Robot Control Library

V1.8.1

Table of Contents

Robot control Library

The **RobControl** is a cross-platform robotics control library that provides interpolated motion control for simple robotics applications.

The currently supported mechanics are: 3ax Cartesian robots, 4ax SCARA, 4ax Delta, 4ax Palletizer and 6ax articulated robots.

It is possible to control several robots at the same time, with each robot working independently from the others. Both real and virtual (simulated) robots can be controlled at the same time.

The RobControl library provides two main components:

- A robot structure data type called Robot_Type used for robots definition
- A function called RobotControl used for robots control

A typical use of the function looks as follows:

```
#include "RobControl.h"
#define ROBOT_NUM 2  //define the number of robots (2 in this example)
Robot_Type Robot[ROBOT_NUM]; // declare an array of robots control elements

/* call the function RobotControl cyclically */
status = RobotControl ((Robot_Type*)&Robot, ROBOT_NUM);
```

Note that the RobotControl function must be called every PLC cycle.

The input parameters to the function are a pointer to the robots structure array and the size of the array (i.e. the number of controlled robots).

The output is a status of the function, normally zero. For error numbers see the error codes section at the end of this document.

Robot structure

The RobControl library provides a Robot_Type structure data type. Each instance of the robot structure includes the following components:

Commands: these are all the actions called by the application software to control the robot. Note that all commands are internally automatically reset after being set by the application.

Parameters: this is the user interface where the robot parameters can be configured.

Monitor: offers a detailed view of the current status of the robot. These values cannot be modified by the application.

Note on the units:

- All rotary axes units are in degrees [°]: e.g. rotary joints, and A,B,C Euler angles of rotation for the path. Their speed unit is [°/s], acceleration is in [°/s^2] and jerk is in [°/s^3].
- All translational axes units are in millimeters [mm]: e.g. X,Y,Z position axes of the path, and workspace limits. Their speed unit is [mm/s], acceleration is in [mm /s^2] and jerk is in [mm /s^3].

Commands substructure

Command	Description	Parameters
RunProgram	Starts the execution of an NC program, given in text string format, from the selected line number until the END block or end of file.	Program; StartLine
RunBlocks	Starts the execution of a series of NC blocks inserted in a ring buffer as individual strings. The execution continues until an empty block is found or until the Stop command is called.	Blocks
Stop	Stops the current movement (block or program or jogging). A stopped movement cannot be continued, it must be restarted.	
Halt	Halts the current movement (block or program). Note that jogging movements cannot be halted.	
Continue	Continue a halted movement.	
JogAxis	Jog the selected axis along the selected direction. It can either be a path axis or a joint axis. Note that the movement is stopped when the JogAxis command is released.	Jog.Mode; AxisIndex; Direction; GotoPos
Reference	Sets the position of the joints and auxiliary axes to the values given in the parameters substructure. This command is normally used for homing purposes at start-up and it can only be executed when the robot is not moving (standstill state). *Note that changing the values of the joint axes causes a jump on the path axes as well! The application must make sure that the servo drives are switched off when calling this command; otherwise, a jump on the set positions is transferred directly to the motors!*	ActFromDrives[]
Reset	Resets all active errors.	

Parameters substructure (INPUT values)

Section	Parameter	Description
JointLimits[]	PositionPos	Maximum position value in the positive direction for each joint.
	PositionNeg	Maximum position value in the negative direction for each joint.
	VelocityPos	Maximum velocity value in the positive direction for each joint.
	VelocityNeg	Maximum velocity value in the negative direction for each joint.
	AccelerationPos	Maximum acceleration value in the positive direction for each joint.
	AccelerationNeg	Maximum acceleration value in the negative direction for each joint.
	JerkPos	Maximum jerk value in the positive direction for each joint (cannot be lower than acceleration). If a jerk value is not assigned by the user then it is automatically set to 10 times the acceleration value.

	JerkNeg	Maximum jerk value in the negative direction for each joint (cannot be lower than acceleration). If a jerk value is not assigned by the user then it is automatically set to 10 times the acceleration value.
AuxLimits[]	PositionPos	Maximum position value in the positive direction for each auxiliary axis.
	PositionNeg	Maximum position value in the negative direction for each auxiliary axis.
	VelocityPos	Maximum velocity value in the positive direction for each auxiliary axis.
	VelocityNeg	Maximum velocity value in the negative direction for each auxiliary axis.
	AccelerationPos	Maximum acceleration value in the positive direction for each auxiliary axis.
	AccelerationNeg	Maximum acceleration value in the negative direction for each auxiliary axis.
	JerkPos	Maximum jerk value in the positive direction for each auxiliary axis (cannot be lower than acceleration). If a jerk value is not assigned by the user then it is automatically set to 10 times the acceleration value.
	JerkNeg	Maximum jerk value in the negative direction for each auxiliary axis (cannot be lower than acceleration). If a jerk value is not assigned by the user then it is automatically set to 10 times the acceleration value.
PathLimits	Linear, Angular	Dynamic limits for linear [mm] and angular [deg] path axes
	Velocity	Maximum path velocity.
	Acceleration	Maximum path acceleration.
	Jerk	Maximum path jerk (cannot be lower than acceleration). If a jerk value is not assigned by the user then it is automatically set to 10 times the acceleration value.
Workspace[]	Type	Each zone defines a cuboid in space (in base coordinate system). 0 ... Disabled 1 ... Safe: robot must stay inside the zone 2 ... Forbidden: robot must stay outside the zone 3 ... Orientation: TCP's orientation cannot exceed safe cone
	PositionMin[3]	XYZ coordinates of first point defining cuboid
	PositionMax[3]	XYZ coordinates of second point defining cuboid
	Orientation[3]	ABC Euler angles of safe orientation cone
	MaxAngle	Maximum allowed deviation from safe orientation (in degrees)
Collision	LinkSize[]	Diameter of each link of the robot. Used for self-collision and inter-collision detections. The element following the last link is the tool. If a link has size zero it is not used for the calculations.
	Self	Activates self-collision detection, i.e. the collision between the TCP and the robot's own body.
	Inter	Activates inter-collision detection, i.e. the collision between this robot and any other robot for which inter-collision is also active.
UnitsRatio.Joints[] **UnitsRatio.Aux[]**	MotorUnits	Number of motor (encoder) steps per each revolution of the physical axis (e.g. 10000 pulses). Note that this value should also include the gearbox ratio, i.e. when one revolution of the physical axis is equivalent to several revolutions of the motor.
	AxisUnits	Number of metric units per each revolution of the physical axis (e.g. 360 degrees)
	Direction	Either +1 or -1 depending if the motor and the axis rotate in the same or opposite direction. Note: this value cannot be zero!
	HomeOffset	Value of the encoder reading (in motor units) when the axis is in its zero position.
Mechanics	Type	0... Cartesian (3ax) 1 ... SCARA (4ax) 2 ... not defined 3 ... Delta (4ax) 4 ... Palletizer (4ax) 5 ... RTCP (5ax) 6 ... Anthropomorphic (6ax) 10 ... User-defined
	Links.Offset	Distance of the next joint from the current joint along the base axes X,Y,Z.
	Links.Rotation	Rotation of the next joint from the current joint along the base axes X,Y,Z. Note that only Links[0] (zeroframe) rotations are currently supported
	Coupling[]	Mechanical coupling between joints. See individual transformations configuration parameters for details.
	UserTrf	Pointers to user-defined Direct and Inverse transformation functions
Calibration.Tool	Points[5]	5 measured poses of robot, with TCP in constant position and different orientations (see calibration section for details)
	Start	A positive edge starts the computation of the tool

	Status	Returns ERR_CALIBRATION if calculations are not successful
	Result	X,Y,Z estimated size of the tool
Calibration.Frame	Points[3]	3 measured points along X and Y axes (see calibration section for details)
	Start	A positive edge starts the computation of the frame
	Status	Returns ERR_CALIBRATION if calculations are not successful
	Result	X,Y,Z,A,B,C estimated frame coordinates with respect to base frame
Override		Path speed override from 0 to 200%.
Program		Pointer to a text string containing the complete NC program.
Blocks[]		A ring buffer of individual NC blocks given as strings. Note that the blocks are deleted as soon as they are processed by the motion planner and new blocks can be inserted by the application to continue execution.
ActFromDrives .Joints[] .Aux[]		Actual positions of joints and auxiliary axes in motor units. These values should be assigned cyclically from the servo drives but they are only used by the command *Reference* when homing the axes.
StartLine		Line number from which the program execution will start.
StopLine		Line number after which the program execution will stop. Disabled if zero.
Points[]		Array of target points for all blocks and programs movements. Note that the Points[0] element is always internally overwritten with zero values.
	Axes[]	Target position for the joint axes (max 6 axes)
	Mode	Joint target point defined in path axes (0) or joint axes (1)
	Aux[]	Target position for the auxiliary axes (max 6 axes)
	ModeAux[]	Aux target point defined with absolute (0) or incremental (1) positions
Tool[]		Array of tools to be used by the robot. It represents the size and orientation of the tool from the mounting point (MP) to the tool center point (TCP) along the path axes. Note that the Tool[0] element is always internally overwritten with zero values, so that omitting the tool in the programmed block will be equivalent to working with no tool.
Frame[]		Array of local work-piece frames to be used by the robot. It represents the location and orientation of the local frame with respect to the base frame of the robot. It can be specified along the path axes. Note that the Frame[0] element is always internally overwritten with zero values, so that omitting the frame in the programmed block will be equivalent to working in the robot's base frame.
Jog	Mode	Jogging mode for the JogAxis command. The movement can be along the Joint frame (0), TCP Base frame (1), TCP Tool frame (2) or Auxiliary axes (3).
	AxisIndex	Specifies which axis is going to move. For example: choose Mode=1 and AxisIndex=0 to jog along the X axis.
	Direction	Positive (0) or Negative (1) for jogging. GoTo (2) for absolute positioning.
	GotoPos	Target position for absolute movement when Direction = GoTo is selected.
Conveyor[]	Angle	Orientation angle [in degrees] of each conveyor with respect to the robot's base frame. E.g. 0 deg is along +X axis, 90 deg is along +Y axis, 180deg is along –X axis. A maximum of two conveyors per robot can be configured.
	Position	Current position of conveyor offset [in mm]. It is automatically reset to zero when a new tracking section starts and must be then cyclically updated by the application according to the conveyor's actual movement.
PathCorrections[]		External offset corrections provided by the application program and applied to the path axes. Note that these values are not included in the path-planning phase and care should be taken to avoid uncontrolled axes movements. Only small and smooth changing values should be applied!
M_synch[]		Specifies which M-function is synchronous (program execution halts).
CycleTime		Specifies the cycle time in which this robot is operated [in seconds]. *It must be a value larger than zero otherwise the RobotControl function will return an error!*
FilterTime		Specifies the filter time for the output position values to the servo drives (Monitor.SetToDrive[]).Note that only the joint output values are filtered, not the internal planned values. The minimum filter time value is zero (no filtering) and the maximum is 0.5 seconds. A larger value will smooth out the movement dynamics but will also cause larger deviations from the planned path.
MaxTransitionAngle		Specifies the maximum angle between two consecutive movements which is still considered tangential, i.e. does not cause a path speed reduction. Transitions larger than this angle will cause the path speed to stop. The minimum angle value is zero (stop at all transitions) and the maximum angle is 180 degrees (never stop between transitions). A larger value will cause the movement to be faster but it will also cause large accelerations at the joint

	axes. A higher filter time might be required in that case.
SingleStep	If this parameter is enabled the program will be executed in single-step mode, that is the execution will halt at the end of each block and wait for the Continue command. Note that the single step mode can be activated only from standstill state, not while a program is already running.
Simulation	If this parameter is enabled the robot will be execute its movements in simulation mode. The *SetToDrive* values will not be updated and the physical axes will not move. This mode is useful for testing purposed, e.g. to detect workspace violations or singularities along the path without physically executing it. Once the simulation mode is switched off the virtual axes in the monitor structure are automatically re-aligned with the actual joint (*SetToDrive*) values. The simulation mode can be toggled only from standstill state, not while a program is already running.
SingularityWarning	It enables the detection of singularity points along the programmed path. The error ERR_SINGULARITY will be activated depending on the chosen mode: 0 … no singularity detection 1 … singularity detection always active when running programs 2 … singularity detection only active when running programs in simulation mode (dry run)

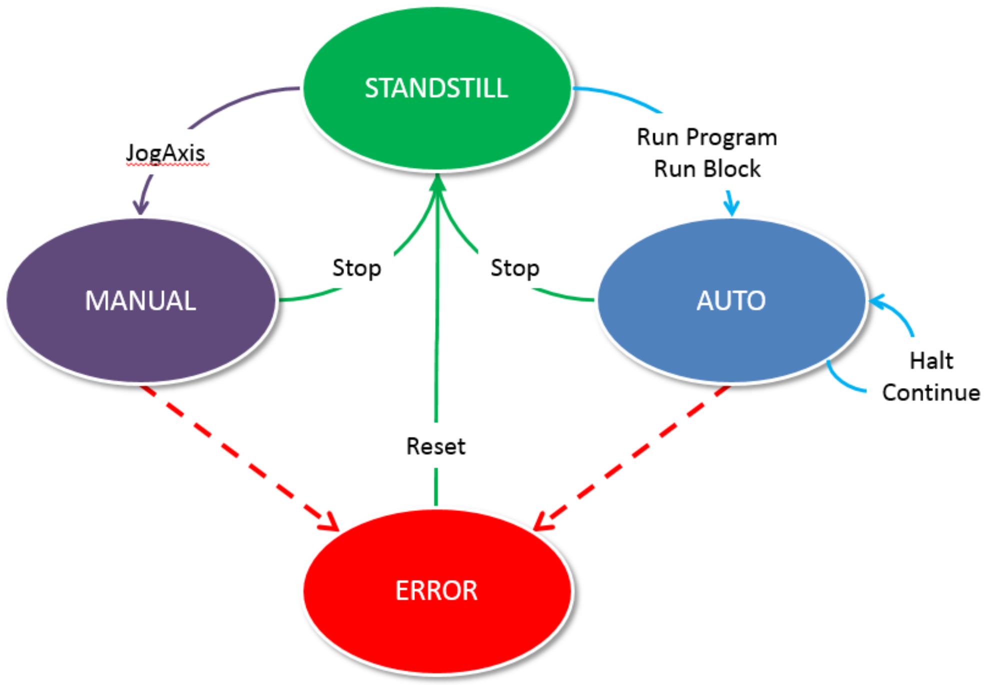
Monitor substructure (OUTPUT values)

Variable	Description
Handle	Memory address of the vision structure instance.
AxesNum	Number of active axes for this robot.
Moving	It is set to true if any movement execution is active, otherwise is false.
Halted	It is set to true if a previously active movement has been halted. The movement can now either be continued or stopped.
PathSpeed	The current speed of the TCP.
PathPosition[]	The current position of the TCP, i.e. the path axes positions.
JointPosition[]	The current position of the joint axes.
JointSpeed[]	The current speed of the joint axes.
AuxPosition[]	The current position of the auxiliary axes.
SetToDrive **.Joints[]** **.Aux[]**	The current position of the joint and auxiliary axes (in motor units) ready to be transferred to the servo drives. Include compensations for gear ratio, encoder mounting direction and home offsets as defined in the UnitsRatio parameters. NOTE: these values are not updated in simulation mode!
MountBasePosition[]	The current position of the mount point, i.e. the TCP point when no tool is mounted, as seen from the base frame. This is equal to the PathPosition when no tool is mounted and no frame is active.
ToolBasePosition[]	The current TCP position as seen from the base frame. This is equal to the PathPosition when no additional frame is active.
Wireframe[]	The current position of each node of the wireframe model, from base to TCP.
LineNumber	Currently executed line number (integer value) in active program.
CurrentBlock	Currently executed block (string) in active program.
BlockLength	Total path length of the current block.
CompletedBlockLength	Already completed length of the current block. Equal to BlockLength at the end of the block. It shows the elapsed time while executing a WAIT command.
TargetPoint	Target point of the current movement. Might be useful for the application to e.g. show the point's coordinates on a monitor page.
Tool	Current active tool. Given as integer index from the Tool array in the Parameters structure. The tool index can also be manually modified here while the robot is in standstill state. The actual PathPosition will be immediately adjusted accordingly.
Frame	Current active frame. Given as integer index from the Frame array in the Parameters structure.
M[]	Active M values set by the current block. An M code is a signal sent from the NC program to the PLC logic, e.g. to set a physical output at the end of a movement. *All M codes must be reset here by the application.* M codes can either be synchronous or not, according to the parameter M_synch[]. Synchronous M codes force the execution of the NC program to stop and wait for the M code to be

	reset by the application. That is the case, for example, of a spindle axis that needs to reach a certain speed before starting a movement. As soon as the M code is reset the NC program execution will resume. On the other hand, a non-synchronous M code does not halt the NC program. It can be used, for example, to switch on/off an HMI status signal.
R_[]	R_ parameters are of Boolean type and can be set by the application. They are evaluated by the IF statement in the NC program.
DI_[]	Status of digital inputs set by the application. They are evaluated by the WAIT DI command and by the IF statement in the NC program.
DO_[]	Status of the digital outputs set and reset in the NC program by the SET DO and RESET DO commands respectively.
TrackActive	It shows if the robot is currently following Conveyor 1 [TrackActive=1] or Conveyor 2 [Track-Active=1] or none of them [TrackActive=0].
TrackSynch	This flag is set when the robot starts and new tracking section with a conveyor. At the raise of this flag the application first needs to update the target positions of the upcoming move-ments (e.g. position of product to be picked on conveyor) and then must reset the TrackSynch flag to allow the tracking movements to start. See example section for more details.
TangActive	It shows if the automatic tangential mode is active.
TangOffset	Angular offset between true path tangent and desired orientation of tangential axis.
ActiveError	It shows the current error number if the robot is in error state. It can be reset by the Reset command.
ErrorLine	It shows the program line number where the error was triggered.
State	Current state of the robot: 0 – STANDSTILL 10 – MANUAL (manual movement – jogging) 20 – AUTO (automatic movement – run program/blocks) 255 – ERROR

State Diagram

Every robot instance is in a well-defined state at all times. The following drawing shows the existing states and some of the actions required to move among them.



Standstill (0): this is the default initial state. All movement states are accessed from here.

Manual (10): Manual jogging movement is executed on single axes. This kind of movements can be stopped but not halted.

Auto (20): Automatic movement is currently running, either as single NC program or as sequence of individual blocks. The execution can be stopped or halted and then continued.

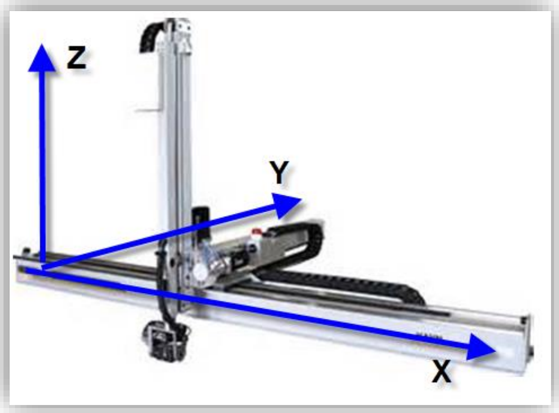
Error (255): the robot is in error state. A reset command can bring it back in standstill state.

Mechanics description

According to the specified mechanical type, the link parameters assume different meanings.

Cartesian Robot (type 0)

No link parameters are needed for Cartesian Robots. The transformations between joints and path axes are linear.

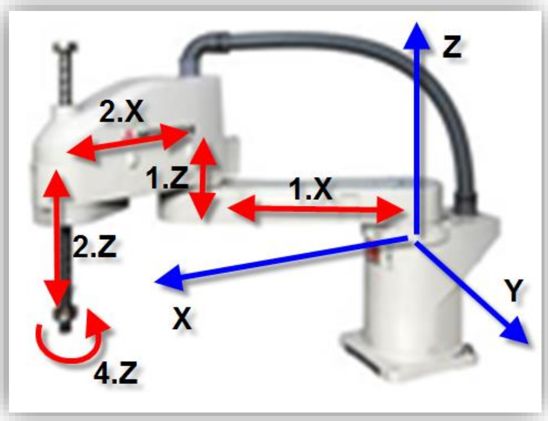


SCARA Robot (type 1)

Distance between first and second joint:
Links[1].Offset.X
Links[1].Offset.Z

Distance between second joint and TCP:
Links[2].Offset.X
Links[2].Offset.Z

Vertical displacement of TCP when rotating fourth joint:
Links[4].Offset.Z [rev^{-1}]



Delta Robot (type 3)

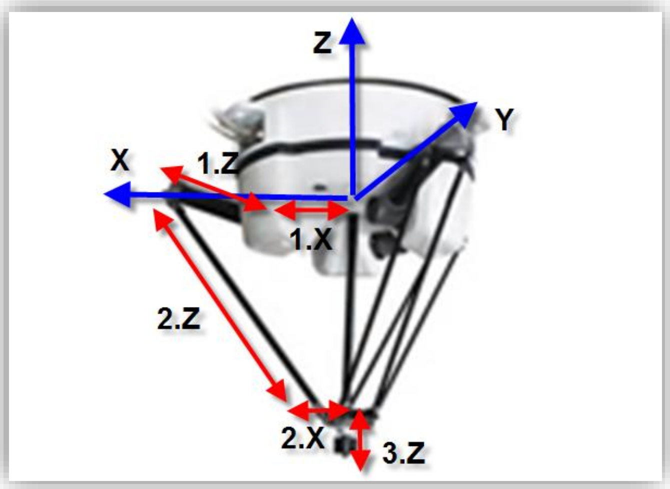
Radius of top platform:
Links[1].Offset.X

Length of upper arm:
Links[1].Offset.Z

Radius of bottom platform:
Links[2].Offset.X

Length of lower arm:
Links[2].Offset.Z

Tool holder length:
Links[3].Offset.Z



The first joint is aligned with the X axis, the second is at 120deg, the third at 240deg.

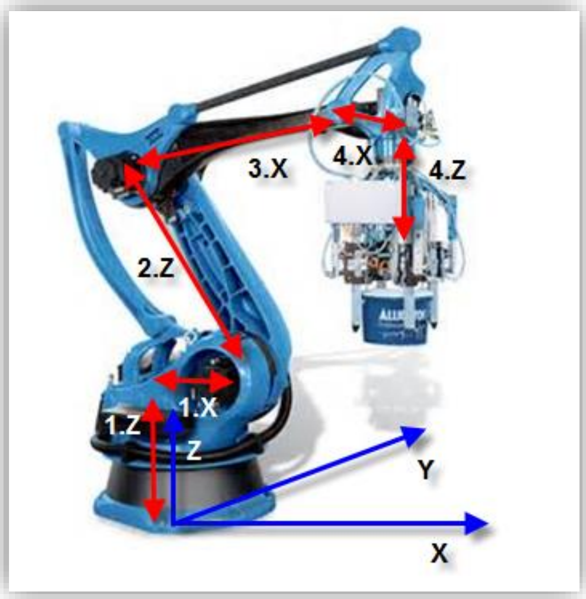
Palletizer Robot (type 4)

Distance between first and second joint:
Links[1].Offset.X
Links[1].Offset.Z

Distance between second and third joint:
Links[2].Offset.Z

Distance between third and passive joint:
Links[3].Offset.X

Distance between passive joint and TCP:
Links[4].Offset.X
Links[4].Offset.Z



A mechanical coupling between the second and third joint can be configured with the parameter

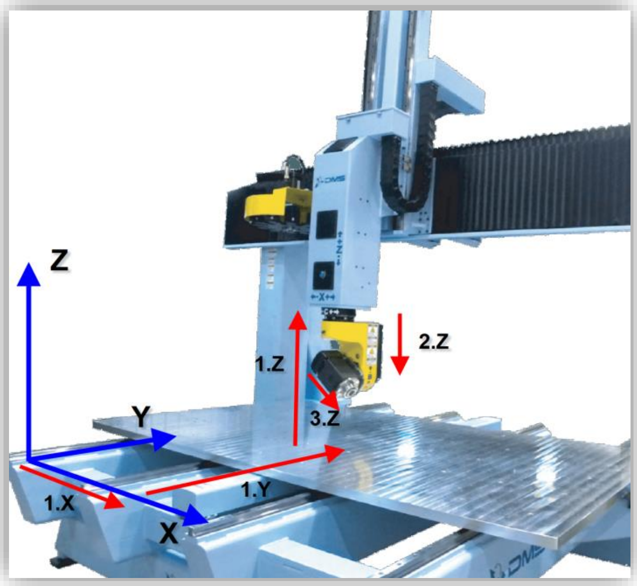
Coupling[1] = J3/J2

5ax RTCP (type 5)

Distance between base and first rotating joint:
Links[1].Offset.X
Links[1].Offset.Y
Links[1].Offset.Z

Distance between first and second rotating joint:
Links[2].Offset.X
Links[2].Offset.Y
Links[2].Offset.Z

Distance between second rotating joint and TCP:
Links[3].Offset.X
Links[3].Offset.Y
Links[3].Offset.Z



There are six different combinations of RTCP wrists (e.g. AB, BC, CA...) and the user needs to specify the rotation axes of the two rotating joints. Following the convention that ABC are the rotation angles around the XYZ axes, the two parameters can be used to describe the RTCP configuration:

Links[2].Rotation.X/Y/Z
Links[3].Rotation.X/Y/Z

For example, the CB wrist in the image would be configured as:

Links[2].Rotation.Z = 1 // C rotation around Z. Set X and Y to 0
Links[3].Rotation.Y = 1 // B rotation around Y. Set X and Z to 0

Anthropomorphic Robot (type 6)

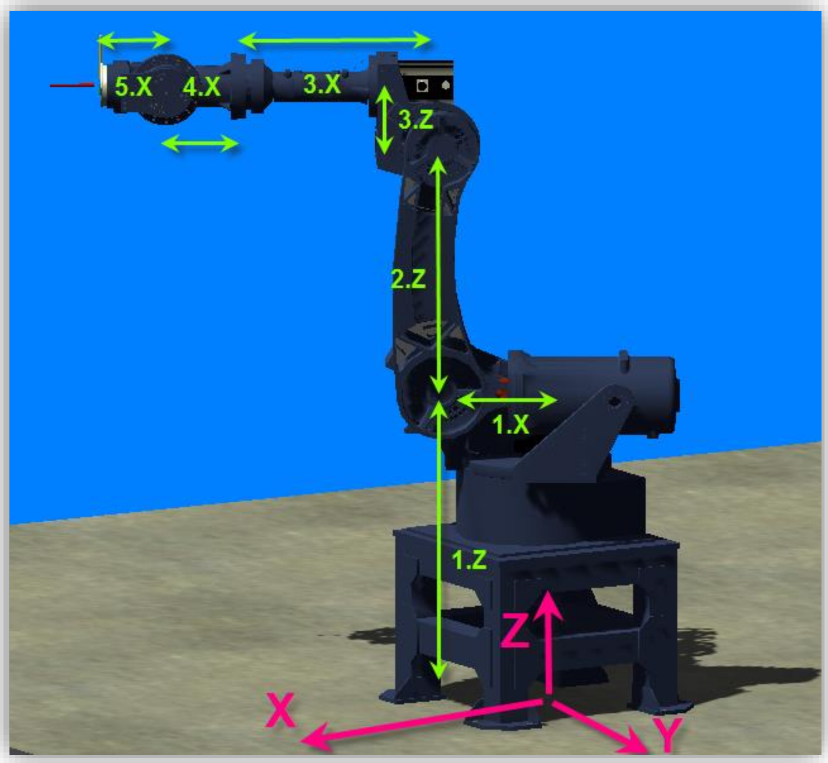
Distance between first and second joint:
Links[1].Offset.X
Links[1].Offset.Z

Distance between second and third joint:
Links[2].Offset.Z

Distance between third and fourth joint:
Links[3].Offset.X
Links[3].Offset.Z

Distance between fourth and fifth joint:
Links[4].Offset.X

Distance between fifth joint and TCP:
Links[5].Offset.X



A mechanical coupling between the joints 4,5,6 can be configured with the parameters

Coupling[3] = J5/J4

Coupling[4] = J6/J4

Coupling[5] = J6/J5

Universal Robot (type 8)

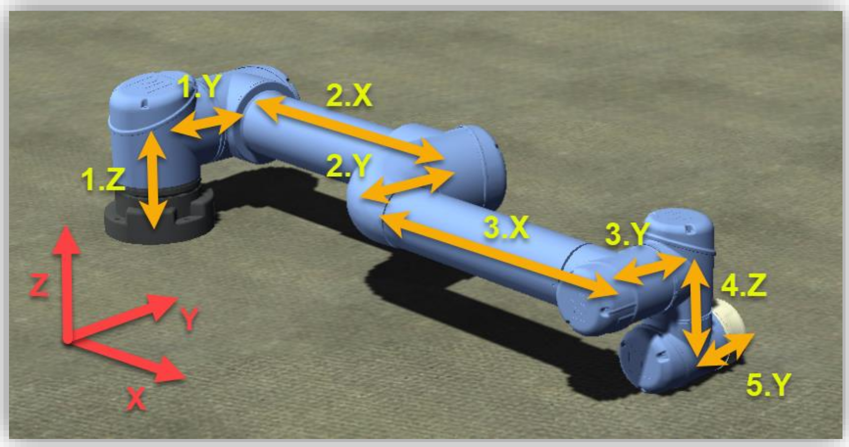
Distance between first and second joint:
Links[1].Offset.Y
Links[1].Offset.Z

Distance between second and third joint:
Links[2].Offset.X
Links[2].Offset.Y

Distance between third and fourth joint:
Links[3].Offset.X
Links[3].Offset.Y

Distance between fourth and fifth joint:
Links[4].Offset.Z

Distance between fifth joint and TCP:
Links[5].Offset.Y



NOTE: all distances must be defined as positive values.

User-defined Robot (type 10)

A customized mechanical system can also be configured and controlled. The number of robot's axes must be defined in the variable UserTrf and can be up to a maximum of 6 joints:

UserTrf.AxesNum

The mechanical body parameters are defined with the Links variable to describe the distance and orientation of each joint to the next (i=1..5):
Links[i].Offset.X
Links[i].Offset.Y
Links[i].Offset.Z
Links[i].Rotation.X
Links[i].Rotation.Y
Links[i].Rotation.Z

The addresses of the user-defined direct and inverse transformations must also be passed to the UserTrf variable:

```
UserTrf.Direct = &myDirectFunc;
UserTrf.Inverse = &myInverseFunc;
```

The prototypes for the user-defined functions are:

```
unsigned short myDirectFunc (
        Link_Type [6], //mechanical description of robot (INPUT)
        float[6], // target position of joints  (INPUT)
        float[6], // current position of TCP (INPUT)
        float[6]) //target position of TCP (OUTPUT)

unsigned short myInverseFunc (
        Link_Type[6], //mechanical description of robot (INPUT)
        float[6], // target position of TCP (INPUT)
        float[6], //current position of joints (INPUT)
        float[6]) //target position of joints (OUTPUT)
```

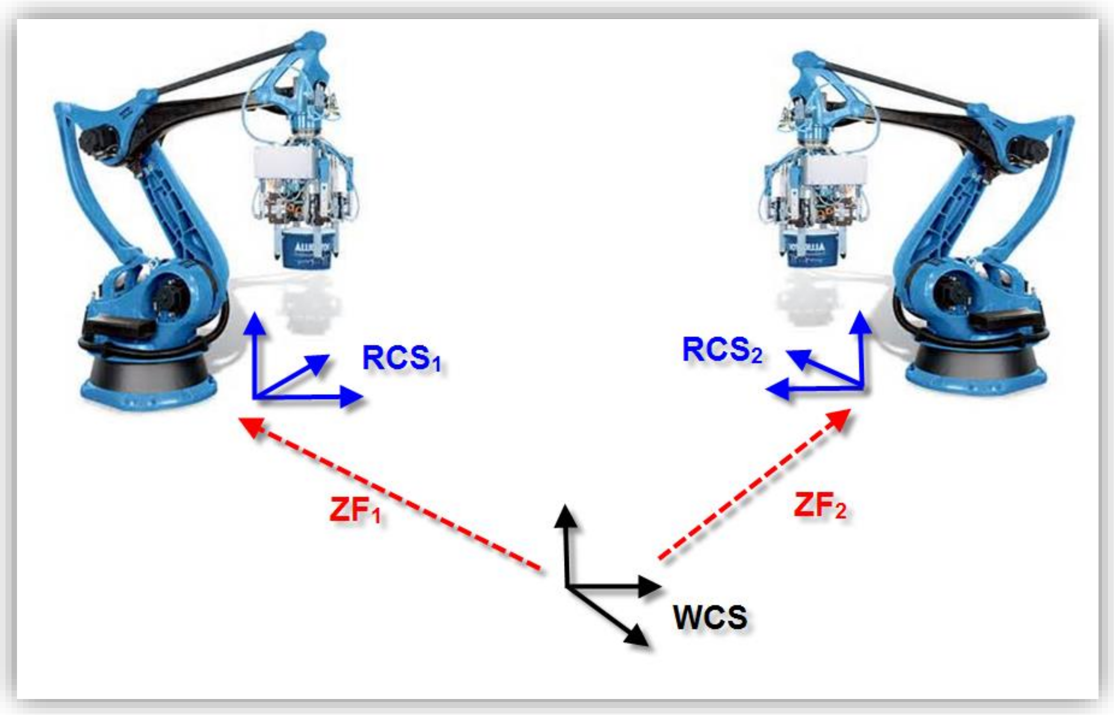
If the user-defined functions return anything different than 0 (STATUS_OK), then the movement will abort with an error.

ZeroFrame

The ZeroFrame (ZF) is used to locate the robot in the world coordinate system (WCS). It is especially useful when more robots share the same working area.

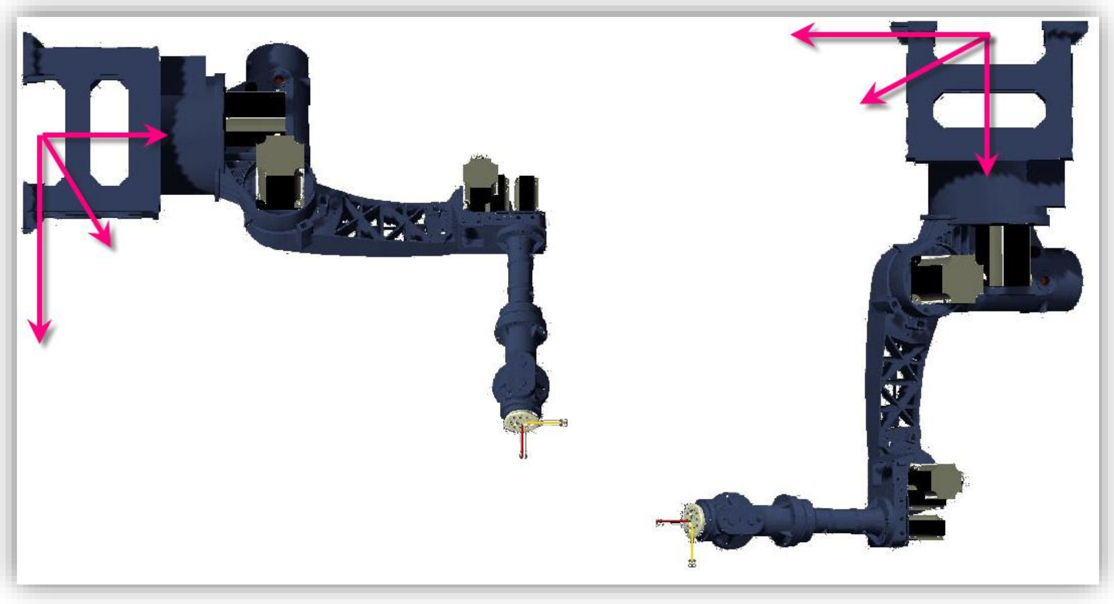
Distance between robot's base and world origin:
Links[0].Offset.X
Links[0].Offset.Y
Links[0].Offset.Z

Rotation between robot's base and world origin around the Z vertical axis:
Links[0].Rotation.Z



Anthropomorphic 6axes robots can have their base frame rotated along all degrees of freedom, e.g. for wall mounting or upside-down configurations:

Links[0].Rotation.X
Links[0].Rotation.Y
Links[0].Rotation.Z



Workspace monitoring

A monitoring for the workspace of the robot is available and is activated automatically as soon as at least one zone is defined in the parameters workspace substructure.

NOTE: workspace monitoring only works when the robot runs in automatic mode. It is not active during manual jogging operations.

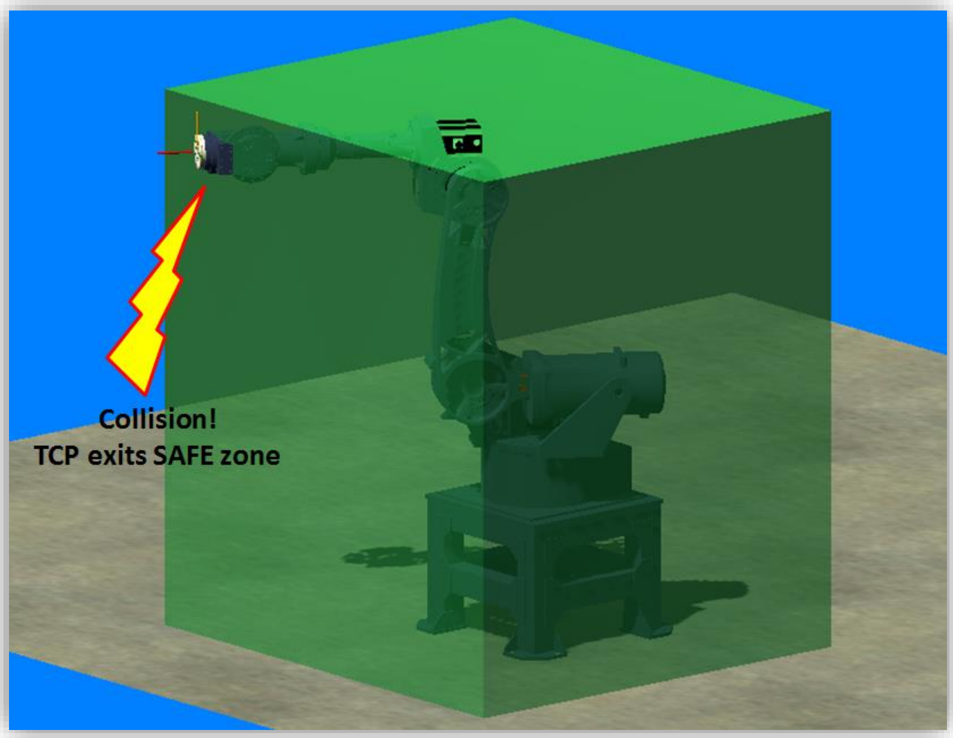
A maximum of 10 zones can be defined, each with its own type and position.

The first kind of zones are essentially cuboids in the robot's base space and can be either configured as ZONE_SAFE (type 1) or as ZONE_FORBIDDEN (type 2).

The second kind of zones are cones and can be used to restrict the orientation of the TCP. They are configured as ZONE_ORIENTATION (type 3).

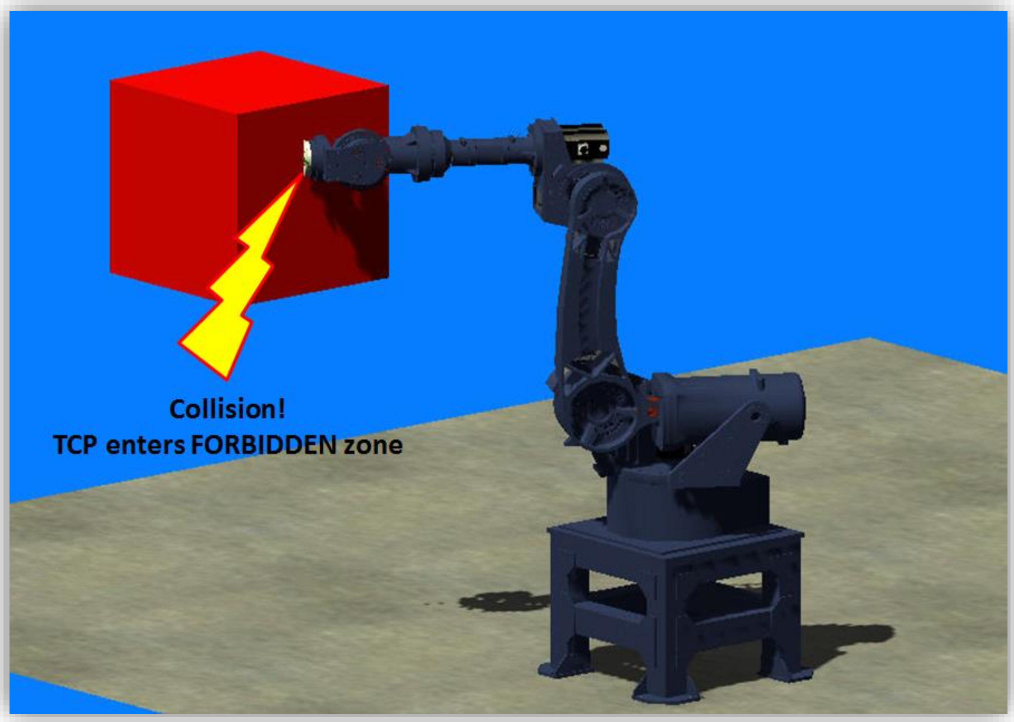
A **safe** zone requires the entire robot to always stay *inside* its boundaries.

```
Robot.Parameters.Workspace[0].Type = ZONE_SAFE;
Robot.Parameters.Workspace[0].PositionMin[0] = -1000; //X min
Robot.Parameters.Workspace[0].PositionMin[1] = -1000; //Y min
Robot.Parameters.Workspace[0].PositionMin[2] = 0; //Z min
Robot.Parameters.Workspace[0].PositionMax[0] = 1000; //X max
Robot.Parameters.Workspace[0].PositionMax[1] = 1000; //Y max
Robot.Parameters.Workspace[0].PositionMax[2] = 2500; //Z max
```



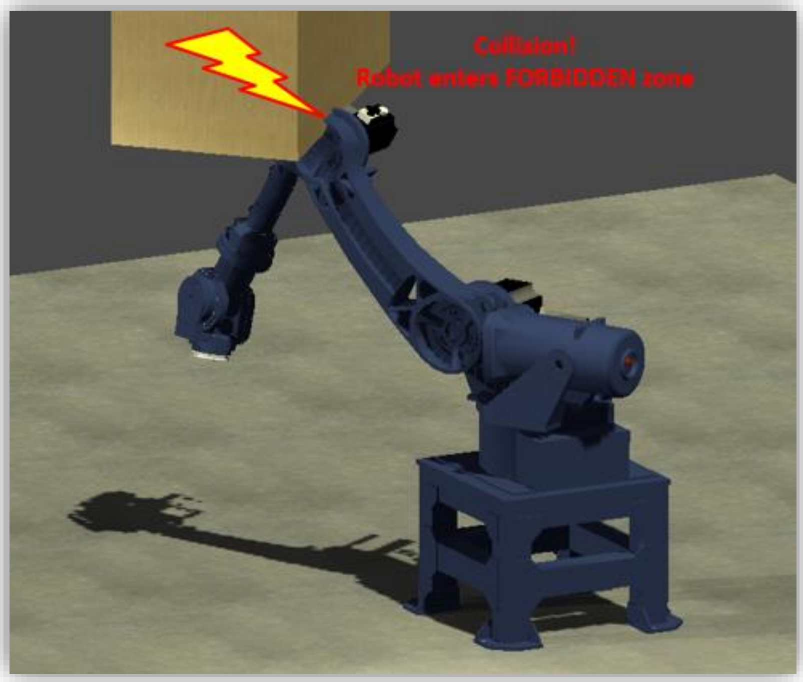
A **forbidden** zone requires the entire robot to always stay *outside* its boundaries.

```
Robot.Parameters.Workspace[1].Type = ZONE_FORBIDDEN;
Robot.Parameters.Workspace[1].PositionMin[0] = 500; //X min
Robot.Parameters.Workspace[1].PositionMin[1] = -200; //Y min
Robot.Parameters.Workspace[1].PositionMin[2] = 500; //Z min
Robot.Parameters.Workspace[1].PositionMax[0] = 700; //X max
Robot.Parameters.Workspace[1].PositionMax[1] = -400; //Y max
Robot.Parameters.Workspace[1].PositionMax[2] = 700; //Z max
```



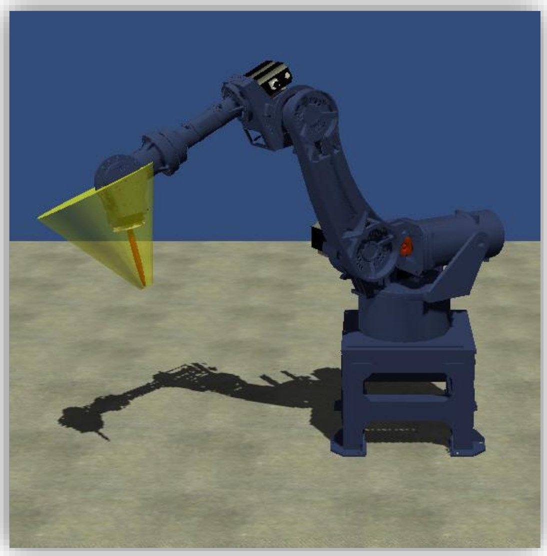
Workspace monitoring is active along the whole TCP path, both at planning time and at execution time.

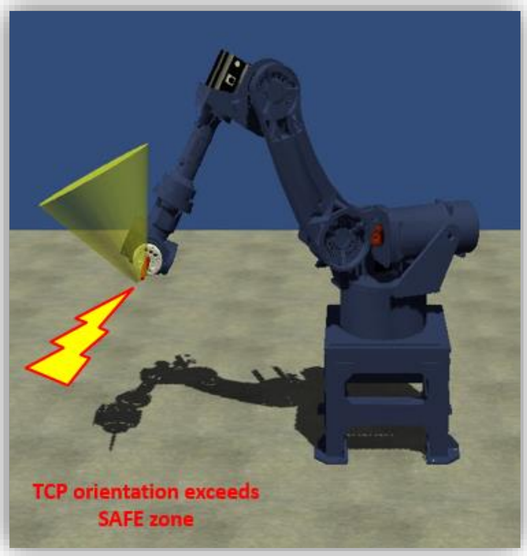
The entire kinematical body of the robot is monitored, not only the front-end-effector:



An **orientation** zone requires the TCP orientation to always stay *inside* its boundaries.

```
Robot.Parameters.Workspace[2].Type = ZONE_ORIENTATION;
Robot.Parameters.Workspace[2].Orientation[0] = 30; //A
Robot.Parameters.Workspace[2].Orientation[1] = 90; //B
Robot.Parameters.Workspace[2].Orientation[2] = 0;  //C
Robot.Parameters.Workspace[2].MaxAngle = 30; //in degrees
```





NOTE: workspace monitoring is computationally expensive, especially for circular and spline movements. It is advised to keep the number of monitored zones as low as possible.

Collision detection

Two kinds of collisions are possible and can be monitored:

- Self-collision: the TCP of the robot collides with its own body
- Inter-collision: the body of the robot collides with the body of other robots

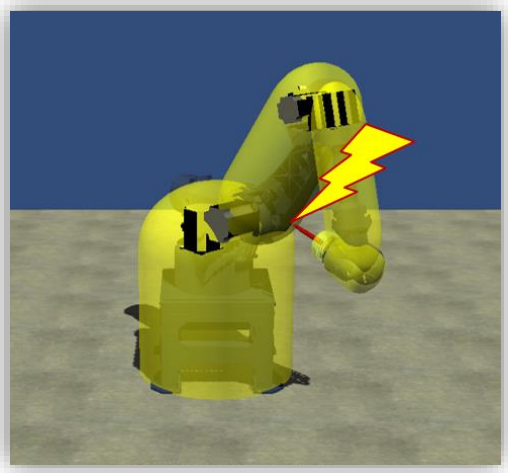
In other to detect any kind of collision, the system needs to know the size (**diameter**) of each link of the robot. These values can be configured in the parameters substructure:

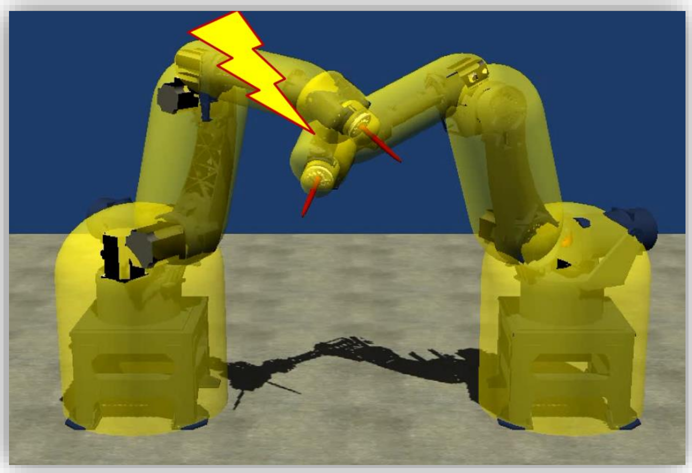
```
Robot.Parameters.Collision.LinkSize[0] = 300; //diameter of Joint 1
Robot.Parameters.Collision.LinkSize[1] = 150; //diameter of Joint 2
Robot.Parameters.Collision.LinkSize[2] = 10;  //diameter of Joint 3
Robot.Parameters.Collision.LinkSize[3] = 0;   //diameter of Joint 4
Robot.Parameters.Collision.LinkSize[4] = 100; //diameter of Joint 5
Robot.Parameters.Collision.LinkSize[5] = 0;   //diameter of Joint 6
Robot.Parameters.Collision.LinkSize[6] = 20;  //diameter of Tool
```

Note that the element following the last joint is used for the tool size (optional). Any element set to zero will be excluded from the calculations.

Activating the monitoring is done by setting the following flags:

```
Robot.Parameters.Collision.Self = 1; //activate self-collision detection
Robot.Parameters.Collision.Inter = 1; //activate inter-collision detection
```





NOTE: collision detection only works when the robot runs in automatic mode. It is not active during manual jogging operations.

NOTE: Inter-collision only works against any other robot for which the detection option has also been activated.

When working with more than one robot the coordinate systems are particularly important. The *zero frame* of each robot must be used to localize the robot with respect to the global coordinate system (see section on mechanics description for details).

NC Program Syntax

The user NC program must be written in robotic language, which includes movements and synchronization commands.

The program must be provided by the application as a string. No file handling is performed by the Robot Control library.

The Parameters.Program variable is a character pointer (char*) and the size of the pointed string is arbitrary.

The following table lists the currently supported commands (required parameters in **bold**, others are optional):

Command	Description	Parameters
ML	Linear interpolation of the path axes	**P**, F, Z, T, R
MJ	Linear interpolation of the joint axes	**P**, F, Z, T, R, Jx
MC	Circular interpolation of the path axes	**P**, **Q**, F, Z, T, H
MS	Spline interpolation of the path axes	**P,** F, Z, T
HOME	Move joints to their home position	F
WAIT	Delay time	[sec]
WAIT DI	Wait for a digital input signal to be 1	[index]
SET DO	Set a digital output signal to 1	[index]
RESET DO	Reset a digital output signal to 0	[index]
M	M-function to communicate with PLC	Max of 10 in a single line

IF
ELSE
ENDIF
	Conditional statement	(!)R[index] or (!)DI[index]

GOTO	Jump to specified label	Loop count
SUB	Execute local subroutine at specified label until next END	Loop count
TRK	Start tracking conveyor	[index = 0,1,2]
LABEL:	Label can be any string followed by "**:**"	
TANG	(De)activate tangential axis C	[0,1], H
T	Modify active tool (without movement)	[index]
END	Abort main program execution or complete subroutine.	
//	comment	

Parameter	Description	Notes
P	Target position	
F (FC/FA)	Path speed (Cartesian/Angular)	[mm/s] (*see additional notes*)
Z	Frame index	0 is always base frame
T	Tool index	0 is always no tool
Q	A point on the circle along the path	Must not be in line with P
H	Rotation angle	Must be a non-negative value
R	Round edge radius	Must be a non-negative value
Jx=	Optional reference value for joint axes	Only used for MJ with target point specified in path axes

Additional notes (*see examples section for more details*):

Not more than one command can be specified on the same line. The only exception is the M command.

M commands: Up to 10 M commands can be called on the same line. M commands can be **synchronous** (program execution halts until reset by the application) or **asynchronous** (program execution does not stop – no speed dip). The default setting for all M commands is asynchronous. Note that only a positive non-zero index is accepted for M commands, i.e. M0 is not valid.

Note that synchronous M-functions also block the path planning process (see examples section).

F (feedrate): The path speed must be defined with the parameter F (feedrate) at least for the first movement in the program. This setting is **modal** and will not be modified until another F parameter is programmed. Movements can also be programmed with Cartesian-only or angular-only speed configurations using the FC/FA keywords (also modal). The speed units of each definition depend on the programmed movement: for more details see the section "Feedrate definitions".

All other settings (Z, T, and R) are **non-modal**: they only influence the current block, not the following blocks. If e.g. no T is given then the T0 (zero tool) is assumed.

T (tool): The active tool is taken in consideration when jogging the robot in manual mode. The command T can be used to modify the current tool without starting a movement. Alternatively, the active tool index can be modified directly in the Monitor structure, as long as the robot is in standstill state.

R (round-edge): A round edge can be inserted between two linear movements using the parameter **R** followed by a non-negative value, which represents the radius of the rounded edge. Note that the radius cannot exceed half of the length of a linear segment and will be automatically reduced otherwise. The user can also program a zero radius (R0) which forces the movement to stop at the transition point, no matter how large the maximum transition angle is.

HOME command: The HOME command is equivalent to MJ P0, where P0 is the default homing position of the robot, i.e. all axes to zero position.

H (rotation angle): If the rotation angle H of the circular interpolation MC is not specified, then the rotation will stop at the target point. If H is programmed then the rotation length is directly defined by the user. Full rotations can be achieved with values of H equal or larger than 360 degrees. If H is small it can happen that the middle point Q and/or the target point P are not reached.

GOTO and SUB commands: The GOTO and SUB commands are followed by a label (**case-sensitive** string). The interpreter scans the program for this label starting from the first line until the Eof. If no label is found an error is output. If more than one label with the same name is defined then only the first occurring one is recognized and used. Note that the target label for the GOTO and SUB commands can only contain one word (e.g. GOTO START_POINT). All leading and trailing empty spaces will be ignored.

The GOTO and SUB commands are similar to each other: they both force the program execution to jump to the target label. However, the major difference is that the SUB command returns back to the calling point as soon as the subroutine ends with an END command.

Both GOTO and SUB commands also support a loop counter, i.e. the jump can be repeated a given number of times. If no loop count is defined then a SUB call is executed only once, while a GOTO loop is executed forever (infinite loop).

The GOTO and SUB commands are not supported for Blocks executions because the corresponding label could be outside of the ring buffer.

Subroutines must be declared after the main program, not before! See examples section for details.

IF..ELSE..ENDIF construct: The IF..ELSE..ENDIF construct is only supported in programs, not for Blocks executions. IF sections can be nested (see examples section).

The IF condition can evaluate the Boolean value of either an R parameter or a DI signal. A negation (C-style !) can be added to the condition (e.g. IF !R3).

TRK command: Conveyor tracking can be activated with the command TRK followed by the conveyor index (either 1 or 2). The command TRK 0 deactivates tracking.

TANG command: The “TANG 1” command switches the C axis into automatic tangential mode, i.e. following the tangent to the path projection on the XY plane. The “TANG 0” command switches back to normal mode. Note that all programmed values for the C axis are ignored during automatic tangential mode. An additional offset can be programmed with the optional parameter H.

Feedrate definitions

Each individual movement can be programmed with its own feedrate. The user has three different feedrate definitions to choose from:

- **F** (default): defined including *all interpolated axes*.

  For point-to-point movements (MJ, HOME) all the joint axes are included:

  $$F = \sqrt{\sum v_j^2}$$

  If all joints are rotatory then the speed units are **deg/s**.

  For path interpolated movements (ML, MC, MS) all the path axes are included:

  $$F = \sqrt{v_X^2 + v_Y^2 + v_Z^2 + v_A^2 + v_B^2 + v_C^2}$$

  The speed units in this case are generally **undefined**, because both Cartesian axes (X,Y,Z) and orientation axes (A,B,C) are mixed together in the calculation.

- **FC** (cartesian): defined including *only Cartesian axes*.

  No matter whether the movement is PTP or path interpolated, only the displacement in the X,Y,Z space is considered when calculating the speed:

  $$F = \sqrt{v_X^2 + v_Y^2 + v_Z^2}$$

  The speed units in this case are **mm/s**.

  If the Cartesian distance between the movement's starting and ending point is zero, then the speed definition automatically falls back to the default configuration.

- **FA** (angular): defined including *only orientation axes*.

  No matter whether the movement is PTP or path interpolated, only the displacement in the A,B,C space is considered when calculating the speed:

  $$F = f(v_A + v_B + v_C)$$

  The speed units in this case are **deg/s**. The angular speed calculation considers the distance between the movement's starting and ending orientation quaternion.

  If the angular distance between the movement's starting and ending point is zero, then the speed definition automatically falls back to the default configuration.

NOTE: PTP and path interpolated movements have a different standard feedrate definition. Dynamic transition between these two different kinds of movements will only be smooth if the same equivalent feedrate definition is used for both (e.g. Cartesian feedrate in mm/s).

Auxiliary axes

Besides the robot’s joints, the library can also control up to 6 external auxiliary axes (Aux). They can be moved independently using the Jog function, or together with the rest of the robot in automatic mode, by including their target position in the destination point structure.

While aux axes are not part of the interpolated path of the robot, they do move synchronously to the trajectory, starting and ending a movement at the same time of the joint axes.

The structure of a target point includes position for the joint/path and for the aux axes:

Point_Type.Axes[0..5] -> target position of either joint or path axes depending on Point_Type.Mode.

Point_Type.Aux[0..5] -> target position of aux axes, either as absolute or incremental value depending on Point_Type.ModeAux.

Point_Type.ModeAux = 0 -> absolute programmed position (default behavior).

Point_Type.ModeAux = 1 -> incremental programmed position, starting from the current position.

If a movement of the auxiliary axes is programmed together with the rest of the robot, the generated trajectory will not violate the maximum speed of the aux axes. This might cause the robot to move slower than expected, in case aux axes limits are set too low.

Calibration

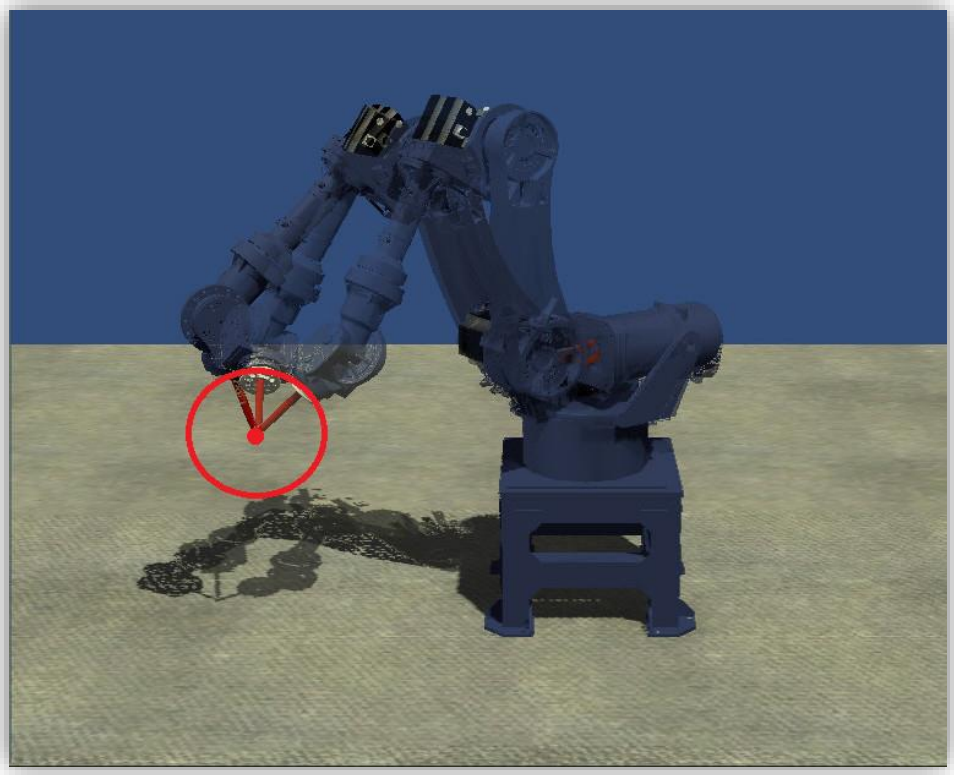
Automatic calibration calculations for tools and frames are described here.

The tool calibration can be used to identify the geometrical size of an unknown tool.

The frame calibration can be used to identify the position and orientation of a user-defined frame with respect to the robot's base frame.

Tool Calibration

The operator needs to manually jog the robot to 5 different orientations (ABC) sharing the same position (XYZ). Note that the actual value of the chosen position is irrelevant; the only requirement is to keep it constant along all the 5 poses.



The recorded values of *Monitor.MountBasePositions* must be inserted in the variables *Parameters.Calibration.Tool.Points[]*.

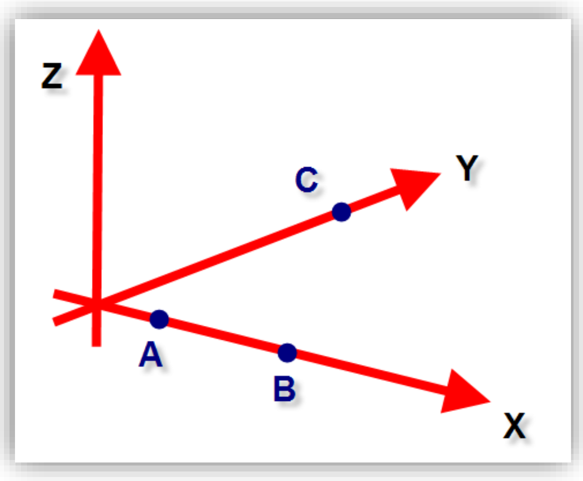
The command *Parameters.Calibration.Tool.Start* can then be activated to start the identification procedure.

The calculation yields the tool size in *Parameters.Calibration.Tool.Result.X/Y/Z*, which can then be copied into the *Parameters.Tool[]* structure.

For best results the orientations should be far apart from each other. In extreme cases, when two or more poses overlap, the procedure will fail and return error ERR_CALIBRATION. In that case the result values are invalid and must not be used.

Frame Calibration

The operator needs to provide the *position* of three points A, B, C as shown in the figure below.



A and B are two points along the X axis, satisfying the condition $X_A<X_B$. Point A can (but does not necessarily have to) be the origin of the axis.

C is a point along the Y axis, satisfying the condition $Y_C>0$.

The origin of the XY plane is found automatically by forcing the two axes to be perpendicular. The direction of the Z axis is found automatically as Z=XxY.

The command *Parameters.Calibration.Frame.Start* can then be activated to start the identification procedure.

The calculation yields the frame position and orientation in *Parameters.Calibration.Frame.Result.Axes*, which can then be copied into the *Parameters.Frame[]* structure.

For best results the points should be far apart from each other. In extreme cases, when two or more points overlap, the procedure will fail and return error ERR_CALIBRATION. In that case the result values are invalid and must not be used.

Examples

The following examples show some simple applications of the robot control library.

Make sure the library is included in the control task:

```
#include "RobControl.h"
```

Robot Declaration

For our examples we define one SCARA robot and one 6axes anthropomorphic robot.

```
#define ROBOT_NUM 2; //our system has 2 robot
Robot_Type Robot[ROBOT_NUM]; // declare an array of robots control elements
```

The robots will be therefore identified by the structures Robot[0] and Robot[1].

```
Robot[0].Parameters.Mechanics.Type = SCARA;
Robot[0].Parameters.Mechanics.Links[1].Offset.X = 200;
Robot[0].Parameters.Mechanics.Links[2].Offset.X = 200;

Robot[1].Parameters.Mechanics.Type = ARM;
Robot[1].Parameters.Mechanics.Links[0].Offset.Z = 480;
Robot[1].Parameters.Mechanics.Links[1].Offset.X = 400;
Robot[1].Parameters.Mechanics.Links[1].Offset.Z = 200;
Robot[1].Parameters.Mechanics.Links[2].Offset.Z = 1100;
Robot[1].Parameters.Mechanics.Links[3].Offset.X = 766;
Robot[1].Parameters.Mechanics.Links[3].Offset.Z = 230;
Robot[1].Parameters.Mechanics.Links[4].Offset.X = 345;
Robot[1].Parameters.Mechanics.Links[5].Offset.X = 244;
```

We will start working with the SCARA robot. We need to assign some initial values to its parameters:

```
for (int i=0;i<4;i++)
{
      Robot[0].Parameters.JointLimits[i].VelocityNeg = 100;
      Robot[0].Parameters.JointLimits[i].VelocityPos = 100;
      Robot[0].Parameters.JointLimits[i].AccelerationNeg = 1000;
      Robot[0].Parameters.JointLimits[i].AccelerationPos = 1000;
      Robot[0].Parameters.JointLimits[i].JerkNeg = 10000;
      Robot[0].Parameters.JointLimits[i].JerkPos = 10000;
}

Robot[0].Parameters.PathLimits.Linear.Velocity = 1000;
Robot[0].Parameters.PathLimits.Linear.Acceleration = 10000;

Robot[0].Parameters.PathLimits.Angular.Velocity = 100;
Robot[0].Parameters.PathLimits.Angular.Acceleration = 1000;
```

NOTE: if jerk limits are not assigned then they are automatically set to 10 times the acceleration values. Keep in mind that path angular limits are given in [deg/s] and are usually much smaller than linear limits, which are given in [mm/s].

```
Robot[0].Parameters.JointLimits[0].PositionNeg = -180;
Robot[0].Parameters.JointLimits[0].PositionPos = 180;
Robot[0].Parameters.JointLimits[1].PositionNeg = -160;
Robot[0].Parameters.JointLimits[1].PositionPos = 160;
Robot[0].Parameters.JointLimits[2].PositionNeg = 0;
Robot[0].Parameters.JointLimits[2].PositionPos = 200;
Robot[0].Parameters.JointLimits[3].PositionNeg = -360;
Robot[0].Parameters.JointLimits[3].PositionPos = 360;
```

```
Robot[0].Parameters.Override = 100;
Robot[0].Parameters.CycleTime = 0.001; //1ms cycle time
Robot[0].Parameters.FilterTime = 0.05; //50ms filter time
Robot[0].Parameters.MaxTransitionAngle = 90.0;
```

NOTE: the RobotControl function will not work correctly if the specified cycletime in the parameters structure is not equal to the controller task cycle

Assigning units ratios to all joint axes is important to make sure that the communication with the servo drives works correctly:

```
for (int i=0;i<4;i++)
{
      Robot[0].Parameters.UnitsRatio[i].AxisUnits = 360; //degrees
      Robot[0].Parameters.UnitsRatio[i].MotorUnits = 10000; //pulses
      Robot[0].Parameters.UnitsRatio[i].Direction = 1; //positive
      Robot[0].Parameters.UnitsRatio[i].HomeOffset = ...; //from encoder
}
```

Function call

In order for the motion control library to work, the function call must be executed cyclically:

```
status = RobotControl(&Robot, ROBOT_NUM);
```

Joint axes homing

When starting up a robot we need to tell the RobotControl function the current position of the joint axes. This can be done via the command *SetJoints* with the parameters *ActFromDrives (expressed in motor units)*.

```
Robot[0].Parameters.ActFromDrives[0] = 1250;
Robot[0].Parameters.ActFromDrives[1] = -555;

Robot[0].Commands.SetJoints = 1;
```

Note that no movement happens on the robot when setting the joint values. The output position of the path axes will jump to the new calculated TCP position and orientation (in mm for translational axes and in degrees for rotational axes):

```
Robot[0].Monitor.PathPosition[0] = 322.6829
Robot[0].Monitor.PathPosition[1] = 225.945
Robot[0].Monitor.PathPosition[2] = 0.0
Robot[0].Monitor.PathPosition[3] = 25.0
```

NOTE: the SetJoints command can only be executed if the robot is in standstill state and if the servo drives are switched off!

Jog Movements

We can now start some simple jogging movements on the robot.

The structure Parameters.Jog is used to configure manual movements:

```
Robot[0].Parameters.Jog.Mode = JOG_JOINTS;
Robot[0].Parameters.Jog.AxisIndex = 0; //first joint
Robot[0].Parameters.Jog.Direction = JOG_NEGATIVE;

Robot[0].Commands.JogAxis = 1; //start jogging movement
```

The first joint of the robot will start moving in the negative direction as long as the JogAxis command is set to 1, or until the axis reaches its limit position.

Alternatively, an interpolated movement along the path axes can be executed:

```
Robot[0].Parameters.Jog.Mode = JOG_BASE;
Robot[0].Parameters.Jog.AxisIndex = 0; //first path axis (X)
Robot[0].Parameters.Jog.Direction = JOG_POSITIVE;

Robot[0].Commands.JogAxis = 1; //start jogging movement
```

In this case the movement will continue until the target X axis position cannot be reached anymore by the mechanics, or until one of the joints reach its limits.

Note that moving along the path axes at constant speed causes the joints axes to move at varying speed, which can easily exceed their dynamic limits when approaching a singularity. In that case the movement will abort with error ERR_SPG_LIMIT_VEL (1206). The movement can be resumed after resetting the error.

WARNING: jogging movements are limited in speed by the joint axes limits (for mode JOG_JOINTS) and by the path speed limits (for modes JOG_BASE and JOG_TOOL). The override parameter should be used to reduce the actual movement speed.

NOTE: the JOG_TOOL mode only works for 6axes robots, where the TCP base frame and TCP tool frame are not equivalent.

Axes jogging movements can also be executed to a specific target position:

```
Robot[0].Parameters.Jog.Mode = JOG_BASE;
Robot[0].Parameters.Jog.AxisIndex = 0; //first path axis (X)
Robot[0].Parameters.Jog.Direction = JOG_GOTO;
Robot[0].Parameters.Jog.GotoPos = 150.0;

Robot[0].Commands.JogAxis = 1; //start jogging movement
```

What if the robot does not move? Check the following parameters:

- Override: is it set to a value larger than 0%?
- Cycletime: is it set correctly equal to the controller cycle time?
- Movement dynamic values: are velocity, acceleration and jerk for joint axes and path set to non-zero values?
- State: is the axis in Error state? If yes, then reset the error before moving again.

Single block movement

A single block movement can be executed with the command *RunBlocks*. A string containing the movement instruction must be passed to the Parameters.Blocks[0] variable.

For example, the string "*ML P1 F100*" will move the robot to the point P1 with a linear interpolation and path speed of 100 mm/s.

The point P1 must be defined in the Parameters.Points structure. In the example below the point P1 is defined in the path coordinate system (X=100,Y=100).

```
Robot[0].Parameters.Points[1].Mode = POINT_PATH; //defined in X,Y,Z,A
```

```
Robot[0].Parameters.Points[1].Axes[0] = 100; // X axis position
Robot[0].Parameters.Points[1].Axes[1] = 100; // Y axis position

strcpy(&Robot[0].Parameters.Blocks[0],"ML P1 F100");

Robot[0].Command.RunBlocks = 1;
```

NOTE: the actual speed of an interpolated movement is not always necessarily equal to the programmed speed, because the joint axes limit velocities can never be violated. An automatic reduction of the path speed might be observed during the movement. This commonly happens around singularities.

A PTP movement to a point defined in the joint coordinate system can be programmed as follows:

```
Robot[0].Parameters.Points[2].Mode = POINT_JOINTS; //defined in Qi
Robot[0].Parameters.Points[2].Axes[0] = -30;    // J1 axis position
Robot[0].Parameters.Points[2].Axes[1] = 0;      // J2 axis position
Robot[0].Parameters.Points[2].Axes[2] = 50;     // J3 axis position
Robot[0].Parameters.Points[2].Axes[3] = 90;     // J4 axis position

strcpy(&Robot[0].Parameters.Blocks[0],"MJ P2 F100");

Robot[0].Command.RunBlocks = 1;
```

All movements can be stopped with the *Stop* command, as already seen for the jogging movements. However, block movements can also be halted (paused) and then resumed with the *Halt* and *Continue* commands.

NOTE: the robot will still be in MOVING state after a *Halt* command. A *Stop* command is needed to go back to STANDSTILL state and be able to start a new movement.

Ring buffer blocks movement

More than one block can be inserted in the Parameters.Blocks array and the execution continues until an empty block is found (or, alternatively, until the *Stop* command is called).

As soon as a block is processed by the motion planner, the corresponding string is deleted from the array, leaving space for the user to insert new blocks.

The Parameters.Blocks array is handled as a ring buffer, which means that the user can keep filling in new blocks and keep the movement active continuously.

NOTE: the application must ensure that the motion parameters (target points, tool, frame) are valid during the actual execution of each block. The parameters should only be modified after the block has been executed and has been removed from the ring buffer.

NC Program execution

The command RunProgram can be used to run a complete NC program. The syntax recognized by the internal interpreter has been described before.

Next is an example of a text program that can run on the RobControl library:

```
HOME F300
MJ P1 F100
LOOP:
 ML P2 T1 Z1
 ML P3 T1 Z1
 ML P4 T1 Z1
 M11
 WAIT 2
 MC P2 Q5 T1 Z1
 MJ P1 T1 Z1
 M10
 WAIT 1
GOTO LOOP 2 //jump back to label “LOOP” for two cycles
END
```

Assuming that the above text program is referenced by the pointer *(char*) myprg* then we can run it by using the following code:

```
Robot[0].Parameters.Program = myprg;

Robot[0].Command.RunProgram = 1;
```

It is also possible to start from any line number (e.g. line 7 = “M11”) in the middle of the program simply by setting:

```
Robot[0].Parameters.StartLine = 7;
```

Programs execution can be halted, continued and stopped as already described for block movements.

By setting the SingleStep parameter the program can be executed one line at the time:

```
Robot[0].Parameters.SingleStep = 1;
```

The monitor structure will show a program *moving* but *halted*, and execution can be resumed to the next step with the *Continue* command.

Note that the single step mode cannot be activated while a movement is already active.

NC Program execution with subroutines

One or more subroutines can be called inside an NC program. Note that all subroutines must be defined at the end of the main program!

```
HOME F3000
MJ P1 F1000
SUB CUT         //subroutine “CUT” executed once here
MJ P2 F1000
SUB CUT 3       //subroutine “CUT” executed three times here
END

//subroutine defined here
CUT:
 ML P20 F50
 MC P21 Q22
 ML P23
END
```

NOTE: if the starting line of the program is set in the middle of a subroutine, the execution will continue only until the next END command and then abort. No jump back to the original calling line can be made, because that part of the code was entirely skipped!

Robot Synchronization

The path movement of a robot can be synchronized with the rest of the application or also with other robots on the same controller. The synchronization is achieved with the help of M commands.

For example, a typical application consists of two robots that move independently to a starting position and then synchronize with each other. We can define an M command to be synchronous for both robots:

```
Robot[0].Parameters.M_synch[20] = 1; //define M20 as synchronous for Robot1
Robot[1].Parameters.M_synch[20] = 1; //define M20 as synchronous for Robot2

Robot[0].Parameters.Program = myprg1;
Robot[1].Parameters.Program = myprg2;

Robot[0].Command.RunProgram = 1;
Robot[1].Command.RunProgram = 1;
```

As soon as each robot will reach the M20 command in the program execution the movement will halt and wait for the application to reset it. The application is responsible for resetting the two M20 commands in the same cycle as soon as they are both set by the two robots.

Blocking M-functions

Synchronous M commands cause both the path planning and the program execution to halt. That is useful when the application needs to modify the target position of the movements programmed after the M-function.

```
MJ P1 F100
M1
MJ P1
```

If M1 is defined synchronous (`Robot[0].Parameters.M_synch[1] = 1`) then the program execution will halt at the M1 line. Now the application can modify the target properties of point P1 (`Robot[0].Parameters.Points[1]…`) and cause a new path to be planned after M1 is reset.

IF..ELSE construct

The IF construct evaluates a Boolean condition, which can be specified as either an R parameter or a DI signal. A C-style negation (!) can be added to the condition.

The ELSE section is optional, while the ENDIF statement is mandatory. IF sections can be nested.

```
MJ P1 F100
IF DI3
  ML P2
ELSE
  ML P3
  IF !R1
    END          //program aborts here if both DI3 and R1 are false
  ENDIF
ENDIF
```

If either no matching ELSE or ENDIF is found after an IF section, then error ERR_IP_IF will be triggered during execution.

Note that IF statements are path-synchronous, that is, the condition is only evaluated when the movement execution reaches that line. This behavior causes the movement to stop at the IF statement, no matter what Boolean value the condition holds.

In case the IF statement also needs to be synchronized with the PLC application, then a synch-M command needs to be called before the IF line. A typical case is when the expression to be evaluated is an R parameter, which is normally set by the application:

```
MJ P1 F100
M5                  // synchronize with PLC application: the value of R1 must be set before it is evaluated
IF R2
  ML P2 F500
ENDIF
```

Transition Angle

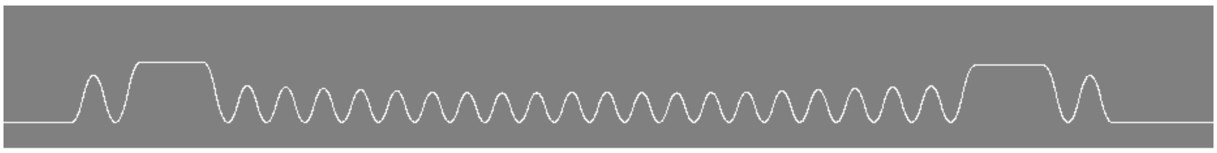
When two movements are programmed consecutively the path speed will either stop at the transition point or continue smoothly from the first to the second block.

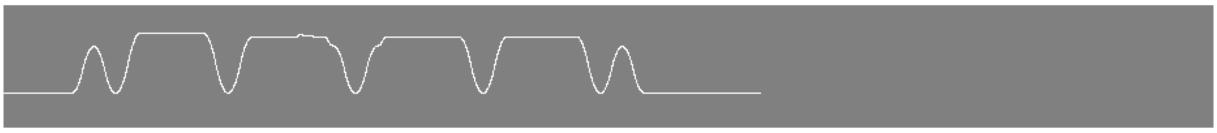
The deciding factor is the vector angle between the two movements in the joint world: if the transition angle is small enough then the two movements are tangential and the path speed will not decrease; if the transition angle is too large then the transition is non-tangential and the path speed will stop.

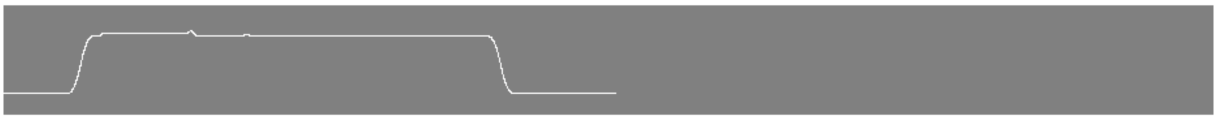
The parameter Robot.Parameters.MaxTransitionAngle is the threshold that separates tangential transitions from non-tangential transitions. Its value can range from 0 to 180 degrees.

Setting the MaxTransitionAngle low causes the path speed to halt at most transitions. On the other hand, setting it high will keep the path speed constant at most transitions. However, the higher the transition angle, the higher will the speed jumps be on the joint axes. This inconvenience can be partly mitigated by increasing the filter time on the output position values to the servo drives.

The following example shows the difference between a MaxTransitionAngle of 0, 90 and 180 degrees.







Allowing higher transition angles to be tangential clearly brings a substantial reduction in the whole movement time since the path speed does not need to stop at each block transition.

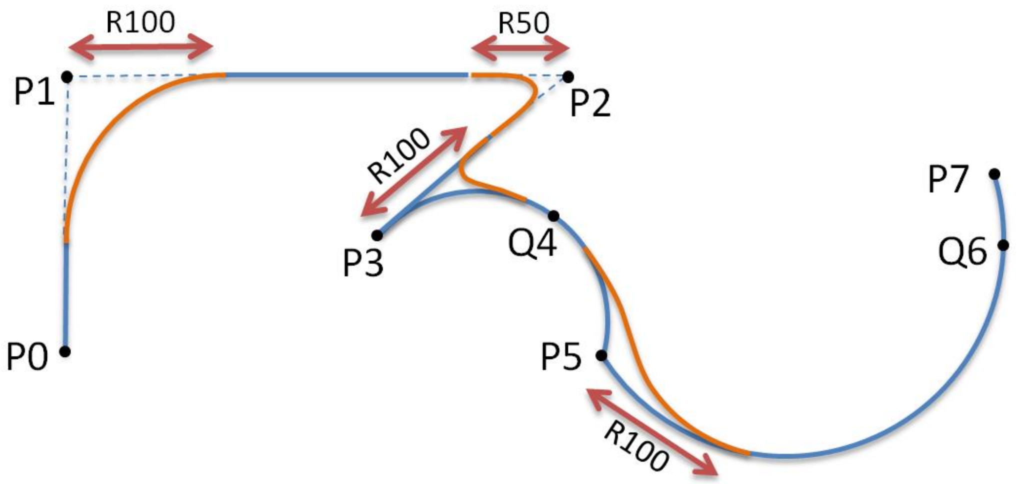
In contrast, allowing only low transition angles to be tangential slows down the movement but normally also increases its accuracy.

The final choice comes down to a compromise between speed and precision.

Round Edge

A round edge can be inserted between any two movements (ML, MJ, MC) with the addition of the R parameter specifying the radius of the edge.

ML P1 R100 FC100
ML P2 R50
ML P3 R100
MC Q4 P5 R100
MC Q6 P7



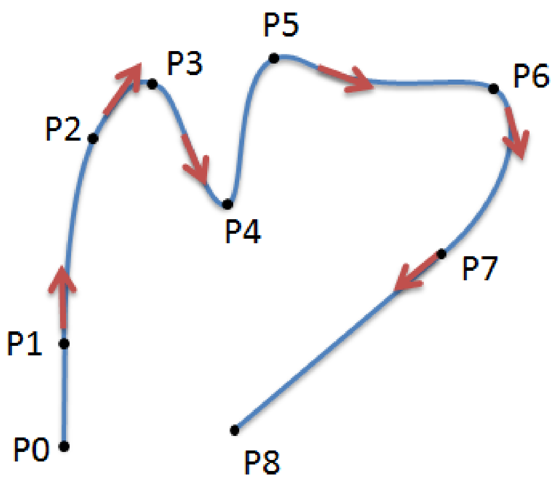
Programming a zero radius can be used to force a speed stop at a point, even if the transition is tangential.

ML P1 R0 F100
ML P2

NOTE: round edges programmed at transitions with spline movements (MS) are ignored because splines are per definition built tangentially to the adjacent movement.

Tangential axis

The C axis of a robot, that is the rotation around the Z direction, can be programmed to be in automatic tangential mode. The axis will always point along the tangent to the path, as seen projected on the XY plane.



The activation of tangential mode is executed by the command *TANG 1*.

The command *TANG 0* deactivates the function and switches back to normal mode.

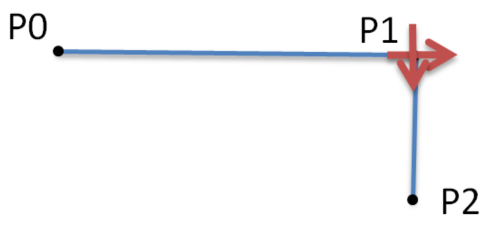
```
MJ P0 F100
ML P1 F500
TANG 1 //activates tangential mode
MS P2
MS P3
MS P4
MS P5
MS P6
MS P7
TANG 0 // deactivates tangential mode
ML P8 F6000
```

NOTE: all the values programmed for the C axis are ignored when the tangential mode is active. That is because the axis orientation is automatically interpolated from the XY path and cannot be manually set by the user.

While active, the tangential axis will align itself with the direction of the upcoming path at the beginning of each movement. This can cause a stop in path speed depending on the angle between the two consecutive movements in the XY plane.

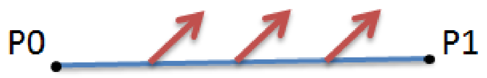
Extreme examples are:

- Splines: the path will never stop because splines are built to be tangential (see figure above). However, the path speed might decrease on high curvature segments.
- Sharp corners between lines: the path is forced to a stop at the transition point to let the C axis change its orientation before continuing (see figure below)



If the application allows, adding a round edge between the two linear segments (see previous paragraph) removes the need for a path stop, at the cost of a small path violation.

An optional offset angle between the actual XY path and the tangential axis can be programmed with the parameter H. For example, the command “TANG 1 H45” forces the C axis to align itself at 45 degrees with respect to the true tangent. This is useful to compensate for fixed mounting angles of the tool on the machine.



NOTE: tangential mode only affects path interpolated movements, not PTP movements!

WARNING: when using tangential axis with a 6 axes robot make sure that the TCP is always aligned perpendicularly to the XY plane (i.e. B=90) before and during the activation of the automatic tangential mode otherwise the behavior could be unpredictable.

Conveyor tracking

The command TRK can be used to activate synchronization with an external conveyor. A maximum of two conveyors can be configured for each existing robot.

```
MJ P0 F100
TRK 1 //synchronize with Conveyor 1
ML P1 F100
ML P2 F100
TRK 0 //de-synchronize from Conveyor 1
ML P3 F100
```

The TRK 1/2 command starts a tracking session, during which the robot follows the Conveyor 1/2 along the direction defined by the Robot.Parameters.Conveyor[1/2].Angle.

The tracking session continues until the command TRK 0 is reached (or until the end of the program). Note that each tracking session (TRK 1/2) must be closed (TRK 0) before a new one is started. In other words, the command TRK 1 cannot be programmed if the TRK 2 is already active.

When the program reaches a new tracking session the path speed stops and the flag Robot.Monitor.TrackSynch is raised. At this point the application needs to synchronize with the program and update the target positions of all movements included in the tracking session. Once the synchronization is completed the flag must be reset and the program can continue running.

```
if (Robot[0].Monitor.TrackSynch)
{
  Robot[0].Monitor.TrackSynch = 0; //reset Synch flag
  Robot[0].Parameters.Points[1].Axes[0] = ...; //actual object's position
  Robot[0].Parameters.Points[2].Axes[0] = ...; //actual object's position
}
```

All points are always programmed in the base frame, no matter if inside or outside a tracking session. Of course, standard frames F[] can be activated at any time.

Note that during a tracking session the robot will continue following the conveyor even if the program is halted, either because of a halt command or because of single-step operation.

Joint reference positions

If a PTP movement is programmed with a target point given in path coordinates the robot needs to make a choice between all the possible joint configurations.

The default rule is to select the configuration “closest” to the initial pose, starting from the first joint. However, there are situations where the default choice leads to unwanted poses, or even to errors because of workspace violations or mechanical limits.

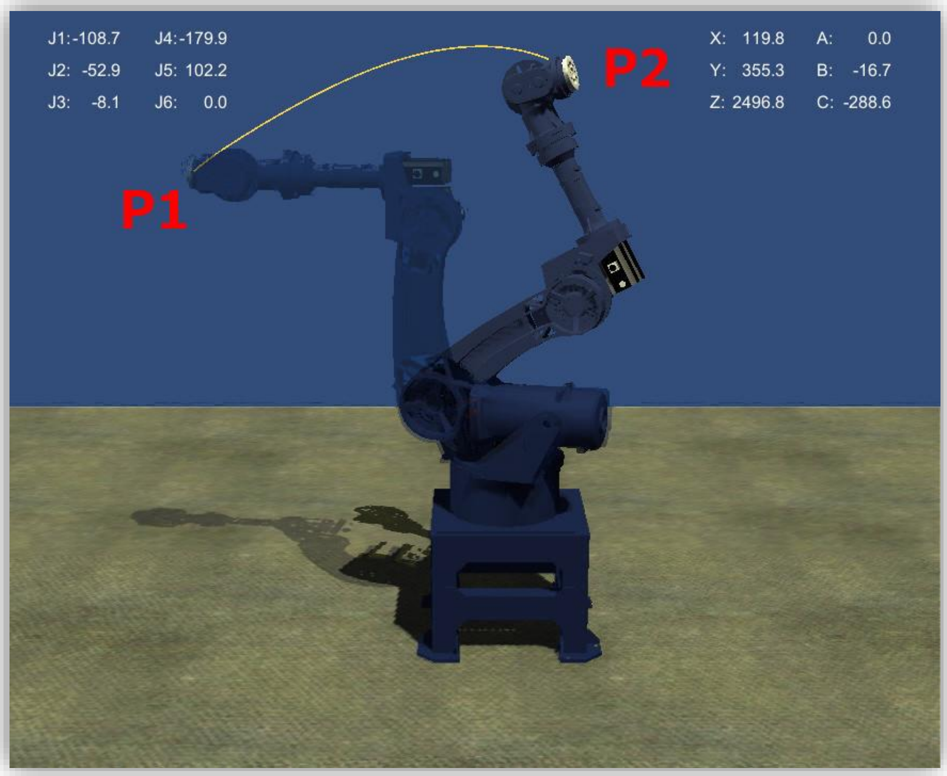
The user is given the possibility to override the internal pose selection by specifying a reference position for each of the joints. The robot will then select the joint configuration closest to the programmed reference values.

The following example shows a movement programmed from P1 to P2, where P2 is given in path coordinates:

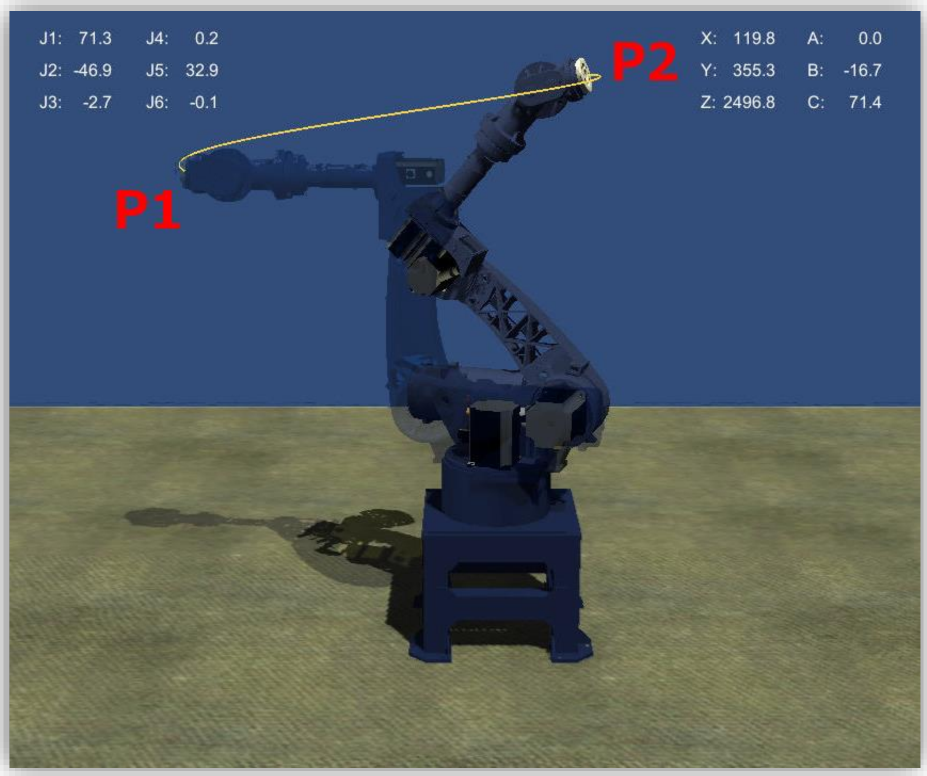
P2 = {X=119.8; Y=355.3; Z=2496.8; A=0.0; B=-16.7; C=-288.6}

The starting position is P1 = {J1=-100; J2..6 = 0.0}

Programming the movement as *MJ P2* leads to a solution with J1=-108.7, which is the closest to the initial pose.



However, the user might prefer (or require) a different final pose, and could program the movement as *MJ P2 J1=0*. That leads to a solution with J1=71.3, which is closer to the reference programmed value of J1=0.


J1: 71.3 J4: 0.2
J2: -46.9 J5: 32.9
J3: -2.7 J6: -0.1

X: 119.8 A: 0.0
Y: 355.3 B: -16.7
Z: 2496.8 C: 71.4

Error codes

The robot motion control can report several *error messages* during operation.

The following list includes some of the possible errors:

Error number	Error code	Description
0	`STATUS_OK`	
1005	`ERR_CYCLETIME`	Cycle time not configured correctly
1006	`ERR_CHECKSUM`	Header file corrupted
1010	`ERR_MAX_ROBOTS`	Maximum number of robots exceeded
1011	`ERR_ROBOT_LICENSE`	License not found
1012	`ERR_JOG_PAR`	Jogging parameters not configured correctly
1013	`ERR_JOG_GOTOPOS`	Jogging absolute target position wrong
1014	`ERR_WRONG_JOINT_LIMITS`	Joint limits not configured correctly
1015	`ERR_UNITS_SCALING`	Axis units scaling not configured correctly
1016	`ERR_POINT_TYPE`	Target point type not supported
1017	`ERR_WRONG_AUX_LIMITS`	Auxiliary axes limits not configured correctly
1021	`ERR_TRF_MECH`	Mechanical parameters not configured correctly
1022	`ERR_TRF_WORKSPACE`	Position outside mechanical reach
1023	`ERR_TRF_MECH_NOT_SUPPORTED`	Robot type not supported
1025	`ERR_TRF_POINTER`	Null pointer to user transformations
1026	`ERR_TRF_AXESNUM`	Wrong number of joints for user transformations
1027	`ERR_TRF_ROT`	Wrong rotation axes configured for RTCP
1030	`ERR_CALIBRATION`	Tool calibration failed (check points)
1050	`ERR_NOT_SUPPORTED`	Command not supported in block movements
1100	`ERR_IP_EMPTYSTRING`	No block programmed
1102	`ERR_IP_SYNTAX`	Syntax error in current block
1103	`ERR_IP_CONFLICT`	Conflicting codes programmed in current block
1104	`ERR_IP_NOPOINT`	No target point programmed
1105	`ERR_IP_NOCENTER`	No center point programmed
1106	`ERR_IP_MAXMFUNC`	Maximum number of M functions on same block reached
1107	`ERR_IP_POINTINDEX`	Point index is not correct
1108	`ERR_IP_TOOLINDEX`	Tool index is not correct
1109	`ERR_IP_FRAMEINDEX`	Frame index is not correct
1110	`ERR_IP_MFUNCINDEX`	M function index is not correct
1111	`ERR_IP_FEEDRATE`	No feedrate programmed
1113	`ERR_IP_LABEL`	Label for GOTO/SUB commands not found
1114	`ERR_IP_TRK_INDEX`	Conveyor index is not correct
1115	`ERR_IP_JUMP`	GOTO/SUB commands not supported in block movements
1116	`ERR_IP_SUBLEVEL`	Subprogram level too deep
1117	`ERR_IP_TANG`	Wrong syntax for tangential command
1118	`ERR_IP_IO_INDEX`	Wrong index for IO command
1119	`ERR_IP_IF`	Error in IF..ELSE..ENDIF construct
1150	`ERR_PP_CIRCLEPOINTS`	Error in circle points (must not be aligned)
1160	`ERR_WORKSPACE_ZONE1`	Zone 1 of Workspace violated
1161	`ERR_WORKSPACE_ZONE2`	Zone 2 of Workspace violated
1162	`ERR_WORKSPACE_ZONE3`	Zone 3 of Workspace violated
1163	`ERR_WORKSPACE_ZONE4`	Zone 4 of Workspace violated
1164	`ERR_WORKSPACE_ZONE5`	Zone 5 of Workspace violated

1165	ERR_WORKSPACE_ZONE6	Zone 6 of Workspace violated
1166	ERR_WORKSPACE_ZONE7	Zone 7 of Workspace violated
1167	ERR_WORKSPACE_ZONE8	Zone 8 of Workspace violated
1168	ERR_WORKSPACE_ZONE9	Zone 9 of Workspace violated
1169	ERR_WORKSPACE_ZONE10	Zone 10 of Workspace violated
1171	ERR_LIMIT_X	Position limit exceeded on axis X
1172	ERR_LIMIT_Y	Position limit exceeded on axis Y
1173	ERR_LIMIT_Z	Position limit exceeded on axis Z
1174	ERR_LIMIT_A	Position limit exceeded on axis A
1175	ERR_LIMIT_B	Position limit exceeded on axis B
1176	ERR_LIMIT_C	Position limit exceeded on axis C
1181	ERR_LIMIT_J1	Position limit exceeded on axis J1
1182	ERR_LIMIT_J2	Position limit exceeded on axis J2
1183	ERR_LIMIT_J3	Position limit exceeded on axis J3
1184	ERR_LIMIT_J4	Position limit exceeded on axis J4
1185	ERR_LIMIT_J5	Position limit exceeded on axis J5
1186	ERR_LIMIT_J6	Position limit exceeded on axis J6
1187	ERR_LIMIT_VEL_J1	Speed limit exceeded on axis J1
1188	ERR_LIMIT_VEL_J2	Speed limit exceeded on axis J2
1189	ERR_LIMIT_VEL_J3	Speed limit exceeded on axis J3
1190	ERR_LIMIT_VEL_J4	Speed limit exceeded on axis J4
1191	ERR_LIMIT_VEL_J5	Speed limit exceeded on axis J5
1192	ERR_LIMIT_VEL_J6	Speed limit exceeded on axis J6
1204	ERR_SPG_DYNCALC	Error in dynamical calculations
1205	ERR_SPG_LIMIT_POS	Position limits reached during movement
1206	ERR_SPG_LIMIT_VEL	Velocity limits reached during movement
1207	ERR_SPG_LIMIT_ACC	Acceleration limits reached during movement
1208	ERR_SPG_LIMIT_JERK	Jerk limits reached during movement
1220	ERR_LIMIT_A1	Position limit exceeded on axis Aux1
1221	ERR_LIMIT_A2	Position limit exceeded on axis Aux2
1222	ERR_LIMIT_A3	Position limit exceeded on axis Aux3
1223	ERR_LIMIT_A4	Position limit exceeded on axis Aux4
1224	ERR_LIMIT_A5	Position limit exceeded on axis Aux5
1225	ERR_LIMIT_A6	Position limit exceeded on axis Aux6
1230	ERR_LIMIT_VEL_A1	Speed limit exceeded on axis Aux1
1231	ERR_LIMIT_VEL_A2	Speed limit exceeded on axis Aux2
1232	ERR_LIMIT_VEL_A3	Speed limit exceeded on axis Aux3
1233	ERR_LIMIT_VEL_A4	Speed limit exceeded on axis Aux4
1234	ERR_LIMIT_VEL_A5	Speed limit exceeded on axis Aux5
1235	ERR_LIMIT_VEL_A6	Speed limit exceeded on axis Aux6
1250	ERR_TRK1	Error while tracking conveyor 1
1251	ERR_TRK2	Error while tracking conveyor 2
1260	ERR_OPTMOT	Invalid optimized trajectory
1270	ERR_COLL_SELF	Self-collision detected
1271	ERR_COLL_INTER	Inter-collision detected
1280	ERR_SINGULARITY	The robot is near a singularity point