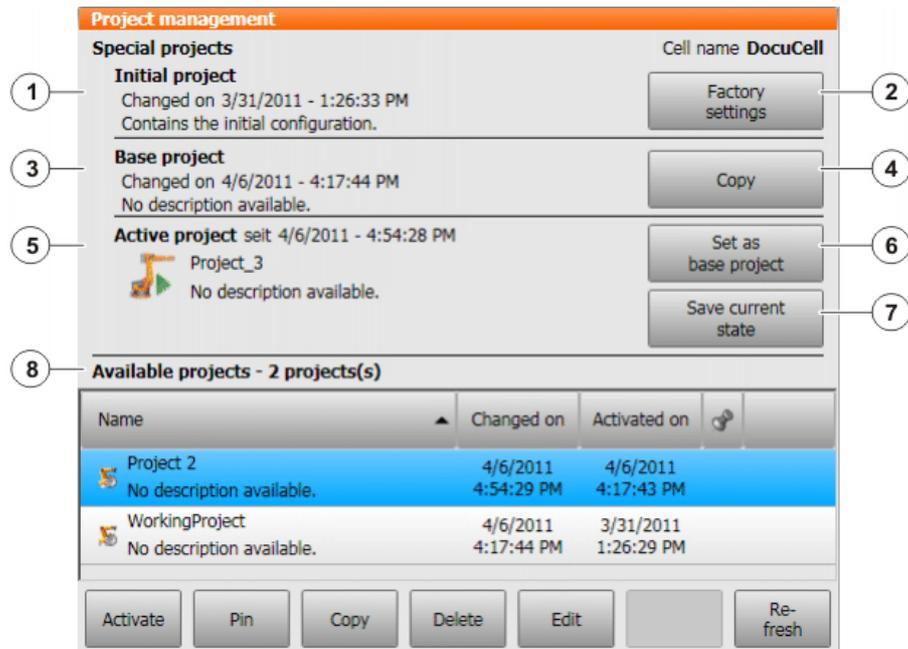


Fig. 9-18: “Projects management” window

- In addition to the regular projects, the **Project management** window contains the following special projects:

Project	Description
Initial project	The initial project is always present. It cannot be changed by the user. It contains the initial state of the robot controller as shipped.
Base project	<p>The user can save the active project as the base project. This functionality is generally used to save a functional, tried-and-tested project state.</p> <p>The base project cannot be activated, but copied. The base project can no longer be changed by the user. It can, however, be overwritten by saving a new base project (after a request for confirmation).</p> <p>If a project is activated which does not contain all the configuration files, the missing information is inserted from the base project. This is the case e.g. if a project is activated from an earlier version of WorkVisual. (The configuration files include machine data files, safety configuration files and many others.)</p>

- Description of buttons

Button	Description
Activate	<p>Activates the selected project.</p> <p>If the selected project is pinned: Creates a copy of the selected project. (A pinned project cannot be activated itself, only a copy of it.) The user can then decide whether to activate this copy or whether the current project should remain active.</p>
Pin	Pinned projects cannot be changed, activated or deleted. They can be copied or unpinned, however. A project can thus be pinned e.g. to prevent it from being accidentally deleted.
Unpin	Unpins the project.
Copy	Copies the selected project.

Button	Description
Delete	Deletes the selected project.
Edit	Opens a window in which the name and/or description of the selected project can be changed.
Refresh	Refreshes the project list. This enables e.g. projects to be displayed which have been transferred to the robot controller since the display was opened.

Procedure

 **Restriction:** If the activation causes changes in the **Communication parameter** area of the safety configuration, the user group "Safety recovery" or higher must be selected.
If the operating mode AUT or AUT EXT is selected: The project can only be activated if this affects only KRL programs. If the project contains settings that would cause other changes, it cannot be activated.

1. Select the menu sequence **File > Project management**. The **Project management** window is opened.
2. Select the desired project and activate it using the **Activate** button.
3. The KUKA smartHMI displays the request for confirmation *Do you want to activate the project [...]?*. In addition, a message is displayed as to whether the activation would overwrite a project, and if so, which.
If no relevant project will be overwritten: Confirm with **Yes** within 30 minutes.
4. An overview is displayed of the changes which will be made in comparison to the project that is still active on the robot controller. The check box **Details** can be used to display details about the changes.
5. The overview displays the request for confirmation *Do you want to continue?*. Confirm with **Yes**. The project is activated on the robot controller.

 **WARNING** After activation of a project on the robot controller, the safety configuration must be checked there! If this is not done, the robot will possibly be operated with incorrect data. Death to persons, severe physical injuries or considerable damage to property may result.

9.2 Editing KRL programs with WorkVisual

- File handling
(>>> 9.2.1 "File handling" Page 98)
- Working with the KRL Editor
(>>> 9.2.2 "Working with the KRL Editor" Page 104)

9.2.1 File handling

Description

- Load existing file into the KRL Editor.
- Add new file from catalog.
- Add external file.

Principle of templates from catalog

- Before templates can be used, the corresponding catalog must be loaded.
- Load via **File > Add catalog** and activate the corresponding templates.
 - KRL templates: *KRL Templates.afc*
 - VW templates: *VW Templates.afc*

- KRL templates catalog

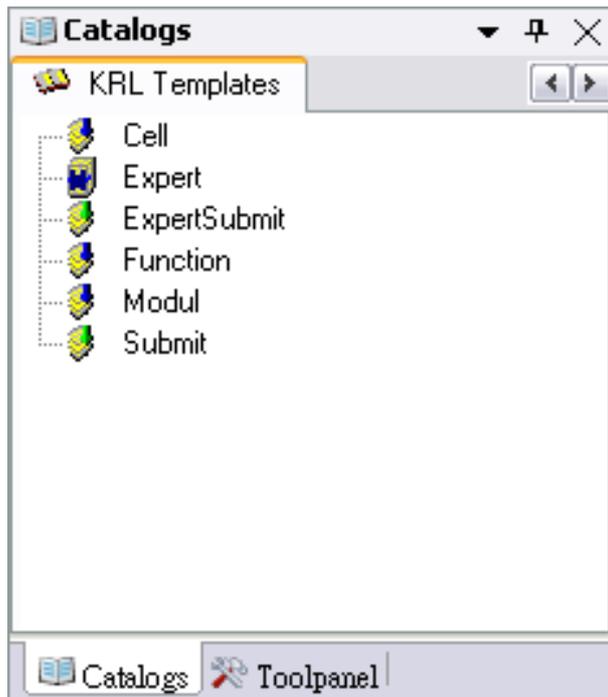


Fig. 9-19: Catalog for KRL templates

Handling instructions

Procedure for opening a file (SRC/DAT) in the KRL Editor

1. Switch to “Files” in the project tree.

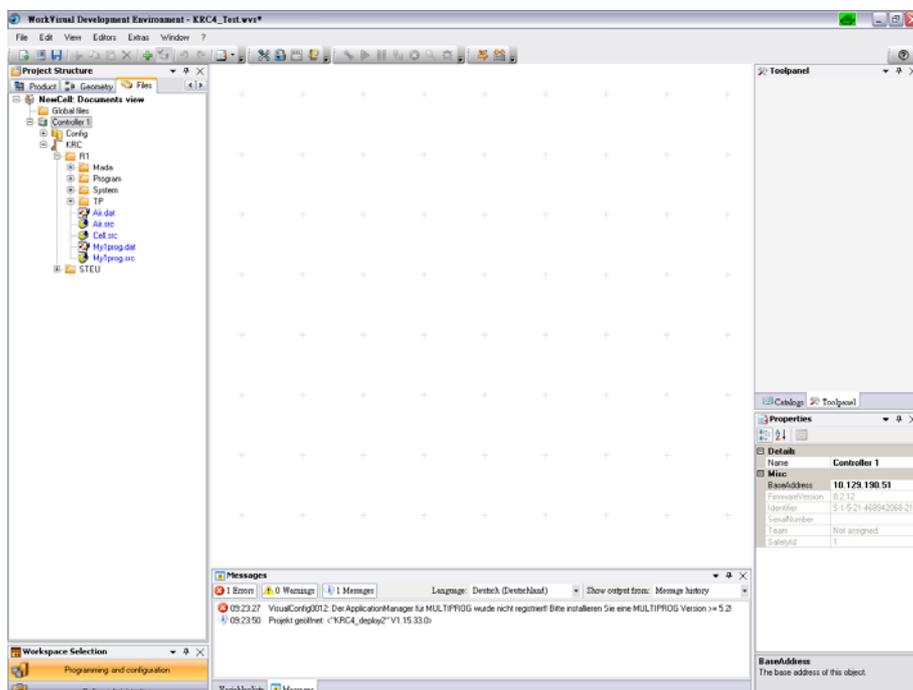


Fig. 9-20: WorkVisual project tree

2. Open the directories as far as directory R1.

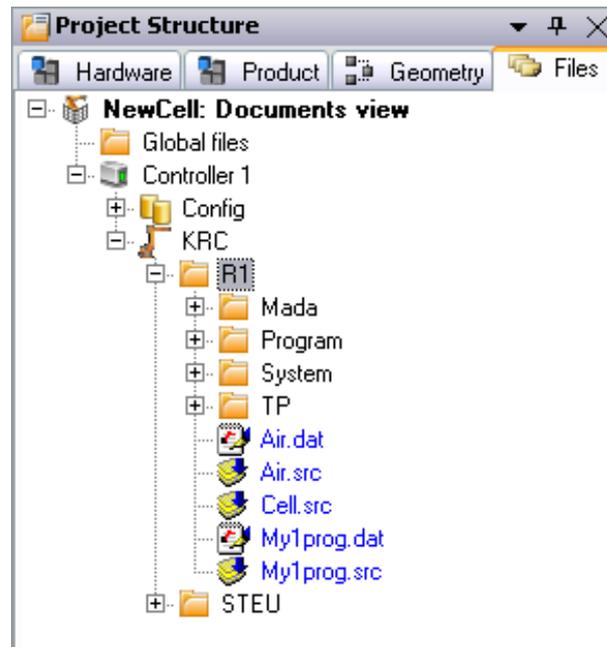


Fig. 9-21: WorkVisual project tree file structure (R1)

3. Select file by means of:

- Double-click.
- Toolbar button 
- Right-click and select  (KRL Editor) in the context menu.

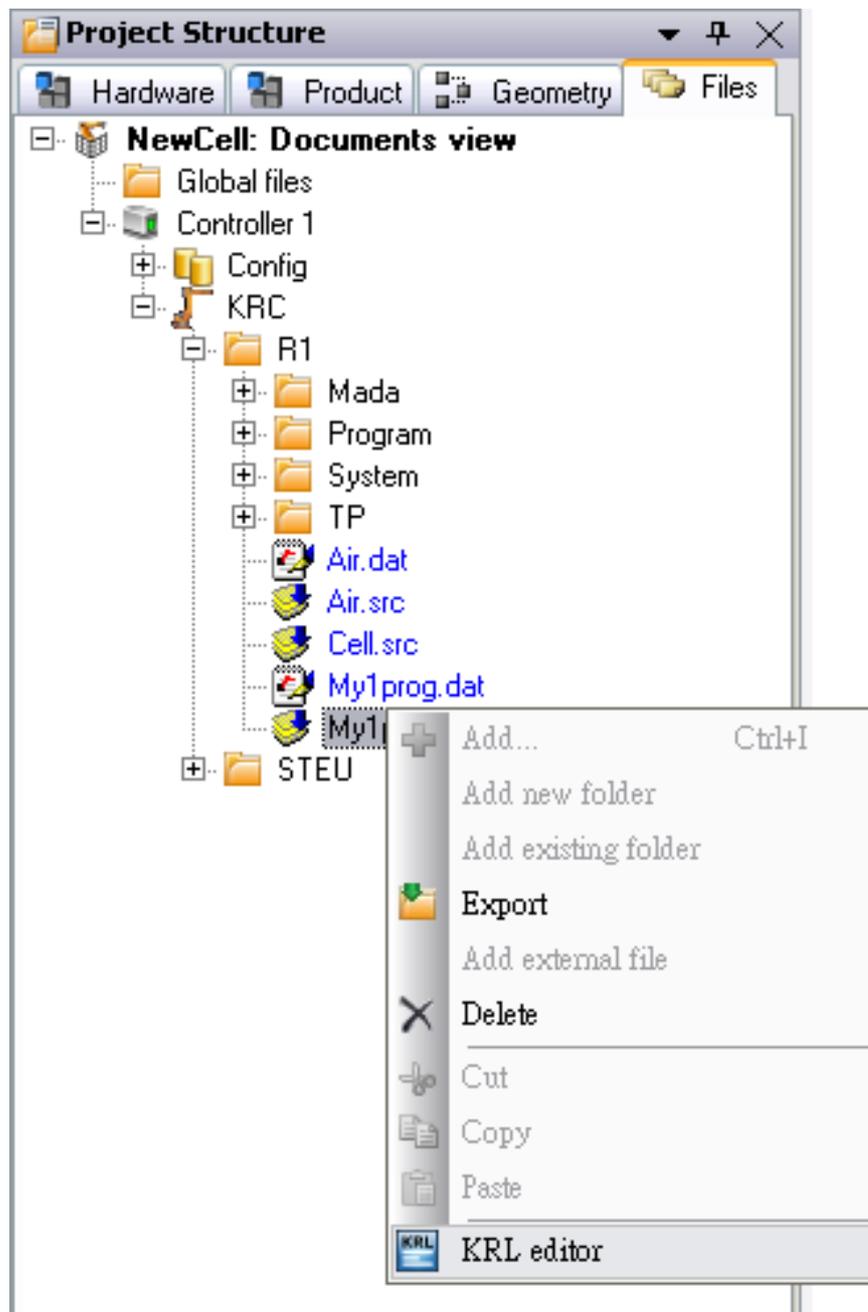


Fig. 9-22: WorkVisual – Context menu (KRL Editor)

Procedure for adding a file by means of KRL templates

1. Switch to “Files” in the project tree.
2. Open the directories as far as directory R1.
3. Select the folder in which the new file is to be created.
4. Right-click and select  (Add) in the context menu.

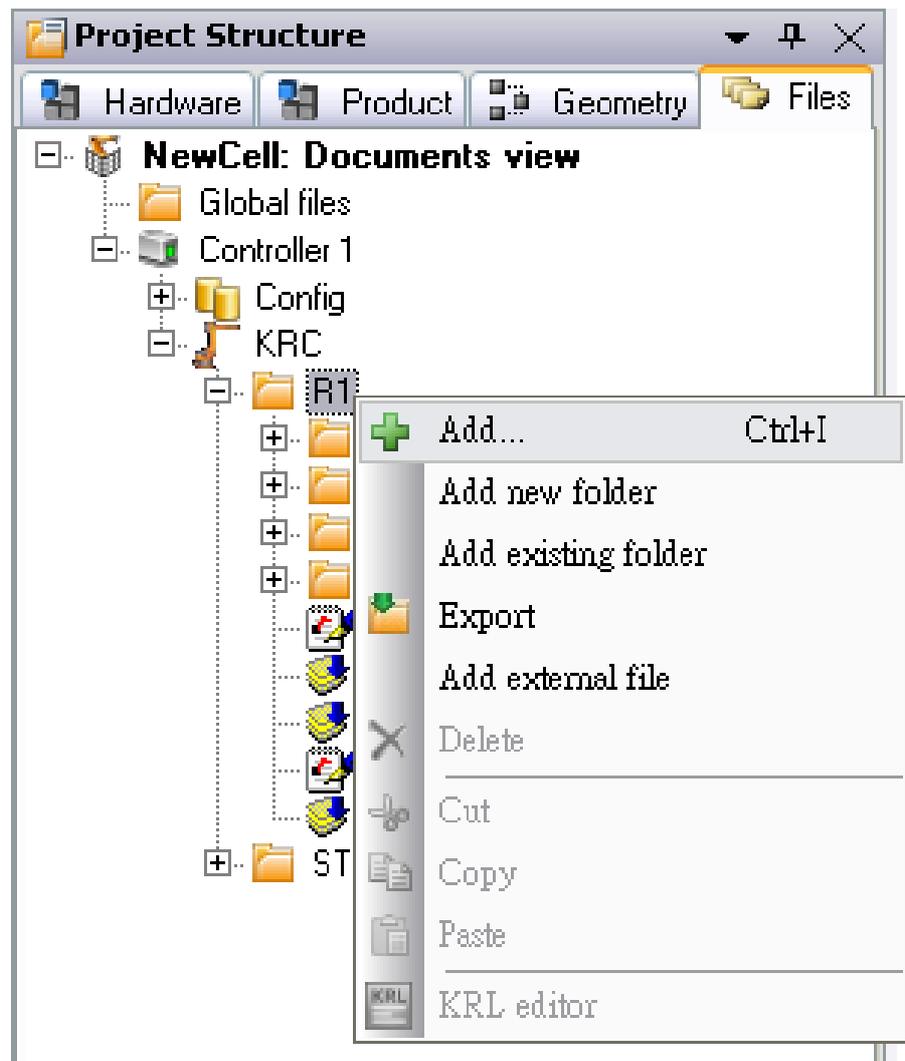


Fig. 9-23: WorkVisual – Context menu (Add)

5. Select template.

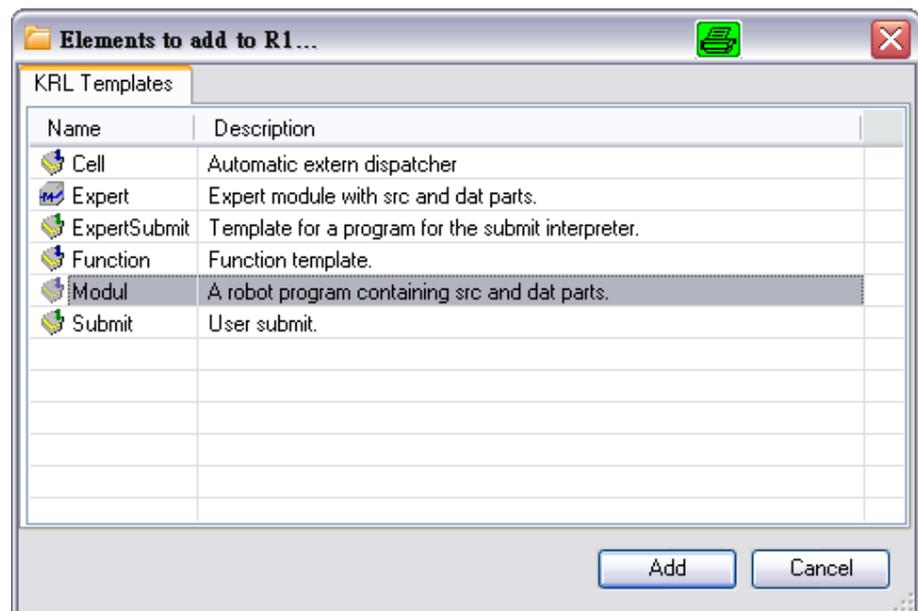


Fig. 9-24: WorkVisual – KRL templates

6. Assign program name.

Procedure for adding an external file

1. Switch to “Files” in the project tree.
2. Open the directories as far as directory R1.
3. Select the folder in which the new file is to be created.
4. Right-click and select “Add external axis file” in the context menu.

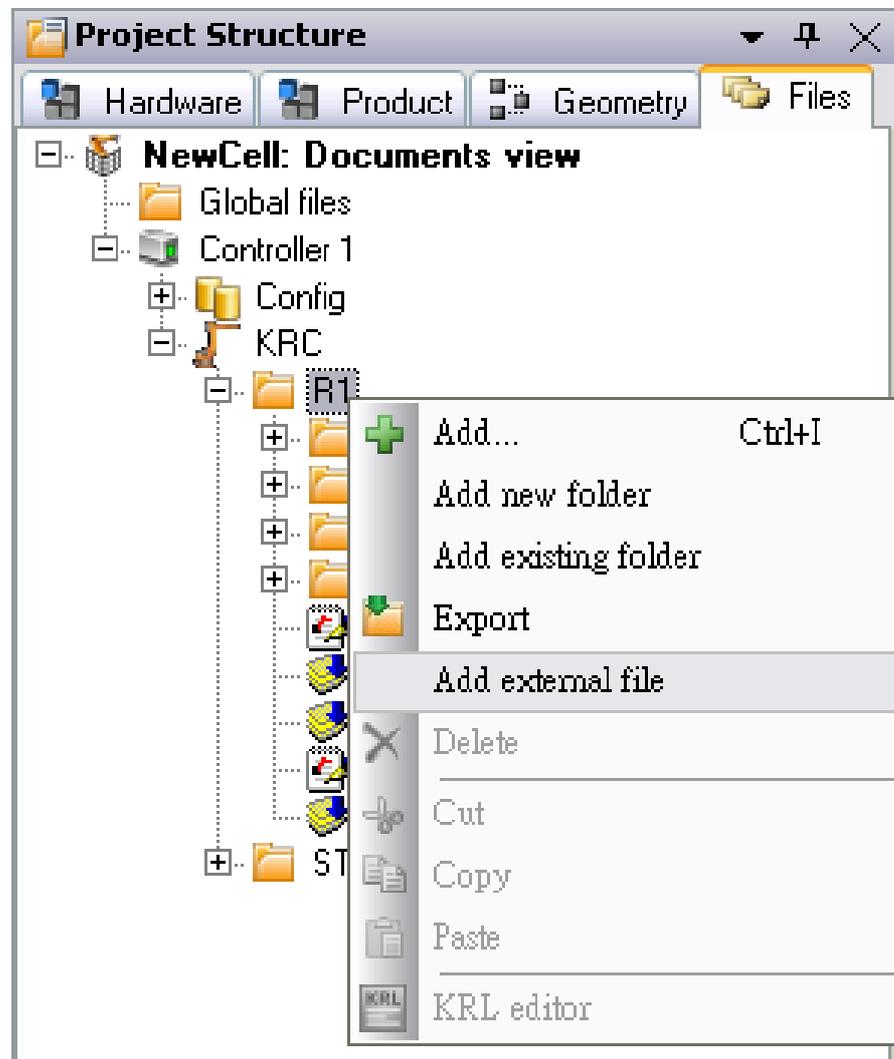


Fig. 9-25: WorkVisual – Context menu (Add external file)

5. Select file(s) and press “Open”.

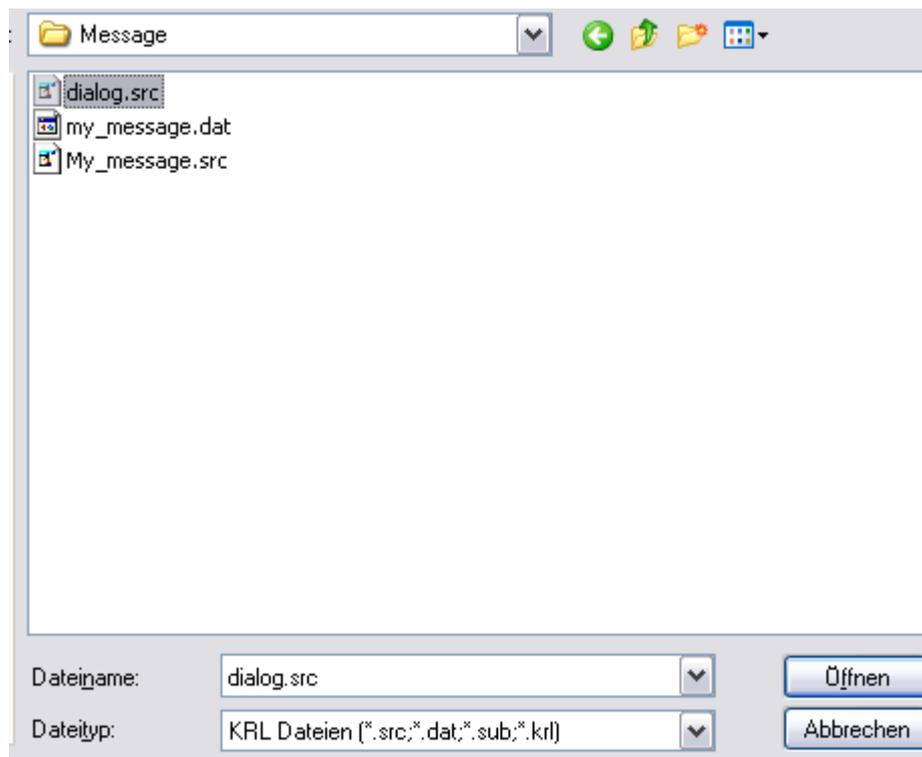


Fig. 9-26: WorkVisual – Add external file

9.2.2 Working with the KRL Editor

Description of KRL Editor

Program editing (SRC/DAT)

- By means of direct KRL entry
- By means of fast entry for KRL instructions (KRL snippets)
- By means of inline forms from the tool panel

Properties of the KRL Editor

- Configuration of the Editor
- Color description in the KRL Editor
- Error detection (KRL Parser)
- Variable list
- Additional editing functions in the KRL Editor

Configuring the KRL Editor

- Select the menu sequence **Extras > Options**. The **Options** window is opened.
- The **Text editor** folder contains the subitems **Appearance** and **Behavior**.
- Appearance

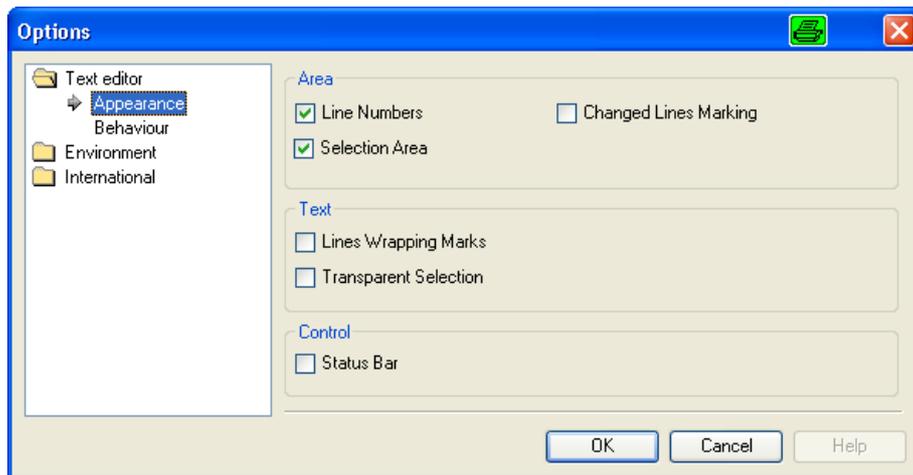


Fig. 9-27: Appearance

Appearance:

Box	Description
Line numbers	Activated: Display line numbers.
Selection area	Activated: Selected code is additionally marked by a vertical red bar on the left-hand side.
Changed lines marking	Activated: Changed lines are marked in yellow at the beginning of the line.
Line wrapping marks	Only relevant if the Word Wrap check box is activated under Behavior . <ul style="list-style-type: none"> ■ Activated: Line breaks are indicated by a small green arrow. ■ Deactivated: Line breaks are not indicated.
Transparent selection	Representation of selected code: <ul style="list-style-type: none"> ■ Activated: Text in original color on light background ■ Deactivated: White text on dark background
Status bar	Activated: A status bar is displayed at the bottom of the KRL Editor. It displays e.g. the program name and the number of the line in which the cursor is currently positioned.

- Behavior

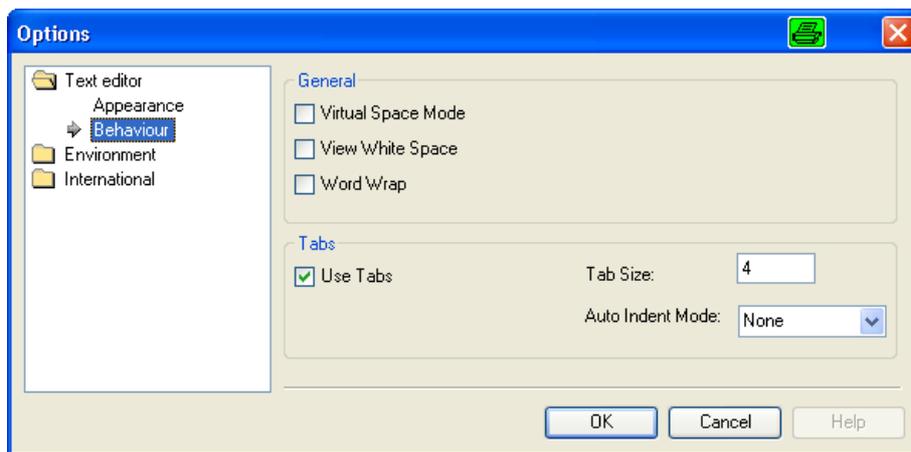


Fig. 9-28: Behavior

Behavior:

Box	Description
Virtual Space mode	<ul style="list-style-type: none"> Activated: The cursor can be placed at any position in a blank line. Deactivated: The cursor can only be placed at the beginning of a blank line.
View White Space	Activated: Control characters are displayed. (Spaces, tabs, etc.)
Word wrap	<ul style="list-style-type: none"> Activated: Lines are broken in accordance with the window width. Deactivated: Lines are not broken. If any lines are longer than the window width, a scroll bar is automatically displayed.
Use Tabs	<ul style="list-style-type: none"> Activated: The tab key inserts a tab. Deactivated: The tab key inserts the number of spaces specified under Tab Size.
Tab Size	The size of a tab is equivalent to x spaces.
Auto Indent Mode	Behavior when making a new line using the Enter key: <ul style="list-style-type: none"> None: The new line is not indented. Block: The new line has the same level of indentation as the preceding line. Smart: Behavior within a fold <ul style="list-style-type: none"> If the previous line is indented, this indentation is used for the new line. If the previous line is not indented, the new line is indented.

Color description of the KRL Editor

■ Description of the colors:

The KRL Editor recognizes the different elements of the code entered and automatically displays them in different colors.

Code element	Color
KRL keywords (except ; FOLD and ; ENDFOLD)	Medium blue
; FOLD and ; ENDFOLD	Gray
Figures	Dark blue

Code element	Color
Strings (text in quotation marks "...")	Red
Comments	Green
Special characters	Blue-green
Other code	Black

- Example of the use of colors

```

15
16 ;FOLD PTP HOME Vel= 100 % DEFAULT ;%{PE}%MKUKATPBASIS
17 $BWDSTART = FALSE
18 PDAT_ACT=PDEFAULT
19 FDAT_ACT=FHOME
20 BAS (#PTP_PARAMS,100 )
21 $H_POS XHOME
22 PTP XHOME
23 ;ENDFOLD
24

```

Fig. 9-29: Example: Colors in the KRL Editor

1	KRL keywords: Blue
2	Comment: Green
3	FOLD: Gray
4	Other code: Black

KRL Editor – Error detection

- The KRL Editor has an automatic error detection function.
- Detected errors in the programming code are underlined with a wavy red line.
- The errors are only visible in the message window if the category **KRL Parser** is selected.
- KRL errors and some structural errors are detected (declaration at the wrong point in the program)
- Typing errors in variables are not detected.



The error detection function does not detect all errors. If nothing is underlined, this is no guarantee that the program is free from errors.

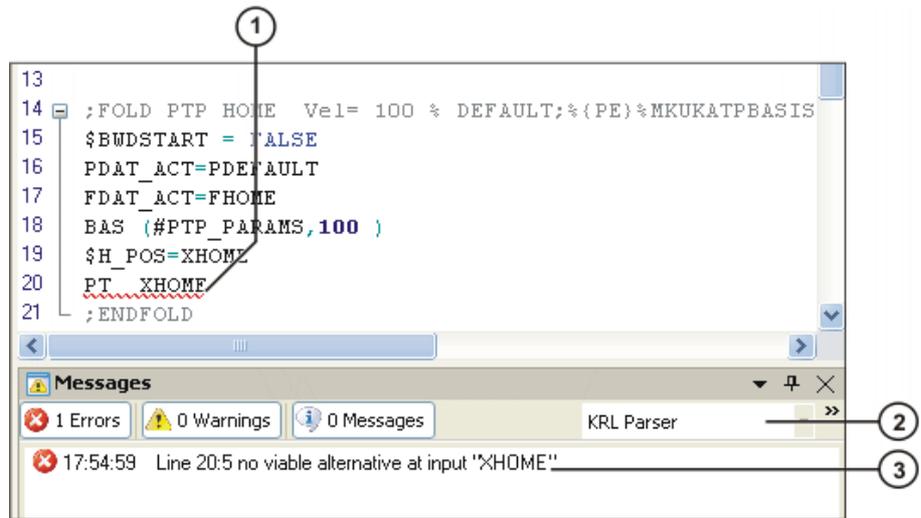


Fig. 9-30: Example: error detection

1	KRL Editor: error underlined with a wavy red line
2	Message window: category KRL Parser is selected.
3	Message window: error description with line and column number

Function of the variable list

- All the KRL variables that are declared in a particular file can be displayed in a list overview.

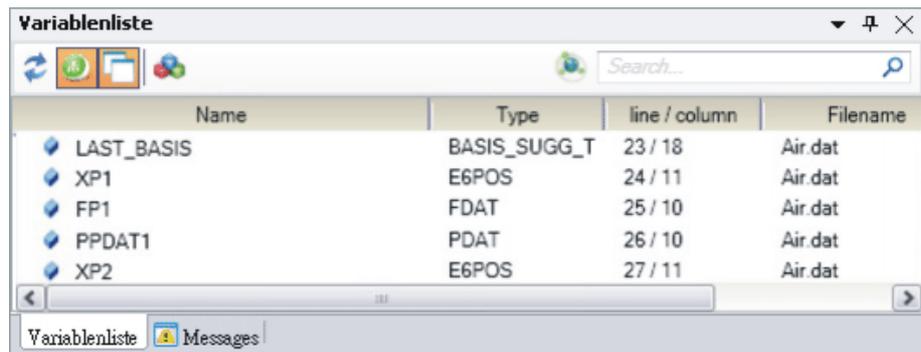


Fig. 9-31: Example of a variable list

- With SRC files, the variables from the corresponding DAT file are always displayed at the same time, and vice versa.

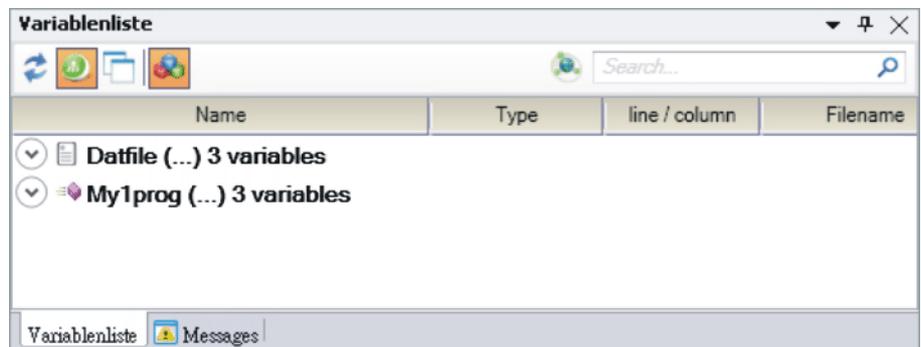


Fig. 9-32: Example of a variable list (variables from SRC and DAT)

- Enter the variable name or part of the name in the search box. The search results are displayed immediately.

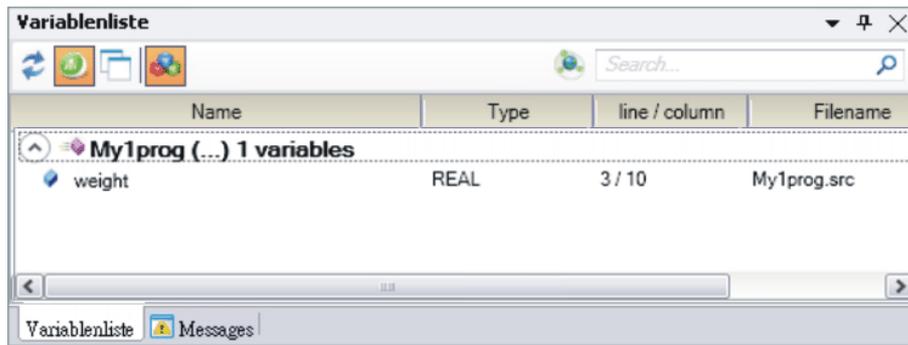


Fig. 9-33: Example of a variable list (search function)



If the search box contains a search term, this remains valid if the focus is moved to another file. It is only possible to display all the variables in a file if the search box is empty.

- Setting options

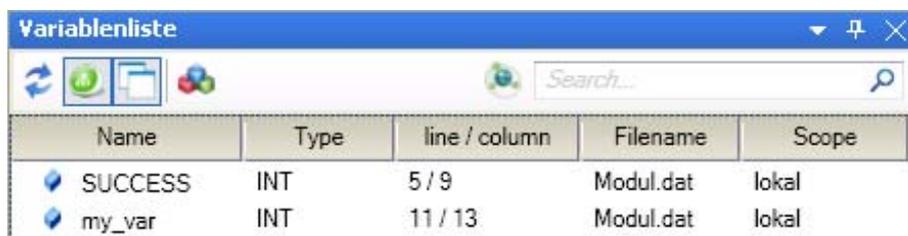


Fig. 9-34: “Variable list” window

Clicking on a column sorts the list by this column.

Button	Description
	Refresh
	Updates the list whenever the selected file in the tree changes
	Updates the list when the current editor changes
	Groups the variables by local subfunctions (SRC/DAT)
	Button is pressed: The search includes all global variables.

Additional editing functions

Standard edit functions can be called via **Edit** in the context menu. These include:

- **Cut, Paste, Copy, Delete**
- **Undo, Redo**
- **Find..., Replace...**
- **Go to...**
- **Select all**

Further edit functions are available in the context menu under **Advanced**:

Advanced > ...	Description
Tabify selection	Replaces the spaces in the selected string with tabs. Precondition: The configuration setting Use tabs is activated.
Untabify selection	Replaces the tabs in the selected string with spaces.
Increase indent	Inserts spaces or a tab at the cursor position.
Decrease indent	Removes the tab or spaces to the left of the cursor position.
Comment selection	Inserts a semicolon at the beginning of the selected line.
Uncomment selection	Removes the semicolon from the beginning of the selected line.
Collapse all folds	Closes all folds in the currently displayed file.
Expand all folds	Opens all folds in the currently displayed file.

Handling instructions

Fast entry for KRL instructions (KRL snippets)

An interrupt declaration must be programmed. KRL snippets are used to avoid having to enter the complete syntax `INTERRUPT DECL ... WHEN ... DO`. All that is then required is to fill out the variable positions in the syntax manually.

Procedure for programming with code snippets

1. Move the cursor to the desired position.
 - Right-click and select **Insert code snippet** from the context menu. A list box is opened. Double-click on the desired instruction.

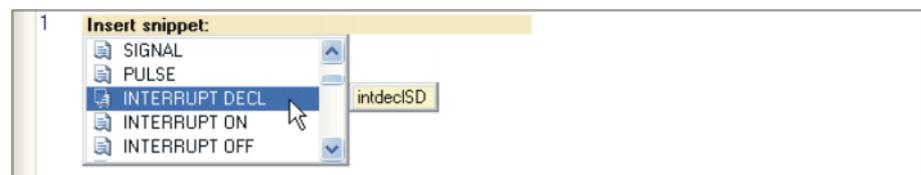


Fig. 9-35: Double-clicking on the instruction

- Or: Type in the abbreviation and press the TAB key.



Fig. 9-36: KRL snippets with search function

2. The KRL syntax is inserted automatically. The first variable position is highlighted in red. Enter the desired value.

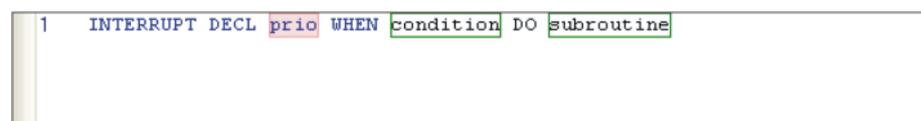


Fig. 9-37: The first variable position is highlighted in red

3. Press the Enter key to jump to the next variable position. Enter the desired value.

4. Repeat step 3 for all other variable positions.

KRL Editor: programming with the tool panel

Activate the tool panel.

1. Menu **Window>Toolpanel**
2. Position or dock the empty tool panel.
3. Load the program into the KRL Editor and the tool panel fills itself with functions.



Fig. 9-38: Filled tool panel

Procedure for programming with the tool panel

1. Move the cursor to the desired position.
2. Select the desired inline form in the tool panel.

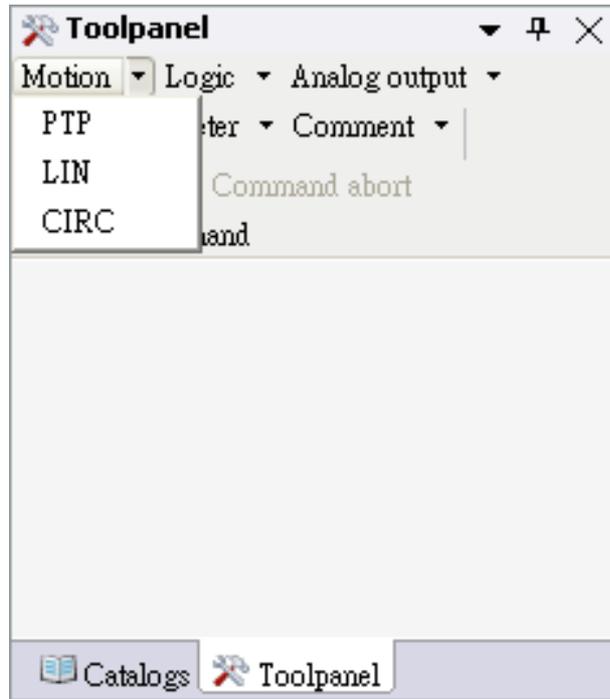


Fig. 9-39: Tool panel – Motion

3. Edit inline form / set parameters.

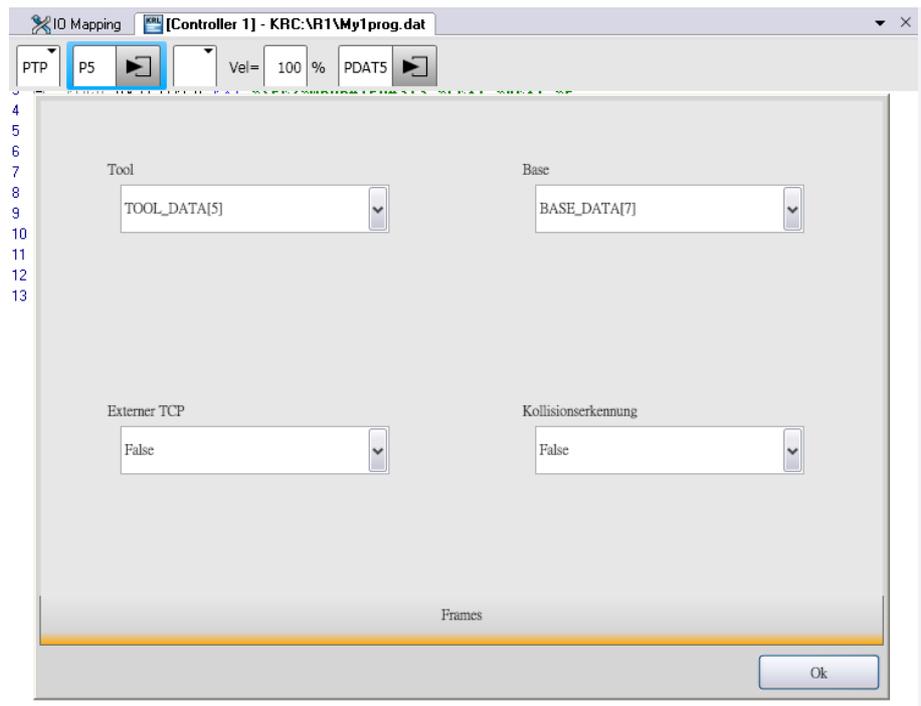


Fig. 9-40: KRL Editor – Inline form

4. Close inline form with the **Cmd OK** key on the tool panel.

Index

Symbols

\$TIMER o 59

A

Administrator 11
Advance run pointer 17
Array 25
Arrays 25

B

Basic arithmetic operations 22
Bit operations 22
Branch 61
Branches 61

C

CASE 63
Comment 5
Comparing projects 88
Comparison operations 22
Conditional statements 61
Counting loop 67
Cycle time 59

D

Data management 15
Data names 8
DECL 17
Declaration 17, 18
Declarations 15
DEFFCT 40
DISTANCE 78

E

Endless loop 65
ENUM 30
Enumeration data type 30
Expert level 11

F

Fold 6
Functions 33, 40, 42
Functions for message generation 43
Functions for string variables 42
Functions, mathematical 42

G

Global 15
Graphical user interface, WorkVisual 86

I

IF ... THEN 61
Importing, WorkVisual project 98
Initialization 20
Installing 92

K

Keyword 17

KRL Editor 104

L

Local 15
Logic operations 22
Loops 65

M

Manipulation 22
Motion programming in KRL 45

N

Non-rejecting loop 70

O

Operator 11

P

Parameter transfer 37
PATH 81
Path-related switching function 78, 81
Priority 24, 78, 81
Program execution control 61
Program flowchart 8
Program flowchart example 10
Program flowchart symbols 8
Programmer 11
Programming methodology, program flowchart example 10
Project Explorer 88
Project, activating 96
Project, loading 88
Project, opening 85
Pulse 76

R

Rejecting loop 69
Return 34, 40

S

Standard functions 22, 42
Structure 28
Structured programming 5
Subprogram 33
Subprogram, global 35
Subprogram, local 33
Subprograms 7
SWITCH 63
Switch statement 62, 63
Switching functions 75
System variables 59

T

Template 98
Timer 59
Transferring 92
TRIGGER 78, 81

U

User group, default 11

V

Variable life 15

Variables 15, 17

W

Wait function, signal-dependent 73

Wait function, time-dependent 72

WHILE 69

WorkVisual 85

