KUKA

Training

Robot Programming 3

KUKA System Software 8

Contents

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. Ease of maintenance
5. Clearly structured
6. Economical

1.2 Tools for creating structured robot programs

What is the point of a comment?

Comments are additional remarks within programming languages. All programming languages consist of instructions for the computer (code) and information for text editors (comments). If a source text undergoes further processing (compilation, interpretation, etc.), the comments are ignored by the software carrying out the processing and thus have no effect on the results.

In the KUKA controller, line comments are used, i.e. the comments automatically end at the end of the line.

Comments on their own cannot render a program legible; they can, however, significantly increase the legibility of a well-structured program. Comments enable the programmer to insert remarks and explanations into the program without the controller registering them as syntax.

It is the responsibility of the programmer to ensure that the contents of the comments are up to date and correspond to the actual program instructions. If programs are modified, the comments must therefore also be checked and adapted as required.

The contents of a comment, and thus its beneficial value, can be freely selected by the programmer/editor and are not subject to rules of syntax. Comments are generally written in "human" language – either the native language of the author or a major world language.

- 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:

At the start of a source text, the author can insert preliminary remarks, followed by details about the author, the license, the creation date, a contact address for questions, a list of other required files, etc.

```
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:

Headers and sections can be indicated as such. Often, not only linguistic means are used, but also graphic aids that are implemented using texts.

```
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:

The working method or the meaning of a section of text (e.g. a program line) can be explained, for example, so that other people (or even the author himself) can subsequently understand it more easily.

```
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:

Comments can be used to label inadequate sections of code or as placeholders for sections of code that are missing completely.

```
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:

If part of the code is to be temporarily deleted, but possibly reinserted in the future, it is commented out. Once it has been turned into a comment, the compiler no longer considers it to be code, i.e. for practical purposes it no longer exists.

```
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?

By default, the system already uses folds on the KUKA controller, e.g. for displaying inline forms. These folds help to structure the display of values in the inline form and hide program sections that are not relevant for the operator.

Furthermore, the user (in user group Expert or higher) is able to make his own folds. These folds can be used by the programmer, for example, to inform the user about what happens in certain program sections while keeping the actual KRL syntax in the background.

Initially, folds are generally first displayed closed after they have been created.

```
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
$BWDSSTART=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?

In programming, subprograms are primarily used to enable multiple use of sections containing identical tasks and thus avoid repetition of code. Apart from anything else, this saves memory.

Another important reason for the use of subprograms is the resultant structuring of the program.

A subprogram should perform a self-contained and clearly definable subtask. For improved ease of maintenance and elimination of programming errors, subprograms today are preferably short and uncluttered, as the time and administrative effort required to call subprograms on modern computers is negligible.

- 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 subprograms

```
DEF MAIN()

INI

LOOP

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

    IF $IN[1]== TRUE THEN
        EXIT
    ENDIF

ENDLOOP

END
```

What is achieved by indenting command lines?

In order to make explicit the structures within program modules, it is advisable to indent nested command sequences in the program text and write instructions at the same nesting depth directly below one another.

The effect obtained is purely optical and serves merely to make the program more comprehensible to other people.

```
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        ; Motion with KRL
    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. This includes e.g.:

- 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?

A program flowchart is a flowchart illustrating the sequence and structure of a program. It is a graphic representation of the implementation of an algorithm in a program and describes the sequence of operations for solving a task. The symbols for program flowcharts are standardized in DIN 66001. Program flowcharts are often used independently of computer programs to represent processes and activities.

The graphic representation of a program algorithm is significantly more legible than the same program algorithm in a code-based description as the structure is much easier to recognize.

Structure and programming errors are much easier to avoid when translating the program into program codes as correct application of a program flowchart

allows direct translation into program codes. At the same time, creation of a program flowchart provides documentation of the program to be created.

■ 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
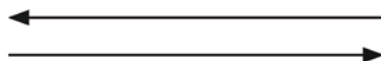
Fig. 1-2

Branch
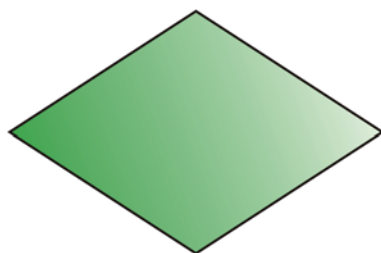
Fig. 1-3

General statements in the program code

Fig. 1-4

Subprogram call

Fig. 1-5

Input/output statement
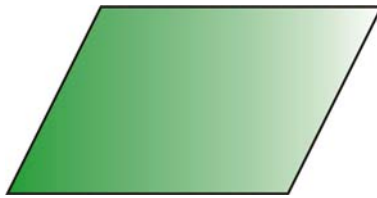
Fig. 1-6

Program flowchart example
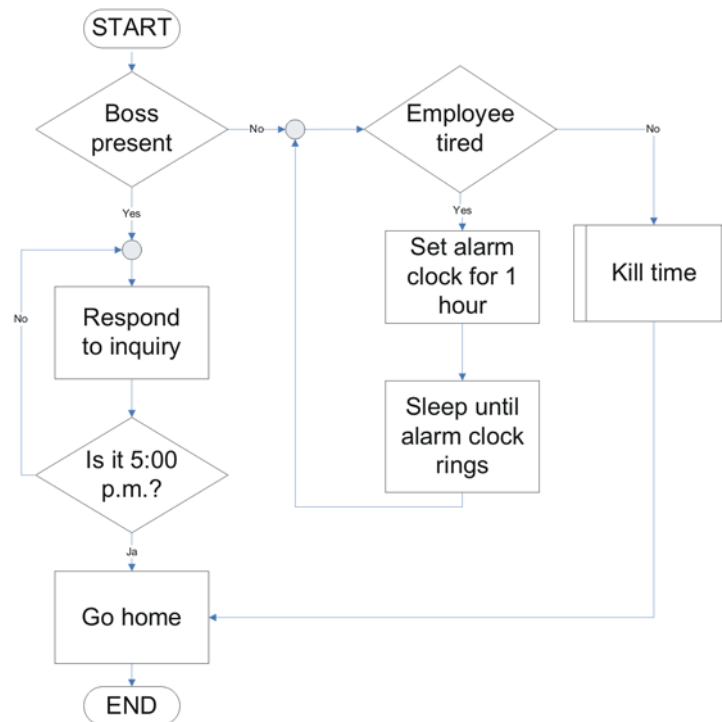
Fig. 1-7

How is a program flowchart drawn up?

Starting from the user expectations, the problem is gradually refined until the resulting components are manageable enough to be converted into KRL.
The drafts made at successive stages of development become steadily more detailed.

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 Submit interpreter

2.1 Using the Submit interpreter

Description of the Submit interpreter

Two tasks run in KSS 8.x.

- Robot interpreter (execution of robot motion programs and their logic)
- Controller interpreter (execution of a parallel control program)

Structure of the program SPS.SUB:

```
 1  DEF  SPS ( )
 2    DECLARATIONS
 3    INI
 4
 5    LOOP
 6      WAIT FOR NOT($POWER_FAIL)
 7      TORQUE_MONITORING()
 8
 9      USER PLC
10    ENDLOOP
```

The status of the Submit interpreter

S (green)	S (red)	S (grey)
Submit interpreter running	Submit interpreter stopped	Submit interpreter deselected

The controller interpreter

- can be started automatically or manually.
- can also be stopped or deselected manually.
- can perform robot operator and control tasks.
- is created by default in the R1/SYSTEM directory with the name **SPS.sub**.
- can be programmed with the KRL command set.
- **cannot** execute KRL commands related to robot motions.
- allows asynchronous motions of external axes.
- has read/write access to system variables.
- has read/write access to inputs/outputs.

Relationships when programming the Submit interpreter

⚠ **CAUTION** **Attention!**
The Submit interpreter must not be used for time-critical applications! A PLC must be used in such cases. Reasons:

- The Submit interpreter shares system resources with the robot interpreter and I/O management, which have the higher priority. The Submit interpreter is thus not executed regularly at the robot controller's interpolation cycle rate of 12 ms.
- Furthermore, the runtime of the Submit interpreter is irregular. The runtime of the Submit interpreter is influenced by the number of lines in the SUB program. Even comment lines and blank lines have an effect.

- Automatic starting of the Submit interpreter
  - The submit interpreter starts automatically when the robot controller is switched on after a cold start.
  - The program defined in the file KRC/STEU/MADA/$custom.dat is started.

```
$PRO_I_O[]="/R1/SPS()"
```

- Manual operator control of the Submit interpreter
  - Select operator control via the menu sequence **Configure** > **SUBMIT Interpreter** > **Start / select**.
  - Direct operator control using the status bar of the **Submit interpreter** status indicator. Selecting it opens a window with the options that can be executed.

> **i** If a system file, e.g. $config.dat or $custom.dat, is modified in such a way that errors are introduced, the Submit interpreter is automatically deselected. Once the error in the system file has been rectified, the Submit interpreter must be reselected manually.

Points to consider when programming the Submit interpreter

- **No** statements for robot motions can be executed, such as:
  - PTP, LIN, CIRC, etc.
  - Subprogram calls containing robot motions.
  - Statements referring to robot motions, TRIGGER or BRAKE.
- Asynchronous axes, such as E1, can be controlled.

```
IF (($IN[12] == TRUE) AND ( NOT $IN[13] == TRUE)) THEN
ASYPTP {E1 45}
ENDIF
...
IF ((NOT $IN[12] == TRUE) AND  ($IN[13] == TRUE)) THEN
ASYPTP {E1 0}
ENDIF
```

- Statements between the LOOP and ENDLOOP lines are processed permanently in the background.
- All stoppages due to wait commands or wait loops must be avoided, as these slow Submit interpreter execution still further.
- Switching of outputs is possible.

> **⚠ CAUTION** **Warning!**
> No check is made to see if the robot interpreter and Submit interpreter are accessing the same output simultaneously, as this may even be desired in certain cases.
> The user must therefore carefully check the assignment of the outputs. Otherwise, unexpected output signals may be generated, e.g. in safety equipment. Death, serious physical injuries or major damage to property may result.

> **⚠ WARNING** In the test modes, $OV_PRO must not be written to by the Submit interpreter, because the change may be unexpected for operators working on the industrial robot. Death to persons, severe injuries and considerable damage to property may result.

> **⚠ WARNING** **Warning!**
> Do not use the submit interpreter to process safety-relevant and time-critical signals and variables.

Procedure for programming the Submit interpreter

1. Programming is carried out with the interpreter stopped or deselected.
2. The default program SPS.sub is loaded into the editor.
3. Carry out the necessary declarations and initializations. The prepared folds should be used for this.

4. Make program expansions in the fold `USER PLC`.

5. Close and save the submit interpreter.

6. If the submit does not start automatically, start it manually.

Program example: blinker programming in the Submit interpreter

```
DEF SPS( )
DECLARATIONS
DECL BOOL flash ;Declaration in $CONFIG.dat
INI
flash = FALSE
$TIMER[32]=0 ; Reset TIMER[32]
$TIMER_STOP[32]=false ; Start TIMER[32]
...
LOOP
...
USER PLC
IF ($TIMER[32]>500) AND (flash==FALSE) THEN
   flash=TRUE
ENDIF
IF $TIMER[32]>1000 THEN
   flash=FALSE
   $TIMER[32]=0
ENDIF
; Assignment to a lamp (output 99)
$OUT[99] = flash
...
ENDLOOP
```

3 Workspaces with KRL

3.1 Using workspaces

Description

"Workspaces for personnel protection" and "workspaces for equipment protection"

- "Workspaces for personnel protection" serve to protect personnel and can only be set up using the additional SafeOperation option.
- KUKA System Software 8.x enables workspaces to be configured for the robot. These serve to **protect the system only**.

"Workspaces for equipment protection"

- These workspaces, suitable for machine protection only, are configured directly in the KUKA System Software.
- 8 axis-specific workspaces can be created.
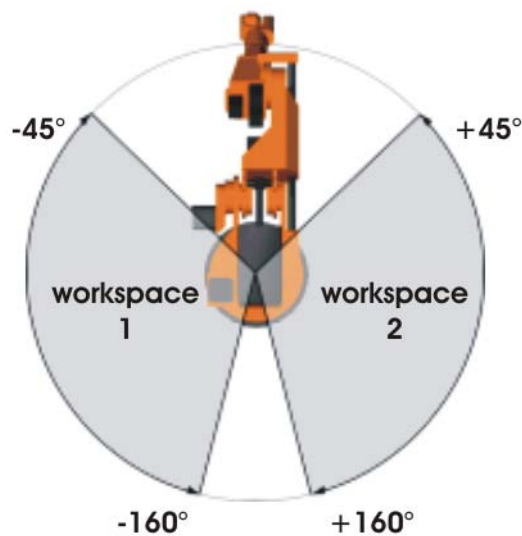
Fig. 3-1: Example of axis-specific workspaces for A1

- Axis-specific workspaces can be used to further restrict the areas defined by the software limit switches in order to protect the robot, tool or workpiece.
- 8 Cartesian workspaces can be created.
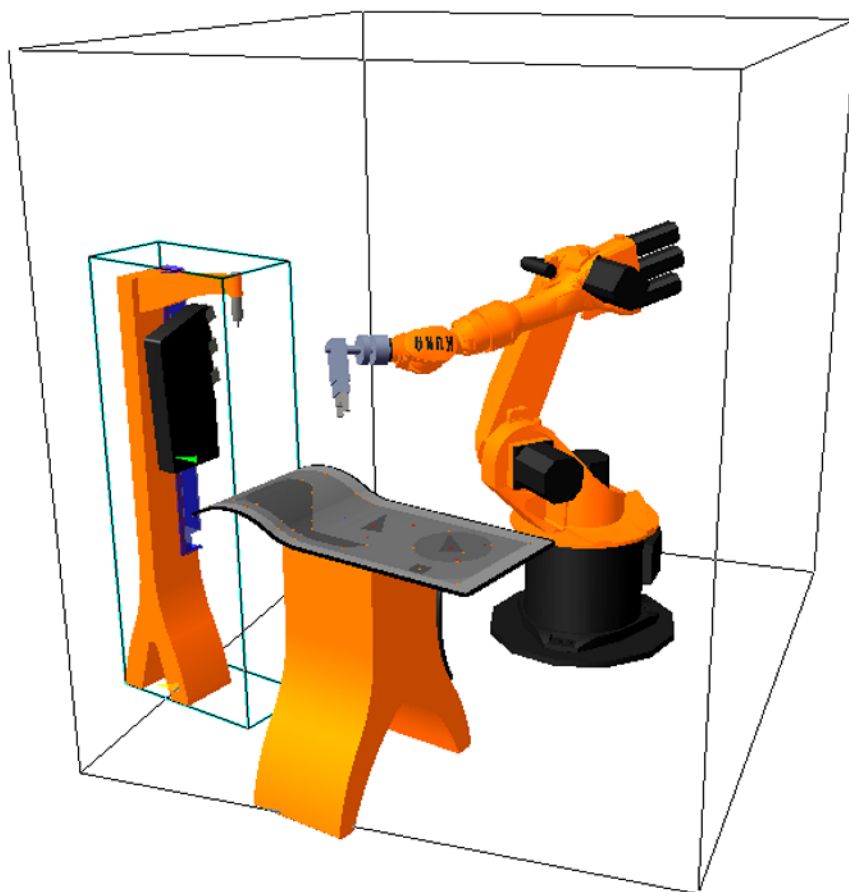
Fig. 3-2: Example of a Cartesian workspace

- In the case of Cartesian workspaces, only the position of the TCP is monitored. It is not possible to monitor whether other parts of the robot violate the workspace.
- Multiple workspaces can be activated in order to form complex shapes; these workspaces may also overlap.
- **Non-permitted spaces**: The robot may only move outside such a space.

Fig. 3-3: Non-permitted spaces

- **Permitted spaces**: The robot must not move outside such a space.
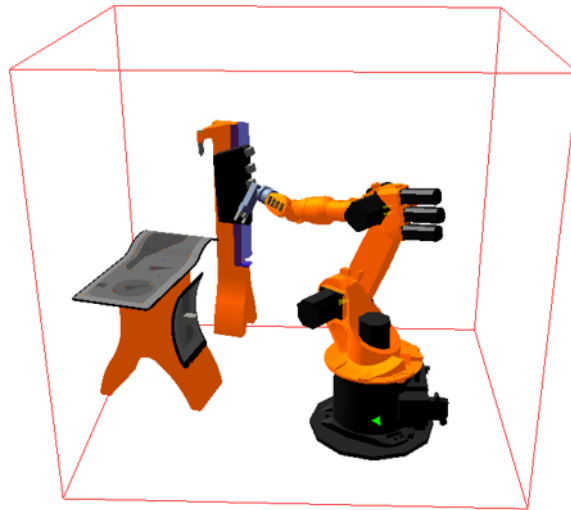
Fig. 3-4: Permitted spaces

■ Exactly what reactions occur when the robot violates a workspace depends on the configuration.

■ One output (signal) can be assigned to each workspace.

Principle of workspace interlocking and workspaces

Workspace interlocking

■ Sequence with direct coupling (without PLC).
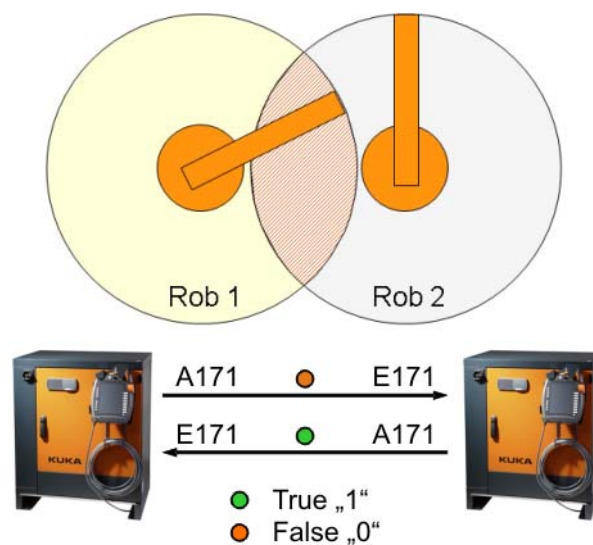
Fig. 3-6

■ Sequence with a PLC, which can only forward the signals or implements additional logic control.
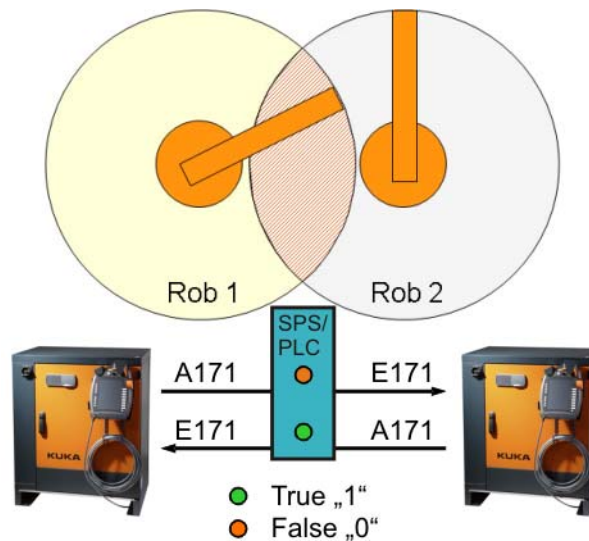
Fig. 3-10

■ Direct signal forwarding (with use of a PLC: without logic)

   ■ **Without wait time**: An entry request is submitted and the robot may enter the space immediately if it is not locked.

> **NOTICE**   If both robots submit an entry request simultaneously, they are both granted permission to enter, generally resulting in a collision.

   ■ **With monitoring time**: An entry request is submitted and the robot's own space is locked. The new space is not checked until a monitoring time has elapsed. If the space is not locked, the robot may enter it immediately. The spaces are locked if both requests are received almost simultaneously.

■ Signal forwarding with logic control (priority)

   ■ The entry requests and entry enable are logically linked. The **priority control** decides which robot may enter the shared workspace even in the case of simultaneous entry requests.

   ■ In addition to the priority control, it can also be checked whether the robot (robot TCP) is located in the workspace before permission to enter is granted. **Workspaces must be defined** for this purpose.

Principle of workspace configuration

Mode for workspaces

■ **#OFF**

   Workspace monitoring is deactivated.

■ **#INSIDE**

   ■ Cartesian workspace: The defined output is set if the TCP or flange coordinate system is located inside the workspace.

   ■ Axis-specific workspace: The defined output is set if the axis is located inside the workspace.

■ **#OUTSIDE**

   ■ Cartesian workspace: The defined output is set if the TCP or flange coordinate system is located outside the workspace.

   ■ Axis-specific workspace: The defined output is set if the axis is located outside the workspace.

■ **#INSIDE_STOP**

   ■ Cartesian workspace: The defined output is set if the TCP, flange coordinate system or wrist root point is located inside the workspace. (Wrist root point = intersection of A4/A5/A6.)

- Axis-specific workspace: The defined output is set if the axis is located inside the workspace.

The robot is also stopped and messages are displayed. The robot cannot be moved again until the workspace monitoring is deactivated or bypassed.

- **#OUTSIDE_STOP**
  - Cartesian workspace: The defined output is set if the TCP or flange coordinate system is located outside the workspace.
  - Axis-specific workspace: The defined output is set if the axis is located outside the workspace.

The robot is also stopped and messages are displayed. The robot cannot be moved again until the workspace monitoring is deactivated or bypassed.

The following parameters define the position and size of a Cartesian workspace:

- Origin of the workspace relative to the WORLD coordinate system
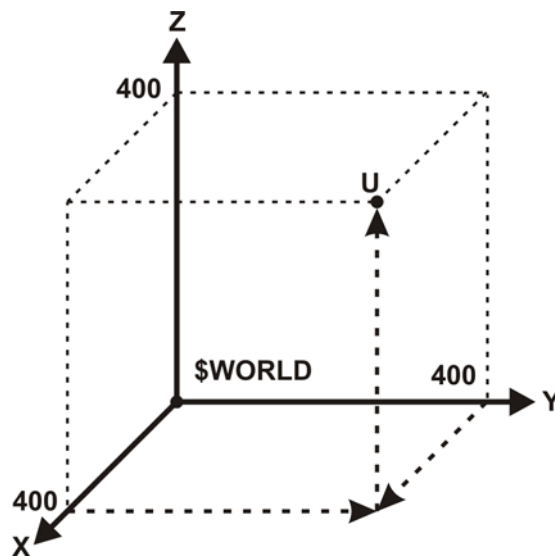
Fig. 3-13: Cartesian workspace, origin U

- Dimensions of the workspace, starting from the origin
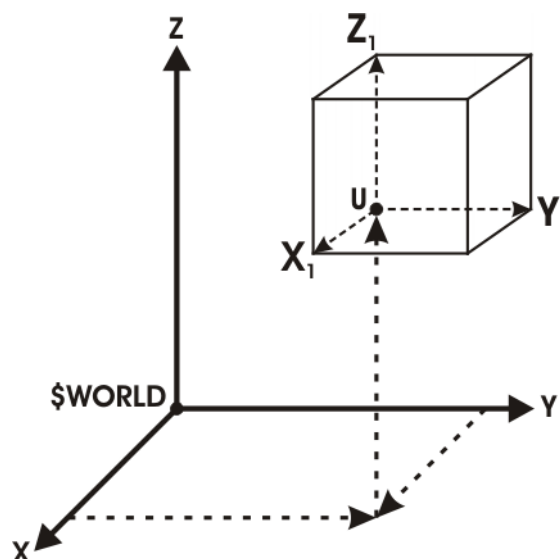
Fig. 3-14: Cartesian workspace, dimensions

Procedure for configuring and using workspaces

Configuring axis-specific workspaces

1. In the main menu, select **Configuration** > **Miscellaneous** > **Workspace monitoring** > **Configuration**.
   The **Cartesian workspaces** window is opened.

2. Press **Axis-spec.** to switch to the window **Axis-specific workspaces**.
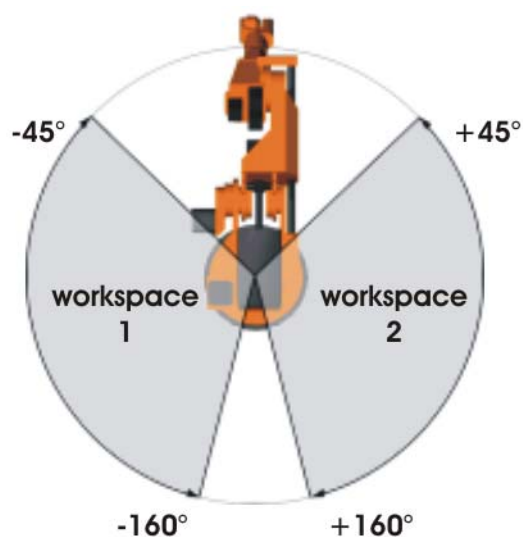
Fig. 3-15: Example of axis-specific workspaces for A1

3. Enter values and press **Save**.

Fig. 3-16: Example of an axis-specific workspace

4. Press **Signal**. The **Signals** window is opened.

Fig. 3-17: Workspace signals

Item	Description
1	Outputs for monitoring of the Cartesian workspaces
2	Outputs for monitoring of the axis-specific workspaces

If no output is to be set when the workspace is violated, the value FALSE must be entered.

5. In the **Axis-specific** group: next to the number of the workspace, enter the output that is to be set if the workspace is violated.
6. Press **Save**.
7. Close the window.

Configuring Cartesian workspaces

1. In the main menu, select **Configuration** > **Miscellaneous** > **Workspace monitoring** > **Configuration**.

   The **Cartesian workspaces** window is opened.
2. Enter values and press **Save**.
3. Press **Signal**. The **Signals** window is opened.
4. In the **Cartesian** group: next to the number of the workspace, enter the output that is to be set if the workspace is violated.
5. Press **Save**.
6. Close the window.

Example of Cartesian workspaces

■ If point "P2" is situated at the origin of the workspace, only the coordinates of "P1" need to be determined.
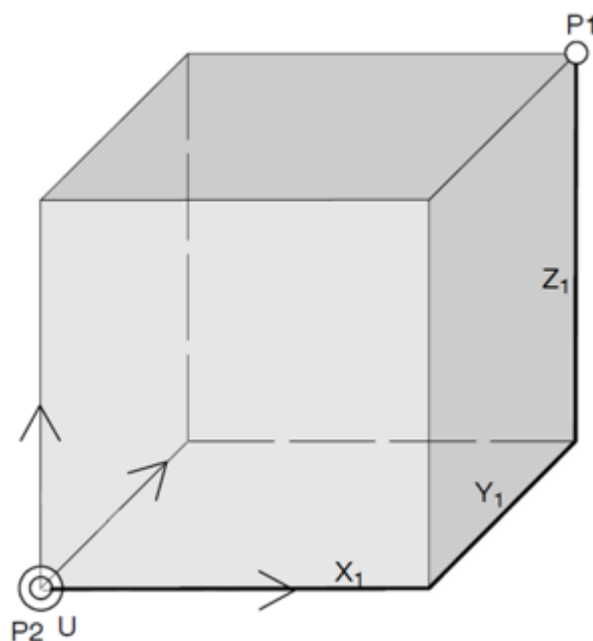
Fig. 3-18: Example of a Cartesian workspace (P2 located at origin)

Fig. 3-19: Example configuration of a Cartesian workspace (P2 located at origin)

- The workspace in this example has the dimensions x = 300 mm, y = 250 mm and z = 450 mm. In relation to the world coordinate system, it is rotated about the Y axis by 30 degrees. The origin "U" is not situated at the center of the cuboid.

KUKA
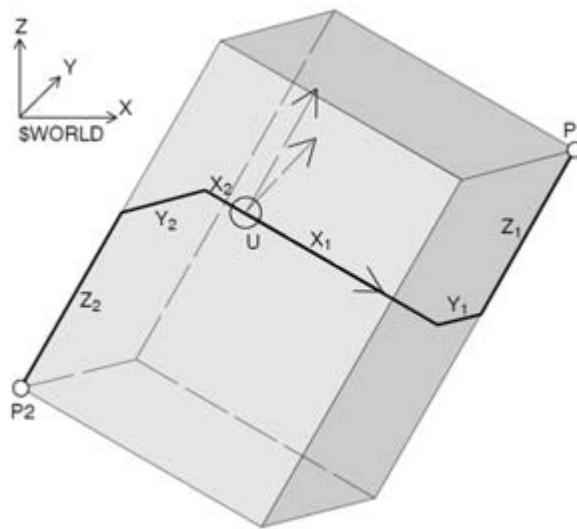
Fig. 3-20: Example of a Cartesian workspace (rotated)

Fig. 3-21: Example configuration of a Cartesian workspace (rotated)

Working with workspaces

■ Axis-specific workspaces (R1\Mada\$machine.dat)

```
DEFDAT  $MACHINE PUBLIC
...
$AXWORKSPACE[1]={A1_N 0.0,A1_P 0.0,A2_N 0.0,A2_P 0.0,A3_N 0.0,A3_P
0.0,A4_N 0.0,A4_P 0.0,A5_N 0.0,A5_P 0.0,A6_N 0.0,A6_P 0.0,E1_N
0.0,E1_P 0.0,E2_N 0.0,E2_P 0.0,E3_N 0.0,E3_P 0.0,E4_N 0.0,E4_P
0.0,E5_N 0.0,E5_P 0.0,E6_N 0.0,E6_P 0.0,MODE #OFF}
$AXWORKSPACE[2]={A1_N 45.0,A1_P 160.0,A2_N 0.0,A2_P 0.0,A3_N
0.0,A3_P 0.0,A4_N 0.0,A4_P 0.0,A5_N 0.0,A5_P 0.0,A6_N 0.0,A6_P
0.0,E1_N 0.0,E1_P 0.0,E2_N 0.0,E2_P 0.0,E3_N 0.0,E3_P 0.0,E4_N
0.0,E4_P 0.0,E5_N 0.0,E5_P 0.0,E6_N 0.0,E6_P 0.0,MODE #INSIDE_STOP}
```

■ Cartesian workspaces (STEU\Mada\$custom.dat)

```
DEFDAT  $CUSTOM PUBLIC
...
$WORKSPACE[1]={X 400.0,Y -100.0,Z 1200.0,A 0.0,B 30.0,C 0.0,X1
250.0,Y1 150.0,Z1 200.0,X2 -50.0,Y2 -100.0,Z2 -250.0,MODE #OUTSIDE}
$WORKSPACE[2]={X 0.0,Y 0.0,Z 0.0,A 0.0,B 0.0,C 0.0,X1 0.0,Y1 0.0,Z1
0.0,X2 0.0,Y2 0.0,Z2 0.0,MODE #OFF}
```

■ Workspace signals (STEU\Mada\$machine.dat)

```
DEFDAT  $MACHINE PUBLIC
...
SIGNAL $WORKSTATE1 $OUT[912]
SIGNAL $WORKSTATE2 $OUT[915]
SIGNAL $WORKSTATE3 $OUT[921]
SIGNAL $WORKSTATE4 FALSE
...
SIGNAL $AXWORKSTATE1 $OUT[712]
SIGNAL $AXWORKSTATE2 $OUT[713]
SIGNAL $AXWORKSTATE3 FALSE
```

■ Switching workspace on/off by means of KRL

```
DEF myprog( )
...
$WORKSPACE[3].MODE = #INSIDE
...
$WORKSPACE[3].MODE = #OFF
...
$AXWORKSPACE[1].MODE = #OUTSIDE_STOP
...
$AXWORKSPACE[1].MODE = #OFF
```

3.2   Exercise: Workspace monitoring

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

■ Configure workspaces

■ Use the different modes with workspaces

■ Override workspace monitoring

Preconditions

The following are preconditions for successful completion of this exercise:

■ Theoretical knowledge of workspace monitoring

Task description

Subtask 1

1. Configure workspace 1 as a cube with edge length 400 mm.
2. Transfer a signal when the workspace is entered. Use output 14 for this.
3. Configure workspace 2 as a cube with edge length 400 mm.
4. Transfer a signal when the workspace is exited. Use output 15 for this.
5. Test both workspaces and compare the information with the control panel display.
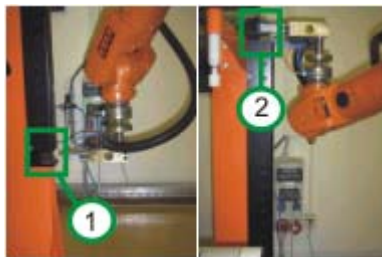
Fig. 3-22

Subtask 2

1. Configure workspace 3 as a cuboid with edge length 400 mm and 200 mm.
2. Lock entry into this workspace and transfer a signal. Use output 16 for this.
3. Test this workspace and compare the information with the control panel display.
4. To exit the workspace, override the monitoring of the workspace by means of the corresponding menu item.
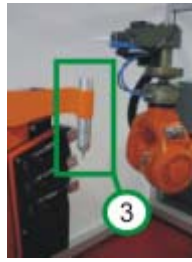
Fig. 3-23

What you should now know:

1. What is the maximum number of workspaces that can be configured?

. . . .

. . . .

2. What MODE options are available during workspace configuration?

. . . .

. . . .

3. Which coordinate system does the ORIGIN refer to in Cartesian workspace configuration?

. . . .

. . . .

4. What are the advantages of a robot interlock implemented using direct I/O coupling and monitoring time?

. . . .

. . . .

5. What are the disadvantages in the above case (Question 4) when working without a monitoring time?

. . . .

. . . .

4 Message programming with KRL

4.1 General information about user-defined messages

Description of user-defined messages

Message programming properties

- The programmer can use KRL to program his own messages.
- Several messages can be generated simultaneously.
- Generated messages are stored in a message buffer until they are deleted.
- Notification messages are not managed in the message buffer ("fire and forget" principle).
- Messages can be easily checked or deleted, but not notification messages.
- Up to 3 parameters can be integrated into each message.

An icon is displayed in the message window of the KUKA.HMI alongside every message. The icons are permanently assigned to the message types and cannot be altered by the programmer.

The following types of message can be programmed:

Icon	Type
	Acknowledgement message
	Status message
	Notification message
	Wait message

	Dialog message (displayed in a separate pop-up window)

 No predefined reactions of the robot system are linked to the different message types (e.g. robot brakes or program is stopped). The desired reactions must be programmed.

The generation, deletion or checking of messages is carried out using ready-made standard KUKA functions. A number of different variables are required for this:

Functions for message programming

- Generate message
- Check message
- Clear message
- Generate dialog
- Check dialog

Complex variables for message programming

- Structure for originator, message number, message text
- Structure as placeholder for the 3 possible parameters
- Structure for the general message reaction

■ Structure for labeling the buttons in the case of dialog messages

Principle of user-defined message programming: variables/structures

Fig. 4-1: Notification message

Structure for originator, message number, message text

■ Predefined KUKA structure: `KrlMsg_T`

```
STRUC KrlMsg_T CHAR Modul[24], INT Nr, CHAR Msg_txt[80]
```

■

```
DECL KrlMsg_T mymessage
mymessage = {Modul[ ] "College", Nr 1906, Msg_txt[ ] "My first
Message"}
```

■ Originator: `Modul[ ]"College"`
  ■ Maximum 24 characters
  ■ When displayed, the originator text is embedded by the system in "< >".
■ Message number: `No. 1906`
  ■ Freely selectable integer
  ■ Duplicated numbers are not recognized.

  Message text: `Msg_txt[ ] "My first Message"`
  ■ Maximum 80 characters
  ■ Text is displayed in the second line of the message.

When sending a message, the message type must be selected:

■ Enumeration data type `EKrlMsgType`

```
ENUM EKrlMsgType Notify, State, Quit, Waiting
```

■ `#Quit`: generates this message as an acknowledgement message

■ `#STATE`: generates this message as a status message

■ `#NOTIFY`: generates this message as a notification message

■ `#WAITING`: generates this message as a wait message

The value of a variable is to be displayed in a message text. For example, the current number of units is to be displayed. For this, a so-called placeholder is required in the message text. The maximum number of placeholders is 3. The notation is `%1`, `%2` and `%3`.

For this reason, 3 parameter sets are required. Each parameter set consists of the KUKA structure `KrlMsgPar_T`:

```
Enum KrlMsgParType_T Value, Key, Empty
STRUC KrlMsgPar_T KrlMsgParType_T Par_Type, CHAR Par_txt[26], INT
Par_Int, REAL Par_Real, BOOL Par_Bool
```

Using the individual elements

- `Par_Type`: Type of parameter/placeholder
  - `#VALUE`: The parameter is integrated directly into the message text in the form in which it is transferred (i.e. as string, INT, REAL or BOOL value).
  - `#KEY`: The parameter is a keyword that must be searched for in the message database in order to load the corresponding text.
  - `#EMPTY`: The parameter is empty.
- `Par_txt[26]`: Text or keyword for the parameter
- `Par_Int`: Transfer of an integer value as a parameter
- `Par_Real`: Transfer of a real value as a parameter
- `Par_Bool`: Transfer of a Boolean value as a parameter; the text displayed is `TRUE` or `FALSE`.

Program examples for the direct transfer of parameters to the placeholders:

The message text is `Msg_txt[ ]` "Fault in %1".

```
DECL KrlMsgPar_T Parameter[3] ; Create 3 parameter sets
...
Parameter[1] = {Par_Type #VALUE, Par_txt[ ] "Finisher"}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
...
```

Message generation: **Fault in finisher**

> Since the parameters are seldom entered using constants, the individual elements are transferred with the point separator.

Program examples for the transfer of parameters to the placeholders using the point separator:

The message text is `Msg_txt[ ]` "%1 components missing".

```
DECL KrlMsgPar_T Parameter[3] ; Create 3 parameter sets
DECL INT missing_part
...
misssing_part = 13
...
Parameter[1] = {Par_Type #VALUE}
Parameter[1].Par_Int = missing_part
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
...
```

Message generation: **13 components missing**
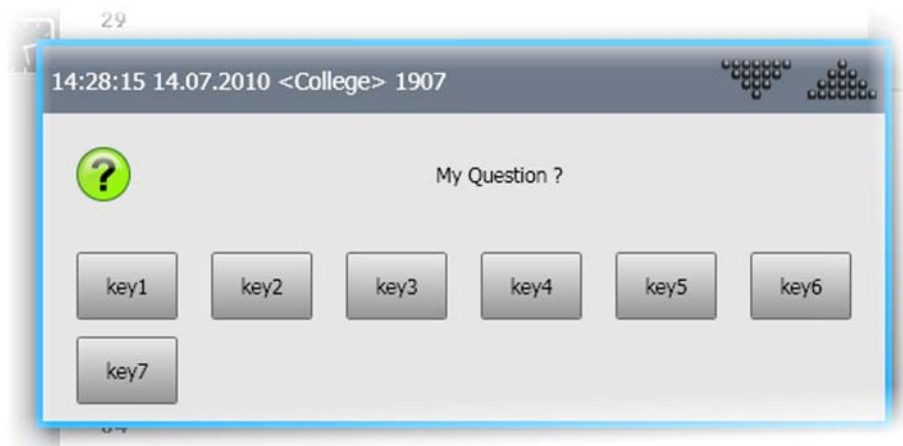
Fig. 4-2: Dialog

Structure for the assignment of buttons in dialogs:

■   Predefined KUKA structure: `KrlMsgDlgSK_T`

```
Enum KrlMsgParType_T Value, Key, Empty
Struc KrlMsgDlgSK_T KrlMsgParType_T Sk_Type, Char SK_txt[10]
```

■   `Sk_Type`: Type of button label
  ■   `#VALUE`: The parameter is integrated directly into the message text in the form in which it is transferred.
  ■   `#KEY`: The parameter is a keyword that must be searched for in the message database in order to load the corresponding text.
  ■   `#EMPTY`: The button is not assigned.
■   `Sk_txt[ ]`: Text or keyword for the button

Program example for the labeling of 7 buttons in a dialog:

```
DECL KRLMSGDLGSK_T Softkey[7] ; Prepare 7 possible softkeys
...
softkey[1]={sk_type #value, sk_txt[] "key1"}
softkey[2]={sk_type #value, sk_txt[] "key2"}
softkey[3]={sk_type #value, sk_txt[] "key3"}
softkey[4]={sk_type #value, sk_txt[] "key4"}
softkey[5]={sk_type #value, sk_txt[] "key5"}
softkey[6]={sk_type #value, sk_txt[] "key6"}
softkey[7]={sk_type #value, sk_txt[] "key7"}
...
```

> **i** A maximum of 10 characters per button can be assigned. The width of the buttons varies according to which characters are used.

When a message or dialog is generated, 4 more message options are transferred. These options can be used to influence the advance run, message deletion and log database.

Structure for general message options:

■   Predefined KUKA structure: `KrlMsgOpt_T`

```
STRUC KrlMsgOpt_T   BOOL VL_Stop, BOOL Clear_P_Reset, BOOL
Clear_SAW, BOOL Log_To_DB
```

■   `VL_Stop`: `TRUE` triggers an advance run stop.
  ■   Default: `TRUE`
■   `Clear_P_Reset` : `TRUE` deletes all status, acknowledgement and wait messages when the program is reset or canceled.

■ Default: `TRUE`

> ℹ️ Notification messages can only be deleted using the buttons **OK** and **Confirm all**. The following always applies for dialog messages: `Clear_P_Reset=TRUE`.

■ `Clear_P_SAW`: `TRUE` deletes all status, acknowledgement and wait messages when block selection is carried out using the "Line Sel." button.

    ■ Default: `FALSE`

■ `Log_To_DB`: `TRUE` causes the message to be logged in the log database.

    ■ Default: `FALSE`

Principle of user-defined message programming: functions

Setting, checking and deleting a message

■ **Setting or generating a message**

This function is used to set a message in the KRL program. This means that the corresponding message is transferred to the internal message buffer. The **exception** to this is notification messages, as these are not managed in the message buffer.

    ■ Built-in functions for generating a message

```
DEFFCT INT Set_KrlMsg(Type:IN, MyMessage:OUT, Parameter[ ]:OUT,
Option:OUT)
DECL EKrlMsgType Type
DECL KrlMsg_T MyMessage
DECL KrlMsgPar_T Parameter[ ]
DECL KrlMsgOpt_T Option
```

    ■ `Type`: Type of message (#Notify, #State, #Quit, #Waiting)

    ■ `MyMessage`: general message information (originator, message number, message text)

    ■ `Parameter[ ]`: the **3** possible parameters for the placeholders %1, %2 and %3 (must also be transferred if used)

    ■ `Option`: general message options (advance run stop, log in message database, implicitly delete message in the case of a program reset or block selection)

    ■ **Return value of the function:** called a "handle" (ticket number). This handle can be used to check whether the message has been successfully generated and also serves as the identification number in the message buffer. In this way, specific messages can be checked or deleted.

```
DEF MyProg( )
DECL INT handle
...
handle = Set_KrlMsg(Type, MyMessage, Parameter[ ], Option)
```

    ■ `handle == -1`: The message could not be generated (e.g. because the message buffer is too full).

    ■ `handle > 0`: The message was successfully generated and is being managed with the corresponding identification number in the message buffer.

> ℹ️ Notification messages are processed on a "fire and forget" basis. In the case of notification messages, the return value is always `handle = 0` if the message was generated successfully.

■ **Checking a message**

This function can be used to check whether a specific message with a defined handle still exists. It also checks whether this message is still present in the internal message buffer.

- Built-in functions for checking a message

```
DEFFCT BOOL Exists_KrlMsg(nHandle:IN)
DECL INT nHandle
```

- `nHandle`: The handle provided for the message by the function "`Set_KrlMsg(...)`"
- **Return value of the function:**

```
DEF MyProg( )
DECL INT handle
DECL BOOL present
...
handle = Set_KrlMsg(Type, MyMessage, Parameter[ ], Option)
...
present= Exists_KrlMsg(handle)
```

- `present == TRUE`: This message still exists in the message buffer.
- `present == FALSE`: This message is no longer in the message buffer (i.e. it has been acknowledged or deleted).

- **Deleting a message**

This function can be used to delete a message. This means that the corresponding message is deleted from the internal message buffer.

- Built-in functions for checking a message

```
DEFFCT BOOL Clear_KrlMsg(nHandle:IN)
DECL INT nHandle
```

- `nHandle`: The handle provided for the message by the function "`Set_KrlMsg(...)`"
- **Return value of the function:**

```
DEF MyProg( )
DECL INT handle
DECL BOOL erase
...
handle = Set_KrlMsg(Type, MyMessage, Parameter[ ], Option)
...
eraser = Clear_KrlMsg(handle)
```

- `eraser == TRUE`: This message was successfully deleted.
- `eraser == FALSE`: This message could not be deleted.

Special functions for deleting with the function `Clear_KrlMsg(handle)`:
`Clear_KrlMsg(-1)`: All messages initiated by this process are deleted.
`Clear_KrlMsg(-99)`: All initiated user-defined KRL messages are deleted.

Principle of user-defined dialog programming: functions

Setting and checking a dialog

- **Setting or generating a dialog**

The function `Set_KrlDlg( )` generates a dialog message. This means that the message is transferred to the message buffer and displayed in a separate message window with buttons.

- Built-in functions for generating a dialog

```
DEFFCT Extfctp Int Set_KrlDlg (MyQuestion:OUT, Parameter[ ]:OUT,
Button[ ]:OUT, Option:OUT)
DECL KrlMsg_T MyQuestion
DECL KrlMsgPar_T Parameter[ ]
DECL KrlMsgDlgSK_T Button[ ]
DECL KrlMsgOpt_T Option
```

- `MyQuestion`: general message information (originator, message number, message text)
- `Parameter[ ]`: the **3** possible parameters for the placeholders %1, %2 and %3 (must also be transferred if used)
- `Button[ ]`: the labeling for the **7** possible buttons (must also be transferred if used)
- `Option`: general message options (advance run stop, log in message database, implicitly delete message in the case of a program reset or block selection)
- **Return value of the function:** handle for the dialog. This handle can be used to check whether the dialog has been successfully generated. It can also serve as the identification number in the message buffer.

```
DEF MyProg( )
DECL INT handle
...
handle = Set_KrlDlg(MyQuestion, Parameter[ ], Button[ ], Option)
```

- `handle == -1`: The dialog could not be generated (e.g. because another dialog is active and has not yet been answered or the message buffer is too full).
- `handle > 0`: The dialog was successfully generated and is being managed with the corresponding identification number in the message buffer.

> **i** A dialog cannot be generated until no other dialog is active.
> The function merely generates the dialog. It does not wait until the dialog has been answered.

- **Checking a dialog**

  The function `Exists_KrlDlg()` can be used to check whether a specific dialog still exists. It also checks whether this dialog is still present in the message buffer.

  - Built-in functions for checking a message

```
DEFFCT BOOL Exists_KrlDlg(INT nHandle:IN, INT Answer:OUT)
DECL INT nHandle, answer
```

- `nHandle`: The handle provided for the dialog by the function "`Set_KrlDlg(…)`"
- `answer`: Return value indicating which button has been pressed. Button 1, defined as "`Button[1]`", thus returns the value 1.

> **i** The function does not wait until the dialog has been deleted, but merely searches the buffer for the dialog with this handle. The KRL program must therefore be polled cyclically until the dialog has been answered or deleted.

■   **Return value of the function:**

```
DEF MyProg( )
DECL INT handle, answer
DECL BOOL present
...
handle = Set_KrlDlg(MyQuestion, Parameter[ ], Button[ ], Option)
...
present = Exists_KrlDlg(handle, answer)
```

■ `present == TRUE`: This dialog still exists in the message buffer.
■ `present == FALSE`: This dialog is no longer in the message buffer (i.e. it has been answered).

> `answer` is now written back with the value of the pressed button. Valid values are in the range 1 to 7, depending on the programmed button numbers.

4.2   Working with a notification message

Description of a user-defined notification message

Fig. 4-3: Notification message

■ Notification messages are not managed in the message buffer.
■ Notification messages can only be deleted using the buttons **OK** and **Confirm all**.

Function of a user-defined notification message

■ Notification messages are suitable for displaying general information.
■ A notification message is only generated. It is possible to check whether the message has arrived.
■ As notification messages are not managed, **approx. 3 million** messages can be generated.

Programming user-defined notification messages

1. Load the main program into the editor.
2. Declare working variables for:
   ■ Originator, message number, message text (from `KrlMsg_T`)
   ■ Arrays with 3 elements for the parameters (from `KrlMsgPar_T`)
   ■ General message options (from `KrlMsgOpt_T`)
   ■ "Handle" (as `INT`)
3. Initialize working variables with the desired values.
4. Program the function call `Set_KrlMsg(…)`.
5. If required, evaluate "Handle" to check that generation was successful.
6. Close and save the main program.

Fig. 4-4: Notification message

Programming example for the above display:

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
...
mymessage={modul[] "College", Nr 1906, msg_txt[] "My first Message"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Placeholders are empty, placeholder[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#NOTIFY, mymessage, Parameter[ ], Option)
```

4.3 Exercise: Programming notification messages

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

- Program customized notification messages
- Freely output parameters in messages

Preconditions

The following are preconditions for successful completion of the exercise:

- Knowledge of the KRL programming language
- Theoretical knowledge of message programming

Task description

Subtask 1: Notification message

1. Create a notification message with the text "Magazine almost empty – Refill".
2. This message is to be displayed using input 13 on the control panel.
3. Test your program in accordance with the instructions.

Subtask 2: Notification message with parameter

1. Create a notification message with the text "Part number xxx is ready".
2. This message is to be displayed using input 16 on the control panel; the part counter for this part is to be incremented and displayed at position xxx.
3. Test your program in accordance with the instructions.

What you should now know:

1. How is a notification message deleted again?

....

....

2. Which component in the message structure is responsible for causing the message text to be generated?

. . . .

. . . .

2. Which component in the message structure is responsible for causing the message text to be generated?

4.4 Working with a status message

Description of a user-defined status message

Fig. 4-5: Status message

- Status messages are managed in the message buffer.
- Status messages **cannot** be deleted again using the "All OK" button.
- Status messages must be deleted by means of a function in the program.
- Status messages can also be deleted by means of settings in the message options when the program is reset or exited, or in the case of a block selection.

Function of a user-defined status message

- Status messages are suitable for indicating a change of status (e.g. elimination of an input).
- A maximum of 100 messages are managed in the message buffer.
- The program is stopped, for example, until the status that triggered it is no longer active.
- The status message is deleted again using the function `Clear_KrlMsg( )`.

> ℹ️ No predefined reactions of the robot system are linked to the different message types (e.g. robot brakes or program is stopped). The desired reactions must be programmed.

Programming user-defined status messages

1. Load the main program into the editor.
2. Declare working variables for:
   - Originator, message number, message text (from `KrlMsg_T`)
   - Arrays with 3 elements for the parameters (from `KrlMsgPar_T`)
   - General message options (from `KrlMsgOpt_T`)
   - "Handle" (as `INT`)
   - Variable for the result of the check (as `BOOL`)
   - Variable for the result of the deletion (as `BOOL`)
3. Initialize working variables with the desired values.
4. Program the function call `Set_KrlMsg(…)`.
5. Stop the program with a loop until the status that triggered the message is no longer applicable.
6. Delete the status message with the function call `Clear_KrlMsg( )`.
7. Close and save the main program.

Fig. 4-6: Status message

Programming example for the above display/message:

> ℹ️ The status message is triggered by the status of input 12 (`FALSE`). Once the message has been generated, the program is stopped. The message is deleted by the status of input 17 (`TRUE`). Program execution is then resumed.
>
> The message also disappears if the program is reset or exited. This is due to the setting `Clear_P_Reset TRUE` in the message options.

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
DECL BOOL present, eraser
...
IF $IN[12]==FALSE THEN
mymessage={modul[] "College", Nr 1909, msg_txt[] "My Messagetext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Placeholders are empty, placeholder[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#STATE, mymessage, Parameter[ ], Option)
ENDIF
eraser=FALSE
;Loop for stopping until this message has been cleared
REPEAT
IF $IN[12]==TRUE THEN
eraser=Clear_KrlMsg(handle) ;Delete message
ENDIF
present=Exists_KrlMsg(handle) ;Additional check
UNTIL NOT(present) or eraser
```

4.5   Exercise: Programming status messages

Aim of the exercise   On successful completion of this exercise, you will be able to carry out the following activities:

- Program customized status messages
- Freely output parameters in messages

Preconditions   The following are preconditions for successful completion of the exercise:

- Knowledge of the KRL programming language
- Theoretical knowledge of message programming

Task description   Subtask 1: Status message

1. Create a status message with the text "Magazine almost empty".
2. This message is to be displayed using input 14 on the control panel.
3. This message is to be deleted again by resetting input 14 on the control panel.
4. Test your program in accordance with the instructions.

Subtask 2: Status message with parameter

1. Create a status message with the text "There are still xxx of yyy cubes in the magazine".
2. This message is to be displayed using input 15 on the control panel.
3. This message is to be deleted again by resetting input 15 on the control panel.

4.   Test your program in accordance with the instructions.

What you should now know:

1. What does %2 mean in the message text?

. . . .

. . . .

4.6 Working with an acknowledgement message

Description of a user-defined acknowledgement message

Fig. 4-7: Acknowledgement message

- Acknowledgement messages are managed in the message buffer.
- Acknowledgement messages can be deleted using the buttons **OK** and **Confirm all**.
- Acknowledgement messages can also be deleted by means of a function in the program.
- Acknowledgement messages can also be deleted by means of settings in the message options when the program is reset or exited, or in the case of a block selection.

Function of a user-defined acknowledgement message

- Acknowledgement messages are suitable for displaying information of which the user must be made aware.
- A maximum of 100 messages are managed in the message buffer.
- In the case of an acknowledgement message (unlike a notification message), it is possible to check whether or not the user has acknowledged it.
- The program is stopped, for example, until the message has been acknowledged.

> ℹ️ No predefined reactions of the robot system are linked to the different message types (e.g. robot brakes or program is stopped). The desired reactions must be programmed.

Programming user-defined acknowledgement messages

1. Load the main program into the editor.
2. Declare working variables for:
   - Originator, message number, message text (from `KrlMsg_T`)
   - Arrays with 3 elements for the parameters (from `KrlMsgPar_T`)
   - General message options (from `KrlMsgOpt_T`)
   - "Handle" (as `INT`)
   - Variable for the result of the check (as `BOOL`)
3. Initialize working variables with the desired values.
4. Program the function call `Set_KrlMsg(…)`.
5. Stop the program with a loop.
6. Use the function call `Exists_KrlMsg(...)` to check whether the message has already been acknowledged by the user; if the message has been acknowledged, the above loop will have been exited.
7. Close and save the main program.

Fig. 4-8: Acknowledgement message

Programming example for the above display/message:

> ℹ️ Once the message has been generated, the program is stopped. The message is deleted by pressing **OK** or **All OK**. Program execution is then resumed.
>
> The message also disappears if the program is reset or exited. This is due to the setting `Clear_P_Reset TRUE` in the message options.

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
DECL BOOL present
...
mymessage={modul[] "College", Nr 1909, msg_txt[] "My Messagetext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Placeholders are empty, placeholder[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#QUIT, mymessage, Parameter[ ], Option)

;Loop for stopping until this message has been cleared
REPEAT
present=Exists_KrlMsg(handle)
UNTIL NOT(present)
```

4.7 Exercise: Programming acknowledgement messages

Aim of the exercise
On successful completion of this exercise, you will be able to carry out the following activities:

- Program customized acknowledgement messages
- Freely output parameters in messages

Preconditions
The following are preconditions for successful completion of the exercise:

- Knowledge of the KRL programming language
- Theoretical knowledge of message programming

Task description
Subtask 1: Acknowledgement message

1. Create an acknowledgement message with the text "Ackn. Fault - Vacuum not established".
2. This message is to be displayed using input 15 on the control panel.
3. Test your program in accordance with the instructions.

Subtask 2: Status message with acknowledgement message

1. Create a status message with the text "Fault - Vacuum not established".
2. This message is to be displayed using input 13 on the control panel.

3.   When the input is reset, the status message is to be cleared and your acknowledgement message programmed in subtask 1 is to be displayed.

4.   Test your program in accordance with the instructions.

What you should now know:

1. What is the maximum number of acknowledgement messages managed in the message buffer.

. . . .

. . . .

4.8 Working with a wait message

Description of a user-defined wait message

Fig. 4-9: Wait message

- Wait messages are managed in the message buffer.
- Wait messages can be deleted again using the **Simulate** button.
- Wait messages **cannot** be deleted again using the **All OK** button.
- Wait messages can also be deleted by means of settings in the message options when the program is reset or exited, or in the case of a block selection.

Function of a user-defined wait message

- Wait messages are suitable for waiting for a state while displaying the wait icon.
- A maximum of 100 messages are managed in the message buffer.
- The program is stopped, for example, until the status that is being waited for is active.
- The wait message is deleted again using the function `Clear_KrlMsg( )`.

Programming user-defined wait messages

1. Load the main program into the editor.
2. Declare working variables for:
   - Originator, message number, message text (from `KrlMsg_T`)
   - Arrays with 3 elements for the parameters (from `KrlMsgPar_T`)
   - General message options (from `KrlMsgOpt_T`)
   - "Handle" (as `INT`)
   - Variable for the result of the check (as `BOOL`)
   - Variable for the result of the deletion (as `BOOL`)
3. Initialize working variables with the desired values.
4. Program the function call `Set_KrlMsg(…)`.
5. Stop the program with a loop until the status that is being waited for is active or the message has been deleted using the **Simulate** button.
6. Delete the wait message with the function call `Clear_KrlMsg( )`.
7. Close and save the main program.

Fig. 4-10: Wait message

Programming example for the above display/message:

> **i** Once the message has been generated, the program is stopped. The message is deleted by the status of input 12 (`TRUE`). Program execution is then resumed.
>
> The message also disappears if the program is reset or exited. This is due to the setting `Clear_P_Reset TRUE` in the message options.

```
DECL KRLMSG_T mymessage
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGOPT_T Option
DECL INT handle
DECL BOOL present, eraser
...
IF $IN[12]==FALSE THEN
mymessage={modul[] "College", Nr 1909, msg_txt[] "My Messagetext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Placeholders are empty, placeholder[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
handle = Set_KrlMsg(#WAITING, mymessage, Parameter[ ], Option)
ENDIF
eraser=FALSE
;Loop for stopping until this message has been cleared
REPEAT
IF $IN[12]==TRUE THEN
eraser=Clear_KrlMsg(handle) ;Meldung loeschen
ENDIF
present=Exists_KrlMsg(handle) ;kann über simuliere geloescht worden
sein
UNTIL NOT(present) or eraser
```

4.9 Exercise: Programming wait messages

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

- Program customized wait messages
- Freely output parameters in messages

Preconditions

The following are preconditions for successful completion of the exercise:

- Knowledge of the KRL programming language
- Theoretical knowledge of message programming

Task description

1. Create a wait message with the text "Wait for operator input" and activate it via input 12.
2. Clear the wait message by resetting input 12.
3. Test your program in accordance with the instructions.

What you should now know:

1. What is the difference between a "STATE" and a "WAITING" message?

....

....

4.10 Working with dialogs

Description of a user-defined dialog

Fig. 4-11: Dialog

- A dialog cannot be generated until no other dialog is active.
- Dialogs can be deleted using a softkey that can be labeled by the programmer.
- Up to 7 softkeys can be defined.

Function of a user-defined dialog

- Dialogs are suitable for displaying questions that must be answered by the user.
- The function `Set_KrlDlg()` generates a dialog.
- The function merely generates the dialog.
- It does not wait until the dialog has been answered.
- The function `Exists_KrlDlg()` can be used to check whether a specific dialog still exists.
- This function also does not wait until the dialog has been deleted, but merely searches the buffer for the dialog with this handle.
- The KRL program must therefore be polled cyclically until the dialog has been answered or deleted.
- How program execution continues can be made dependent on which softkey the user selects.

Fig. 4-12: Dialog

Evaluation of the buttons

- Declaration and initialization of the buttons

```
DECL KRLMSGDLGSK_T Softkey[7] ; Prepare 7 possible softkeys
softkey[1]={sk_type #value, sk_txt[] "key1"}
softkey[2]={sk_type #value, sk_txt[] "key2"}
softkey[3]={sk_type #value, sk_txt[] "key3"}
softkey[4]={sk_type #value, sk_txt[] "key4"}
softkey[5]={sk_type #value, sk_txt[] "key5"}
softkey[6]={sk_type #value, sk_txt[] "key6"}
softkey[7]={sk_type #value, sk_txt[] "key7"}
```

- Evaluation by means of `Exists_KrlDlg()`: The button created with index 4 sends 4 as the return value.

```
; Softkey no. 4 sends 4 as the return value
softkey[4]={sk_type #value, sk_txt[] "key4"}
```
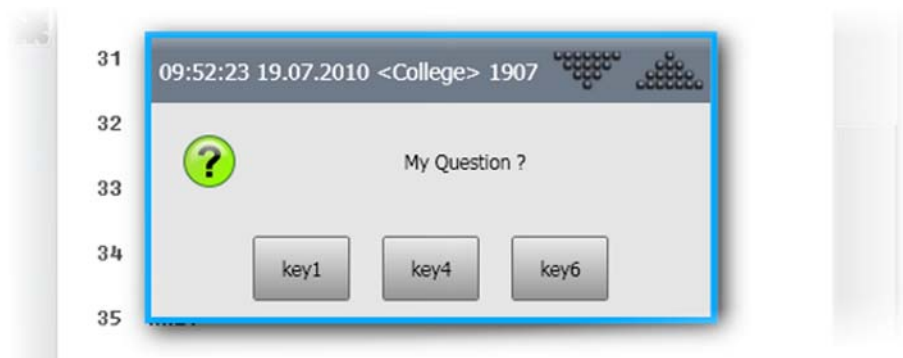

Fig. 4-13: Dialog with 3 buttons

> If not all buttons are programmed, i.e. gaps are left (no. 1, 4, 6), the buttons are arranged consecutively. If only buttons 1, 4 and 6 are used, only return values 1, 4 and 6 are possible.

Programming user-defined dialogs

1. Load the main program into the editor.
2. Declare working variables for:
   - Originator, message number, message text (from `KrlMsg_T`)
   - Arrays with 3 elements for the parameters (from `KrlMsgPar_T`)
   - 7 possible buttons (from `KrlMsgDlgSK_T`)
   - General message options (from `KrlMsgOpt_T`)
   - "Handle" (as `INT`)
   - Variable for the result of the check (as `BOOL`)
   - Variable for the result of the answer as to which button was pressed (as `INT`)
3. Initialize working variables with the desired values.
4. Program the function call `Set_KrlDlg(…)`.
5. Stop the program with a loop until the dialog has been answered.
6. Evaluate the dialog with the function call `Exists_KrlDlg( )`.
7. Plan and program additional branches in the program.
8. Close and save the main program.

Fig. 4-14: Dialog

Programming example for the above display/message:

> **i** Once the dialog has been generated, the program is stopped. The message is deleted once it has been answered. Program execution is then resumed. A switch statement is then programmed.
> The message also disappears if the program is reset or exited. This is due to the setting `Clear_P_Reset TRUE` in the message options.

```
DECL KRLMSG_T myQuestion
DECL KRLMSGPAR_T Parameter[3]
DECL KRLMSGDLGSK_T Softkey[7] ;Prepare 7 possible softkeys
DECL KRLMSGOPT_T Option
DECL INT handle, answer
DECL BOOL present
...

myQuestion={modul[] "College", Nr 1909, msg_txt[] "My Questiontext"}
Option= {VL_STOP FALSE, Clear_P_Reset TRUE, Clear_P_SAW FALSE,
Log_to_DB TRUE}
;Placeholders are empty, placeholder[1..3]
Parameter[1] = {Par_Type #EMPTY}
Parameter[2] = {Par_Type #EMPTY}
Parameter[3] = {Par_Type #EMPTY}
softkey[1]={sk_type #value, sk_txt[] "key1"} ; Button 1
softkey[2]={sk_type #value, sk_txt[] "key2"} ; Button 2
softkey[3]={sk_type #value, sk_txt[] "key3"} ; Button 3
softkey[4]={sk_type #value, sk_txt[] "key4"} ; Button 4
softkey[5]={sk_type #value, sk_txt[] "key5"} ; Button 5
softkey[6]={sk_type #value, sk_txt[] "key6"} ; Button 6
softkey[7]={sk_type #value, sk_txt[] "key7"} ; Button 7
...
```

```
...; Generate dialog
handle = Set_KrlDlg(myQuestion, Parameter[ ],Softkey[ ], Option)
answer=0
REPEAT ; Loop for stopping until this dialog has been answered
present = exists_KrlDlg(handle ,answer) ; Answer is written by the
system
UNTIL NOT(present)
...
SWITCH answer
CASE 1 ; Button 1
; Action for button 1
...
CASE 2 ; Button 2
; Action for button 2
...
...
CASE 7 ; Button 7
; Action for button 7
ENDSWITCH
...
```

4.11   Exercise: Programming a dialog

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

- Program customized notification, status and acknowledgement messages
- Program customized query dialogs
- Freely output parameters in messages

Preconditions

The following are preconditions for successful completion of the exercise:

- Knowledge of the KRL programming language
- Theoretical knowledge of message programming

Task description

Subtask 1: Dialog message

1. Create a dialog message with the text "Select a new part".
2. Make 4 different parts available and assign the 5th softkey the name "END".
3. On selection of a part, generate a notification message with the text "Part xxx selected". Use any existing basic modules for this.
4. Test your program in accordance with the instructions.

What you should now know:

1. How are the softkeys in the dialog labeled?

....

....

5 Interrupt programming

5.1 Programming interrupt routines

Description of interrupt routines
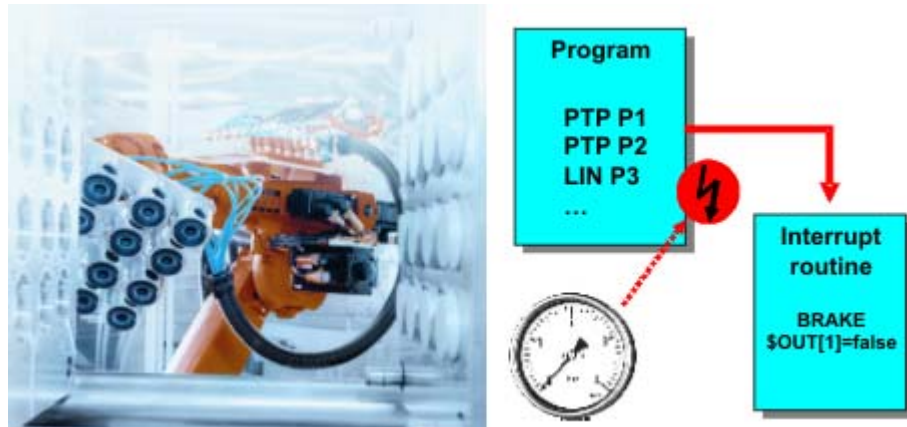
Fig. 5-1: Working with interrupt routines

- In the case of a defined event, e.g. an input, the controller interrupts the current program and executes a defined subprogram.
- A subprogram called by an interrupt is called an interrupt program.
- A maximum of 32 interrupts may be declared simultaneously.
- Up to 16 interrupts may be active at any one time.

Important steps when using an interrupt

- Interrupt declaration
- Interrupt activation/deactivation or disabling/enabling
- Possibly stopping the robot
- Possibly rejecting the current path planning and executing a new path

Principle of interrupt declaration

General information about the declaration of interrupts

- In the case of a defined event, e.g. an input, the controller interrupts the current program and executes a defined subprogram.
- The event and the subprogram are defined by `INTERRUPT ... DECL ... WHEN ... DO ....`
-

> ℹ The interrupt declaration is a statement. It must be situated in the statements section of the program and not in the declaration section!

-

> ℹ When first declared, an interrupt is deactivated. The interrupt must be activated before the system can react to the defined event!

Syntax of the interrupt declaration

-

```
<GLOBAL> INTERRUPT DECL Prio WHEN Ereignis DO Interruptprogramm
```

- `Global`
    - An interrupt is only recognized at, or below, the level in which it is declared.
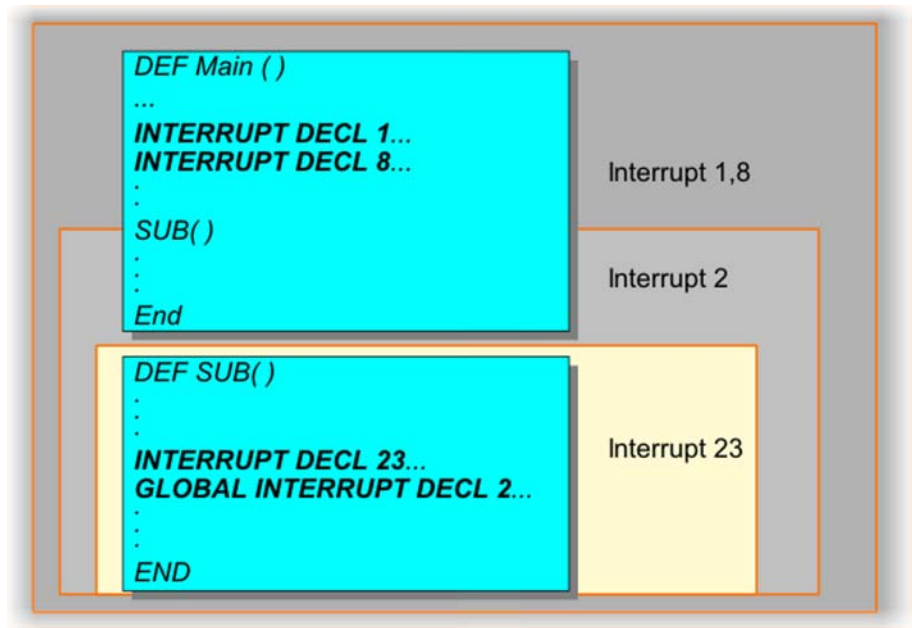
Fig. 5-2: Validity of interrupts

- ■ An interrupt declared in a subprogram is not recognized in the main program (in this case: interrupt 23).
- ■ An interrupt preceded by the keyword `GLOBAL` in the declaration is also recognized at the higher levels (in this case: interrupt 2).

■

```
<GLOBAL> INTERRUPT DECL Prio WHEN Ereignis DO Unterprogramm
```

- ■ *Prio*: Priority
  - ■ Priorities 1, 2, 4 to 39 and 81 to 128 are available.
  - ■ Priorities 3 and 40 to 80 are reserved for use by the system.
  - ■ Interrupt 19 may be preset for the brake test.
  - ■ If several interrupts occur at the same time, the interrupt with the highest priority is processed first, then those of lower priority. **(1 = highest priority)**
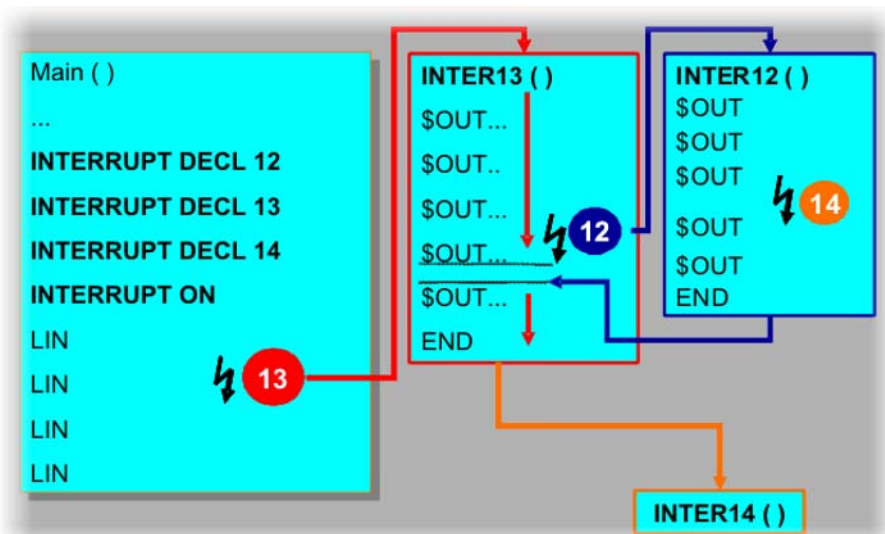
Fig. 5-3: Priorities of interrupts

■

```
<GLOBAL> INTERRUPT DECL Prio WHEN Ereignis DO Unterprogramm
```

■ *Event*: Event that is to trigger the interrupt.

> ℹ This event is detected by means of an edge when it occurs (edge-trig-gered).

■ *Interrupt program*
  ■ The name of the interrupt program to be executed.
  ■ This subprogram is referred to as the interrupt program.
  ■ Runtime variables must not be transferred to the interrupt program as parameters.
  ■ Variables declared in a data list are permissible.
■ Example: declaration of an interrupt

```
INTERRUPT DECL 23 WHEN $IN[12]==TRUE DO INTERRUPT_PROG(20,VALUE)
```

  ■ No global interrupt
  ■ Priority: 23
  ■ Event: positive edge of input 12
  ■ Interrupt program: INTERRUPT_PROG(20,VALUE)
  ■

> ℹ When first declared, an interrupt is deactivated. The interrupt must be activated before the system can react to the defined event!

Description of interrupt activation/deacti-vation/disabling/ enabling

Once an interrupt has been declared, it must then be activated.

Possibilities of the command `INTERRUPT ...`:

■ Activates an interrupt.
■ Deactivates an interrupt.
■ Disables an interrupt.
■ Enables an interrupt.

Syntax

■ `INTERRUPT` *Action* `<`*Number*`>`

  Action

  ■ `ON`: Activates an interrupt.
  ■ `OFF`: Deactivates an interrupt.
  ■ `DISABLE`: Disables an activated interrupt.
  ■ `ENABLE`: Enables a disabled interrupt.

  Number

  ■ Number (= priority) of the interrupt to which the *Action* is to refer.
  ■ *Number* can be omitted.
    In this case, `ON` or `OFF` refers to all declared interrupts, while `DISABLE` or `ENABLE` refers to all active interrupts.

Activating and deactivating interrupts

```
INTERRUPT DECL 20 WHEN $IN[22]==TRUE DO SAVE_POS( )
...
INTERRUPT ON 20
;Interrupt is recognized and executed (positive edge)
...
INTERRUPT OFF 20 ; Interrupt is switched off
```

> **i** ■In this case, the interrupt is triggered by a change of state, e.g. in the case of $IN[22]==TRUE by the change from FALSE to TRUE. The state must therefore not already be present at INTERRUPT ON, as the interrupt is not then triggered!
>
> ■ Furthermore, the following must also be considered in this case: the change of state must not occur until at least one interpolation cycle after INTERRUPT ON.
>
> (This can be achieved by programming a WAIT SEC 0.012 after INTERRUPT ON. If no advance run stop is desired, a CONTINUE command can also be programmed before the WAIT SEC.)
>
> The reason for this is that INTERRUPT ON requires one interpolation cycle (= 12 ms) before the interrupt is actually activated. If the state changes before this, the interrupt cannot detect the change.

Activating and deactivating interrupts: contact bouncing

> **i** If there is a risk of an interrupt being incorrectly triggered twice because of sensitive sensors ("contact bouncing"), you can prevent this by switching off the interrupt in the first line of the interrupt program. However, a genuine interrupt arising during interrupt processing can now no longer be recognized. If the interrupt is to remain active, it must be switched back on before returning to the main program.

Disabling and enabling interrupts

```
INTERRUPT DECL 21 WHEN $IN[25]==TRUE DO INTERRUPT_PROG( )
...
INTERRUPT ON 21
;Interrupt is recognized and immediately executed (positive edge)
...
INTERRUPT DISABLE 21
;Interrupt is recognized and saved but not executed (positive edge)
...
INTERRUPT ENABLE 21
; Saved interrupts are not executed until now
...
INTERRUPT OFF 21 ; Interrupt is switched off
...
```

> **i** A disabled interrupt is recognized and saved. The interrupt is executed directly after it has been enabled. In the case of motions, it must be ensured that there is no risk of collision.

Description of braking the robot or canceling the current motion by means of an interrupt routine

Braking the robot

■ A robot is to stop immediately when a certain event occurs.

■ There are two braking ramps available (STOP 1 and STOP 2).

■ The interrupt program is not continued until the robot has come to a stop.

■ The robot motion that had been started is resumed as soon as the interrupt program has been completed.

■ Syntax:

- `BRAKE`: STOP 2
- `BRAKE F`: STOP 1

> **i** BRAKE may only be used in an interrupt program.

Motions and interrupt routines

- The robot moves while the interrupt routine is being executed in parallel.
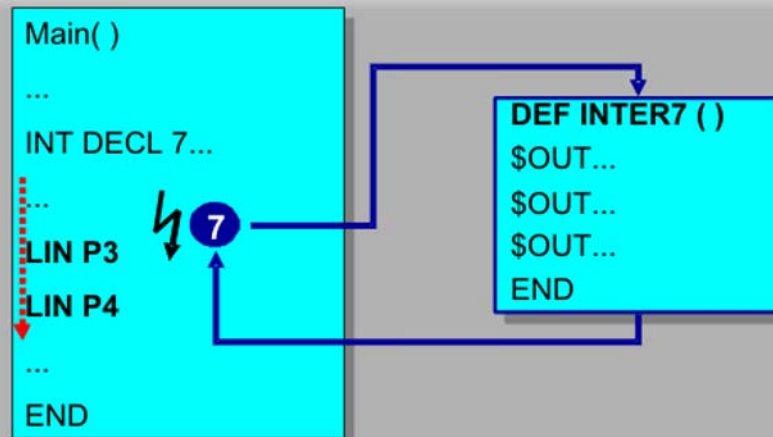
Fig. 5-4: Execution of interrupt routines

> **i** If the time required for execution of the interrupt routine is shorter than that for the path planning in the main program, the robot can continue moving without stopping. If the time required for the interrupt routine is greater than the planned path, the robot stops at the end of the path planning and resumes motion as soon as the interrupt routine has been executed.

- Inline forms for the initialization (`INI`) or for motions (e.g. `PTP` or `LIN` ...) are not permissible. These result in error messages during execution.
- The robot is stopped with `BRAKE` and resumes motion at the end of the interrupt routine with the path planned in the main program.
- The robot is stopped with `BRAKE` and moved in the interrupt routine. On completion of the interrupt routine, the path from the main program is resumed.

> **⚠ CAUTION** It must be ensured that there is no risk of collision!
> Failure to observe this may result in death to persons, physical injuries or damage to property.

- The robot is stopped with `BRAKE` and is to execute a new path after completion of the interrupt routine. This can be implemented using the `RESUME` command.
- The robot is stopped with `BRAKE` and moved in the interrupt routine. On completion of the interrupt routine, the current path is no longer to be resumed and new path planning is carried out. This can also be implemented using the `RESUME` command.

> **⚠ CAUTION** Since it cannot be predicted precisely when the interrupt will be triggered, it must be ensured that collision free motion during the interrupt routine and the subsequent motion is possible at all possible positions along the current robot path.
> Failure to observe this may result in death to persons, physical injuries or damage to property.

Canceling the current motion by means of `RESUME`

- `RESUME` cancels all running interrupt programs and subprograms up to the level at which the current interrupt was declared.
- When the `RESUME` statement is activated, the advance run pointer must not be at the level where the interrupt was declared, but at least one level lower.
- `RESUME` may only occur in interrupt programs.
- As soon as an interrupt has been declared as `GLOBAL`, **no** `RESUME` command may be used in the interrupt routine.
- Changing the variable $BASE in the interrupt program only has an effect there.
- The computer advance run, i.e. the variable $ADVANCE, must not be modified in the interrupt program.
- Motions that are to be canceled by means of BRAKE and RESUME must be programmed in a subprogram.
- The behavior of the robot controller after `RESUME` depends on the following motion instruction:
    - PTP instruction: is executed as a PTP motion.
    - LIN instruction: is executed as a LIN motion.
    - CIRC instruction: is always executed as a LIN motion!

    Following a RESUME statement, the robot is not situated at the original start point of the CIRC motion. The motion will thus differ from how it was originally planned; this can potentially be very dangerous, particularly in the case of CIRC motions.

> ⚠ **WARNING**   If the first motion instruction after RESUME is a CIRC motion, this is always executed as LIN! This must be taken into consideration when programming RESUME statements. The robot must be able to reach the end point of the CIRC motion safely, by means of a LIN motion, from any position in which it could find itself when the RESUME statement is executed.
> Failure to observe this may result in death to persons, injuries or damage to property.

- Useful system variables for exact positioning
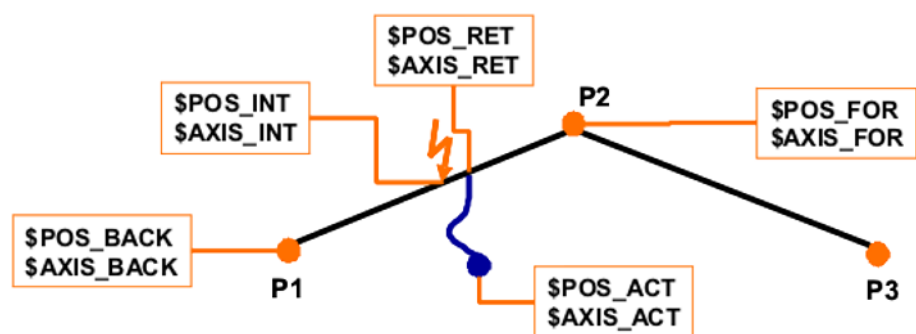
Fig. 5-5: System variables for exact positioning

- Useful system variables for approximate positioning
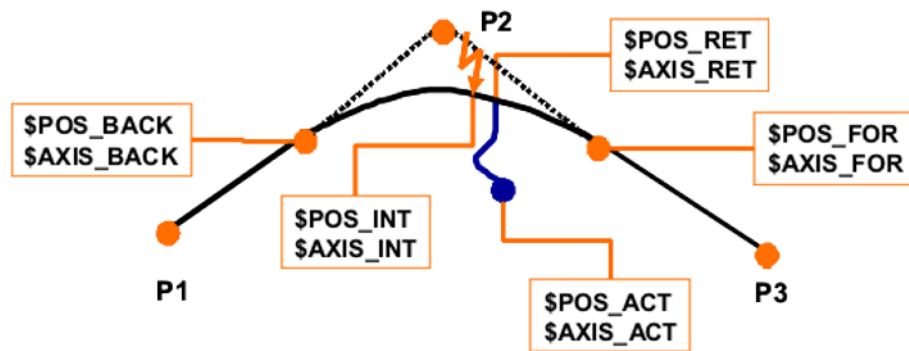
Fig. 5-6: System variables for approximate positioning

Programming interrupt routines

Execution of logic parallel to the robot motion

1. Interrupt declaration
   - Define priority.
   - Determine trigger event.
   - Define and create interrupt routine.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )

END

_____
DEF ERROR()

END
```

2. Activate and deactivate the interrupt.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
...
...
INTERRUPT OFF 25

END

_____
DEF ERROR()

END
```

3. Expand the program with motions and define actions in the interrupt routine.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
INTERRUPT OFF 25
END
_____
DEF ERROR()
$OUT[20]=FALSE
$OUT[21]=TRUE
END
```

Execution of logic after the robot has been stopped, followed by resumption of the robot motion

1. Interrupt declaration
    ■ Define priority.
    ■ Determine trigger event.
    ■ Define and create interrupt routine.
    ■ Activate and deactivate the interrupt.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
...
...
INTERRUPT OFF 25

END
_____
DEF ERROR()

END
```

2. Expand the program with motions and define robot braking and logic in the interrupt routine.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
INTERRUPT OFF 25
END
_____
DEF ERROR()
BRAKE
$OUT[20]=FALSE
$OUT[21]=TRUE
END
```

Stopping the current robot motion, repositioning, rejecting the current path planning and executing a new path

1. Interrupt declaration
    ■ Define priority.

■ Determine trigger event.

■ Define and create interrupt routine.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
...
END
_____
DEF ERROR()
...
END
```

2. Expand program with motions.

■ In order to be able to cancel a motion, it must be executed in a subprogram.

■ Advance run pointer must remain in the subprogram.

■ Activate and deactivate the interrupt.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
SEARCH()
END
_____
DEF SEARCH()
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
WAIT SEC 0 ; Stop advance run pointer
INTERRUPT OFF 25
END
_____
DEF ERROR()
...
END
```

3. Edit the interrupt routine.

■ Stop the robot.

■ Reposition the robot to $POS_INT.

■ Reject current motion.

■ New motion in main program.

```
DEF MY_PROG( )

INI
INTERRUPT DECL 25 WHEN $IN[99]==TRUE DO ERROR( )
SEARCH()
END
_____
DEF SEARCH()
INTERRUPT ON 25
PTP HOME Vel=100% DEFAULT
PTP P1 Vel=100% PDAT1
PTP P2 Vel=100% PDAT2
PTP HOME Vel=100% DEFAULT
WAIT SEC 0 ; Stop advance run pointer
INTERRUPT OFF 25
END
_____
DEF ERROR()
BRAKE
PTP $POS_INT
RESUME
END
```

5.2   Exercise: Working with interrupts

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

- Declare an interrupt
- Create an interrupt subprogram
- Evaluate and process interrupts in the program sequence

Preconditions

The following are preconditions for successful completion of this exercise:

- Knowledge of the KRL programming language
- Theoretical knowledge of interrupt programming

Task description

The aim of this exercise is to detect the positions of 3 cubes using a defined measurement run and to save these positions.

1. Create a new program with the name "SEARCH".
2. Remove three cubes from the magazine (you, not the robot) and place them in a line on the table
3. Teach a LIN motion as a search run to pass over the three cubes The velocity is to be set to 0.2 m/s.
4. The sensor must be activated/deactivated via output 27. Checkback signals for position determination are received at input 27.
5. Output 10 is to be switched for 1 second when a cube is detected. At the same time, the position of the detected cube must be saved. Create and use an array in the local DAT file or $config.dat for this purpose.
6. On completion of the search run, the robot is to indicate the 3 saved positions by moving to each one in turn and waiting for 1 second before moving on to the next position.
7. Test your program in accordance with the instructions.

What you should now know:

1. In which program section is the interrupt declared?

....

....

2. What is the difference between INTERRUPT OFF 99 and INTERRUPT DIS-
ABLE 99?

. . . .

. . . .

. . . .

. . . .

3. When is the interrupt subprogram called?

. . . .

. . . .

4. What is the effect of the command INTERRUPT OFF at the start of an in-
terrupt subprogram?

. . . .

. . . .

5. What priority range is not enabled for the interrupt?

. . . .

. . . .

5.3   Exercise: Canceling motions with interrupts

Aim of the exercise   On successful completion of this exercise, you will be able to carry out the following activities:

■ Declare an interrupt

■ Create an interrupt subprogram

■ Evaluate and process interrupts in the program sequence

■ Brake the robot motion by means of a KRL command

■ Brake and cancel the robot motion by means of KRL commands

Preconditions   The following are preconditions for successful completion of this exercise:

■ Knowledge of the KRL programming language

■ Theoretical knowledge of interrupt programming

■ Theoretical knowledge of the KRL commands for braking and canceling robot motion and how to use them

Task description   Using a defined measurement run, detect the positions of 3 cubes and save these positions. Additionally, the measurement run is to be terminated as soon as the third cube has been detected.

1. Duplicate your program SEARCH and rename it CANCEL_SEARCH.

2. Remove three cubes from the magazine (you, not the robot) and place them in a line on the table

3. Teach a LIN motion as a search run to pass over the three cubes The velocity is to be set to 0.2 m/s. The sensor must be activated/deactivated via output 27. Checkback signals for position determination are received at input 27.

4. Output 10 is to be switched for 1 second when a cube is detected. At the same time, the position of the detected cube must be saved. Create and use an array in the local DAT file for this purpose.

5. Once the 3rd cube has been found, the robot is to stop immediately and the search run is to be canceled.

6. On completion of the search run, the robot is to indicate the 3 saved positions by moving to each one in turn and waiting for 1 second before moving on to the next position.

7. Test your program in accordance with the instructions.

What you should now know:

1. What is the difference between BRAKE and BRAKE F?

. . . .

. . . .

2. Why does the RESUME command not function correctly here?
INTERRUPT DECL 21 WHEN $IN[1] DO located( )
INTERRUPT ON 21
LIN Strtpt
LIN Endpt
$ADVANCE = 0
INTERRUPT OFF 21 ...
END

―――――――――――――――――――――――――――――

DEF located( )
INTERRUPT OFF 21
BRAKE
;pick_part
RESUME
END

3. When is an interrupt triggered?

. . . .

. . . .

6 Programming return motion strategies

6.1 Programming return motion strategies

What is a return motion strategy?

Once an application program has been created and tested in practical operation, the question additionally arises as to how the program will react to malfunctions.

Of course it is desirable for the system to react automatically to a malfunction.

Return motion strategies are used for this purpose.

A return motion strategy involves return motions that the robot executes in the event of a malfunction in order to move automatically to the home position, for example, no matter where it is currently located.

These return motions must be freely programmed by the programmer.

Where are return motion strategies used?

Return motion strategies are used wherever full automation of a production cell is desired, even in the case of malfunctions.

A correctly programmed return motion strategy might only give the operator an opportunity to decide what is to happen in the further procedure.

The need to jog the robot out of a hazardous situation can thus be avoided.
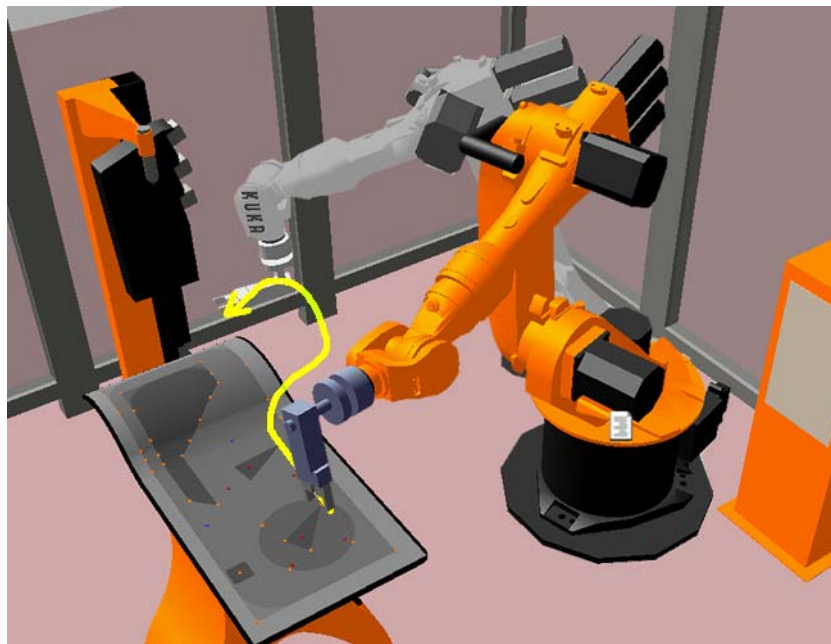
Fig. 6-1

How is a return motion strategy programmed?

- Subdivide motion range into workspaces
- Configure I/Os
- Declare interrupts
- Save positions
- Program user messages
- Define various home positions if necessary
- Use global points if necessary
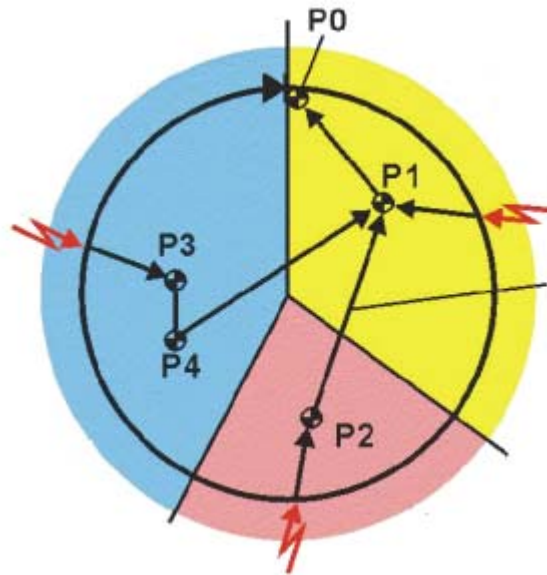
Fig. 6-2
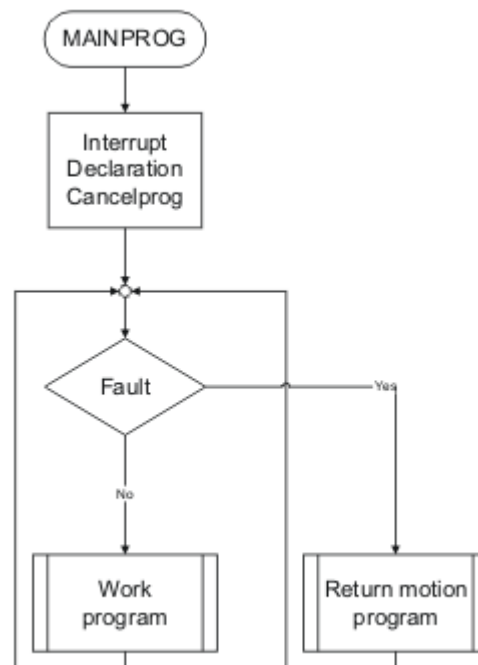
Fig. 6-3

6.2 Exercise: Programming a return motion strategy

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

- Program automatic return motions
- Integrate messages into the work process
- Detect faults using interrupts
- Terminate the robot motion, depending on the process

Preconditions

The following are preconditions for successful completion of this exercise:

- Knowledge of the KRL programming language
- Knowledge of message programming

- Knowledge of interrupt programming
- Knowledge of the KRL commands for braking and canceling robot motion and how to use them
- Theoretical knowledge of the Trigger command

Task description   The basic program involves fetching the cube from the magazine and inserting it back into the magazine. The enabling is withdrawn by the PLC via an input (no. 11). The robot must be stopped immediately. The operator is to decide, by answering a dialog query, whether the robot should return to the home position or resume the process. In either case, the robot cannot move, following the decision, until the enabling is present once again and this fault has been acknowledged. If the home position is selected, the motion to this position is executed at reduced velocity (POV = 10%). In the home position, another dialog will ask whether the system is ready. If "yes", the robot can continue program execution with the program override that was set before the fault occurred. Answering with "no" terminates the program.
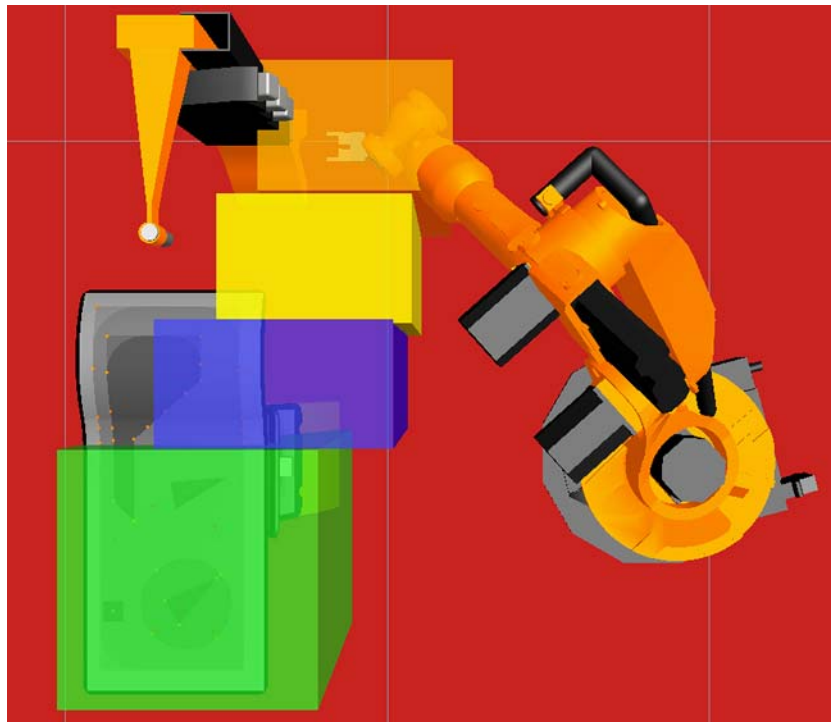
Fig. 6-4

1. Start by creating the program flowchart.
2. Pay attention to the structure of your overall concept when implementing the program structure.
3. The aim of the project is clear, well-structured programming and functioning programs/modules.
4. When assigning file and variable names, make sure they are easy to understand.
5. Ensure that it is always possible to move to the home position without the risk of collision.
6. When restarting from the home position, ensure that the correct procedure (fetch or set down) is carried out, depending on the gripper position. Note: input 26 means that the gripper is open.
7. Test your program in accordance with the instructions.

What you should now know:

1. Which expert command can be used to switch user-defined variables on the path?

. . . .

. . . .

2. What are the KRL commands for the immediate termination of a subprogram and an interrupt subprogram?

. . . .

. . . .

3. What is the purpose of a BCO run?

. . . .

. . . .

4. What variables can be used to influence the program override?

. . . .

. . . .

5. What can additionally be switched using the Trigger command that cannot be switched using the SYNOUT inline form?

. . . .

. . . .

7 Working with analog signals

7.1 Programming analog inputs

Description

Fig. 7-1: Analog signals

- The KR C4 features 32 analog inputs.
- For the analog signals, an optional bus system is required, which must be configured via WorkVisual.
- Analog inputs are read via the system variables `$ANIN[1]` ... `$ANIN[32]`.
- Cyclical reading (every 12 ms) of an analog input.
- The values of `$ANIN[no]` range between 1.0 and -1.0 and represent an input voltage of +10 V to -10 V.

Function

Static value assignment

- Direct value assignment

```
...
REAL value

value = $ANIN[2]
...
```

- Value assignment of a signal declaration

```
...
SIGNAL sensor $ANIN[6]
REAL value

value = sensor
...
```

Dynamic value assignment

- All of the variables used in an ANIN statement must be declared in **data lists** (locally or in `$CONFIG.DAT`).
- A maximum of **three** ANIN ON statements can be used at the same time.
- A maximum of two ANIN ON statements can use the same variable *Value* or access the same analog input.
- Syntax
  - Starting cyclical reading:
    ANIN ON *Value* = *Factor* * *Signal name* <*±Offset*>

Element	Description
Value	Type: REAL
	The result of the cyclical reading is stored in *Value*. *Value* can be a variable or a signal name for an output.
Factor	Type: REAL
	Any factor. It can be a constant, variable or signal name.
Signal name	Type: REAL
	Specifies the analog input. *Signal name* must first have been declared with SIGNAL . It is not possible to specify the analog input $ANIN[x] directly instead of the signal name.
	The values of an analog input $ANIN[x] range between +1.0 and -1.0 and represent a voltage of +10 V to -10 V.
Offset	Type: REAL
	It can be a constant, variable or signal name.

- ▪ Ending cyclical reading:

  ANIN OFF *Signal name*

■ Example 1

```
DEFDAT myprog
DECL REAL value = 0
ENDDAT
```

```
DEF myprog( )
SIGNAL sensor $ANIN[3]
...
ANIN ON value = 1.99*sensor-0.75
...
ANIN OFF sensor
```

■ Example 2

```
DEFDAT myprog
DECL REAL value = 0
DECL REAL corr = 0.25
DECL REAL offset = 0.45
ENDDAT
```

```
DEF myprog( )
SIGNAL sensor $ANIN[7]
...
ANIN ON value = corr*sensor-offset
...
ANIN OFF sensor
```

Procedure for programming analog inputs

> **NOTICE** A precondition for using the analog signals is correct configuration of the bus system with the connected analog signals.

Programming of ANIN ON/OFF

1. Selection of the correct analog input
2. Performance of the signal declaration
3. Declaration of the necessary variables in a data list
4. **Switching on:** Programming of the ANIN ON statement
5. Verification that no more than 3 dynamic inputs are active
6. **Switching off:** Programming of the ANIN OFF statement

7.2 Programming analog outputs
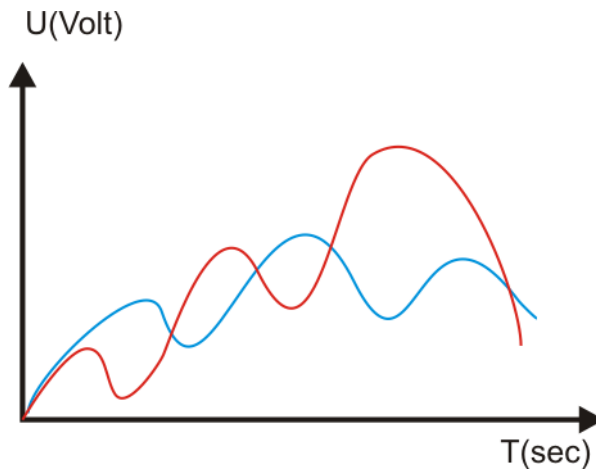
Description
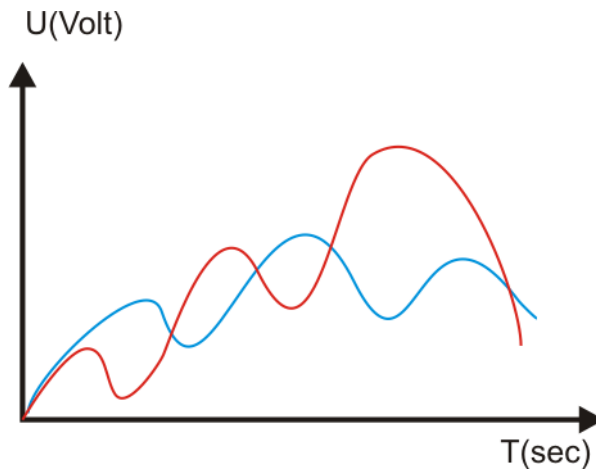
Fig. 7-2: Analog signals

- The KR C4 features 32 analog outputs.
- For the analog signals, an optional bus system is required, which must be configured via WorkVisual.
- Analog inputs are read via the system variables $ANOUT[1] ... $ANOUT[32].
- Cyclical writing (every 12 ms) to an analog output.
- The values of $ANOUT[no] range between 1.0 and -1.0 and represent an output voltage of +10 V to -10 V.

Function

> **NOTICE** A maximum of 8 analog outputs (static and dynamic together) can be used at any one time. ANOUT triggers an advance run stop.

Static value assignment

- Direct value assignment

```
...
ANOUT[2] = 0.7 ; 7 V at analog output 2
...
```

- Value assignment by variables

```
...
REAL value
value = -0.8
ANOUT[4] = value ; -8 V at analog output 4
...
```

- Programming by inline form

Fig. 7-3: Inline form "ANOUT" (static)

Item	Description
1	Number of the analog output ■ **CHANNEL_1 … CHANNEL_32**
2	Factor for the voltage ■ **0 … 1** (intervals: 0.01)

Dynamic value assignment

■ All of the variables used in an ANOUT statement must be declared in **data lists** (locally or in $CONFIG.DAT).

■ A maximum of **four** ANOUT ON statements can be used at the same time.

■ ANOUT triggers an advance run stop.

■ Syntax

   ■ Starting cyclical writing:

      ANOUT ON *Signal name* = *Factor* * *Control element* <±*Offset*> <DELAY = ±*Time*> <MINIMUM = *Minimum value*> <MAXIMUM = *Maximum value*>

Element	Description
Signal name	Type: REAL Specifies the analog output. *Signal name* must first have been declared with SIGNAL . It is not possible to specify the analog output $ANOUT[x] directly instead of the signal name. The values of an analog output $ANOUT[x] range between +1.0 and -1.0 and represent a voltage of +10 V to -10 V.
Factor	Type: REAL Any factor. It can be a constant, variable or signal name.
Control element	Type: REAL It can be a constant, variable or signal name.
Offset	Type: REAL It can be a constant, variable or signal name.
Time	Type: REAL Unit: seconds. By using the keyword DELAY and entering a positive or negative amount of time, the output signal can be delayed (+) or set early (-).
Minimum value, Maximum value	Type: REAL Minimum and/or maximum voltage to be present at the output. The actual value does not fall below/exceed these values, even if the calculated values fall outside this range. Permissible values: -1.0 to +1.0 (corresponds to -10 V to +10 V). It can be a constant, variable, structure component or array element. The minimum value must always be less than the maximum value. The sequence of the keywords MINIMUM and MAXIMUM must be observed.

   ■ Ending cyclical writing:

      ANOUT OFF *Signal name*

■ Example 1

```
DEF myprog( )
SIGNAL motor $ANOUT[3]
...
ANOUT ON motor = 3.5*$VEL_ACT-0.75 DELAY=0.5
...
ANOUT OFF motor
```

■ Example 2

```
DEFDAT myprog
DECL REAL corr = 1.45
DECL REAL offset = 0.25
ENDDAT
```

```
DEF myprog( )
SIGNAL motor $ANOUT[7]
...
ANOUT ON motor = corr*$VEL_ACT-offset
...
ANOUT OFF motor
```

Procedure for programming analog inputs

> **NOTICE** A precondition for using the analog signals is correct configuration of the bus system with the connected analog signals.

Programming of ANOUT ON/OFF

1. Selection of the correct analog output
2. Performance of the signal declaration
3. Declaration of the necessary variables in a data list
4. **Switching on:** Programming of the `ANOUT ON` statement
5. Verification that no more than 4 dynamic outputs are active
6. **Switching off:** Programming of the `ANOUT OFF` statement

Example:
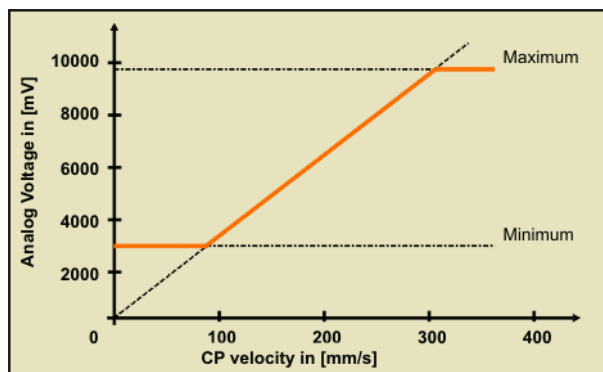
Fig. 7-4: Example of an analog output signal

```
DEF myprog( )
SIGNAL motor $ANOUT[3]
...
ANOUT ON motor = 3.375*$VEL_ACT MINIMUM=0.30 MAXIMUM=0.97
...
ANOUT OFF motor
```

7.3   Exercise: Working with analog I/Os

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

■ Use signal declarations with inputs/outputs
■ Integrate analog inputs into work processes statically or dynamically

■   Integrate analog outputs into work processes statically or dynamically

Preconditions   The following are preconditions for successful completion of this exercise:

■   Theoretical knowledge of signal declarations

■   Theoretical knowledge of integrating analog inputs/outputs

Task description   Configure your system so that you can alter the program override via an analog input. Additionally, the actual robot velocity is to control an analog output.

Subtask 1

1. Create a program with the name "Velocity".
2. Use analog input 1, which is controlled by the potentiometer.
3. Adapt the program override in the Submit interpreter.
4. Test your program in accordance with the instructions.

Subtask 2

1. Expand your program with CP motions (velocity: up to 2 m/s) in an endless loop.
2. Use analog output 1 of the control panel display.
3. Use the system variable $VEL_ACT for the current velocity.
4. Test your program in accordance with the instructions.
5. Additional instructions: Even if the velocity is below 0.2 m/s, the output must nonetheless be connected to 1.0 V, and if the velocity is greater than 1.8 m/s, the output voltage must not exceed 9.0 V.

Make sure that you only activate the analog I/O once.

What you should now know:

1. How many analog I/Os can be used in the KRC controller?

. . . .

. . . .

2. How many predefined digital inputs, analog inputs and analog outputs can the KUKA controller use simultaneously?

. . . .

. . . .

3. What are the KRL commands for starting and stopping the analog output cyclically?

. . . .

. . . .

4. How is an analog input statically polled?

. . . .

. . . .

8 Sequence and configuration of Automatic External

8.1 Configuring and implementing Automatic External

Description

Fig. 8-1: PLC connection

- The Automatic External interface allows robot processes to be controlled by a higher-level controller (e.g. a PLC).
- The higher-level controller transmits the signals for the robot processes (e.g. motion enable, fault acknowledgement, program start, etc.) to the robot controller via the Automatic External interface. The robot controller transmits information about operating states and fault states to the higher-level controller.

To enable use of the Automatic External interface, the following configurations must be carried out:

1. Configuration of the CELL.SRC program.
2. Configuration of the inputs/outputs of the Automatic External interface.

Using the inputs/ outputs of the Automatic External interface

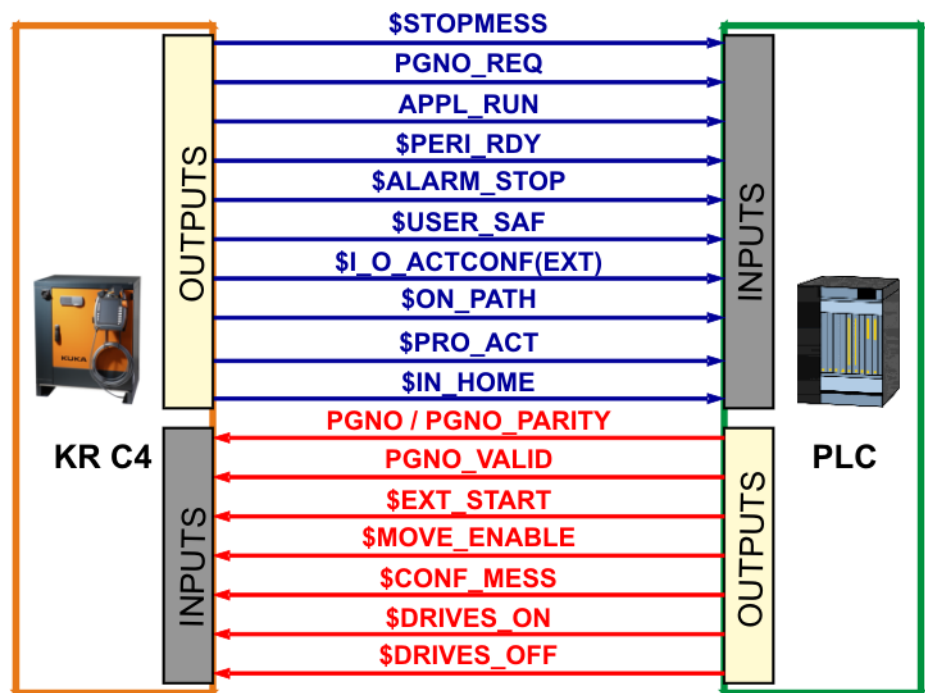Overview of the principal signals of the interface

Fig. 8-2: Overview of the principal Automatic External signals

Inputs (from robot controller's point of view)

■ **PGNO_TYPE** – Program number type

This variable defines the format in which the program number sent by the higher-level controller is read.

Value	Description	Example
1	Read as binary number. The program number is transmitted by the higher-level controller as a binary coded integer.	0 0 1 0 0 1 1 1 => PGNO = 39
2	Read as BCD value. The program number is transmitted by the higher-level controller as a binary coded decimal.	0 0 1 0 0 1 1 1 => PGNO = 27
3	Read as "1 of n"*. The program number is transmitted by the higher-level controller or the periphery as a "1 of n" coded value.	0 0 0 0 0 0 0 1 => PGNO = 1 0 0 0 0 1 0 0 0 => PGNO = 4

* When using this transmission format, the values of PGNO_REQ, PGNO_PARITY and PGNO_VALID are not evaluated and are thus of no significance.

■ **PGNO_LENGTH** – Program number length

This variable determines the number of bits in the program number sent by the higher-level controller. Range of values: 1 … 16.

If PGNO_TYPE has the value 2, only 4, 8, 12 and 16 are permissible values for the number of bits.

■ **PGNO_PARITY** – Program number parity bit

Input to which the parity bit is transferred from the higher-level controller.

Input	Function
Negative value	Odd parity
0	No evaluation
Positive value	Even parity

If PGNO_TYPE has the value 3, PGNO_PARITY is not evaluated.

- **PGNO_VALID** – Program number valid

  Input to which the command to read the program number is transferred from the higher-level controller.

Input	Function
Negative value	Number is transferred at the falling edge of the signal.
0	Number is transferred at the rising edge of the signal on the EXT_START line.
Positive value	Number is transferred at the rising edge of the signal.

- **$EXT_START** – External start

  If the I/O interface is active, this input can be set to start or continue a program (normally CELL.SRC).

> **i** Only the rising edge of the signal is evaluated.

> ⚠ **WARNING** There is no BCO run in Automatic External mode. This means that the robot moves to the first programmed position after the start at the programmed (not reduced) velocity and does not stop there.

- **$MOVE_ENABLE** – Motion enable

  This input is used by the higher-level controller to check the robot drives.

Signal	Function
TRUE	Jogging and program execution are possible.
FALSE	All drives are stopped and all active commands inhibited.

> **i** If the drives have been switched off by the higher-level controller, the message "GENERAL MOTION ENABLE" is displayed. It is only possible to move the robot again once this message has been reset and another external start signal has been given.

> **i** During commissioning, the variable $MOVE_ENABLE is often configured with the value $IN[1025]. If a different input is not subsequently configured, no external start is possible.

- **$CONF_MESS** – Message acknowledgement

  Setting this input enables the higher-level controller to acknowledge error messages automatically as soon as the cause of the error has been eliminated.

> **i** Only the rising edge of the signal is evaluated.

- **$DRIVES_ON** – Drives on

  If there is a high-level pulse of at least 20 ms duration at this input, the higher-level controller switches on the robot drives.

- **$DRIVES_OFF** – Drives off

    If there is a low-level pulse of at least 20 ms duration at this input, the higher-level controller switches off the robot drives.

Outputs (from robot controller's point of view)

- **$ALARM_STOP** – Emergency Stop

    This output is reset in the following EMERGENCY STOP situations:

    - The EMERGENCY STOP button on the KCP is pressed (Int. E-Stop).
    - External EMERGENCY STOP

> **i** In the case of an EMERGENCY STOP, the nature of the EMERGENCY STOP can be recognized from the states of the outputs **$ALARM_STOP** and **Int. E-Stop**:
>
> - Both outputs are FALSE: the EMERGENCY STOP was triggered on the KCP.
> - **$ALARM_STOP** is FALSE, **Int. E-Stop** is TRUE: external EMERGENCY STOP.

- **$USER_SAF** – Operator safety / safety gate

    This output is reset if the safety fence monitoring switch is opened (AUT mode) or an enabling switch is released (T1 or T2 mode).

- **$PERI_RDY** – Drives ready

    By setting this output, the robot controller communicates to the higher-level controller the fact that the robot drives are switched on.

- **$STOPMESS** – Stop messages

    This output is set by the robot controller in order to communicate to the higher-level controller any message occurring which requires the robot to be stopped. (Examples: EMERGENCY STOP, Motion enable or Operator safety)

- **$I_O_ACTCONF** – Automatic External active

    This output is TRUE if Automatic External mode is selected and the input $I_O_ACT is TRUE (normally always at $IN[1025]).

- **$PRO_ACT** – Program is active/running

    This output is set whenever a process is active at robot level. The process is therefore active as long as a program or an interrupt is being processed. Program processing is set to the inactive state at the end of the program only after all pulse outputs and all triggers have been processed.

- **PGNO_REQ** – Program number request

    A change of signal at this output requests the higher-level controller to send a program number.

    If PGNO_TYPE has the value 3, PGNO_REQ is not evaluated.

- **APPL_RUN** – Application program running

    By setting this output, the robot controller communicates to the higher-level controller the fact that a program is currently being executed.

- **$IN_HOME** – Robot in HOME position

    This output communicates to the higher-level controller whether or not the robot is in its HOME position.

- **$ON_PATH** – Robot is on the path

    This output remains set as long as the robot stays on its programmed path. The output ON_PATH is set after the BCO run. This output remains set until the robot leaves the path, the program is reset or block selection is carried out. The ON_PATH signal has no tolerance window, however; as soon as the robot leaves the path the signal is reset.

Principle of Automatic External communication

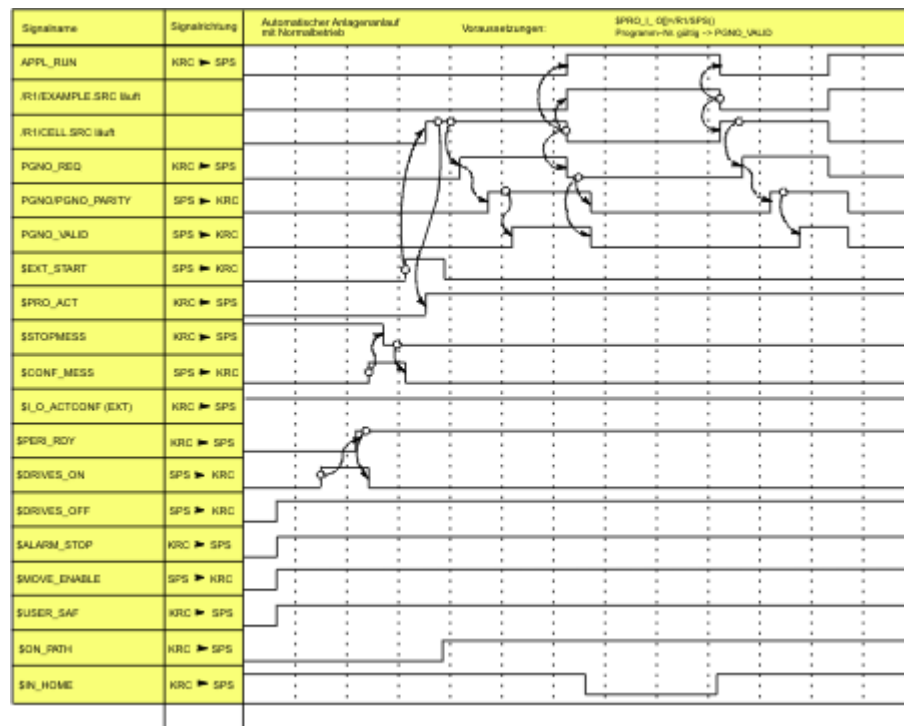Overview of the complete procedure

Fig. 8-3: Automatic system start and normal operation with program number acknowledgement by means of PGNO_VALID

Subdivision into subareas

1. Switch on drives
2. Acknowledge messages
3. Start Cell program
4. Transfer program number and execute application

For each of these areas, conditions have to be fulfilled and the possibility of reporting robot states to the PLC must be created.

Fig. 8-4: Handshake

It is advisable to use these predefined handsakes.
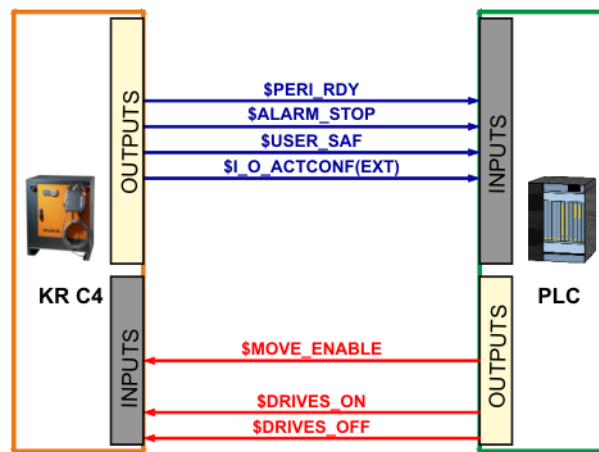
Switch on drives

Fig. 8-7

- Preconditions
    - $USER_SAF – Safety gate closed
    - $ALARM_STOP – No Emergency Stop active
    - $I_O_ACTCONF – Automatic External active
    - $MOVE_ENABLE – Motion enable present
    - $DRIVES_OFF – Drives off not activated
- Switch on drives

    $DRIVES_ON – Switch drives on for at least 20 ms
- Drives ready

    $PERI_RDY – The $DRIVES_ON signal is reset as soon as the check-back signal for the drives is received.

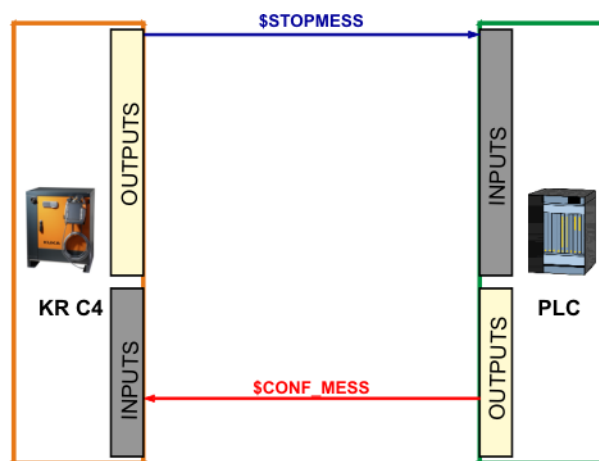**Acknowledge messages**

Fig. 8-11

- Preconditions

    $STOPMESS – Stop message is active
- Acknowledge message

    $CONF_MESS – Acknowledge message
- Acknowledgeable messages are cleared

    $STOPMESS – Stop message is no longer active; $CONF_MESS can now be reset.

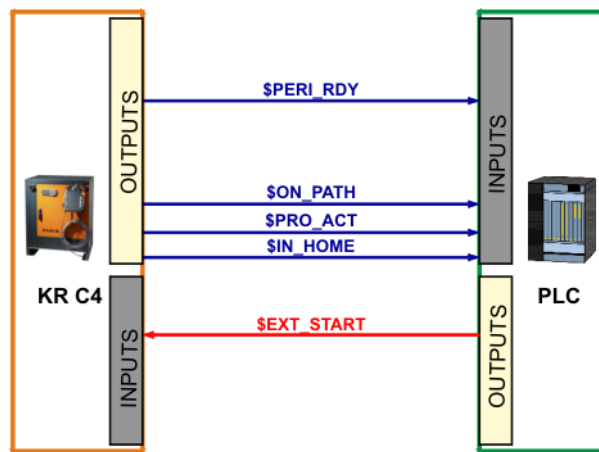**Start program (CELL.SRC) externally**

Fig. 8-16

- Preconditions
    - $PERI_RDY – Drives are ready
    - $IN_HOME – Robot is in HOME position
    - No $STOPMESS – No stop message is active
- External start
    $EXT_START – Activate external start (positive edge)
- CELL program running
    - $PRO_ACT – Signals that CELL program is running
    - $ON_PATH – The $EXT_START signal is reset as soon as the robot is on its programmed path.

Execute program number transfer and application program
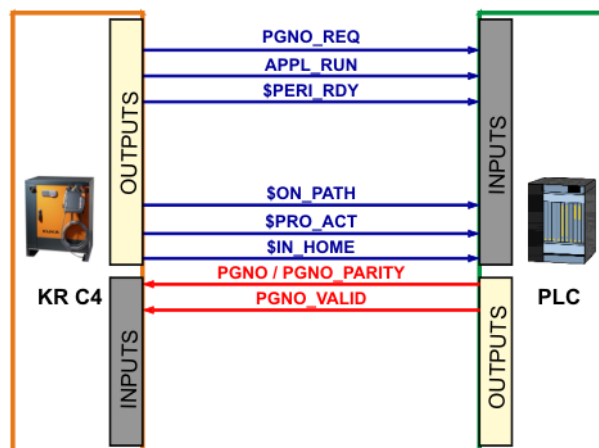
Fig. 8-23

- Preconditions
    - $PERI_RDY – Drives are ready
    - $PRO_ACT – CELL program is running
    - $ON_PATH – Robot is on the path
    - $IN_HOME – Robot is in HOME position, not required for restart
    - PGNO_REQ – Program number request is active
- Program number transfer and confirmation
    - Program number transfer
        (Correct data type (PGNO_TYPE), program number length (PGNO_LENGTH) and first bit for program nummer (PGNO_FBIT) are set)

■ PGNO_VALID – Set program number valid (confirmation) (positive edge)

■ Application program is running

■ APPL_RUN – Signals that application program is running

■ Robot leaves the HOME position; on termination of the application program the robot returns to the HOME position.

Procedure

1. In the main menu, select **Configuration** > **Inputs/outputs** > **Automatic External**.
2. In the **Value** column, select the cell to be edited and press **Edit**.
3. Enter the desired value and save it by pressing **OK**.
4. Repeat steps 2 and 3 for all values to be edited.
5. Close the window. The changes are saved.

	Term	Type	Name	Value
1	Type programno.	Var	PGNO_TYPE	1
2	programno. reflection	Var	REFLECT_PROG_NR	0
3	Bitwidth programno.	Var	PGNO_LENGTH	8
4	First bit programno.	I/O	PGNO_FBIT	33
5	Parity bit	I/O	PGNO_PARITY	41
6	Programno. valid	I/O	PGNO_VALID	42
7	Programstart	I/O	$EXT_START	1026
8	Move enable	I/O	$MOVE_ENABLE	1025
9	Error confirmation	I/O	$CONF_MESS	1026
10	Drives off (invers)	I/O	$DRIVES_OFF	1025
11	Drives on	I/O	$DRIVES_ON	140
12	Activate interface	I/O	$I_O_ACT	1025

Fig. 8-26: Configuring Automatic External inputs

Item	Description
1	Number
2	Long text name of the input/output
3	Type ■ **Green**: Input/output ■ **Yellow**: Variable or system variable ($...)
4	Name of the signal or variable
5	Input/output number or channel number
6	The outputs are thematically assigned to tabs.

Fig. 8-27: Configuring Automatic External outputs

8.2 Exercise: Automatic External

Aim of the exercise

On successful completion of this exercise, you will be able to carry out the following activities:

- Targeted integration of a robot program into Automatic External operation
- Adaptation of the "Cell" program
- Configuration of the Automatic External interface
- Recognition of sequence in Automatic External mode

Preconditions

The following are preconditions for successful completion of this exercise:

- Knowledge of how to edit the "Cell" program
- Knowledge of how to configure the Automatic External interface
- Theoretical knowledge of the signal sequence in Automatic External

Task description

1. Configure the Automatic External interface to the requirements of your control panel.
2. Expand your Cell program by any 3 modules, the functions of which you have verified beforehand.
3. Test your program in the modes T1, T2 and Automatic. Observe the relevant safety instructions.
4. Use buttons to simulate the functionality of the PLC.

What you should now know:

1. What is the precondition for PGNO_REQ not to be evaluated?

. . . .

. . . .

2. What signal is used to activate the drives and what must be taken into consideration?

. . . .

. . . .

3. Which Automatic External interface variable also influences jogging?

. . . .

. . . .

4. Which Fold in the CELL program checks the HOME position?

. . . .

. . . .

5. What are the preconditions for Automatic External mode?

. . . .

. . . .

9 Programming collision detection

9.1 Programming motions with collision detection

Description

Fig. 9-1: Collision

Monitoring of axis torques is implemented in robotics in order to detect whether the robot has collided with an object. This collision is undesirable in most cases and can result in destruction of the robot, tool or components.

Collision detection

- If the robot collides with an object, the robot controller increases the axis torques in order to overcome the resistance. This can result in damage to the robot, tool or other objects.
- Collision detection reduces the risk and severity of such damage. It monitors the axis torques.
- The user can define the procedure to be executed after a collision once the algorithm has detected a collision and the robot has stopped:
    - The robot stops with a STOP 1.
    - The robot controller calls the program tm_useraction. This is located in the Program folder and contains the HALT statement. Alternatively, the user can program other reactions in the program tm_useraction.
- The robot controller automatically calculates the tolerance range.
- A program must generally be executed 2 or 3 times before the robot controller has calculated a practicable tolerance range.
- The user can define an offset via the user interface for the tolerance range calculated by the robot controller.
- If the robot is not operated for a longer period (e.g. over the weekend), the motors, gear units, etc., cool down. Different axis torques are required in the first few runs after such a break than in the case of a robot that is already at operating temperature. The robot controller automatically adapts the collision detection to the changed temperature.

Limitations

- Collision detection is **not** possible in **T1 mode**.
- Collision detection is not possible for HOME positions and other global positions.
- Collision detection is not possible for external axes.
- Collision detection is not possible during backward motion.

■ High axis torques arise when the stationary robot starts to move. For this reason, the axis torques are not monitored in the starting phase (approx. 700 ms).

■ The collision detection function reacts much less sensitively for the first 2 or 3 program executions after the program override value has been modified. Thereafter, the robot controller has adapted the tolerance range to the new program override.

Principle of collision detection

Teaching a program with collision detection

■ Acceleration adaptation must be activated with the system variable $ADAP_ACC.

■ The system variable is located in the file C:\KRC\Roboter\KRC\R1\MaDa\$ROBCOR.DAT

■ $ADAP_ACC = #NONE Acceleration adaptation not activated

■ $ADAP_ACC = #STEP1 Dynamic model without kinetic energy

■ $ADAP_ACC = #STEP2 Dynamic model with kinetic energy

■ To activate collision detection for a motion, the parameter **Collision detection** must be set to TRUE during programming. This can be seen from the addition CD in the program code:

```
PTP P2  Vel= 100 % PDAT1 Tool[1] Base[1] CD
```

> The parameter **Collision detection** is only available if the motion is programmed via an inline form.

■ The tolerance range is only calculated for motion blocks that have been executed completely.

Setting the offset values

■ An offset for the torque and for the impact can be defined for the tolerance range.

■ **Torque:** The torque is effective if the robot meets a continuous resistance. Examples:

■ The robot collides with a wall and pushes against the wall.

■ The robot collides with a container. The robot pushes against the container and moves it.

■ **Impact:** The impact is effective if the robot meets a brief resistance. Example:

■ The robot collides with a panel which is sent flying by the impact.

■ The lower the offset, the more sensitive the reaction of the collision detection.

■ The higher the offset, the less sensitive the reaction of the collision detection.

> If the collision detection reacts too sensitively, do not immediately increase the offset. Instead, recalculate the tolerance range first and test whether the collision detection now reacts as desired.

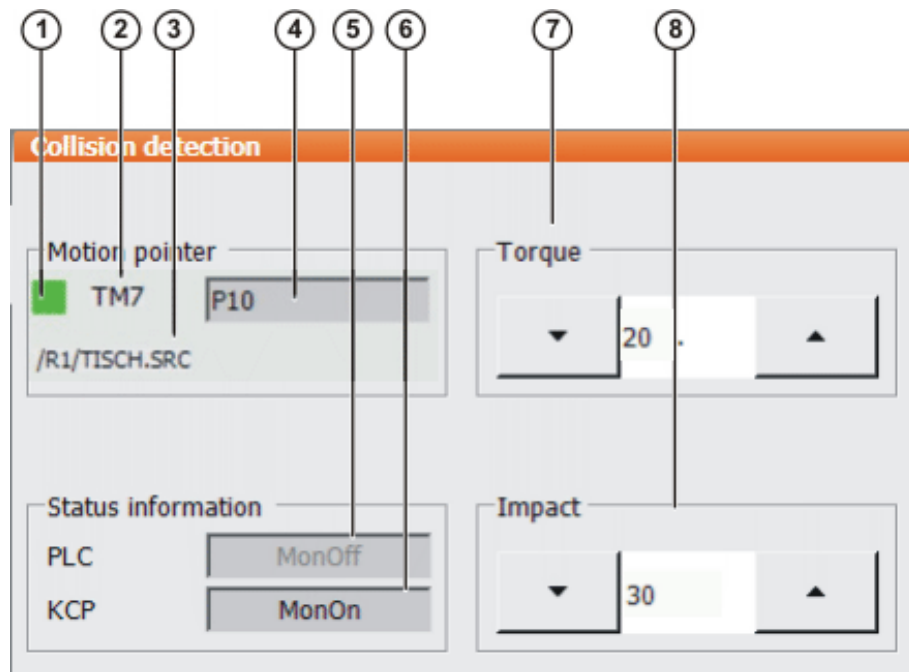■ Option window "Collision detection"

Fig. 9-2: Option window "Collision detection"

> ℹ️ The values in the option window **Collision detection** do not always refer to the current motion. Deviations are particularly possible in the case of points which are close together and approximated motions.

Item	Description
1	The button indicates the status of a motion. ■ Red: the current motion is not monitored. ■ Green: the current motion is monitored. ■ Orange: a button for setting the numeric value of the torque or impact on the left or right has been pressed. The window remains focused on the motion and the offset can be modified. The change can be applied by pressing **Save**. ■ Pixelated: a program must generally be executed 2 or 3 times before the robot controller has calculated a practicable tolerance range. While the robot controller is in the learning phase, the button remains pixelated.
2	Number of the TMx variable The robot controller creates a TMx variable for each motion block in which the parameter **Collision detection** is set to TRUE. TMx contains all the values for the tolerance range of this motion block. If 2 motion blocks refer to the same point Px, the robot controller creates 2 TMx variables.
3	Path and name of the selected program
4	Point name

Item	Description
5	This box is only active in "Automatic External" mode. It appears gray in all other modes.
	MonOn: collision detection has been activated by the PLC.
	If collision detection is activated by the PLC, the PLC sends the input signal **sTQM_SPSACTIVE** to the robot controller. The robot controller responds with the output signal **sTQM_SPSSTATUS**. The signals are defined in the file $config.dat.
	Note: Collision detection is only active in Automatic External mode if both the **PLC** box and the **KCP** box show the entry **MonOn**.
6	**MonOn**: collision detection has been activated from the KCP.
	Note: Collision detection is only active in Automatic External mode if both the **PLC** box and the **KCP** box show the entry **MonOn**.
7	Offset for the torque. The lower the offset, the more sensitive the reaction of the collision detection. Default value: 20.
	The window remains focused on the motion and the offset can be modified. The change can be applied by pressing **Save**.
	N.A.: the option **Collision detection** in the inline form is set to FALSE for this motion.
8	Offset for the impact. The lower the offset, the more sensitive the reaction of the collision detection. Default value: 30.
	The window remains focused on the motion and the offset can be modified. The change can be applied by pressing **Save**.
	N.A.: the option **Collision detection** in the inline form is set to FALSE for this motion.

Button	Description
Activate	Activates collision detection.
	This button is not displayed if the torque or impact has been changed, but the changes have not yet been saved.
Deactivate	Deactivates collision detection.
	This button is not displayed if the torque or impact has been changed, but the changes have not yet been saved.
Save	Saves changes to the torque and/or impact.
Cancel	Rejects changes to the torque and/or impact.

Procedure

> Alternatively, the lines with the torque monitoring in such programs can be deleted and collision detection can be used instead. Collision detection must not be used together with torque monitoring in a program.
> Acceleration adaptation is activated when system variable $ADAP_ACC **is not equal to** #NONE. (This is the default setting.) The system variable can be found in the file C:\KRC\Roboter\KRC\R1\MaDa\$ROBCOR.DAT.

Programming collision detection

1. Create a motion using an inline form.
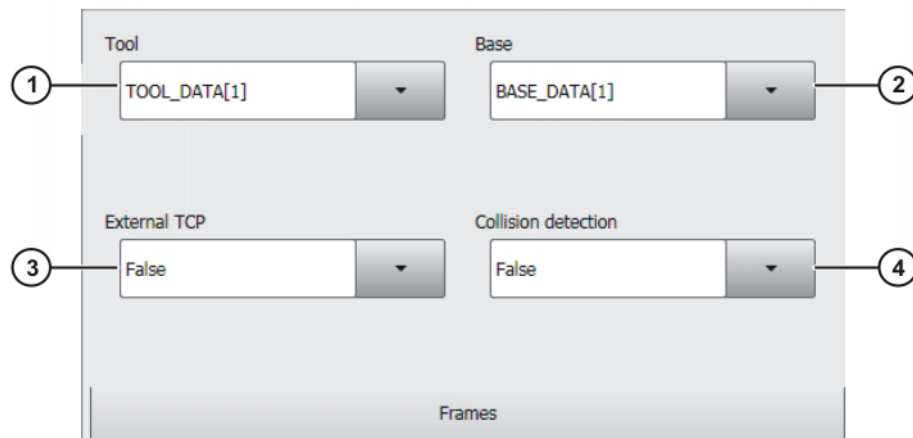2. Open the option window "Frames" and activate collision detection.

Fig. 9-3: Option window: Frames

Item	Description
1	Tool selection. If **True** in the box **External TCP**: workpiece selection. Range of values: [1] … [16]
2	Base selection. If **True** in the box **External TCP**: fixed tool selection. Range of values: [1] … [32]
3	Interpolation mode ■ **False**: The tool is mounted on the mounting flange. ■ **True**: The tool is a fixed tool.
4	■ **True**: For this motion, the robot controller calculates the axis torques. These are required for collision detection. ■ **False**: For this motion, the robot controller does not calculate the axis torques. Collision detection is thus not possible for this motion.

3. Complete the motion.

Calculating the tolerance range and activating collision detection

1. In the main menu, select **Configuration** > **Miscellaneous** > **Collision detection**.

    (>>> Fig. 9-2)

2. The box **KCP** must contain the entry **MonOff**. If this is not the case, press **Deactivate**.

3. Start the program and execute it several times. After 2 or 3 program executions, the robot controller has calculated a practicable tolerance range.

4. Press **Activate**. The box **KCP** in the **Collision detection** window now contains the entry **MonOn**.

    Save the configuration by pressing **Close**.

Adapting the offset for motions

1. Select program.

2. In the main menu, select **Configuration** > **Miscellaneous** > **Collision detection**.

3. The offset for a motion can be modified while a program is running: If the desired motion is displayed in the **Collision detection** window, press the buttons next to the torque or impact. The window remains focused on this motion. Change the offset using these buttons.
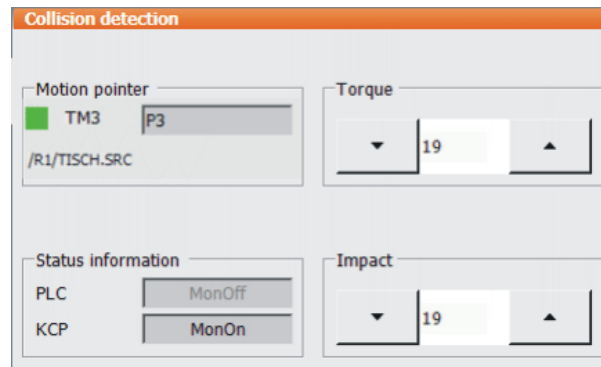
Fig. 9-6: Modified collision detection values

Alternatively, a block selection to the desired motion can be carried out.

4. Save the change by pressing **Save**.

5. Save the configuration by pressing **Close**.

6. Set the original operating mode and program run mode.

Index

Symbols
$ADAP_ACC 88

A
Acknowledgement message 29, 42
Analog I/Os, exercise 73
Analog inputs 69
Analog signals 69
Automatic External 75

C
Canceling a motion with interrupt, exercise 62
Collision detection 85, 89
Collision detection (menu item) 89
Collision detection, Automatic External 88
Collision detection, variable 87
Comment 5
Configuration 75
Configuring Automatic External, exercise 83

D
Data names 9
Declaring an interrupt, exercise 60
Dialog 47
Dialog message 29

E
EKrlMsgType 30

F
Fold 7

G
Global 51

I
Impact 86
Interpolation mode 89
Interrupt 51

K
KrlMsg_T 30
KrlMsgDlgSK_T 32
KrlMsgOpt_T 32

M
Message number 30
Message text 30
Message type 30
Messages 29

N
Notification message 29, 36

O
Originator 30
Outputs, analog 71

P
Priority 52
Program flowchart 9
Program flowchart example 11
Program flowchart symbols 10
Programming a dialog, exercise 50
Programming acknowledgement messages, exercise 43
Programming methodology, program flowchart example 11
Programming notification messages, exercise 37
Programming status messages, exercise 40
Programming wait messages, exercise 46

R
Return motion strategies 65
Return motion strategy, exercise 66

S
Status message 29, 39
Structured programming 5
Submit 13
Submit interpreter 13
Subprograms 8

T
tm_useraction 85
TMx 87
Torque 86

U
User messages 29

V
Voltage 72

W
Wait message 29, 45
Workspace monitoring 26
Workspace monitoring, exercise 26
Workspaces 17
Workspaces, mode 20
Wrist root point 20