MECADEMIC

INDUSTRIAL ROBOTICS

MC-PM-MCS500

Revision number: 11.1.43

Mecademic Robotics

July 17, 2025

Contents

Programming Manual for the MCS500 Industrial Robot

```python
with initializer.RobotWithTools() as robot:
    # CHECK THAT IP ADDRESS IS CORRECT! #
    try:
        robot.Connect(address='192.168.0.100')
    except mdr.CommunicationError as e:
        logger.info(f'Robot failed to connect. Is the IP address correct? {e}')
        raise e

    try:
        # Reset the robot configuration (joint limits, work zone, etc.)
        #initializer.reset_robot_configuration(robot)

        # Send the commands to get the robot ready for operation.
        logger.info('Activating and homing robot...')
        initializer.reset_motion_queue(robot, activate_home=True)
        initializer.reset_vacuum_module(robot)

        # Pause execution until robot is homed.
        robot.WaitHomed()
        logger.info('Robot is homed and ready.')

        # Send motion commands to have the robot draw out a square.
        if tools.robot_model_is_meca500(robot.GetRobotInfo().robot_model):
            robot.MovePose(200, 0, 300, 0, 90, 0)
```

For firmware version: 11.1

Document revision: B

Online release date: July 17, 2025

Document ID: MC-PM-MCS500

About this manual

This manual describes the key concepts for industrial robots and the communication methods used with our robots through an Ethernet-enabled computing device (IPC, PLC, PC, Mac, Raspberry Pi, etc.): using either TCP/IP, EtherCAT, EtherNet/IP, or PROFINET protocols. To maximize flexibility, we do not use a proprietary programming language. Instead, we provide a set of robot-related instructions, an API, making it possible to use any modern programming language that can run on your computing device.

The default communication protocol for Mecademic robots is TCP/IP; it consists of a set of text-based motion and request commands sent to and returned by the robot. Additional cyclic communication protocols (EtherCAT, EtherNet/IP, and PROFINET) are also available and described in this manual. However, even if you do not intend to use the TCP/IP protocol, it is necessary to read the chapter that describes its text-based commands.

Furthermore, we offer a fully-fledged Python API, available from our GitHub account. That API is self-documented, but you still need to read the present programming manual.

Danger

Reading the user manual of your robot (MC-UM-MCS500) and understanding the robot's operating principles is a prerequisite to reading this programming manual.

All of our robot models are programmed similarly, with only minor differences. For instance, certain commands and messages are specific to particular models and their optional accessories. To streamline your experience, this programming manual has been tailored specifically for the MCS500 robot and its accessories.

Symbol definitions

The following table lists the symbols that may be used in Mecademic documents to denote certain conditions. Particular attention must be paid to the warning and danger messages in this manual.

Note

Identifies information that requires special consideration.

Warning

Provides indications that must be respected in order to avoid equipment or work (data) on the system being damaged or lost.

Danger

Provides indications that must be respected in order to avoid a potentially hazardous situation, which could result in injury.

Revision history

The firmware that is installed on Mecademic products has the following numbering convention:

{major}.{minor}.{patch}.{build}

Each Mecademic manual is written for a specific {major}.{minor}.{*}.{*} firmware version. On a regular basis, we revise each manual, adding further information and improving certain explanations. We only provide the latest revision for each {major}.{minor}.{*}.{*} firmware version. Below is a summary of the changes made in each revision.

Revision	Date	Comments
B	April 21, 2025	Substantial revision of Sections 3 and 4. Addition of *Cyclic protocols* subsection to every API command.
A	March 17, 2025	Original version

The document ID for each Mecademic manual in a particular language is the same, regardless of the firmware version and the revision number.

Basic theory and definitions

We place a high value on technical accuracy, detail, and consistency, and use terminology that may not always align with standard industry terms. Therefore, it is important to read this section carefully, even if you have prior experience with industrial robot arms.

Definitions and conventions

Units

Distances that are displaced to or defined by the user are in millimeters (mm), angles are in degrees (°) and time is in seconds (s), except for timestamps.

Joint numbering

The joints of the MCS500 are also numbered in ascending order, starting from the base, but the last two "joints" are actually independent degrees of freedom, a translation and a rotation, and are achieved by driving a ball nut and a spline nut about a ball-screw spline. Thus, the translational motion is arbitrarily designated as joint 3, while the rotation about the spline shaft axis is designated as joint 4. Figure 1 shows the joint numbering, the joint directions and the zero joint position in the MCS500 robot. In that "zero" robot position, the axes of joints 1, 2, and 4 are coplanar and the spline shaft is all the way up.

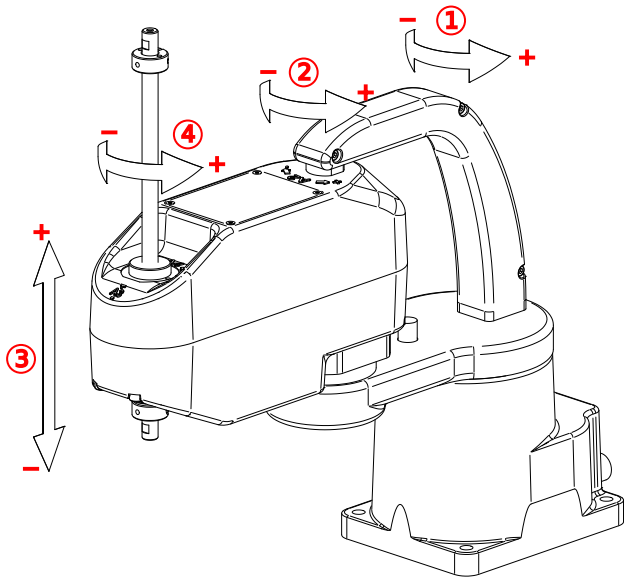


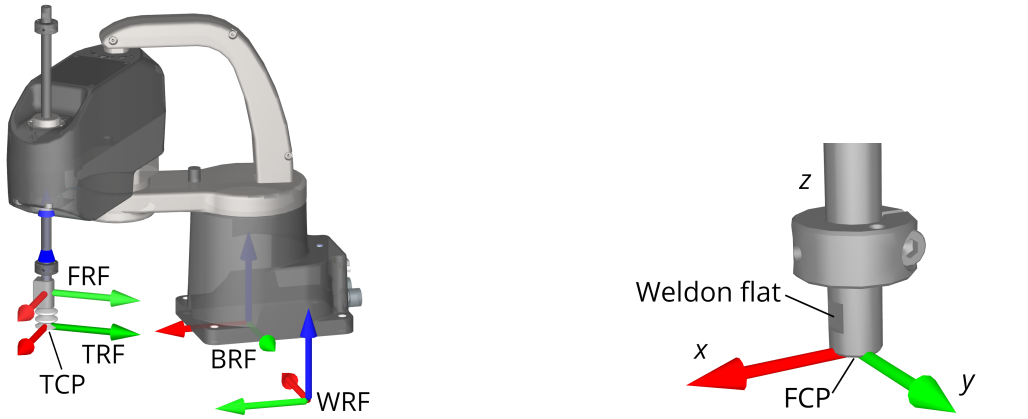
Figure 1: MCS500's joint numbering, joint directions and zero joint positions

Reference frames

We use right-handed Cartesian coordinate systems (reference frames). There are only four of them that you need to be familiar with, as shown in Figure 2 (x axes are red, y axes are green, and z axes are blue). These four reference frames and the key term related to them are:

- *BRF* (page 339): *Base reference frame* (page 339). Static reference frame fixed to the robot base. Its z axis coincides with the axis of joint 1 and points upwards, its origin lies on the bottom of the robot base, and its x axis is normal to the base front edge and points forward.

- *WRF* (page 341): *World reference frame* (page 341). The main static reference frame coincides with the BRF by default. It can be defined with respect to the BRF using the *SetWrf* (page 168) command.
- *FRF* (page 339): *Flange reference frame* (page 339). Mobile reference frame fixed to the robot's flange. The FRF is fixed to the end of the spline shaft that is closer to the robot's base, so that its z axis coincides with the axis of the spline shaft and points away from the robot's base, its origin is at the very end of the spline shaft, and its x axis is perpendicular to the plane of the Weldon flat.
- *FCP* (page 339): *Flange center point* (page 339). Origin of the FRF.
- *TRF* (page 341): *Tool reference frame* (page 341). The mobile reference frame associated with the robot's end-effector. By default, the TRF coincides with the FRF. It can be defined with respect to the FRF with the *SetTrf* (page 166) command.
- *TCP* (page 341): *Tool center point* (page 341). Origin of the TRF. (Not to be confused with the Transmission Control Protocol acronym, which is also used in this document.)



(a) All frames

(b) Flange Reference Frame (FRF)

Figure 2: Reference frames for the MCS500

Pose

Some Mecademic commands accept *pose* (page 340) (position and orientation of one reference frame with respect to another) as arguments. In these commands, and in the the MecaPortal web interface, a pose consists of a Cartesian position, {x, y, z}, and the single orientation angle γ, about a vertical axis. *All reference frames have their z axes upwards. It is impossible, for example, to specify a TRF with the z-axis pointing downwards.*

Joint positions and last joint turn configuration

The angle associated with the rotational joints 1, 2, and 4, θ_1, θ_2, and θ_4, respectively, and the position of the translational joint 3, d_3, will collectively be referred to as *joint position* (page 340) i (i = 1, 2, 3, 4). Since the last joint of the robot (joint 4) can rotate more than one revolution, you should think of the joint angle as a motor angle, rather than as the angle

between two consecutive robot links. Unless you attach an end-effector with cabling to the robot flange, there is no way of knowing the value of the last joint angle just by observing the robot.

Note that the directions for each joint are engraved on the robot's body and shown in Figure 1. Also, as previously mentioned, all joint positions are zero in that figure.

The mechanical limits for the first three robot joints are as follows:

$-140° \leq \theta_1 \leq 140°$,
$-145° \leq \theta_2 \leq 145°$,
$-102 \text{ mm} \leq d_3 \leq 0 \text{ mm}$.

Joint 4 has no mechanical limits, but its software limits are ±10 turns. Finally, we define the integer c_t as the joint 4 *turn configuration parameter* (page 341), so that

$-180° + c_t 360° < \theta_4 \leq 180° + c_t 360°$.

Joints can be further constrained using the *SetJointLimits* (page 190) command (or via the MecaPortal).

Joint set and robot posture

There are several possible solutions for joint positions, for a desired pose of the robot end-effector with respect to the robot base, i.e., several possible sets $\{\theta_1, \theta_2, d_3, \theta_4\}$. The simplest way to describe how the robot is postured, is by giving its set of joint positions. This set will be referred to as the *joint set* (page 340).

A joint set completely defines the relative poses of each pair of adjacent links, i.e., the *robot posture* (page 341). However, the joint sets $\{\theta_1, \theta_2, d_3, \theta_4\}$ and $\{\theta_1, \theta_2, d_3, \theta_4 + c_t\ 360°\}$, where $-180° < \theta_4 \leq 180°$ and c_t is the turn configuration for joint 4, correspond to the same robot posture.

Therefore, *a joint set conveys the same information as a robot posture AND the turn configuration of the last joint.*

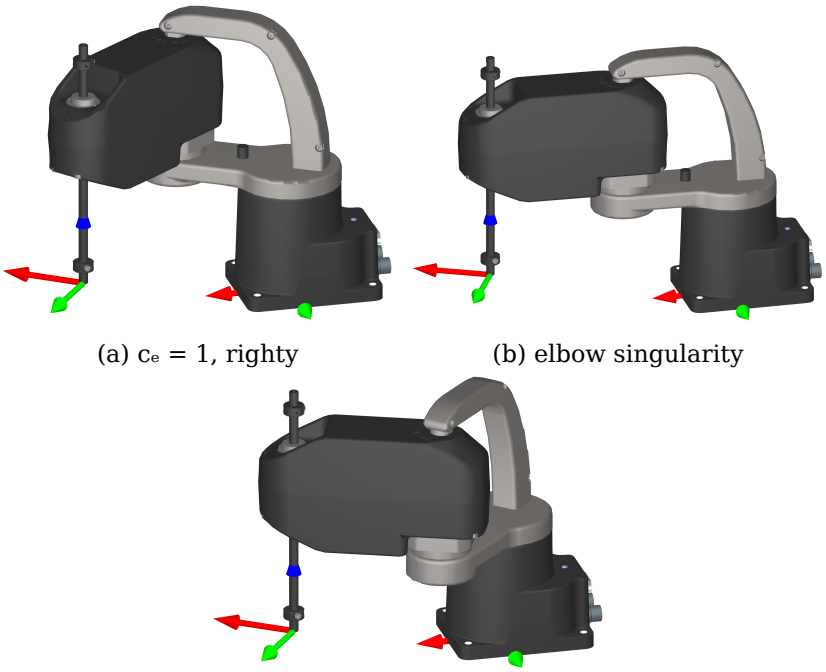
Configurations, singularities and workspace

Inverse kinematic solutions and configuration parameters

The *inverse kinematics* (page 339) is the problem of obtaining the robot joint sets that correspond to a desired end-effector pose.

The inverse kinematics of our SCARA robots provide two feasible robot postures for a desired pose of the TRF with respect to the WRF, as shown in Figure 3, and many more joint sets (since if θ_4 is a solution, then $\theta_4 \pm n360°$, where n is an integer, is also a solution). These two solutions are distinguished with a binary parameters called the *robot posture configuration parameter* (page 340): c_e:

- c_e (elbow configuration parameter)
  - $c_e = 1$, if $\theta_3 > 0°$. The condition $c_e = 1$ is also referred to as "righty" (see Figure 3a);
  - $c_e = -1$, if $\theta_3 < 0°$. The condition $c_e = -1$ is also referred to as "lefty" (see Figure 3b);



(a) $c_e = 1$, righty (b) elbow singularity

(c) $c_e = -1$, lefty

Figure 3: Posture configuration parameter and the singularity in the MCS500

The robot calculates the solution to the inverse kinematics that corresponds to the desired posture configuration, c_e, defined by the *SetConf* (page 147) command. In addition, it solves θ_4 by choosing the angle that corresponds to the desired turn configuration, c_t, defined by the *SetConfTurn* (page 148) command. The turn is therefore the last inverse kinematics configuration parameter.

Both the turn configuration and the robot posture configuration parameter are needed

to pinpoint the solution to the robot inverse kinematics (i.e., to pinpoint the joint set corresponding to the desired pose). However, there are major differences between the turn and robot posture configuration parameters; mainly that the change of turn does not involve singularities. This is why different commands are used (*SetConf* (page 147) and *SetConfTurn* (page 148), *SetAutoConf* (page 139) and *SetAutoConfTurn* (page 140), etc.).

Although it is possible to calculate the optimal inverse kinematic solution (the shortest move from the current robot position) using the commands *SetAutoConf* (page 139) and *SetAutoConfTurn* (page 140), we strongly recommend always specifying the desired values for the configuration parameters with *SetConf* (page 147) and *SetConfTurn* (page 148). This should be done for every Cartesian motion command (e.g., *MovePose* (page 138) and the various MoveLin* commands) when programming your robot in *online mode programming* (page 340).

If you are teaching the *robot position* (page 341) and later want the end-effector to move to the current pose along a linear path, you must record not only the current pose of the TRF relative to the WRF (using *GetRtCartPos* (page 259)), but also the definitions of both the TRF and the WRF (using *GetTrf* (page 251) and *GetWrf* (page 253)). Additionally, you need to capture the corresponding configuration parameters (using *GetRtConf* (page 261) and *GetRtConfTurn* (page 262)). Then, to ensure accurate execution of the command *MoveLin* (page 132) when approaching the previously recorded robot position from a starting position, you must verify that the robot is already in the same posture configuration and that θ_4 is within half a revolution of the desired value. If you do not require the robot's TCP to follow a linear trajectory, it is preferable to retrieve only the current joint set using *GetRtJointPos* (page 263). You can later move the robot to that joint set with the *MoveJoints* (page 126) command, eliminating the need to record or specify the four configuration parameters and the definitions of the TRF and WRF.

Automatic configuration selection

The automatic configuration selection should only be used once you understand how this selection is done, and mainly while programming and testing. For example, when jogging in Cartesian space with the MecaPortal, the automatic configuration selection is always enabled. Or, if a target pose is identified in real-time based on input from a sensor (e.g., a camera), enabling the automatic configuration selection will increase your chances of reaching that pose, and in the fastest way.

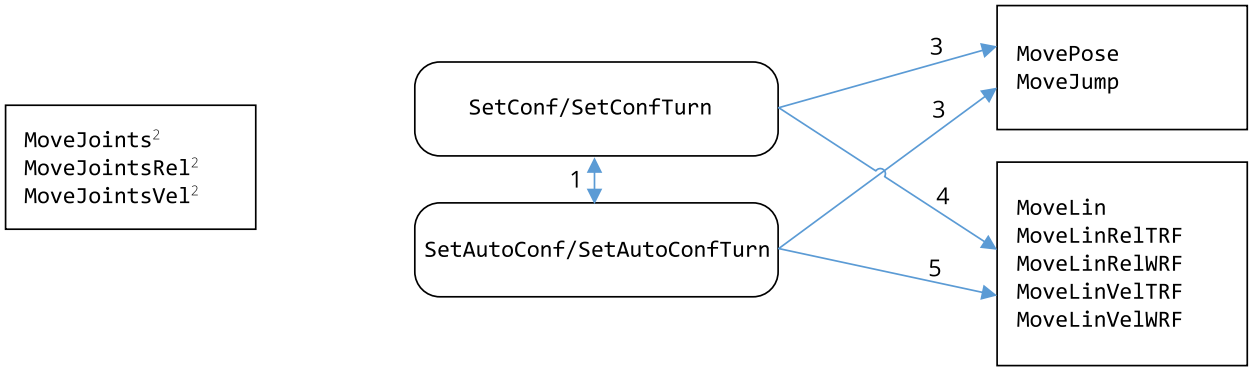


Figure 4: Effect of configuration parameters on robot movement commands

Figure 4 illustrates how the automatic and manual configuration selections work, with the following five remarks:

1. Setting a desired posture or turn configuration (with *SetConf* (page 147) or *SetConfTurn* (page 148), respectively) disables the automatic posture or turn configuration selection, respectively, which are both set by default. Inversely, enabling the automatic posture or turn configuration selection, with *SetAutoConf(1)* (page 139) or *SetAutoConfTurn(1)* (page 140), respectively, removes the desired posture or turn configuration, respectively. At any moment, if *SetAutoConf(0)* (page 139) or *SetAutoConfTurn(0)* (page 140) is executed, the robot posture or turn configuration of the current robot position is set as the desired posture or turn configuration, respectively.
2. The commands *MoveJoints* (page 126), *MoveJointsRel* (page 128), and *MoveJointsVel* (page 129) ignore the automatic and manual configuration selections. Thus, the robot may end up in a posture or turn configuration different from the desired ones, if such were set. If you want to update the desired configurations with the current ones, simply execute the commands *SetAutoConf(0)* (page 139) or *SetAutoConfTurn(0)* (page 140).
3. The command *MovePose* (page 138) respects any desired posture or turn configuration, as long as the desired robot position is reachable. In contrast, if automatic posture and/or turn configuration selection is enabled, *MovePose* (page 138) will choose the joint set corresponding to the desired end-effector pose, that is fastest to reach and that satisfies the desired posture or turn configuration, if any.
4. In the case of MoveLin* commands, the desired posture and turn configurations will force the linear move to remain within the specified posture and turn configurations. This means that a *MoveLin* (page 132) or MoveLinRel* command will be executed only if the posture and turn configurations of the initial and final robot positions are the same as the desired configurations. In the case of MoveLinVel*, the robot will start to move only if the posture and turn configurations of the initial and final robot positions are the same as the desired configurations, and will stop if desired configuration parameter has to change.
5. When automatic configuration selection is enabled, a MoveLin* command may lead

to changing the posture (if passing through a wrist or shoulder singularity) or turn configuration along the path.

Workspace and singularities

The workspace of our MCS500, or more specifically the set of possible positions for the origin of its FRF, is essentially the set of two zones of height $d_{3,max}$ (i.e., 102 mm), one for the righty ($c_e > 0$) configuration and one of the lefty ($c_e < 0$) configuration, as shown in Figure 5. In Cartesian mode, you can move only in one of these two zones. These overlapping zones are "separated" by the elbow singularity (the thick circular arc in Figure 5).

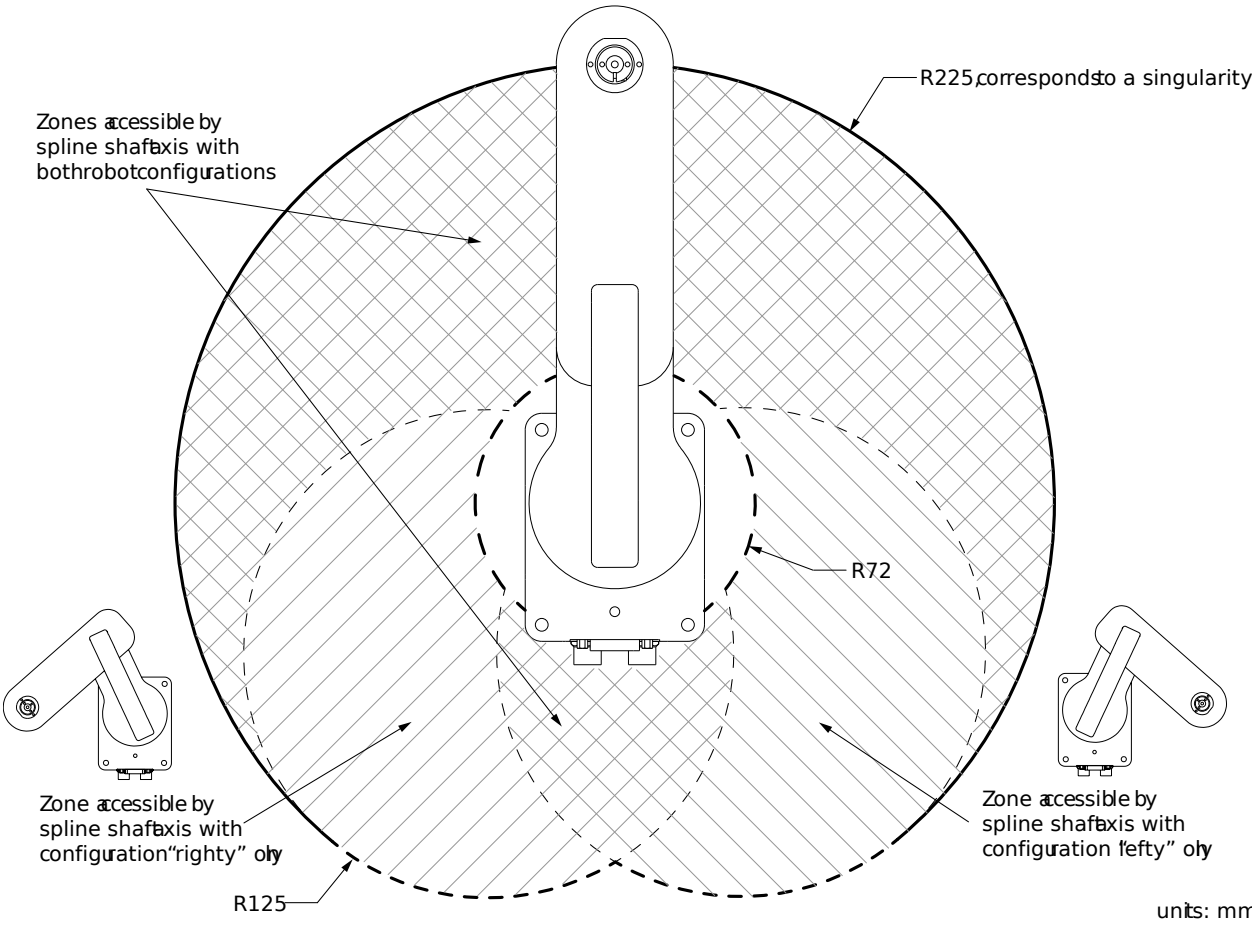


Figure 5: The working range of the MCS500 robot

Crossing singularities with linear Cartesian-space movements

For convenience, our SCARA robots allow you to move the robot in Cartesian mode even from a singular configuration (i.e., one in which $\theta_3 = 0°$) or end up in a singular configuration. This feature is mainly usefull when jogging the robot.

Key concepts for Mecademic robots

Homing

In the case of the MCS500, homing is not needed as the robot is equipped with high-accuracy absolute encoders. However, you should never manually rotate joint 4 beyond its software-defined limits.

Note

The range of the absolute encoder and of the software limits of joint 4 of the MCS500 is only ±3,600°. *Do not manually rotate joint 4 beyond its software limits (e.g., by disabling the brakes).*

Recovery mode

Once activated, if the robot is outside the user-defined joint limits (*SetJointLimits* (page 190)), if the work zone has been breached (*SetWorkZoneLimits* (page 290), *SetWorkZoneCfg* (page 289)), if a collision has been detected (*SetCollisionCfg* (page 287)), or if the robot is too close to an obstacle, it may be necessary to move the robot to a secure position even if these conditions prevent activating the robot or resuming motion. We have implemented the recovery mode (see the command *SetRecoveryMode* (page 198)) for these situations. In this mode, virtually all motion commands are permitted without restrictions such as software joint limits, work zone boundaries, self-collision avoidance, or torque overload protection, as long as the robot is activated. However, *joint speeds are significantly reduced*.

Enabling the recovery mode will significantly limit the joint and Cartesian velocities and accelerations, for safety reasons.

Blending

Industrial robots are most often controlled in *position mode* (page 340), using two groups of commands:

- With *Cartesian-space* (page 339) commands (*MoveLin* (page 132), *MoveLinRelWrf* (page 135), *MoveLinRelTrf* (page 134)), the robot is instructed to move its end-effector towards a target pose along a specified Cartesian path. To follow a complex Cartesian path, it must be broken down into small linear segments and executed using multiple Cartesian-space commands. Recall that singularities can pose challenges when using Cartesian-space commands.
- with *joint-space* (page 340) commands (*MoveJoints* (page 126), *MoveJointsRel* (page 128), *MovePose* (page 138), *MoveJump* (page 130)), the robot is instructed — directly or indirectly — to move its joints linearly towards a target joint set. Recall that when using joint-space commands, singularities are generally not an issue (except possibly for the *MovePose* (page 138) command).

Often, the target poses or joint sets act as “via points,” where the goal is not to reach the target precisely but simply to pass near it. Blending enables the robot to transition smoothly between motion segments instead of stopping at the end of each segment and making sharp changes in direction. Blending can be thought of as taking a rounded shortcut.

Blending allows the trajectory planner to maintain the end-effector’s acceleration to a minimum between two position-mode joint-space movements or two position-mode Cartesian-space movements. When blending is activated, the trajectory planner will transition between the two paths using a blended curve (Figure 6). *The higher the TCP speed, the more rounded the transition will be* (the radius of the blending cannot be explicitly controlled, only the blending duration is configurable).

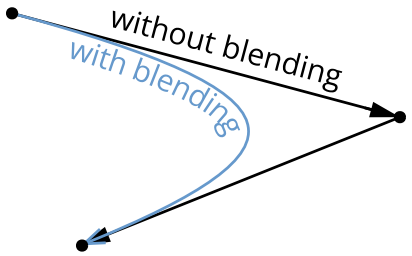


Figure 6: TCP path for two consecutive linear movements, with and without blending

Even if blending is enabled, the robot will come to a full stop after a joint-space movement that is followed by a Cartesian-space movement, or vice-versa. When blending is disabled, each motion will begin from a full stop and end in a full stop. Blending is enabled by default. It can be completely disabled or only partially enabled with the *SetBlending* (page 141) command.

Position and velocity modes

As mentioned in the previous section, the conventional method for moving an industrial robot involves either commanding its end-effector to reach a desired pose along a specified Cartesian path or directing its joints to rotate to a desired joint set. This basic control method is called *position mode* (page 340). If the robot needs to follow a linear path, the Cartesian-space motion commands *MoveLin* (page 132), *MoveLinRelTrf* (page 134), and *MoveLinRelWrf* (page 135) should be used. To move the robot's end-effector to a specific pose (without concern for the path followed by the end-effector) or to rotate the robot's joints to a given joint set or by a specified amount, the joint-space motion commands *MovePose* (page 138), *MoveJoints* (page 126), or *MoveJointsRel* (page 128) should be used, respectively.

In position mode, with Cartesian-space motion commands, it is possible to specify the maximum linear and angular velocities, and the maximum accelerations for the end-effector. Alternatively, you can specify the time duration of your movement. However, you cannot set a limit on the joint accelerations. Thus, if the robot executes a Cartesian-space motion command and passes very close to a singular robot posture, even if its end-effector speed and accelerations are very small, some joints may rotate at maximum speed (as defined by *SetJointVelLimit* (page 152)) and with maximum acceleration. Similarly, with joint-space motion commands, it is possible to specify the maximum velocity and acceleration of the joints or the time duration of the movement. However, it is impossible to limit either the velocity or the acceleration of the robot's end-effector. Figure 7 summarizes the possible settings for the velocity and acceleration in position mode.

As mentioned, in position mode, you can specify either the desired velocities (*SetJointVel* (page 151) or *SetCartLinVel* (page 144) and *SetCartAngVel* (page 143)) or the movement's time duration (*SetMoveDuration* (page 153)). This choice is made using the *SetMoveMode* (page 159) command. In *velocity-based position mode*, the robot attempts to follow the specified velocities without exceeding them while respecting acceleration limits. However, portions of the movement may not maintain the exact desired velocities, and the robot will NOT notify you of these deviations. In *time-based position mode*, you can use *SetMoveDurationCfg* (page 154) to define how the robot should respond if it cannot complete the movement within the specified duration.

There is a second method to control a Mecademic robot, by defining either its end-effector velocity or its joint velocities. This robot control method is called the *velocity mode* (page 341). Velocity mode is designed for advanced applications such as force control, dynamic path corrections, or telemanipulation (for example, the jogging feature in the MecaPortal is implemented using velocity-mode commands).

Controlling the robot in velocity mode requires one of the three velocity-mode motion commands: *MoveJointsVel* (page 129), *MoveLinVelTrf* (page 136) or *MoveLinVelWrf* (page 137). Note that the effect from a velocity-mode motion command lasts the time specified in the *SetVelTimeout* (page 167) command or until a new velocity-mode command is received. This timeout must be very small (the default value is 0.05 s, and the maximum value 1 s). For the robot to continue moving after this timeout, another velocity-mode command can be sent before this timeout. This new command will immediately replace the previous command and

restart the timeout. Position-mode and velocity-mode motion commands can be sent to the robot, in any order. However, if the robot is moving in velocity mode, the only commands that will be executed immediately, rather than after the velocity timeout, are other velocity-mode motion commands and *SetCheckpoint* (page 145), *GripperOpen* (page 310) and *GripperClose* (page 309) commands.

Note

There is a significant difference in the behavior of position- and velocity-mode motion commands. In position mode, if a Cartesian-space motion command cannot be completely performed due to a singularity or a joint limit, the motion will normally not start and a motion error will be raised, that must be reset.

In velocity mode, if the robot runs into a singularity that cannot be crossed or a joint limit, it will simply stop without raising an error. Furthermore, the velocity of the robot's end-effector or of the robot joints is directly controlled, but is subject to the constraint set by the *SetJointVelLimit* (page 152) command. The *SetJointVelLimit* (page 152) command affects the position-mode commands too. See Figure 7 for a complete description of how velocity and acceleration settings affect the two modes.

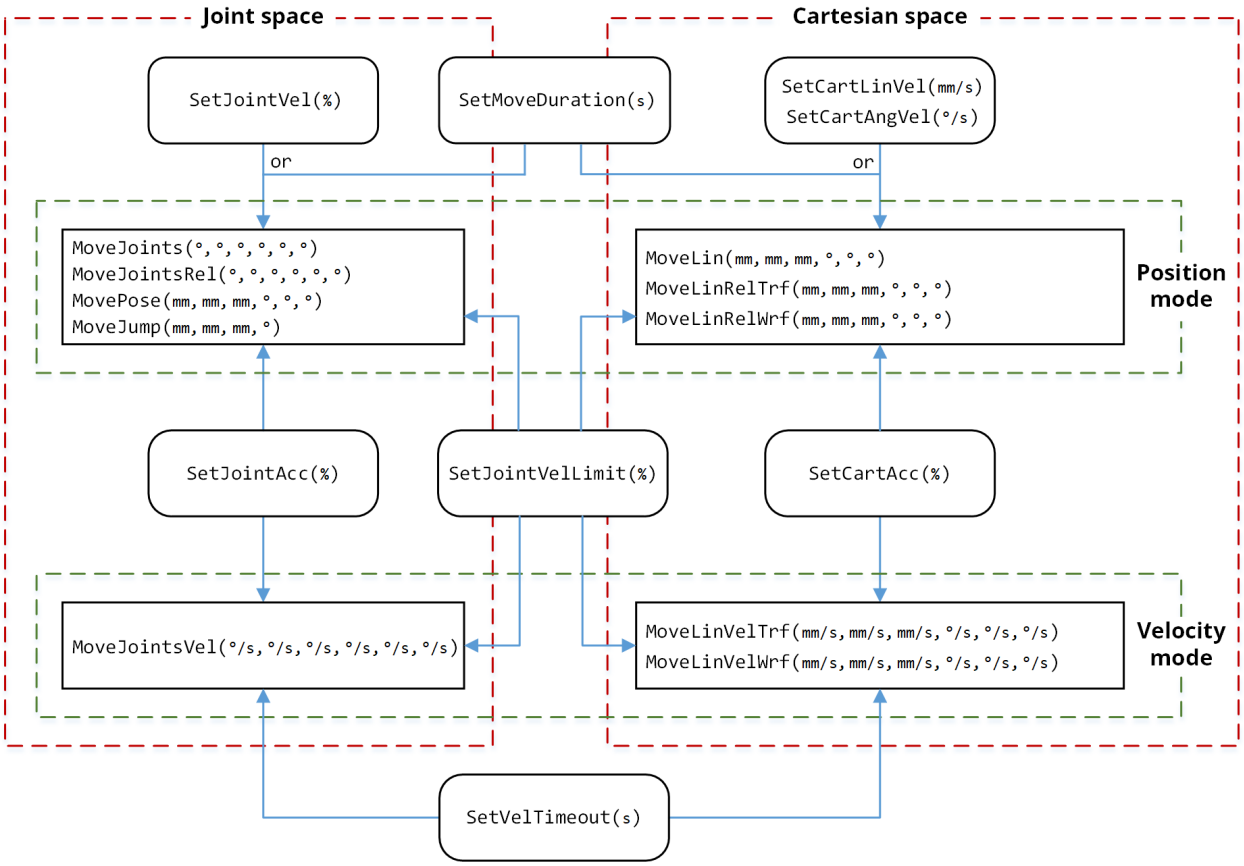


Figure 7: Settings that influence the robot motion in position and velocity modes. (Note: *MoveJump* (page 130) is always executed in velocity-based position mode)

Note

The instantaneous command *SetTimeScaling* (page 202) affects all velocities, accelerations and even time durations (including the timeout set with *SetVelTimeout* (page 167) and the pause set with the *Delay* (page 125) command).

TCP/IP communication

Mecademic robots must be connected to a computer or to a PLC over Ethernet. API commands may be sent through Mecademic's web interface, the MecaPortal, or through a custom computer program using either the TCP/IP protocol, as detailed in the remainder of this section, or any of three cyclic protocols, as detailed in the next three sections. When the robot communicates using the TCP/IP protocol, it uses null-terminated ASCII strings. *The default robot IP address is 192.168.0.100, and its default TCP port is 10000*, referred to as the *control port* (page 339). Commands to and messages from the robot are sent over the control port. The robot will periodically send data over *TCP port 10001*, referred to as the *monitoring port* (page 340), at the rate specified by the *SetMonitoringInterval* (page 192) command. This data includes the joint set and TRF pose (only when it changes), and other optional data enabled with the *SetRealTimeMonitoring* (page 196) command. To avoid desynchronization between the data received from both parts, it is possible to send a copy of the monitoring port data to the control port data with the *SetCtrlPortMonitoring* (page 187) command.

API commands are regrouped in the following categories, in terms of functionality:

- *motion commands* (page 340), which are the commands used to construct the robot trajectory (e.g., *Delay* (page 125), *MoveJoints* (page 126), *SetTRF* (page 166), *SetBlending* (page 141)),
- *robot control commands* (page 340), which are commands used to control the robot (e.g., *ActivateRobot* (page 171), *PauseMotion* (page 182), *SetNetworkOptions* (page 193)),
- *data request commands* (page 339), which are commands used to request some data regarding the robot (e.g., *GetTRF* (page 251), *GetBlending* (page 215), *GetJointVel* (page 229)),
- *real-time data request commands* (page 340), which are commands used to request some real-time data regarding the robot (e.g., *GetRtTrf* (page 273), *GetRtCartPos* (page 259), *GetStatusRobot* (page 277)),
- work zone supervision and collision detection commands, which are commands used to set a bounding box for the robot and its tooling and define collision conditions, and query these settings and related statuses,
- optional accessories commands, which are commands used to control or request data from the optional tools and modules for our robots (i.e.,

  the vacuum and I/O module).
- commands for managing variables, which allow the definition and management of persistent variables.

However, commands can also be categorized in terms of whether they are executed immediately or not. *Queued commands* (page 340) are placed in a *motion queue* (page 340), once received by the robot, and are executed on a FIFO basis. All motion commands and

some external tool commands are queued. *Instantaneous commands* (page 339) are executed immediately, as soon as received by the robot. All data request commands (Get*), all robot control commands, all Work zone supervision and collision prevention commands and some optional accessories (*_Immediate) are instantaneous.

Finally, some command descriptions refer to *default values* (page 339): these are essentially variables that are initialized every time the robot boots. Of these, those that correspond to motion commands are also initialized every time the robot is deactivated (e.g., after an emergency stop). In contrast, certain parameter values are *persistent* (page 340): they have manufacturer's default values, but the changes you make to these are written on an SD drive and persist even if you power off the robot.

Note

For convenience, since TCP API commands used in the TCP/IP protocol form the backbone of other communication protocols, they are presented in a separate part of this manual.

Responses and messages

Every Mecademic robot sends responses and messages over its control port in various situations: when it encounters an error, receives a request or certain motion commands, or experiences a status change. Additionally, the robot periodically or occasionally sends similar responses and messages, along with other information, on its monitoring port.

All responses and messages from the robot are formatted as ASCII strings in the following structure:

[4-digit code][text message OR comma-separated return values]

The second part of a response or message consists of either a descriptive text or a set of comma-separated return values. Descriptive text is intended to provide information to the user and is subject to change without prior notice. For example, the description “Homing failed” may later be updated to “Homing has failed.” Therefore, you should rely solely on the four-digit code when processing messages.

Any changes to these codes or the format of comma-separated return values will always be documented in the firmware upgrade manual. Return values are provided as either integers or IEEE-754 floating-point numbers with up to nine decimal places.

Error messages

When the robot encounters an error while executing a command, it goes into error mode. See Section 4 for details on how to manage these errors. The following table lists all command error messages. *These messages are sent on the control port.*

Table 1: Error messages; sent on the control port only

Message	Explanation
[1000][Command buffer is full.]	Maximum number of queued commands reached. Retry by sending commands at a slower rate.
[1001][Empty command or command unrecognized. - Command: ‘…’]	Unknown or empty command.
[1002][Syntax error, symbol missing. - Command: ‘…’]	A parenthesis or a comma has been omitted.
[1003][Argument error. - Command: ‘…’]	Wrong number of arguments or invalid input (e.g., the argument is out of range).
[1005][The robot is not activated.]	The robot must be activated, before executing the command that caused this error.

continues on next page

Table 1 – continued from previous page

Message	Explanation
[1006][The robot is not homed.]	The robot must be homed, before executing the command that cased this error.
[1007][Joint over limit (... is not in range [...,...] for joint ...). - Command: '...'.]	The robot cannot execute the *MoveJoints* (page 126) or *MoveJointsRel* (page 128) command because at least one joint is either currently outside or will move beyond the user-defined limits.
[1010][Linear move is blocked because a joint would rotate by more than 180deg. - Command: '...']	The linear motion cannot be executed because it requires a reorientation of 180° of the end-effector, and there may be two possible paths.
[1011][The robot is in error.]	A command has been sent but the robot is in error mode and cannot process it until a *ResetError* (page 184) command is sent.
[1012][Linear move is blocked because it requires a reorientation of 180 degrees of the end- effector - Command: '...'.]	The *MoveLin* (page 132) or *MoveLinRel** command sent requires that the robot pass through a singularity that cannot be crossed or pass too close to a singularity with excessive joint rotations.
[1013][Activation failed.]	Activation failed (for example, because the SWStop is active).
[1014][Homing failed.]	Homing procedure failed. Try again.
[1016][Destination pose out of reach for any configuration. - Command: '...'] [1016][Destination pose out of reach for selected conf(...,...,... turn ...). - Command: '...'] [1016][The requested linear move is not possible due to a pose out of reach along the path. - Command: '...']	The pose requested in the *MoveLin* (page 132), MoveLinRel*, *MovePose* (page 138) or *MoveJump* (page 130) command is out of reach, with the desired (or with any) configurations. In the case of the *MoveLin* (page 132) command, this error code is also produced if a pose along the path is out of reach.
[1022][Robot was not saving the program.]	The *StopSaving* (page 206) command was sent, but the robot was not saving a program.
[1023][Ignoring command for offline mode. - Command: '...']	The command cannot be executed in the offline program.
[1024][Mastering needed. - Command: '...']	Mastering was lost. Contact Mecademic.
[1025][Impossible to reset the error. Please, power-cycle the robot.]	Deactivate and reactivate the robot, in order to reset the error.

continues on next page

Table 1 – continued from previous page

Message	Explanation
[1026][Deactivation needed to execute the command. - Command: '...']	The robot must be deactivated in order to execute this command.
[1027][Simulation mode can only be enabled/ disabled while the robot is deactivated.]	The robot must be deactivated in order to execute this command.
[1029][Offline program full. Maximum program size is 13,000 commands. Saving stopped.]	The program saving was interrupted because the limit of 13,000 commands was reached.
[1030][Already saving.]	The robot is already saving a program. Wait until finished to save another program.
[1031][Program saving aborted after receiving illegal command. - Command: '...']	The command cannot be executed because the robot is currently saving a program.
[1033][Start conf mismatch]	Requested move blocked because start robot position is not in the requested configuration.
[1038][No gripper connected.]	Available only on the Meca500.
[1040][Command failed.]	General error for various commands.
[1041][No Vbox]	Available only on the Meca500
[1042][Ext tool sim must deactivated]	Switching external tool type is only possible when the robot is deactivated.
[1043][The specified IO bank is not present on this robot]	The argument for the I/O bank ID is different than 1.`
[1044][There is no vacuum module present on this robot.]	No MVK01 vacuum and I/O module present or simulated, but a command such as *VacuumGrip* (page 324) was sent.
[1550][...]	Variables could not be listed with *ListVariables* (page 337) for the reason specified in the error message.
[1551][...]	The variable could not be retrieved with *GetVariable* (page 336) for the reason specified in the error message.
[1552][...]	The variable creation failed with *CreateVariable* (page 334) for the reason specified in the error message.
[1553][...]	The variable deletion failed with *DeleteVariable* (page 335) for the reason specified in the error message.
[1554][...]	The variable modification failed with *SetVariable* (page 337) for the reason specified in the error message.

continues on next page

Table 1 – continued from previous page

Message	Explanation
[3001][Another user is already connected, closing connection.]	Another user is already connected to the robot. The robot disconnects from the user immediately after sending this message.
[3002][A firmware upgrade is in progress (connection refused).]	The firmware of the robot is being updated.
[3003][Command has reached the maximum length.]	Too many characters before the NULL character. Possibly caused by a missing NULL character
[3005][Error of motion.]	Motion error. Possibly caused by a collision or overload. Correct the situation and send the *ResetError* (page 184) command. If the motion error persists, try power-cycling the robot.
[3006][Error of communication with drives]	This error cannot be reset. The robot needs to be rebooted to recover from this error.
[3009][Robot initialization failed due to an internal error. Restart the robot.]	Error in robot startup procedure. Contact our technical support team if restarting the robot did not resolve the issue.
[3014][Problem with saved program, save a new program.]	There was a problem saving the program.
[3017][No offline program saved.]	There is no program in memory.
[3020][Offline program ... is invalid]	There was a problem starting a particular program with *StartProgram* (page 203).
[3025][Gripper error.]	This command is available only on the Meca500.
[3026][Robot's maintenance check has discovered a problem. Mecademic cannot guarantee correct movements. Please contact Mecademic.]	A hardware problem was detected. Contact our technical support team.
[3027][Internal error occurred.]	In case of internal, software error.
[3029][Excessive torque error occurred]	Excessive motor torque was detected.
[3031][A previously received text API command was incorrect.]	When using EtherNet/IP, this code (received in the input tag assembly only) indicates that the last command sent by TCP/IP was invalid.
[3037][Pneumatic module error]	A communication error with the pneumatic module was detected. Contact our technical support team.
[3039][External tool firmware must be updated.]	Activation has failed, because the robot has detected that the firmware of the EOAT is older than the firmware of the robot.

continues on next page

Table 1 – continued from previous page

Message	Explanation
[3041][Robot error due to imminent collision.]	Sent when robot is in error due to imminent collision detected while severity is configured to generate an error.
[3042][Detected failure in previous firmware update. Please re-install the firmware again.]	An error was detected during the firmware update. Try to reinstall software.
[3043][Excessive communication errors with external tool.]	Too many communication errors were detected between the I/O port and the EOAT connected to that port. This may mean that the cable is damaged and needs to be replaced or that it is not screwed tightly enough on either side. There may also be a hardware problem with the I/O port.
[3044][Abnormal communication error with external port.]	Detected internal communication errors with the robot's I/O port. Please contact Mecademic support for further diagnostic.
[3045][Imminent collision detected, robot will decelerate now.]	Sent when the robot is in error due to the detection of an imminent collision while severity is configured to Pause or Clear Motion.
[3046][Power-supply detected a non-resettable power error. Please check power connection then power-cycle the robot]	Try to power-cycle the robot.
[3047][Robot failed to mount drive. Please try to power-cycle the robot. If the problem persists contact Mecademic support.]	Robot has unexpectedly booted in safe mode. Try to power-cycle the robot.
[3049][Robot error at work zone limit]	Sent when robot is in error due to imminent work zone breach while severity is configured to generate an error.
[3050][A power lost error was detected, robot is going to shutdown]	Power has been lost and the robot is shutting down.

Command responses

The following provides a summary of all possible non-error responses to commands *sent via the control port*. Some of these responses are also transmitted on the monitoring port, as discussed in the next section. Note that motion commands do not generate any non-error responses, except for the optional EOB and EOM messages and any messages generated by the *SetCheckpoint* (page 145) command.

Table 2: Possible responses to commands, sent on the control port

Response code	Command
[2000][Motors activated.]	*ActivateRobot* (page 171)
[2002][Homing done.]	*Home* (page 179)
[2004][Motors deactivated.]	*DeactivateRobot* (page 175)
[2005][The error was reset.] [2006][There was no error to reset.]	*ResetError* (page 184)
[2007][as, hs, sm, es, pm, eob, eom]	*GetStatusRobot* (page 277)
[2013][x, y, z , ɣ]	*GetWrf* (page 253)
[2014][x, y, z , ɣ]	*GetTrf* (page 251)
[2015][p]	*SetTimeScaling* (page 202), *GetTimeScaling* (page 248)
[2028][e]	*GetAutoConf* (page 213)
[2029][c_e]	*GetConf* (page 222)
[2031][e]	*GetAutoConfTurn* (page 214)
[2036][c_t]	*GetConfTurn* (page 223)
[2042][Motion paused.]	*PauseMotion* (page 182)
[2043][Motion resumed.]	*ResumeMotion* (page 185)
[2044][The motion was cleared.]	*ClearMotion* (page 173)
[2045][The simulation mode is enabled.]	*ActivateSim* (page 172)
[2046][The simulation mode is disabled.]	*DeactivateSim* (page 176)
[2047][External tool simulation mode has changed.]	*SetExtToolSim* (page 312)
[2049][Robot is in recovery mode] [2050][Robot is not in recovery mode]	*SetRecoveryMode* (page 198)
[2051][Joint velocity/acceleration ... will be limited to ... due to recovery mode]	*MoveJointsVel* (page 129), *MoveLinVelTrf* (page 136), *SetCartAcc* (page 142), *MoveLinVelWrf* (page 137), *SetCartAngVel* (page 143), *SetCartAcc* (page 142), *SetJointAcc* (page 150), *SetJointVel* (page 151)
[2052][End of movement is enabled.] [2053][End of movement is disabled.]	*SetEom* (page 189)

continues on next page

Table 2 – continued from previous page

Response code	Command
	SetEob (page 188)
[2054][End of block is enabled.] [2055][End of block is disabled.]	
[2056][b_{id}, e]	*GetIoSim* (page 296)
[2056][b_{id}, e]	*SetIoSim* (page 316)
[2060][Start saving program.]	*StartSaving* (page 204)
[2061][n commands saved.]	*StopSaving* (page 206)
[2063][Offline program n started.]	*StartProgram* (page 203)
	StopSaving (page 206)
[2064][Offline program looping is enabled.] [2065][Offline program looping is disabled.]	
[2080][n]	*GetCmdPendingCount* (page 256)
[2081][vx.x.x]	*GetFwVersion* (page 225)
[2085][Command successful. ...]	Response to various instantaneous commands
[2088][vx.x.x]	*GetExtToolFwVersion* (page 292)
[2083][robot's serial number]	*GetRobotSerial* (page 246)
[2084][MCS500]	*GetProductType* (page 241)
[2086][vx.x.x]	*GetExtToolFwVersion* (page 292)
[2090][n, $\theta_{n,min}$, $\theta_{n,max}$]	*GetJointLimits* (page 227)
[2092][n]	*SetJointLimits* (page 190)
	SetJointLimitsCfg (page 191)
[2093][User-defined joint limits enabled.] [2093][User-defined joint limits disabled.]	
[2094][e]	*GetJointLimitsCfg* (page 228)
[2095][s]	*GetRobotName* (page 245)
[2096][Monitoring on control port enabled/disabled]	*SetCtrlPortMonitoring* (page 187)
[2097][n]	*SyncCmdQueue* (page 208)
[2113][q_1, q_2, q_3...]	*GetModelJointLimits* (page 231)
[2116][t]	*GetMonitoringInterval* (page 232)
[2117][n_1, n_2, ...]	*GetRealTimeMonitoring* (page 243), *SetRealTimeMonitoring* (page 196)
[2119][n_1, n_2, n_3, n_4, n_5, n_6]	*GetNetworkOptions* (page 238)
[2140][t]	*GetRtc* (page 275)

continues on next page

Table 2 – continued from previous page

Response code	Command
[2149][n]	*GetCheckpointDiscarded* (page 221)
[2150][p]	*GetBlending* (page 215)
[2151][t]	*GetVelTimeout* (page 252)
[2152][p]	*GetJointVel* (page 229)
[2153][p]	*GetJointAcc* (page 226)
[2154][v]	*GetCartLinVel* (page 219)
[2155][ω]	*GetCartAngVel* (page 218)
[2156][n]	*GetCartAcc* (page 217)
[2157][n]	*GetCheckpoint* (page 220)
[2158][p]	*GetGripperForce* (page 293)
[2159][p]	*GetGripperVel* (page 295)
[2160][l, m]	*GetTorqueLimitsCfg* (page 250)
[2161][τ_1, τ_2, τ_3, τ_4]	*GetTorqueLimits* (page 249)
[2162][d_{closed}, d_{open}]	*GetGripperForce* (page 293)
[2163][l, m]	*GetWorkZoneCfg* (page 284)
[2164][Workspace configuration set successfully.]	*SetWorkZoneCfg* (page 289)
[2165][x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]	*GetWorkZoneLimits* (page 285)
[2166] [Workspace limits set successfully.]	*SetWorkZoneLimits* (page 290)
[2167][x, y, z, r]	*GetToolSphere* (page 283)
[2168][Tool sphere set successfully.]	*SetToolSphere* (page 288)
[2169][p]	*GetJointVelLimit* (page 230)
[2172][p_h, p_r]	*GetVacuumThreshold* (page 308)
[2173][t_p]	*GetVacuumPurgeDuration* (page 307)
[2174][h_{start}, h_{end}, h_{min}, h_{max}]	*GetMoveJumpHeight* (page 236)
[2175][v_{start}, p_{start}, v_{end}, p_{end}]	*GetMoveJumpApproachVel* (page 235)
[2176][m]	*GetOperationMode* (page 257)
[2177][l]	*ConnectionWatchdog* (page 174)
[2178][PStop2 configuration set successfully]	*SetPStop2Cfg* (page 195)
[2179][l]	*GetPStop2Cfg* (page 239)
[2181][l]	*GetCollisionCfg* (page 281)
[2182][v, g_1, $o_{id,1}$, g_2, $o_{id,2}$]	*GetCollisionStatus* (page 282)
[2183][v, g, o_{id}]	*GetWorkZoneStatus* (page 286)
[2189][m]	*GetMoveMode* (page 237)
[2190][s]	*GetMoveDurationCfg* (page 234)
[2191][t]	*GetMoveDuration* (page 233)
[2192][t]	*GetPayload* (page 240)
[2200][t, θ_1, θ_2, d_3, θ_4]	*GetRtTargetJointPos* (page 270)
[2201][t, x, y, z, γ]	*GetRtTargetCartPos* (page 266)

continues on next page

Table 2 – continued from previous page

Response code	Command
[2202][t, ω_1, ω_2, v_3, ω_4]	*GetRtTargetJointVel* (page 272)
[2203][t, τ_1, τ_2, τ_3, τ_4]	*GetRtTargetJointTorq* (page 271)
[2204][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	*GetRtTargetCartVel* (page 267)
[2208][t, c_e]	*GetRtTargetConf* (page 268)
[2209][t, c_t]	*GetRtTargetConfTurn* (page 269)
[2210][t, θ_1, θ_2, d_3, θ_4]	*GetRtJointPos* (page 263)
[2211][t, x, y, z, γ]	*GetRtCartPos* (page 259)
[2212][t, ω_1, ω_2, v_3, ω_4]	*GetRtJointVel* (page 265)
[2213][t, τ_1, τ_2, τ_3, τ_4]	*GetRtJointTorq* (page 264)
[2214][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	*GetRtCartVel* (page 260)
[2218][t, c_e]	*GetRtConf* (page 261)
[2219][t, c_t]	*GetRtConfTurn* (page 262)
[2220][t, n, a_x, a_y, a_z]	*GetRtAccelerometer* (page 258)
[2228][t, x, y, z, γ]	*GetRtWrf* (page 274)
[2229][t, x, y, z, γ]	*GetRtTrf* (page 273)
[2300][t, simType, phyType, hs, es, oh]	*GetRtExtToolStatus* (page 297)
[2310][t, v_1, v_2]	*GetRtValveState* (page 306)
[2320][t, hp, dr, gc, go]	*GetRtGripperState* (page 300)
[2321][t, p]	*GetRtGripperForce* (page 298)
[2322][t, d]	*GetRtGripperPos* (page 299)
[2330][t, b_{id}, present, simMode, errorCode]	*GetRtIoStatus* (page 302)
[2340][t, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]	*GetRtOutputState* (page 303)
[2341][t, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]	*GetRtInputState* (page 301)
[2342][t, v, h, p]	*GetRtVacuumState* (page 305)
[2343][t, p]	*GetRtVacuumPressure* (page 304)
[3000][Connected to ... x_x_x.x.x.]	Confirms connection to robot.
[3000][Connected to ... x_x_x.x.x.]	Confirms connection to robot. Sent only at initial connection.
[3004][End of movement.]	The robot has stopped moving.
[3012][End of block.]	No motion command in queue and robot joints do not move.
[3013][End of offline program.]	The offline program has finished.
[3028][s]	A torque limit was exceeded.
[3030][n]	Checkpoint n was reached.
[3032][2/1/0]	A P-Stop 2 is active (1), is no longer active but needs to be reset (2) or is already cleared (0).
[3035][TCP dump capture started for x seconds]	Sent to indicate that the requested TCP dump capture has started and confirms the maximum duration of x seconds.

continues on next page

Table 2 – continued from previous page

Response code	Command
[3036][TCP dump capture stopped]	Sent after a previously started TCP dump capture has finished.
[3040][n]	Checkpoint n is discarded before reaching it, because motion was cleared.
[3051][Move duration too short: ... is too short. Fastest possible ... Command: ...]	Sent by the robot if in time-based move mode, requested duration impossible to meet, and severity was set to 4 with *SetMoveDurationCfg* (page 154).
[3069][0/1/2]	Response to *GetSafetyStopStatus(3069)* (page 276).
[3070][0/1/2]	Response to *GetSafetyStopStatus(3070)* (page 276).
[3080][0/1/2]	Response to *GetSafetyStopStatus(3080)* (page 276).
[3081][0/1/2]	Response to *GetSafetyStopStatus(3081)* (page 276).
[3082][0/1/2]	Response to *GetSafetyStopStatus(3082)* (page 276).
[3083][0/2]	Response to *GetSafetyStopStatus(3083)* (page 276).
[3084][0/1]	Response to *GetSafetyStopStatus(3084)* (page 276).
[3085][0/2]	Response to *GetSafetyStopStatus(3085)* (page 276).
[3086][0/1/2]	Response to *GetSafetyStopStatus(3086)* (page 276).
[3087][0/1/2]	Response to *GetSafetyStopStatus(3087)* (page 276).

Monitoring port messages

Mecademic robots are configured to send immediate feedback over TCP port 10001, also known as the monitoring port. Several types of messages are transmitted via this port, as shown in Table 3.

Most messages related to state changes are sent either as soon as the state changes or upon establishing a connection. These messages are marked with a tick in the “chg” column of Table 3.

Some messages are sent periodically (every 15 ms or as defined by the *SetMonitoringInterval* (page 192) command) and are marked with a tick in the “prd” column of Table 3.

Other messages, which are neither state-related nor periodic, are sent as appropriate when specific conditions occur on the robot. Some of these messages are also sent upon connection.

Additionally, note that some messages are optional and will not be sent by default unless explicitly enabled using the *SetRealTimeMonitoring* (page 196) command. Optional messages are marked with a tick in the “opt” column of Table 3.

Table 3: Monitoring port messages

chg	opt	prd	Message	Description
✓			[2007][as, hs, sm, es, pm, eob, eom]	Same as response of *GetStatusRobot* (page 277)
✓			[2015][p]	Same as response of *GetTimeScaling* (page 248)
			[2026][θ_1, θ_2, d_3, θ_4]	The same as the response of the legacy command GetPose
			[2027][x, y, z , γ]	The same as the response of the legacy command GetJoints
			[2044][The motion was cleared.]	Same as response of *ClearMotion* (page 173)
✓			[2049][Robot is in recovery mode]	Same as response of *SetRecoveryMode* (page 198)
✓			[2050][Robot is not in recovery mode]	Same as response of *SetRecoveryMode* (page 198)
			[2051][Joint velocity/acceleration ... will be limited to ... due to recovery mode]	Response when requested velocity/acceleration is limited due to recovery mode
✓			[2079][ge, hs, hp, lr, es, oh]	Same as response to legacy GetStatusGripper
✓			[2082][vx.x.x.xxxxx]	Complete firmware version of robot
✓			[2086][vx.x.x]	External firmware version
✓			[2095][s]	Same as response of *GetRobotName* (page 245)

continues on next page

Table 3 – continued from previous page

chg	opt	prd	Message	Description
✓			[2163][l, m]	Same as response of *GetWorkZoneCfg* (page 284)
✓			[2165][x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]	Same as response of *GetWorkZoneLimits* (page 285)
✓			[2167][x, y, z, r]	Same as response of *GetToolSphere* (page 283)
✓			[2181][l]	Same as response of *GetCollisionCfg* (page 281)
✓			[2182][v, g_1, $o_{id,1}$, g_2, $o_{id,2}$]	Same as response of *GetCollisionStatus* (page 282)
✓			[2183][v, g, o_{id}]	Same as response of *GetWorkZoneStatus* (page 286)
✓			[2189][m]	Same as response of *GetMoveMode* (page 237)
	✓	✓	[2200][t, θ_1, θ_2, d_3 θ_4]	Same as response of *GetRtTargetJointPos* (page 270)
	✓	✓	[2201][t, x, y, z, γ]	Same as response of *GetRtTargetCartPos* (page 266)
	✓	✓	[2202][t, ω_1, ω_2, v_3, ω_4]	Same as response of *GetRtTargetJointVel* (page 272)
	✓	✓	[2203][t, τ_1, τ_2, τ_3, τ_4]	Same as response of *GetRtTargetJointTorq* (page 271)
	✓	✓	[2204][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	Same as response of *GetRtTargetCartVel* (page 267)
			[2208][t, c_e]	Same as response of *GetRtTargetConf* (page 268)
✓			[2209][t, c_t]	Same as response of *GetRtTargetConfTurn* (page 269)
	✓	✓	[2210][t, θ_1, θ_2, d_3, θ_4]	Same as response of *GetRtJointPos* (page 263)
	✓	✓	[2211][t, x, y, z, , γ]	Same as response of *GetRtCartPos* (page 259)
	✓	✓	[2212][t, ω_1, ω_2, v_3, ω_4]	Same as response of *GetRtJointVel* (page 265)
	✓	✓	[2213][t, τ_1, τ_2, τ_3, τ_4]	Same as response of *GetRtJointTorq* (page 264)
	✓	✓	[2214][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]	Same as response of *GetRtCartVel* (page 260)
✓	✓		[2218][t, c_e]	Same as response of *GetRtConf* (page 261)
✓	✓		[2219][t, c_t]	Same as response of *GetRtConfTurn* (page 262)

continues on next page

Table 3 – continued from previous page

chg	opt	prd	Message	Description
	✓	✓	[2220][t, n, a_x, a_y, a_z]	Same as response of *GetRtAccelerometer* (page 258)
✓			[2226][t,hardDecel,linDistance,	When starting or finishing hard deceleration
✓			[2227][t,n]	When latest checkpoint reached changes
✓		✓	[2228][t, x, y, z, γ]	Same as response of *GetRtWrf* (page 274)
✓		✓	[2229][t, x, y, z, γ]	Same as response of *GetRtTrf* (page 273)
		✓	[2230][t]	Same as response of *GetRtTrf* (page 273)
✓			[2300][t, simType, phyType, hs, es, oh]	Same as response of *GetRtExtToolStatus* (page 297)
✓			[2310][t, v_1, v_2]	Same as response of *GetRtValveState* (page 306)
✓			[2320][t, hp, dr, gc, go]	Same as response of *GetRtGripperState* (page 300)
	✓	✓	[2321][t,p]	Same as response of *GetRtGripperForce* (page 298)
	✓	✓	[2322][t, p]	Same as response of *GetRtGripperPos* (page 299)
✓			[2330][t, b_{id}, present, simMode, errorCode]	Same as response of *GetRtIoStatus* (page 302)
✓			[2340][t, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]	Same as response of *GetRtOutputState* (page 303)
✓			[2341][t, b_{id}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8]	Same as response of *GetRtInputState* (page 301)
✓			[2342][t, v, h, p]	Same as response of *GetRtVacuumState* (page 305)
	✓	✓	[2343][t, p]	Same as response of *GetRtVacuumPressure* (page 304)
✓			[3000][Connected to ... x_x_x.x.x.]	Confirms connection to robot
✓			[3028][s]	Torque limit exceeded status
✓			[3032][2/1/0]	Response to a change in the state of the P-Stop 2 safety stop signal.
✓			[3048][0/1, 0/1, ...]	Sent by the robot whenever a drive is near to or in overload. The first value corresponds to drive 1, and so on, and it is 1 if the drive is near or in overload, and zero otherwise.
✓			[3069][0/1/2]	Response to a change in the state of the P-Stop 1 safety stop signal
✓			[3070][0/1/2]	Response to a change in the state of the E-Stop safety stop signal

continues on next page

Table 3 – continued from previous page

chg	opt	prd	Message	Description
✓			[3080][0/1/2]	Response to a change in the state of the operation mode safety stop signal state
✓			[3081][0/1/2]	Response to a change in the state of the Enabling Device Released safety stop signal, when the robot is in manual mode
✓			[3082][0/1/2]	Occurs when supply voltage fluctuation is detected
✓			[3083][0/2]	Occurs after robot is rebooted, and after Reset button is pressed for the first time
✓			[3084][0/1]	Redundancy fault. Occurs if a safety signal mismatch is detected
✓			[3085][0/2]	Standstill fault. Occurs if robot moves while in pause mode
✓			[3086][0/1/2]	Response to a change in the connection drop safety signal change
✓			[3087][0/1/2]	Minor error (e.g., power supply reset button pressed too long). Motor voltage is removed and robot is deactivated.

Note that multiple ASCII messages are separated by a single null-character and that there are no blank spaces in any of these messages. Here is an example of messages sent over TCP port 10001 in one interval (for clarity, the null-characters have been replaced by line breaks):

[2026][-102.6011,-0.0000,-78.9239,-0.0000,15.7848,110.3150]
[2027][-3.7936,-16.9703,457.5125,26.3019,-5.6569,9.0367]
[2230][58675156984]

Management of errors and safety stops

Errors detected by the robot

The robot enters *error mode* (page 339) when it encounters an issue while executing a command (see Table 1) or due to a hardware problem (e.g., exceeding a torque limit). When this occurs, the robot sets the value of es (error state) to 1 in the response [2007][as, hs, sm, es, pm, eob, eom] of the *GetStatusRobot* (page 277).

This message can also be received over the monitoring port (see Section 4). Additionally, if you send other commands to the robot while it is in error mode, it will respond with the message [1011][The robot is in error.].

When the robot is in error mode, all pending motion commands are canceled (i.e., the motion queue is cleared). The robot stops and ignores subsequent commands but responds with the [1011][The robot is in error.] message until it receives a *ResetError* (page 184) command.

After the error is reset, the robot will execute all request commands and begin accumulating motion commands in its motion queue. However, these motion commands will only be executed once the *ResumeMotion* (page 185) command is received by the robot.

P-Stop 2 and SWStop

As soon as the externally wired P-Stop 2 is activated (see the robot's user manual), the robot motion is immediately decelerated to a stop, and the response [3032][1] is sent by the robot. The motors and the EOAT remain active (i.e., the brakes are not applied) but stay immobilized until the stop is reset.

If a motion command is sent to the robot while the stop signal is still active (and the robot is still activated and homed), the command will be ignored if the P-Stop 2 is configured in "Clear motion" mode (*SetPStop2Cfg* (page 195)). In this case, the message [3032][1] will be sent again by the robot. However, if the P-Stop 2 is configured in "Pause motion" mode, the commands will continue to be accepted (queued) even while the robot is in a P-Stop 2 state.

Once the stop signal is removed, the message [3032][2] is returned. The P-Stop 2 condition is now ready to be reset using *ResumeMotion* (page 185). The robot will then respond with the messages [2043][Motion resumed.] and [3032][0].

E-Stop and P-Stop 1

When the E-Stop is activated, the robot decelerates to a full stop, *power to the motors is cut*, the brakes are applied, and the robot is *deactivated*. The robot then sends the message [3070][1], along with the messages [2044][The motion was cleared.] and [2004][Motors deactivated.].

To reactivate the motors, you must first clear the E-Stop condition, which will produce the message [3070][2]. Then, press the RESET button or activate the external Reset, which will produce the message [3070][0]. Afterward, you need to re-activate the robot with the *ActivateRobot* (page 171) command.

The robot can differentiate between the Emergency Stop function signal and the P-Stop 1. When the P-Stop 1 signal is activated and the robot is in automatic mode, the robot decelerates to a full stop, *power to the motors and the vacuum generator of the MVK01 is cut*, the brakes are applied, and the robot is *deactivated*. The robot then sends the message [3069][1], along with the messages [2044][The motion was cleared.] and [2004][Motors deactivated.].

To reactivate the motors (and the I/O and vacuum module), you must first clear the P-Stop 1 condition, which will produce the message [3069][2]. Then, press the RESET button or activate the external Reset, which will produce the message [3069][0]. Afterward, you need to re-activate the robot with the *ActivateRobot* (page 171) command.

Enabling device

When the operation mode switch is turned to the “Manual Mode” position, the robot monitors the position of the enabling device. Whenever the enabling device is not pressed halfway, the safety signal [3081][1] is sent, and the robot motion is paused.

To move the robot, you must press the enabling device halfway (or release and press it halfway again), which will produce the message [3081][2]. At this point, using the *ResumeMotion* (page 185) command is allowed, and it will produce the messages [3081][0] and [2043][Motion resumed.].

Operation mode switch

When the robot is powered, any change in the operation mode switch position causes the robot to decelerate to a full stop and removes power from the robot motors. If the switch is set to “Locked,” the message [3080][1] is sent. If the switch is set to “Automatic” or “Manual,” the message [3080][2] is sent. You must then press the Reset button on the power supply (or the external reset button), which will produce the message [3080][0].

To activate the robot after switching to “Manual” mode, in addition to pressing the Reset button, the enabling device must be pressed halfway (or released and pressed halfway again). Once this is done, the robot can be activated using the *ActivateRobot* (page 171) command.

Communication drop

For safety reasons, the robot continuously supervises the TCP connection. If the robot detects that the TCP connection has dropped while it is moving, it will immediately stop the motion and send the message [3081][1].

If the connection watchdog is enabled and the robot does not receive a *ConnectionWatchdog* (page 174) message before the established timeout, the robot will drop the TCP connection and raise the safety signal [3081][1], regardless of whether the robot is moving or not.

Once a new TCP/IP connection is established, the robot will send the message [3081][2]. Afterward, you must send the *ResumeMotion* (page 185) command, to which the robot will respond with the message [3081][0].

Supply voltage fluctuation

If the robot detects a short supply voltage fluctuation, it will generate the signal [3082][1], decelerate to a full stop, remove power from its motors, and then send the message [3082][2]. You must press the Reset button on the power supply (or the external reset button), which will cause the robot to send the message [3082][0].

If the supply voltage fluctuations are more significant or prolonged, the robot will shut itself off and attempt to reboot a few seconds later.

Robot reboot

After a reboot, the robot motors are not powered, and the message [3083][2] is sent. Once the Reset button is pressed, the message [3083][0] is sent, and the robot motors are powered.

Redundancy fault

If a redundant safety signal mismatch is detected for more than 1 second, the robot will immediately decelerate to a full stop, remove power from its motors, and send the message [3084][1].

Redundant safety signals include the E-Stop, the P-Stop 1, the P-Stop 2, and the three-position enabling device.

Standstill fault

If the robot is supposed to be in pause mode but moves, the message [3085][1] is sent, power to the motors is removed, and once the robot comes to a complete stop, the message [3085][2] is sent.

Once the Reset button is pressed, the message [3085][0] is sent.

Minor error

Certain minor errors unrelated to safety signals can result in the removal of power from the motors and the sending of the message [3087][1]. These include prolonged reset signals and some drive errors. The exact cause of these minor errors can be found in the robot logs.

Once the error source is resolved, the message [3087][2] is sent. Afterward, following a reset, the message [3087][0] is sent.

Communicating over cyclic protocols

Our robots can also be controlled using the following three cyclic protocols: EtherCAT, EtherNetIP, and PROFINET. These protocols are described in Section 6, Section 7, and Section 8, respectively. While inherently different, they share the same cyclic data format and API, so we will cover all common concepts in this section.

With cyclic protocols, the robot is controlled using cyclic data fields, which are detailed in this section. PLCs use these fields to activate, configure, move, and monitor the robot. The cyclic data payload format is identical across all supported protocols.

Some TCP/IP commands are not available when using cyclic protocols, such as the command *SetNetworkOptions* (page 193) for changing network settings, or the commands for creating, modifying, and deleting offline programs.

Types of cyclic protocol commands

Cyclic data output is used to control the robot's state, trigger actions, or send motion-related commands. Cyclic data input provides feedback from the robot such as status and joint positions.

Below, we briefly describe how our cyclic protocol API handles each type of action.

Status change commands

Some fields (bits) directly control the robot's state, such as:

- *PauseMotion*
- *ClearMotion*
- *SimMode*
- *RecoveryMode*
- *BrakesControl*

Set these bits to change the robot's state. The robot confirms completion via the corresponding status bit in the cyclic data input (Section 5, Section 5).

> **Note**
>
> Do not rely on cycle count or time delay to confirm a state change. Always check the corresponding confirmation bit (Section 5, Section 5) before assuming the robot has completed the state change.

Triggered actions

Some fields (bits) in the cyclic data directly trigger actions on the robot, such as:

- *Activate*
- *Deactivate*
- *Home*
- *ResetError*
- *ResumeMotion*

To trigger an action, set the corresponding bit to 1, and clear it (reset it to 0) only after the action is completed. Completion is confirmed by the corresponding status bit in the cyclic data input (Section 5, Section 5).

Note

Do not rely on cycle count or time delay to confirm action completion. Always check the corresponding confirmation bit (Section 5, Section 5) before assuming the robot has completed the action.

Motion-related commands

There are three types of motion-related commands that can be sent using cyclic protocols:

- Instantaneous commands (e.g., *SetWorkZoneLimits* (page 290)) that are executed immediately by the robot;
- Queued commands (e.g., *MoveJoints* (page 126)) that are queued and executed one after another;
- Velocity-mode commands (e.g., *MoveJointsVel* (page 129)) that are executed continuously until a new command is received (or until the configured velocity timeout is reached).

The next section provides a detailed explanation of how these motion-related commands are used.

Using motion-related commands

Motion-related commands are sent to the robot via:

- three cyclic data fields, *MotionCommandID*, *MoveID*, and *SetPoint* (Table 5 and Table 6), and
- six command arguments (Table 7).

MotionCommandID

Each motion-related command has a unique ID (Table 8). Entering this ID in the *MotionCommandID* field specifies which command to send.

Note

The field name *MotionCommandID* is retained for historical reasons, although it is also used to send commands other than motion commands, such as configuration commands (e.g., *SetWorkZoneLimits* (page 290)).

MoveID and SetPoint

The combination of *MoveID* and *SetPoint* fields is used to send cyclic or non-cyclic commands (Section 5):

- The *SetPoint* bit enables or disables command reception by the robot. When cleared, the robot ignores the *MotionCommandID* and *MoveID* fields;
- The *MoveID* field determines the command type: cyclic (*MoveID* is 0) or non-cyclic (*MoveID* is not 0, with a new command queued each time the *MoveID* value changes).

Warning

Ensure *SetPoint* is cleared (0) when connecting to the robot, or it may unexpectedly execute a command.

Sending non-cyclic commands

This section explains how to send *non-cyclic* commands (i.e., configuration or motion commands). For cyclic commands (i.e., velocity-mode commands), see Section 5.

Commands are sent by changing the *MoveID* field to a different non-zero integer (while *SetPoint* is set to 1).

- Configuration commands (e.g., *SetWorkZoneLimits* (page 290)) execute immediately. The robot confirms completion by updating its *MoveID* field (Section 5) to match yours;

- Motion commands (e.g., *MoveJoints* (page 126)) are added to the motion queue and processed sequentially. The robot confirms that the command was added to the queue (not yet executed) by updating its *MoveID* field (Section 5) to match yours.

The following sequence must be followed:

- At connection, clear both the *MoveID* and *SetPoint* fields;
- Then, to add a motion command to the robot's motion queue:
  - Set *MotionCommandID* to the desired command;
  - Enter the command arguments;
  - Change *MoveID* to a different non-zero integer value;
  - Set *SetPoint* to 1.
- To stop the robot immediately, set the *PauseMotion* or *ClearMotion* bit.

Warning

The *MoveID*, *MotionCommandID*, and command arguments (Section 5) must not be changed until the robot acknowledges the previous command by returning the corresponding *MoveID* (Section 5).

Warning

Change the *MoveID* only after (or in the same cycle as) *MotionCommandID* and arguments, or the robot may receive a mix of old and new *MotionCommandID* and arguments.

Sending cyclic commands

Cyclic commands, i.e., velocity-mode commands (*MoveJointsVel* (page 129), *MoveLinVelWrf* (page 137), and *MoveLinVelTrf* (page 136)), are executed continuously while *MoveID* is 0 and *SetPoint* is 1. The desired velocity can be changed at any time during this period.

The following sequence must be followed:

- At connection, clear both the *MoveID* and *SetPoint* fields;
- To start moving the robot:
  - Set *MotionCommandID* to the desired velocity-mode command ID;
  - Enter the six command arguments;
  - Set *SetPoint* to 1.
- To change the velocity, modify the six arguments. The robot will apply the new velocities on the next cycle;

- To stop the robot, reset *SetPoint* to 0, set all velocity arguments to zero, or set the *PauseMotion* or *ClearMotion* bit.

Warning

To change to a different velocity-mode command ID, ensure that you change *MotionCommandID* and all the arguments *in the same cycle* to prevent a command to be executed with the arguments that belongs to another command. Alternatively, you may change *SetPoint* to 0 before changing the command and arguments.

Warning

Using a position-mode command in cyclic mode (e.g., *MoveJoints* (page 126)), with *MoveID* set to 0 and *SetPoint* set to 1, will quickly fill the motion queue with copies of the same command every cycle, which is certainly not the desired result.

Cyclic data format

The robot cyclic data includes output fields for sending commands and actions to the robot, as well as input fields that report the complete robot status, position, and configuration.

Section 5 and Section 5 provide the necessary details to identify each field across all supported cyclic protocols. The binary format of the cyclic data is identical for all protocols.

Protocol-specific details are provided in Section 6, Section 7, and Section 8.

You will also find standard cyclic protocol definition files in the robot firmware package:

- EtherNetIP: *Mcs500_vX.X.X.X.eds*
- PROFINET: *GSDML-V2.42-Mecademic-mcs500-XXXXXXXX.xml*
- EtherCAT: *Mcs500_EtherCAT_ESI_vX.X.X.X.xml*

These files can be imported into your PLC to automatically describe the robot and populate a structure with all its cyclic fields.

Cyclic output format

The cyclic output (sent to the robot) contains fields for sending commands and actions.

The total size of the cyclic output is 60 bytes, divided into the following sections:

- *Robot control* (page 44): Controls the general robot state (activation, simulation mode, recovery mode, etc.);
- *Motion control* (page 47): Controls robot motion (pausing, resuming, sending motion commands, etc.);
- *Motion-related commands* (page 50): ID and arguments of the command to execute;
- *Host time* (page 55): Synchronizes the robot's time with the host time;
- *Brake control* (page 57): Controls the robot's brakes when deactivated;
- *Dynamic data configuration* (page 59): Selects which dynamic data the robot reports in its cyclic input payload.

Robot control

The *RobotControl* section in the cyclic output controls general robot states. Changes to bits in this output trigger robot actions, depending on the conditions.

Table 4: *RobotControl* (Offset 0, size 4, EtherCAT index 7200h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
DeactivateRobo	Bool	0:0	1	1600h:1	Deactivates the robot (see *DeactivateRobot* (page 175)) when set to 1. Deactivation is confirmed by the *Activated* status bit (Section 5).
ActivateRobot	Bool	0:1	1	1600h:2	Activates the robot (see *ActivateRobot* (page 171)) when set to 1, but only if the *Deactivate* bit is 0. Activation is confirmed by the *Activated* status bit (Section 5).
Home	Bool	0:2	1	1600h:3	Not available on this robot.
ResetError	Bool	0:3	1	1600h:4	Resets the error (see *ResetError* (page 184)) when set to 1. The reset is confirmed when *ErrorCode* becomes 0 (Section 5).

continues on next page

Table 4 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
ActivateSim	Bool	0:4	1	1600h:5	Enables (set to 1) or disables (set to 0) the default simulation mode type (only applies when the robot is deactivated and has no safety stop signal). See *ActivateSim* (page 172). The simulation mode status is confirmed by the *SimActivated* bit (Section 5). Note that the type of simulation mode (fast or normal) is not reported in cyclic protocols. Also, to change the default simulation mode type, use the TCP command *SetSimModeCfg* (page 201).
EnableRecovery	Bool	0:5	1	1600h:6	Enables (set to 1) or disables (set to 0) recovery mode (see *SetRecoveryMode* (page 198)). The recovery mode state is confirmed by the *RecoveryMode* status bit (Section 5).
DisableEtherCA	Bool	0:6	1	1600h:7	Disables the EtherCAT protocol when set to 1.
(Reserved)		0:7	25		Reserved for future use.

Motion control

The *MotionControl* section in the cyclic output controls robot motion. Changes to bits in this output trigger robot actions, depending on the conditions.

Table 5: *MotionControl* (Offset 4, size 4, EtherCAT index 7310h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
MoveID	Integer	4	16	1601h:1	A user-defined number. Changing it triggers the command specified in *MotionCommandID* to be added to the motion queue. Reception of the command is confirmed by the *MoveID* status bit (Section 5). See Section 5 for details.
SetPoint	Bool	6:0	1	1601h:2	Must be set to 1 for commands to be sent to the robot. See Section 5 for details.
PauseMotion	Bool	6:1	1	1601h:3	Pauses robot motion without clearing commands in the queue (*PauseMotion* (page 182)). Pause is confirmed by the *Paused* status bit (Section 5).
ClearMotion	Bool	6:2	1	1601h:4	Clears the motion queue and pauses the robot (*ClearMotion* (page 173)). Clear is confirmed by the *Cleared* status bit (Section 5).

continues on next page

Table 5 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
ResumeMotion	Bool	6:3	1	1601h:5	A rising edge (value changed from 0 to 1) resumes robot motion (*ResumeMotion* (page 185)) if the following conditions are met: - *PauseMotion* and *ClearMotion* are cleared; - No safety stop signals are active. Resuming also clears resettable safety stops (such as P-Stop 2 or *enabling device released* safety stop signals). It also clears collision and work zone events. Motion resume is confirmed when the *Paused* status bit is cleared (Section 5).
UseVariables	Bool	6:4	1	1601h:6	When set, the robot interprets float arguments (Section 5) as the cyclic ID of variables to use as function arguments. See *Managing variables with cyclic protocols* (page 332) for details.
(Reserved)		6:5	11		Reserved for future use. Must be 0.

Motion-related commands

The *MotionCommand* section in the cyclic output is used to specify the command to execute (Table 6) and its arguments (Table 7).

The list of valid *MotionCommandID* values is provided in Table 8, along with the expected arguments for each command.

Table 6: *MotionCommand* (Offset 8, size 4, EtherCAT index 7305h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
MotionCommandI	Integer	8	32	1602h:1	The ID of the motion-related command to execute. See Table 8 for command IDs and *Using motion-related commands* (page 40) for more information.

Table 7: *MotionCommandArgs* (Offset 12, size 24, EtherCAT index 7306h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
Argument 1	Real	12	32	1602h:2	First argument of the motion-related command, if applicable.
Argument 2	Real	16	32	1602h:3	First argument of the motion-related command, if applicable.
Argument 3	Real	20	32	1602h:4	First argument of the motion-related command, if applicable.
Argument 4	Real	24	32	1602h:5	First argument of the motion-related command, if applicable.
Argument 5	Real	28	32	1602h:6	First argument of the motion-related command, if applicable.
Argument 6	Real	32	32	1602h:7	First argument of the motion-related command, if applicable.

Table 8: *MotionCommandID* numbers

ID	Description
0	No movement: all six arguments are ignored.
1	*MoveJoints* (page 126)†
2	*MovePose* (page 138)†
3	*MoveLin* (page 132)†
4	*MoveLinRelTrf* (page 134)†
5	*MoveLinRelWrf* (page 135)†
6	*Delay* (page 125)†
7	*SetBlending* (page 141)†
8	*SetJointVel* (page 151)†
9	*SetJointAcc* (page 150)†
10	*SetCartAngVel* (page 143)†
11	*SetCartLinVel* (page 144)†
12	*SetCartAcc* (page 142)†
13	*SetTrf* (page 166)†
14	*SetWrf* (page 168)†
15	*SetConf* (page 147) (the posture configuration parameter, c_e, must be provided as the second argument).
16	*SetAutoConf* (page 139)†
17	*SetCheckpoint* (page 145)†
18	Gripper action: argument 1 is 0 for *GripperClose* (page 309) and 1 for *GripperOpen* (page 310) (not available on this robot)
19	*SetGripperVel* (page 315)† (not available on this robot)
20	*SetGripperForce* (page 313)† (not available on this robot)
21	*MoveJointsVel* (page 129)†
22	*MoveLinVelWrf* (page 137)†
23	*MoveLinVelTrf* (page 136)†
24	*SetVelTimeout* (page 167)†
25	*SetConfTurn* (page 148)†
26	*SetAutoConfTurn* (page 140)†
27	*SetTorqueLimits* (page 162)†
28	*SetTorqueLimitsCfg* (page 164)†
29	*MoveJointsRel* (page 128)†
30	*SetValveState* (page 323)† (not available on this robot)
31	*SetGripperRange* (page 314)† (not available on this robot)
32	*MoveGripper* (page 311)† (not available on this robot)
33	*SetJointVelLimit* (page 152)†

continues on next page

Table 8 – continued from previous page

ID	Description
34	*SetOutputState* (page 317), arguments are 32-bit integers: [*BankId*, *output-values*, *output-mask*]. The mask specifies which outputs will change; only outputs with the corresponding bit set in the mask are affected.
35	*SetOutputState_Immediate* (page 318), uses the same arguments as *SetOutputState* (page 317).
36	*SetIoSim* (page 316)†
37	*VacuumGrip* (page 324), all six arguments are ignored.
38	*VacuumGrip_Immediate* (page 325), all six arguments are ignored.
39	*VacuumRelease* (page 326), all six arguments are ignored.
40	*VacuumRelease_Immediate* (page 327), all six arguments are ignored.
41	*SetVacuumThreshold* (page 321)†
42	*SetVacuumThreshold_Immediate* (page 322)†
43	*SetVacuumPurgeDuration* (page 319)†
44	*SetVacuumPurgeDuration_Immediate* (page 320)†
45	*MoveJump* (page 130)†
46	*SetMoveJumpHeight* (page 157)†
47	*SetMoveJumpApproachVel* (page 156)†
48	*SetTimeScaling* (page 202)†
49	*SetMoveMode* (page 159)†
50	*SetMoveDurationCfg* (page 154)†
51	*SetMoveDuration* (page 153)†
60	*SetPayload* (page 160)†
100	*StartProgram* (page 203)†
150	*SetJointLimits* (page 190)†
151	*SetJointLimitsCfg* (page 191)†
152	*SetWorkZoneCfg* (page 289)†
153	*SetWorkZoneLimits* (page 290)†
154	*SetCollisionCfg* (page 287)†
155	*SetToolSphere* (page 288)†
156	*SetCalibrationCfg* (page 186)†
200	*RebootRobot* (page 183)
10,000 to 19,999	Set a robot variable (see *Setting a variable* (page 332))

† Argument count and type match those of the related TCP/IP command. Extra arguments are ignored.

Note

For commands with a pose (or Cartesian velocity vector), the fourth and fifth arguments must be zero, and the sixth argument is the gamma angle (or rotational velocity about the z-axis). For example, for command ID 2 (*MovePose* (page 138)), use `[x, y, z, 0, 0, γ]`.

Host time

The *HostTime* section in the cyclic output synchronizes the robot's date/time with the host's (see *SetRtc* (page 200)).

Table 9: *HostTime* (Offset 36, size 4, EtherCAT index 7400h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
HostTime	Integer	36	32	1610h:1	Current time in seconds since UNIX epoch (00:00:00 UTC, January 1, 1970). If non-zero, the robot updates its time to this value (same as *SetRtc* (page 200)). This ensures accurate timestamps in robot logs, as the robot resets its time on reboot.`

Brake control

The *BrakesControl* section in the cyclic output controls the robot's brakes when it is deactivated.

The robot has brakes on joints 3 and 4.

The brakes behave as follows:

- Brakes disengage automatically when the robot is activated (it holds position when not moving);
- Brakes engage automatically when the robot is deactivated (including safety signals or power off);
- While deactivated, the brakes can be controlled using the fields in Table 10.

Table 10: *BrakesControl* (Offset 40, size 4, EtherCAT index 7410h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
BrakesControlA	Bool	40:0	1	1611h:1	Must be set to 1 to allow brakes control through cyclic data. This bit ensures that the brakes are not inadvertently disengaged if cyclic data sent to the robot contains all zeros.
BrakesEngaged	Bool	40:1	1	1611h:2	If set to 1, the brakes are engaged. If 0, the brakes are disengaged, and the robot may fall under the effects of gravity. This bit is ignored if the *BrakesControlAllowed* bit is cleared or if the robot is activated.
(Reserved)		40:2	30		Reserved for future use. Must be 0.

Dynamic data configuration

The *DynamicDataConfiguration* section in the cyclic output determines which dynamic data the robot reports in each of the 4 available dynamic-data slots in its cyclic input payload.

When a specific dynamic data type is chosen (in Table 11), the robot will return the corresponding values in Table 22, Table 23, Table 24, or Table 25.

If dynamic data type 0 (*Automatic*) is used, the robot cycles through available dynamic data types, reporting a different type each cycle.

See Table 12 for a list of available dynamic data types.

Note

A delay of one or two cycles may occur before a change to the requested dynamic data type takes effect.

Table 11: *DynamicDataConfiguration* (Offset 44, size 16, EtherCAT indices 7420h, 7421h, 7422h, 7423h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
DynamicDataTyp	Integer	44:0	32	1620h:1	Dynamic data type for index #1 (see Table 12).
DynamicDataTyp	Integer	48:0	32	1621h:1	Dynamic data type for index #2 (see Table 12).
DynamicDataTyp	Integer	52:0	32	1622h:1	Dynamic data type for index #3 (see Table 12).
DynamicDataTyp	Integer	56:0	32	1623h:1	Dynamic data type for index #4 (see Table 12).

Table 12: List of *DynamicDataTypeID* values

ID	Description
0	*Automatic*. The robot will automatically choose a dynamic data type and change it every cycle, going through all of them in around-robin manner. This is the easiest way for the host to receive all possible values periodically.
1	*Firmware version* (*GetFwVersion* (page 225)). Values: [major version, minor version, patch version, build number].
2	*Product type* (*GetProductType* (page 241)). Values: [product type] where 3 = Meca500 R3, 4 = Meca500 R4, 20 = MCS500 R1.
3	*Serial number* (*GetRobotSerial* (page 246)). Values: [serial number].
4–10	Reserved.
11	*Joint limits configuration* (*GetJointLimitsCfg* (page 228)). Values: [1/0].
12	*Model joint limits* (*GetModelJointLimits* (page 231)), for joints 1, 2, and 3. Values: [$q_{1,min}$, $q_{2,min}$, $q_{3,min}$, $q_{1,max}$, $q_{2,max}$, $q_{3,max}$], in or mm (for joint 3).
13	*Model joint limits* (*GetModelJointLimits* (page 231)), for joint 4. Values: [$q_{4,min}$, 0, 0, $q_{4,max}$, 0, 0], in °.
14	*Effective joint limits* (*GetJointLimits* (page 227)), for joints 1, 2, and 3. Values: [$q_{1,min}$, $q_{2,min}$, $q_{3,min}$, $q_{1,max}$, $q_{2,max}$, $q_{3,max}$], in or mm (for joint 3).

continues on next page

Table 12 – continued from previous page

ID	Description
15	*Effective joint limits* (*GetJointLimits* (page 227)), for joint 4. Values: [$q_{4,min}$, 0, 0, $q_{4,max}$, 0, 0], in °.
17	*Work zone configuration* (*GetWorkZoneCfg* (page 284)). Values: [work zone limits severity, work zone limits detection mode].
18	*Work zone limits* (*GetWorkZoneLimits* (page 285)). Values: [x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}], in mm.
19	*Tool sphere* (*GetToolSphere* (page 283)). Values: [x, y, z, r], in mm.
20	*Conf* and *Conf turn* (*GetConf* (page 222), *GetConfTurn* (page 223), *GetAutoConf* (page 213), *GetAutoConfTurn* (page 214)). Values: [0, elbow −1/1/NaN, 0, last joint turn or NaN]. NaN indicates auto-conf or auto-conf-turn.
21	*Motion queue parameters* (*GetBlending* (page 215), *GetVelTimeout* (page 252)). Values: [blending ratio percent, velocity timeout in seconds].
22	*Motion queue velocities and accelerations* (*GetJointVel* (page 229), *GetJointAcc* (page 226), *GetCartLinVel* (page 219), *GetCartAngVel* (page 218), *GetCartAcc* (page 217), *GetJointVelLimit* (page 230)). Values: [joint velocity, joint acceleration, Cartesian linear velocity, Cartesian angular velocity, Cartesian acceleration, joint velocity limit], in percent.

continues on next page

Table 12 – continued from previous page

ID	Description
23	*Gripper parameters* (*GetGripperForce* (page 293), *GetGripperVel* (page 295), *GetGripperRange* (page 294)). Values: [gripper force, gripper velocity, fingers opening corresponding to closed state, fingers opening corresponding to open state]. Arguments 1 and 2 are in percentage, while arguments 3 and 4 are in mm.
24	*Torque limits configuration* (*GetTorqueLimitsCfg* (page 250)). Values: [severity, detection mode].
25	*Torque limits* (*GetTorqueLimits* (page 249)). Values: [motor 1 limit, motor 2 limit, ...], in percent.
26	*Vacuum configuration* (*GetVacuumThreshold* (page 308), *GetVacuumPurgeDuration* (page 307)). Values: [holdThreshold, releaseThreshold, purgeDuration]. Arguments 1 and 2 are in kPa, argument 3 is in seconds.
27	*Move jump height* (*GetMoveJumpHeight* (page 236)). Values: [h_{start}, h_{end}, h_{min}, h_{max}], in mm.
28	*Move jump approach velocity* (*GetMoveJumpApproachVel* (page 235)). Values: [v_{start}, p_{start}, v_{end}, p_{end}], in mm or mm/s.
29	*Move mode configuration* (*GetMoveMode* (page 237), *GetMoveDurationCfg* (page 234), *GetMoveDuration* (page 233)). Values: [move mode, severity, duration].

continues on next page

Table 12 – continued from previous page

ID	Description
30	*Robot calibration status* (*GetCalibrationCfg* (page 216), *GetRobotCalibrated* (page 244)). Values: [calibrationEnabled, calibrated].
31	*Robot payload* (*GetPayload* (page 240)). Values: [m, c_x, c_y, c_z], in kg or mm.
32	*Target real-time joint velocity* (*GetRtTargetJointVel* (page 272)). Values: [ω_1, ω_2, ...], in mm/s (for joint 3) or °/s.
33	*Target real-time joint torque* (*GetRtTargetJointTorq* (page 271)). Values: [motor 1 torque, motor 2 torque, ...], in percent.
34	*Target real-time Cartesian velocity* (*GetRtTargetCartVel* (page 267)). Values: [$\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z], in mm/s or °/s.
36	*Collision configuration* (*GetCollisionCfg* (page 281)). Values: [collision severity level].
37	*Collision status* (*GetCollisionStatus* (page 282)). Values: [collision boolean state, group of colliding object 1, ID of colliding object 1, group of colliding object 2, ID of colliding object 2].
38	*Work zone status* (*GetWorkZoneStatus* (page 286)). Values: [work zone breach Boolean state, group of object in breach, ID of object in breach].

continues on next page

Table 12 – continued from previous page

ID	Description
40	*Actual joint position* (*GetRtJointPos* (page 263)). Values: [q_1, q_2, q_3, ...]. Unit is mm (for joint 3) or °.
41	*Actual end-effector pose* (*GetRtCartPos* (page 259)). Values: [x, y, z, α, β, γ]. Units are mm or °.
42	*Actual joint velocity* (*GetRtJointVel* (page 265)). Values: [ω_1, ω_2, ...], in mm/s (for joint 3) or °/s.
43	*Actual joint torque* (*GetRtJointTorq* (page 264)). Values: [joint 1 torque, joint 2 torque, ...], in percent.
44	*Actual Cartesian velocity* (*GetRtCartVel* (page 260)). Values: [$\dot{x}$, $\dot{y}$, $\dot{z}$, ω_x, ω_y, ω_z], in mm/s or °/s.
45	*Actual conf and conf turn* (*GetRtConf* (page 261), *GetRtConfTurn* (page 262)). Values: [0, elbow −1/0/1, 0, last joint turn].
46	Not available on this robot.
52	Not available on this robot.
53	Not available on this robot.
54	*Time scaling* (*GetTimeScaling* (page 248)). Values: [p], in percent.
61	*Configured joint limits* (*GetJointLimits* (page 227)), for joints 1, 2, and 3 (ignored if joint limits are disabled). Values: [$q_{1,min}$, $q_{2,min}$, $q_{3,min}$, $q_{1,max}$, $q_{2,max}$, $q_{3,max}$], in or mm (for joint 3).

continues on next page

Table 12 – continued from previous page

ID	Description
62	*Configured joint limits* (*GetJointLimits* (page 227)), for joint 4 (ignored if joint limits are disabled). Values: [$q_{4,min}$, 0, 0, $q_{4,max}$, 0, 0], in °.
72	*Digital input and output states* (*GetRtIoStatus* (page 302), *GetRtInputState* (page 301), *GetRtOutputState* (page 303)). Unlike most other dynamic data types that report up to six float values, this one maps the 12 bytes of the first three float values as follows: - Byte index 0: - Bit 0: simMode - Bit 1: present - Bits 1-7: reserved - Byte index 1: errorCode. - Byte index 2: number of digital outputs. - Byte index 3: number of digital inputs. - Bytes indices 4–7: digital outputs values. One bit per output. Least significant bit/byte first. - Bytes indices 8–11: digital inputs values. One bit per input. Least significant bit/byte first.
73	*Vacuum state* (*GetRtVacuumState* (page 305), *GetRtVacuumPressure* (page 304)). Values: [vacuumOn, purgeOn, holdingPart, pressure]. Pressure is in kPa.

Note

Each *DynamicDataTypeID* returns six values as defined in Section 5. Unused values are set to 0. For example, ID 19 provides four meaningful values, and the last two are 0. In SCARA robots, some values (such as the Euler angles α and β) are always zero.

Note

No dynamic data IDs are defined for information that is already available in other cyclic data fields, such as *Target joint set* (page 75), *Target end-effector pose* (page 77), *Target*

configuration (page 79), *Target WRF* (page 81), etc. See *Cyclic input format* (page 68) for details.

Cyclic input format

The cyclic input (received from the robot) provides the complete status, position, and configuration of the robot.

The total size of the cyclic input is 252 bytes, divided into the following sections:

- *Robot status* (page 68): General robot state (e.g., activation, simulation mode, recovery mode, etc.);
- *Motion status* (page 70): Robot motion status (e.g., paused state, motion queue status, and other motion-related conditions);
- *Target joint set* (page 75): Real-time calculated joint positions (*GetRtTargetJointPos* (page 270));
- *Target end-effector pose* (page 77): Real-time calculated Cartesian position (*GetRtTargetCartPos* (page 266));
- *Target configuration* (page 79): Real-time calculated shoulder, elbow, wrist, and turn configuration (*GetRtTargetConf* (page 268) and *GetRtTargetConfTurn* (page 269));
- *Target WRF* (page 81): World reference frame used in the real-time calculated position (*GetRtWrf* (page 274), *GetWrf* (page 253));
- *Target TRF* (page 83): Tool reference frame used in the real-time calculated position (*GetRtTrf* (page 273), *GetTrf* (page 251));
- *Robot timestamp* (page 85): Precise monotonic robot timestamp associated with this cyclic data (*GetRtc* (page 275));
- *Safety status* (page 87): Safety-related information (e.g., safety signals, power supply states, operating mode);
- *Dynamic data* (page 90): Additional robot information not included in the above; contents may vary.

Robot status

The *RobotStatus* section in the cyclic input reports the general robot state (similar to *GetStatusRobot* (page 277)).

Table 13: *RobotStatus* (Offset 0, size 4, EtherCAT index 6010h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
Busy	Bool	0:0	1	1A00h:2	True only while the robot is being activated, homed, or deactivated.
Activated	Bool	0:1	1	1A00h:3	Indicates whether the motors are on (powered) and the robot is ready to receive motion commands.
Homed	Bool	0:2	1	1A00h:4	Not available on this robot.
SimActivated	Bool	0:3	1	1A00h:5	Indicates whether the robot simulation mode is activated.
BrakesEngaged	Bool	0:4	1	1A00h:6	Indicates whether the brakes are engaged.
RecoveryMode	Bool	0:5	1	1A00h:7	Indicates whether the robot recovery mode is activated.
EStop (deprecated)	Bool	0:6	1	1A00h:8	Indicates whether the emergency stop safety signal is activated. Deprecated; use *EStop* bit from *SafetyStatus* instead.
CollisionStatu	Bool	0:7	1	1A00h:9	Indicates whether the robot has detected an imminent collision (*SetCollisionCfg* (page 287)).
WorkZoneStatus	Bool	1:0	1	1A00h:10	Indicates whether the robot has detected a work zone breach (*SetWorkZoneCfg* (page 289), *SetWorkZoneLimits* (page 290)).
MonitoringMode	Bool	1:1	1	1A00h:11	1 if this connection with the robot only allows monitoring, not controlling.
(Reserved)		1:2	6		Reserved for future use.
ErrorCode	Integer	2	16	1A00h:1	Indicates the error code (see Table 1 and Table 3) or 0, if there is no error.

Motion status

The *MotionStatus* section in the cyclic input reports the robot's motion status (similar to *GetStatusRobot* (page 277)).

Table 14: *MotionStatus* (Offset 4, size 12, EtherCAT index 6015h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
CheckpointReac	Integer	4	16	1A01h:1	Indicates the last checkpoint number reached (*GetCheckpoint* (page 220)). The value remains the same until another checkpoint is reached.
CheckpointDisc	Integer	6	16	1A01h:2	Indicates the last checkpoint number discarded (*GetCheckpointDiscarded* (page 221)). The value remains unchanged until another checkpoint is discarded.
MoveID	Integer	8	16	1A01h:3	Acknowledges the *MoveID* of the last command the robot received (*Motion control* (page 47)). For details, refer to *Using motion-related commands* (page 40).
FifoSpace	Integer	10	16	1A01h:4	The number of commands that can be added to the robot's motion queue at any time (the maximum is 13,000). If 0 (too many commands sent), subsequent commands will be ignored.

continues on next page

Table 14 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
Paused	Bool	12:0	1	1A01h:6	Indicates whether motion is paused. This bit stays set (and the robot remains paused) until motion is resumed with *Motion control* (page 47) bit *ResumeMotion*.
EOB	Bool	12:1	1	1A01h:7	The End of Block (*EOB*) bit is set when the robot is not moving and there are no motion commands left in the queue. Note that the *EOB* bit may occasionally be set before all commands are completed due to network or processing delays. Therefore, don't rely on this flag to determine when all movements have finished. Use a checkpoint instead (*SetCheckpoint* (page 145)).
EOM	Bool	12:2	1	1A01h:8	The End of Motion (*EOM*) bit is set when the robot is not moving. Note that the *EOM* bit may occasionally be set between two consecutive motion commands. Therefore, do not rely on this flag to determine when all movements have finished. Use a checkpoint instead (*SetCheckpoint* (page 145)).

continues on next page

Table 14 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
Cleared	Bool	12:3	1	1A01h:9	Indicates whether the motion queue is cleared. If the queue is cleared, the robot is not moving. This bit remains set (and the robot remains paused) when the motion queue is cleared due to the *ClearMotion* control bit, robot deactivation, or a safety signal. It remains set until motion is resumed with *Motion control* (page 47) bit *ResumeMotion* (if the robot is still activated) or the robot is reactivated (if it was deactivated).
PStop2 (deprecated)	Bool	12:4	1	1A01h:10	Indicates whether the P-Stop 2 safety signal is set. Deprecated; use the *PStop2* bit from *SafetyStatus* instead.
ExcessiveTorqu	Bool	12:5	1	1A01h:11	Indicates whether a joint torque is exceeding the corresponding user-defined torque limit (*SetTorqueLimits* (page 162), *GetTorqueLimitsStatus* (page 278)).
(Reserved)		12:6	10		Reserved for future use.

continues on next page

Table 14 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
OfflineProgram	Integer	14	16	1A01h:5	ID of the offline program currently running; 0 if none (*StartProgram* (page 203)).

Target joint set

The *TargetJointSet* section in the cyclic input reports the robot's real-time calculated joint position (similar to *GetRtTargetJointPos* (page 270)).

Table 15: *TargetJointSet* (Offset 16, size 24, EtherCAT index 6030h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
Joint 1	Real	16	32	1A02h:1	Real-time calculated target position for joint 1, in °.
Joint 2	Real	20	32	1A02h:2	Real-time calculated target position for joint 2, in °.
Joint 3	Real	24	32	1A02h:3	Real-time calculated target position for joint 3, in mm.
Joint 4	Real	28	32	1A02h:4	Real-time calculated target position for joint 4, in °.
Joint 5	Real	32	32	1A02h:5	n/a
Joint 6	Real	36	32	1A02h:6	n/a

Target end-effector pose

The *TargetEndEffectorPose* section in the cyclic input reports the robot's real-time calculated Cartesian position of the origin of the TRF with respect to the WRF (similar to *GetRtTargetCartPos* (page 266)).

Table 16: *TargetEndEffectorPose* (Offset 40, size 24, EtherCAT index 6031h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
X coordinate	Real	40	32	1A03h:1	X coordinate of the origin of the TRF with respect to the WRF, in mm.
Y coordinate	Real	44	32	1A03h:2	Y coordinate of the origin of the TRF with respect to the WRF, in mm.
Z coordinate	Real	48	32	1A03h:3	Z coordinate of the origin of the TRF with respect to the WRF, in mm.
α angle	Real	52	32	1A03h:4	α Euler angle representing the orientation of the TRF with respect to the WRF, in °.
β angle	Real	56	32	1A03h:5	β Euler angle representing the orientation of the TRF with respect to the WRF, in °.
γ angle	Real	60	32	1A03h:6	γ Euler angle representing the orientation of the TRF with respect to the WRF, in °.

Target configuration

The *TargetConfiguration* section in the cyclic input reports robot real-time posture and turn configurations that correspond to the calculated joint set (*GetRtTargetConf* (page 268), *GetRtTargetConfTurn* (page 269)). For more details, see Section 3.

Table 17: *TargetConfiguration* (Offset 64, size 4, EtherCAT index 6046h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
Shoulder	Integer	64	8	1A08h:1	(Not used on this robot)
Elbow	Integer	65	8	1A08h:2	Real-time elbow posture configuration corresponding to the calculated joint position. The value is typically −1 or 1 but may also be 0 when the robot is near the elbow singularity. See Section 3.
Wrist	Integer	66	8	1A08h:3	(Not used on this robot)
Turn	Integer	67	8	1A08h:4	Real-time turn configuration for the last joint. See Section 3.

Target WRF

The *TargetWrf* section in the cyclic input reports the WRF (with respect of the BRF) used for reporting the current end-effector pose (*Target end-effector pose* (page 77)), similar to the *GetRtWrf* (page 274) and *GetWrf* (page 253) command.

Table 18: *TargetWrf* (Offset 68, size 24, EtherCAT index 6050h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
X coordinate	Real	68	32	1A09h:1	X coordinate of the origin of the WRF with respect to the BRF, in mm.
Y coordinate	Real	72	32	1A09h:2	Y coordinate of the origin of the WRF with respect to the BRF, in mm.
Z coordinate	Real	76	32	1A09h:3	Z coordinate of the origin of the WRF with respect to the BRF, in mm.
α angle	Real	80	32	1A09h:4	α Euler angle representing the orientation of the WRF with respect to the BRF, in °.
β angle	Real	84	32	1A09h:5	β Euler angle representing the orientation of the WRF with respect to the BRF, in °.
γ angle	Real	88	32	1A09h:6	γ Euler angle representing the orientation of the WRF with respect to the BRF, in °.

Target TRF

The *TargetTrf* section in the cyclic input reports the TRF (with respect of the FRF) used for reporting the current end-effector pose (*Target end-effector pose* (page 77)), similar to the *GetRtTrf* (page 273) and *GetTrf* (page 251) commands.

Table 19: *TargetTRF* (Offset 92, size 24, EtherCAT index 6051h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
X coordinate	Real	92	32	1A0Ah:1	X coordinate of the origin of the TRF with respect to the FRF, in mm.
Y coordinate	Real	96	32	1A0Ah:2	Y coordinate of the origin of the TRF with respect to the FRF, in mm.
Z coordinate	Real	100	32	1A0Ah:3	Z coordinate of the origin of the TRF with respect to the FRF, in mm.
α angle	Real	104	32	1A0Ah:4	α Euler angle representing the orientation of the TRF with respect to the FRF, in °.
β angle	Real	108	32	1A0Ah:5	β Euler angle representing the orientation of the TRF with respect to the FRF, in °.
γ angle	Real	112	32	1A0Ah:6	γ Euler angle representing the orientation of the TRF with respect to the FRF, in °.

Robot timestamp

The *RobotTimestamp* section in the cyclic input reports a precise, monotonic robot timestamp associated with the cyclic data (similar to the command *GetRtc* (page 275)).

Table 20: *RobotTimestamp* (Offset 116, size 12, EtherCAT index 6060h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
Seconds	Integer	116	32	1A10h:1	Robot's monotonic timestamp in seconds, based on an arbitrary reference.
Microseconds	Integer	120	32	1A10h:2	Robot's monotonic timestamp in microseconds, within the current second.
DynamicDataCyc	Integer	124	32	1A10h:3	Incremented each time the robot cycles through all available dynamic data to report. Applies only if at least one dynamic data slot (Table 11) is configured with ID 0 (*Automatic*).

Safety status

The *SafetyStatus* section in the cyclic input reports safety-related information (safety signals, power supply input states, operating mode, etc.). See *Management of errors and safety stops* (page 34).

Table 21: *SafetyStatus* (Offset 128, size 12, EtherCAT index 6065h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
EStop	Bool	128:0	1	1A11h:1	E-Stop safety stop signal state†
PStop1	Bool	128:1	1	1A11h:2	P-Stop 1 safety stop signal state†
PStop2	Bool	128:2	1	1A11h:3	P-Stop 2 safety stop signal state†
(Reserved)		128:3	1		Reserved for future use.
OperationModeC	Bool	128:4	1	1A11h:5	Operation mode change safety stop signal state†
EnablingDevice	Bool	128:5	1	1A11h:6	Enabling device released safety stop signal state†
VoltageFluctua	Bool	128:6	1	1A11h:7	Voltage fluctuation safety stop signal state†
Reboot	Bool	128:7	1	1A11h:8	Robot rebooted safety stop signal state†
RedundancyFaul	Bool	129:0	1	1A11h:9	Redundancy fault safety stop signal state†
StandstillFaul	Bool	129:1	1	1A11h:10	Standstill fault safety stop signal state†
ConnectionDrop	Bool	129:2	1	1A11h:11	TCP/IP connection dropped safety stop signal state†
MinorError	Bool	129:3	1	1A11h:12	Minor error safety stop signal state†
(Reserved)		129:4	20		Reserved for future use.
EStopResettabl	Bool	132:0	1	1A11h:33	E-Stop safety stop signal ready to be reset (Reset button)
PStop1Resettab	Bool	132:1	1	1A11h:34	P-Stop 1 safety stop signal ready to be reset (Reset button)
PStop2Resettab	Bool	132:2	1	1A11h:35	P-Stop 2 safety stop signal ready to be reset (with *ResumeMotion*)
(Reserved)		132:3	1		Reserved for future use.
OperationModeC	Bool	132:4	1	1A11h:37	Operation mode change safety stop signal ready to be reset (Reset button)
EnablingDevice	Bool	132:5	1	1A11h:38	Enabling device released safety stop signal ready to be reset (with *ResumeMotion*)
VoltageFluctua	Bool	132:6	1	1A11h:39	Voltage fluctuation safety stop signal ready to be reset (Reset button)
RebootResettab	Bool	132:7	1	1A11h:40	Robot rebooted safety stop signal ready to be reset (Reset button)

continues on next page

Table 21 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
RedundancyFaul	Bool	133:0	1	1A11h:41	Always 0. A redundancy fault requires rebooting the robot; it cannot be reset.
StandstillFaul	Bool	133:1	1	1A11h:42	Standstill fault safety stop signal ready to be reset (Reset button)
ConnectionDrop	Bool	133:2	1	1A11h:43	Connection dropped safety stop signal ready to be reset (with *ResumeMotion*)
MinorErrorRese	Bool	133:3	1	1A11h:44	Minor error safety stop signal ready to be reset (with *ResumeMotion*)
(Reserved)		133:4	20		Reserved for future use.
OperationMode	Integer	136	8	1A11h:65	0 for locked, 1 for automatic, 2 for manual
ResetReady	Bool	137:0	1	1A11h:66	If no more safety signals causing motor power to be removed are present, and the robot is ready to be reset with the Reset button
VMotorOn	Bool	137:1	1	1A11h:67	Robot motors powered or not
(Reserved)		137:2	6		Reserved for future use
PsuInputs_Esto	Bool	138:0	1	1A11h:74	Set (1) when at least one of the two power supply E-Stop inputs is asserted
PsuInputs_PSto	Bool	138:1	1	1A11h:75	Set (1) when at least one of the two power supply P-Stop 1 inputs is asserted
PsuInputs_PSto	Bool	138:2	1	1A11h:76	Set (1) when at least one of the two power supply P-Stop 2 inputs is asserted
PsuInputs_Rese	Bool	138:3	1	1A11h:77	Set (1) when the power supply Reset input is asserted
PsuInputs_Rese	Bool	138:4	1	1A11h:78	Set (1) when the power supply keypad reset button is pressed
PsuInputs_Enab	Bool	138:5	1	1A11h:79	Set (1) when both enabling device inputs are asserted (i.e., the enabling device is pressed)
(Reserved)		138:6	10		Reserved for future use.

† 1 when safety signal is present or resettable, 0 when safety signal has been successfully reset

Dynamic data

The *DynamicData* section in the cyclic input reports additional robot information that is not covered by other cyclic input fields.

The contents of each dynamic data slot are controlled by Table 11. Slots can be set to a specific dynamic data type (Table 12) or configured in *Automatic* mode, in which case the robot will automatically cycle through all available dynamic data types, changing the reported data every cycle.

Table 22: *DynamicData0* (Offset 140, size 28, EtherCAT index 6070h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
DynamicType	Integer	140	32	1A20h:1	Dynamic data type (see Table 12 for available values).
ValueIdx_0	Real	144	32	1A20h:2	Value index 0 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_1	Real	148	32	1A20h:3	Value index 1 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_2	Real	152	32	1A20h:4	Value index 2 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_3	Real	156	32	1A20h:5	Value index 3 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_4	Real	160	32	1A20h:6	Value index 4 (the meaning depends on the *DynamicType*, see Table 12).

continues on next page

Table 22 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
ValueIdx_5	Real	164	32	1A20h:7	Value index 5 (the meaning depends on the *DynamicType*, see Table 12).

Table 23: *DynamicData1* (Offset 168, size 28, EtherCAT index 6071h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
DynamicType	Integer	168	32	1A21h:1	Dynamic data type (see Table 12 for available values).
ValueIdx_0	Real	172	32	1A21h:2	Value index 0 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_1	Real	176	32	1A21h:3	Value index 1 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_2	Real	180	32	1A21h:4	Value index 2 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_3	Real	184	32	1A21h:5	Value index 3 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_4	Real	188	32	1A21h:6	Value index 4 (the meaning depends on the *DynamicType*, see Table 12).

continues on next page

Table 23 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
ValueIdx_5	Real	192	32	1A21h:7	Value index 5 (the meaning depends on the *DynamicType*, see Table 12).

Table 24: *DynamicData2* (Offset 196, size 28, EtherCAT index 6072h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
DynamicType	Integer	196	32	1A22h:1	Dynamic data type (see Table 12 for available values).
ValueIdx_0	Real	200	32	1A22h:2	Value index 0 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_1	Real	204	32	1A22h:3	Value index 1 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_2	Real	208	32	1A22h:4	Value index 2 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_3	Real	212	32	1A22h:5	Value index 3 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_4	Real	216	32	1A22h:6	Value index 4 (the meaning depends on the *DynamicType*, see Table 12).

continues on next page

Table 24 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
ValueIdx_5	Real	220	32	1A22h:7	Value index 5 (the meaning depends on the *DynamicType*, see Table 12).

Table 25: *DynamicData3* (Offset 224, size 28, EtherCAT index 6073h)

Field	Type	Offset	Size (bits)	ECAT PDO	Description
DynamicType	Integer	224	32	1A23h:1	Dynamic data type (see Table 12 for available values).
ValueIdx_0	Real	228	32	1A23h:2	Value index 0 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_1	Real	232	32	1A23h:3	Value index 1 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_2	Real	236	32	1A23h:4	Value index 2 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_3	Real	240	32	1A23h:5	Value index 3 (the meaning depends on the *DynamicType*, see Table 12).
ValueIdx_4	Real	244	32	1A23h:6	Value index 4 (the meaning depends on the *DynamicType*, see Table 12).

continues on next page

Table 25 – continued from previous page

Field	Type	Offset	Size (bits)	ECAT PDO	Description
ValueIdx_5	Real	248	32	1A23h:7	Value index 5 (the meaning depends on the *DynamicType*, see Table 12).

EtherCAT communication

EtherCAT is an open real-time Ethernet protocol originally developed by Beckhoff Automation. When communicating with a Mecademic robot over EtherCAT, you can achieve guaranteed response times of 1 ms. Furthermore, you no longer need to parse strings as you do with the TCP/IP protocol.

Overview

Connection types

With EtherCAT, you can connect several Mecademic robots in various network topologies, including line, star, tree, or ring, as each robot has a unique node address. This allows targeted access to a specific robot, even if your network topology changes.

ESI file

Each EtherCAT slave device is described by an EtherCAT Slave Information (ESI) file that describes its identity, capabilities, and cyclic payload. The EtherCAT controllers (PLC) use this file to properly identify detected EtherCAT slave devices, such as a Mecademic robot.

It can be found in the zip file that contains your robot's firmware update package

(Mcs500_EtherCAT_ESI_vX.X.X.X.xml).

Enabling EtherCAT

The default communication protocol of the robot is the Ethernet TCP/IP protocol. This protocol is required for jogging the robot through its web interface.

To switch to EtherCAT, use the Network configuration panel in the MecaPortal configuration menu.

Alternatively, you can use the *SwitchToEtherCAT* (page 207) command, which can be entered in the MecaPortal code editor or sent to the robot via the *TCP/IP API* (page 18).

This command is persistent. The robot will remain in EtherCAT mode even after being rebooted.

⚠ Warning

When the robot is in EtherCAT mode, TCP/IP or EtherNet/IP communication is not possible (e.g., you cannot use the robot's web interface or other cyclic protocols).

Disabling EtherCAT

To disable EtherCAT (and restore standard TCP/IP communication mode), use the `RobotControl` PDO (see Table 4) or perform a network-configuration-reset.

LEDs

Your robot has three green LEDs on its base, labeled Link/Act IN, Link/Act OUT, and Run. When EtherCAT communication is enabled, these three LEDs indicate the state of the EtherCAT connection, as summarized in Table 26. The MCS500 also has a red LED, labeled ERR.

Table 26: EtherCAT LED description

LED	Name	LED State	EtherCAT state
Link/Act IN	IN port link	On	Link is active but there is no activity
		Blinking	Link is active and there is activity
		Off	Link is inactive
Link/Act OUT	OUT port link	On	Link is active but there is no activity
		Blinking	Link is active and there is activity
		Off	Link is inactive
Run	Run	On	Operational
		Blinking	Pre-Operational
		Single flash	Safe-Operational
		Flashing	Initialization or Bootstrap
		Off	Init
ERR	Error	On	PDI Watchdog Timeout
		Flickering	Booting Error
		Double flash	Application Watchdog Timeout
		Single flash	Unsolicited State Change
		Blinking	Invalid Configuration
		Off	No Error

PDO Mapping

The process data objects (PDOs) provide the interface to the application objects. PDOs are used to transfer data via cyclic communications in real time. PDOs can be reception PDOs (RxPDOs), which receive data from the EtherCAT master (the PLC or the industrial PC), or transmission PDOs (TxPDOs), which send the current value from the slave (the Mecademic robot) to the EtherCAT master. In the previous subsection, we listed the PDOs in the object dictionary. PDO assignment is summarized in the next two tables.

Table 27: RxPDOs

PDO	Object(s)	Name	Note
1600h	7200h	RobotControl	Mandatory. See Table 4.
1601h	7310h	MotionControl	Mandatory. See Table 5.
1602h	7305h, 7306h	Movement	Mandatory. See Table 6.
1610h	7400h	HostTime	Mandatory. See Table 9.
1611h	7410h	BrakesControl	Mandatory. See Table 10.
1620h	7420h	DynamicDataConfiguration 1	Mandatory. See Table 11.
1621h	7421h	DynamicDataConfiguration 2	Mandatory. See Table 11.
1622h	7422h	DynamicDataConfiguration 3	Mandatory. See Table 11.
1623h	7423h	DynamicDataConfiguration 4	Mandatory. See Table 11.

Table 28: TxPDOs

PDO	Object	Name	Note
1A00h	6010h	RobotStatus	Mandatory. See Table 13.
1A01h	6015h	MotionStatus	Mandatory. See Table 14.
1A02h	6030h	TargetJointSet	Mandatory. See Table 15.
1A03h	6031h	TargetEndEffectorPose	Mandatory. See Table 16.
1A08h	6046h	TargetConfiguration	Mandatory. See Table 17.
1A09h	6050h	WRF	Mandatory. See Table 18.
1A0Ah	6051h	TRF	Mandatory. See Table 19.
1A10h	6060h	RobotTimestamp	Mandatory. See Table 20.
1A11h	6065h	SafetyStatus	Mandatory. See Table 21.
1A20h	6070h	DynamicData index 0	Mandatory. See Table 22.
1A21h	6071h	DynamicData index 1	Mandatory. See Table 23.
1A22h	6072h	DynamicData index 2	Mandatory. See Table 24.
1A23h	6073h	DynamicData index 3	Mandatory. See Table 25.

PDO data

Using the PDO data to control and monitor Mecademic robots with EtherCAT is explained in Section 5 of this manual.

The cyclic data format is the same for PROFINET, EtherNet/IP, and EtherCAT protocols. Therefore, it is easy to migrate a robot-controlling application on a controller/PLC between these different protocols.

Please refer to the robot's *ESI file* (page 100) for the list of cyclic input/output fields. Refer to Section 5 for instructions on how to use these cyclic fields.

Note that 16- and 32-bit integer values in the cyclic data use big-endian byte order. Some PLCs may need to be configured accordingly.

EtherNet/IP communication

Mecademic robots are compatible with the EtherNet/IP protocol. The MCS500 is certified by ODVA. A common industry standard, EtherNet/IP can be used with many different PLC brands. Tested to work at 10 ms, faster response times are also possible. Our robots typically use implicit (cyclic) messaging.

Refer to our Support Center for specific PLC examples.

Connection types

When using EtherNet/IP, you can connect several Mecademic robots in the same way as with TCP/IP. Either Ethernet port on the robot can be used. The robots can either be daisy-chained together or connected in a star pattern. The two ports on the Mecademic robot act as a switch in EtherNet/IP mode.

EDS file

Each EtherNet/IP slave device is described by an Electronic Data Sheet (EDS) file that describes its identity, capabilities, and cyclic payload. The EtherNet/IP controllers (PLC) use this file to properly identify detected EtherNet/IP slave devices, such as a Mecademic robot.

It can be found in the zip file that contains your robot's firmware update package

(Mcs500_vX.X.X.X.eds).

Forward open exclusivity

A Mecademic robot allows only one controlling connection at a time (either a TCP/IP connection or through an EtherNet/IP forward-open request).

If the robot is already being controlled, it will refuse a forward-open request with status error 0x106, Ownership Conflict, in EtherNet/IP. It will refuse a TCP/IP connection with error [3001]. However, the web interface can still be used in monitoring mode.

Enabling Ethernet/IP

The Ethernet/IP protocol can be enabled using the *Network configuration* panel in the MecaPortal configuration menu.

Alternatively, you can use the *EnableEtherNetIp()* (page 177) command, which can be entered in the MecaPortal code editor or sent to the robot via the *TCP/IP API* (page 18).

This is a persistent configuration and only needs to be set once.

Note that Ethernet/IP can remain permanently enabled, as it does not interfere with the use of the TCP/IP protocol, unlike EtherCAT and the *SwitchToEtherCAT* (page 207) command.

Cyclic data

Using cyclic data to control and monitor Mecademic robots with Ethernet/IP is explained in Section 5 of this manual.

The cyclic data format is the same for PROFINET, EtherNet/IP, and EtherCAT protocols. Therefore, it is easy to migrate a robot-controlling application on a controller/PLC between these different protocols.

Please refer to the robot's *EDS file* (page 106) for the list of cyclic input/output fields. Refer to Section 5 for instructions on how to use these cyclic fields.

Note that 16- and 32-bit integer values in the cyclic data use big-endian byte order. Some PLCs may need to be configured accordingly.

PROFINET communication

Mecademic robots are compatible with the PROFINET protocol, a common industry standard that can be used with many different PLC brands. The MCS500 is certified by PROFIBUS. Cyclic times up to 1 ms (though not as “hard-real-time” as EtherCAT) are possible. PROFINET—like EtherCAT or EtherNet/IP protocols—controls the robot using cyclic messaging (‘CR Input’ and ‘CR Output’ in PROFINET terms).

PROFINET conformance class

The Mecademic robots PROFINET stack conforms to class-A, as described in the *GSDML file* (page 115).

PROFINET limitations on Mecademic robots

Mecademic robots do not support the following PROFINET features:

- Startup mode: legacy startup mode (only advanced startup mode is supported).
- SNMP: part of PROFINET conformance class B (the robot supports class A only).
- DHCP: the robot does not support selecting DHCP mode via the PROFINET protocol. Note that configuring the robot to use DHCP mode remains possible through the MecaPortal.
- Fast startup.

Connection types

When using PROFINET, you can connect several Mecademic robots, just like with TCP/IP. Either Ethernet port on the robot can be used. The robots can be either daisy-chained together or connected in a star pattern.

Limitations when daisy-chaining robots

Please note that the two Ethernet ports on the robot act as an unmanaged Ethernet switch, not as a “PROFINET-aware” switch. In fact, this Ethernet switch will not respond to LLDP (Local Link Discovery Protocol) packets like a PROFINET-enabled switch would (instead, it forwards LLDP through the daisy chain). As a consequence, the LLDP protocol will not properly identify the network topology when the two Ethernet ports of the robots are connected (in a daisy-chain configuration, for example). Fortunately, this does not prevent the use of the PROFINET protocol, since daisy-chained robots will still be detected by the PROFINET controller.

If you need full network topology discovery using LLDP, we recommend connecting the robot to a PROFINET-enabled Ethernet switch rather than in a daisy chain.

PROFINET protocol over your Ethernet network

The PROFINET protocol uses non-IP packets to communicate real-time data over the Ethernet network. Please ensure that your Ethernet network and switches are properly forwarding these packets between the PROFINET controller (PLC) and the Mecademic robots.

Ethernet packets of type LLDP (0x88CC) are used for the LLDP protocol. This protocol makes it possible to discover the network topology.

Ethernet packets of type PN-DCP (0x8892) are used for the DCP protocol (Discovery and Configuration Protocol). This protocol is used to discover PROFINET devices on the network. It is also used to set host names and IP addresses to detect PROFINET devices.

Ethernet packets of type PROFINET RT (0x8892) are used for PROFINET cyclic data exchanges between the Mecademic robots and the PROFINET controller (PLC).

Enabling PROFINET

The PROFINET protocol can enabled using the *Network configuration* panel in the MecaPortal configuration menu.

Alternatively, you can use the *EnableProfinet()* (page 178) command, which can be entered in the MecaPortal code editor or sent to the robot via the *TCP/IP API* (page 18).

This is a persistent configuration and only needs to be set once.

Note that PROFINET can remain permanently enabled, as it does not interfere with the use of the TCP/IP protocol, unlike EtherCAT and the *SwitchToEtherCAT* (page 207) command.

Also note that LLDP forwarding on the robot is enabled only when PROFINET is enabled on the robot (so it will not be possible to detect a robot using LLDP until PROFINET is enabled on it).

Exclusivity of AR

Only one AR (Application Relationship) can be established with the robot. Only one PROFINET controller (PLC) can control a Mecademic robot.

Controlling the robot is also exclusive between TCP/IP, EtherNet/IP, and PROFINET protocols. The first connection to the robot on any of these cyclic protocols will prevent any other connections on any protocol.

If a PROFINET connection request is refused because the robot is already being controlled by another PROFINET controller (PLC), the refused connect request will be returned with standard error codes and the following values:

- Error code “connect” (0xDB)
- Error decode “PNIO” (0x81)
- Error1 “CMRPC” (0x40)
- Error2 “No AR resource” (0x04)

If a PROFINET connection request is refused because the robot is already being controlled by another protocol (TCP/IP or EtherNet/IP), the refused connect request will be returned with a vendor-specific error code and the following values:

- Error code “connect” (0xDB)
- Error decode “Manufacturer specific” (0x82)
- Error1 “Mecademic Access denied” (0x11)

GSDML file

Each Profinet slave device is described by a GSDML (.xml) file that describes its identity, capabilities, cyclic payload, PROFINET Modules and SubModules that it supports. The PROFINET controllers (PLC) use this file to properly identify detected PROFINET slave devices, such as a Mecademic robot.

It can be found in the zip file that contains your robot's firmware update package

(GSDML-V2.42-Mecademic-mcs500-XXXXXXXX.xml).

Since the GSDML file contains necessary information to identify and list the robot capabilities, this manual provides only a quick summary of the GSDML file.

Robot modules and sub-modules

The robot supports only one module and one sub-module, fixed in a predefined slot.

- Module: “RobotControlModule”, ID=0x32, fixed in slot 1
- Sub-module: ID=0x132, fixed in sub-slot 1

This module provides fixed cyclic data input and output, used to control and monitor the robot.

Cyclic data

Using cyclic data to control and monitor Mecademic robots with PROFINET is explained in Section 5 of this manual.

The cyclic data format is the same for PROFINET, EtherNet/IP, and EtherCAT protocols. Therefore, it is easy to migrate a robot-controlling application on a controller/PLC between these different protocols.

Please refer to the robot's *GSDML file* (page 115) for the list of cyclic input/output fields. Refer to Section 5 for instructions on how to use these cyclic fields.

Note that 16- and 32-bit integer values in the cyclic data use big-endian byte order. Some PLCs may need to be configured accordingly.

Alarms

Mecademic robots will not generate any PROFINET alarms. Any alarm or error condition will be reported by the robot through the corresponding cyclic data fields. This allows the robots to behave the same across various cyclic protocols (such as PROFINET, EtherNet/IP, or EtherCAT).

Refer to Section 5 for more information about robot status and error states reported in the cyclic input data.

Troubleshooting

Log files

From the MecaPortal, you can download three different log files that record state changes, commands sent, responses received, and other data, as described below:

- *User log* (page 341): A simplified log containing user-friendly traces of major events (e.g., robot activation, movement, E-Stop activation).
- *Robot log* (page 341): A more detailed version of the user log, intended primarily for the support team.
- *Detailed event log* (page 339): This file mirrors the content of the event log panel in the MecaPortal when in detailed mode, i.e., when all of the options are selected in the event log panel settings menu, .

Robot log files are stored on the robot's disk. The user log is also saved on the disk, except for the Meca500 robot, where it is volatile. When a log file exceeds 10 MB, a new file is created, and older files are moved to the backup (see next subsection). As a result, some log files may contain only a few lines of data, in which case you may need to check the robot's backup.

The detailed event log is volatile and not saved, meaning it will be lost after a robot reboot. It is also a circular buffer, storing only the most recent data.

You can enable additional details in the user and robot logs using the commands *LogTrace* (page 180) and *LogUserCommands* (page 181).

Finally, the user and robot log files can be downloaded from the Log files tab of the configuration menu, , in the MecaPortal. The detailed event log can be downloaded by clicking the icon in the event log panel of the MecaPortal.

Backup files

The full backup of the robot is a TAR archive file that contains the complete robot configuration, the latest user and robot log files, and their archived versions. You can download the full backup from the “Log files” tab of the configuration menu in the MecaPortal by clicking the “Get all log and configuration files” button with your primary mouse button.

Alternatively, to download a smaller backup file without the archived user and robot logs, click the same button with your secondary mouse button.

Motion commands

Motion commands are used to generate a trajectory for the robot. When a Mecademic robot receives a motion command, it places it in a motion queue. The command will be run once all preceding motion commands have been executed. In other words, motion commands are synchronous.

Most motion commands have arguments, but not all have default values (e.g., the argument for the command *Delay* (page 125)). The arguments for most motion commands are IEEE-754 floating-point numbers, separated by commas and spaces (optional).

Motion commands do not generate a direct response and the only way to know exactly when a certain motion command has been executed is to use the command *SetCheckpoint* (page 145) (a response is then sent when the checkpoint has been reached).

The robot sends an end-of-movement message (*EOM* (page 339), code 3004) whenever it has stopped moving for at least 1 ms, if this option is activated with *SetEom* (page 189). The EOM message is sent whether or not all queued commands have been executed.

Furthermore, by default, the robot sends an end-of-block message (*EOB* (page 339), code 3012) every time the robot has stopped moving AND its motion queue is empty. For example, if both EOM and EOB messages are enabled, and you immediately send a *MoveJoints* (page 126), *SetTrf* (page 166), *MovePose* (page 138) and *Delay* (page 125) command one after the other, the robot will send an EOM message when it has stopped, and then an EOB message as soon as the delay has elapsed.

Note that EOB and EOM messages should NOT be used to detect whether a sequence of motion commands has been executed: communication delays mean that the robot may send an EOB message when it has finished processing all the previously received commands, even though there are more commands stacking up to be processed in the communication channel (between robot and application). Using the *SetCheckpoint* (page 145) command is the best way to follow the sequence of execution of commands. Finally, motion commands can generate errors, explained in Section 4.

The motion commands are listed below in several groups.

Joint-space, position-mode movement commands

- *MoveJoints* (page 126)
- *MoveJointsRel* (page 128)
- *MovePose* (page 138)
- *MoveJump* (page 130)
- *SetMoveJumpApproachVel* (page 156)
- *SetMoveJumpHeight* (page 157)

- *SetJointAcc* (page 150)
- *SetJointVel* (page 151)

Cartesian-space, position-mode movement commands

- *MoveLin* (page 132)
- *MoveLinRelTrf* (page 134)
- *MoveLinRelWrf* (page 135)
- *SetCartAcc* (page 142)
- *SetCartAngVel* (page 143)
- *SetCartLinVel* (page 144)

Velocity-mode movement commands

- *MoveJointsVel* (page 129)
- *MoveLinVelTrf* (page 136)
- *MoveLinVelWrf* (page 137)
- *SetVelTimeout* (page 167)

Robot posture and turn configuration commands

- *SetAutoConf* (page 139)
- *SetAutoConfTurn* (page 140)
- *SetConf* (page 147)
- *SetConfTurn* (page 148)

Other motion commands

- *Delay* (page 125)
- *SetBlending* (page 141)
- *SetCheckpoint* (page 145)
- *SetJointVelLimit* (page 152)
- *SetMoveDuration* (page 153)
- *SetMoveDurationCfg* (page 154)
- *SetMoveMode* (page 159)
- *SetPayload* (page 160)
- *SetTorqueLimits* (page 162)

Delay

This command is used to add a time delay after a motion command. In other words, the robot completes all movements sent before the *Delay* (page 125) command and stops temporarily. (In contrast, the *PauseMotion* (page 182) command interrupts the motion as soon as received by the robot.)

Syntax

Delay(t)

Arguments

- t: desired pause duration in seconds.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *Delay* (page 125) command is represented by *MotionCommandID* 6. See Section 5 for more details.

MoveJoints

This command makes the robot simultaneously move all its joints to the target joint set, as fast as possible but subject to the limits set by the commands *SetJointVel* (page 151) and *SetJointVelLimit* (page 152). All joints start and stop moving at the same time, so there is generally only one joint that moves at the joint velocity indirectly specified in *SetJointVel* (page 151) and *SetJointVelLimit* (page 152). The robot takes a linear path in the joint space, but nonlinear in the Cartesian space. Therefore, the TCP trajectory is not easily predictable (Figure 8). Finally, with *MoveJoints* (page 126), the robot can cross singularities without any problem.

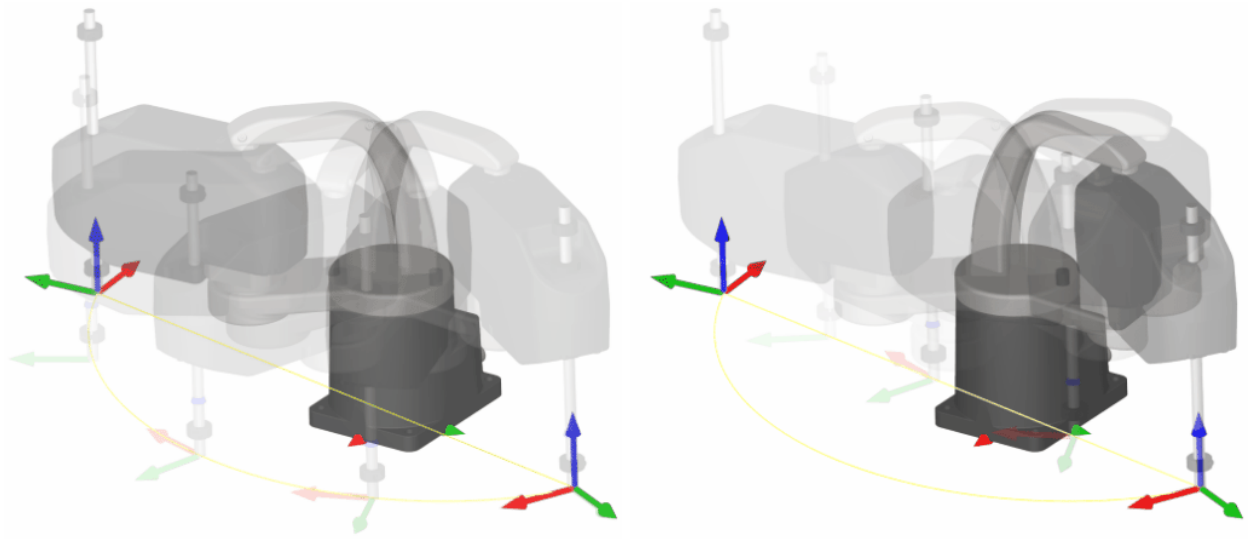
Syntax

MoveJoints(θ_1,θ_2,d_3,θ_4)

Arguments

- the target position of each joint, in degrees (for the revolute joints) and in mm (for the linear joint).

The default ranges for the robot joints are given in *technical-specifications-MCS500* of the robot's user manual. Note that these ranges can be further limited with the command *SetJointLimits* (page 190). The target joints position must be within the allowable joint limits or else the command will not be executed.



(a) MoveJoints (b) MoveLin

Figure 8: An example showing the difference between a path that is linear in joint space (often referred to as a point-to-point motion) and one that is linear in Cartesian space (the TCP follows a line)

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveJoints* (page 126) command is represented by `MotionCommandID` 1. See Section 5 for more details.

MoveJointsRel

This command has the exact behavior as the *MoveJoints* (page 126) command, but instead of accepting the desired (target) joint set as arguments, it takes the desired relative joint displacements. The command is particularly useful when you need to displace certain joints a certain amount, but you do not know the current joint set and wish to avoid having to use the command *GetRtTargetJointPos* (page 270).

Syntax

MoveJointsRel($\Delta\theta_1$,$\Delta\theta_2$,Δd_3,$\Delta\theta_4$)

Arguments

- the desired relative displacement of each joint, in degrees (for the revolute joints) and in mm (for the linear joint). The value of each of the arguments can be positive, negative or zero.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveJointsRel* (page 128) command is represented by *MotionCommandID* 29. See Section 5 for more details.

MoveJointsVel

This displaces the robot's joints simultaneously at the specified joint speeds. All joint movements begin and end at the same time. The robot will decelerate to a complete stop after a period defined by the command *SetVelTimeout* (page 167), unless a subsequent *MoveJointsVel* (page 129) command is issued. Unlike position-mode motion commands, the *MoveJointsVel* (page 129) command *does not generate motion errors when a joint limit is reached*; instead, the robot halts slightly before the limit. Additionally, as with all MoveJoints* commands, the robot can cross singularities when using the *MoveJointsVel* (page 129) command.

Syntax

MoveJointsVel(ω_1,ω_2,v_3,ω_4)

Arguments

- the desired velocity of each joint, in °/s (for the revolute joints) and in mm/s (for the linear joint). The value of each of the arguments can be positive, negative or zero.

The maximum joint velocities are given in *technical-specifications-MCS500* of the robot's user manual.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveJointsVel* (page 129) command is represented by *MotionCommandID* 21. See Section 5 for more details.

MoveJump

With this command, the robot moves up/down its end-effector a certain distance (retract motion along a line and without changing its orientation), then moves it to a pose that is a certain distance over/under the target pose (lateral motion), and finally moves it down/up to the target pose (approach linear motion without changing its orientation), as illustrated in Figure 9. The *MoveJump* (page 130) command is highly optimized for fast pick and place motion and results in much faster cycle times than when using a simple sequence of *MoveLin* (page 132) - *MovePose* (page 138) - *MoveLin* (page 132) commands.

The *MoveJump* (page 130) command works only in velocity-base position mode (see Section 3).

The parameters defining the trajectory of the end-effector in a *MoveJump* (page 130) motion are set by the commands *SetMoveJumpHeight* (page 157) and *SetMoveJumpApproachVel* (page 156) (see Figure 10).

Syntax

MoveJump(x,y,z,γ)

Arguments

- *x*, *y*, *z*: the target position for the TRF with respect to the WRF, in mm;
- *γ*: the target orientation of the TRF about its z-axis with respect to the WRF, in degrees.

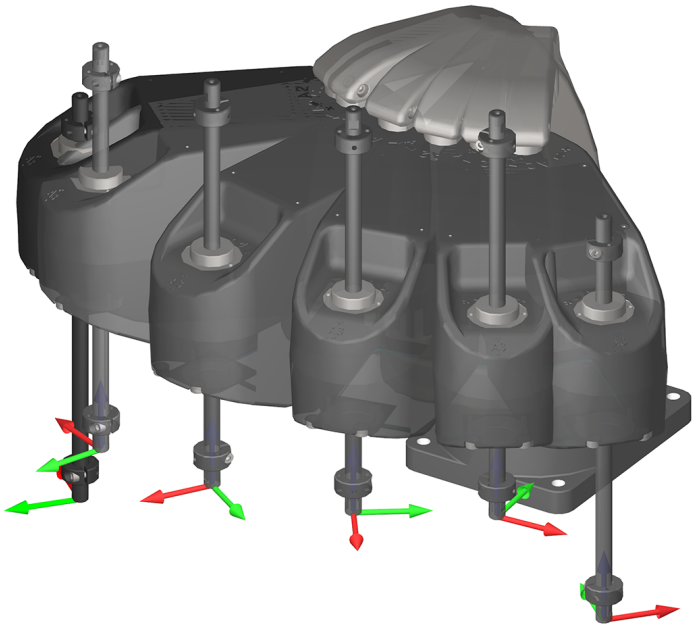


Figure 9: End-effector motion when using the MoveJump command

Further details

As with the *MovePose* (page 138) command, the joint set corresponding to the target pose is calculated according to the desired robot posture and turn configurations, if such were set, or the one that is fastest to reach. Also, as with the *MovePose* (page 138) command, if the complete motion cannot be performed due to joint limits, it will not even start, and an error will be generated. Note that since the robot uses the optimal (quickest) path in joint space, in the lateral motion, *the end-effector follows a complex non-linear path*. Finally, the speed and the acceleration during the *MoveJump* (page 130) motion are defined by the *SetJointVel* (page 151) and *SetJointAcc* (page 150) commands.

Note

The *MoveJump* (page 130) command is a joint-space command like *MovePose* (page 138). Therefore, if you execute two successive *MoveJump* (page 130) commands or a *MoveJump* (page 130) followed by a *MovePose* (page 138), you must first deactivate the blending, or else the *MoveJump* (page 130) will not be completed as planned.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveJump* (page 130) command is represented by `MotionCommandID` 45. See Section 5 for more details.

MoveLin

This command makes the robot move its end-effector, so that its TRF ends up at a target pose with respect to the WRF while the TCP moves along a linear path in Cartesian space, as illustrated in Figure 8b. If the target (desired) orientation of the TRF is different from the initial orientation, the orientation will be modified along the path using a minimum-torque path.

Syntax

MoveLin(x,y,z,γ)

Arguments

- x, y, z: the target position for the TRF with respect to the WRF, in mm;
- γ: the target orientation of the TRF about its z-axis

  with respect to the WRF, in degrees.

Further details

It is physically impossible to follow a linear path with this command, while changing the configuration (i.e., crossing a singularity).

If you specify a desired turn configuration, the *MoveLin* (page 132) command will be executed only if the initial and final robot positions have the same turn configuration as the desired one.

If the complete motion cannot be performed due to singularities or joint limits, it will not even start, and an error will be generated. Similarly, the robot will not accept the *MoveLin* (page 132) command if the required end-effector reorientation is exactly 180°, because there could be two possible paths.

Use the *MoveLin* (page 132) command only when precise linear motion of the TCP is required. For most cases, moving the robot between positions is faster using the *MoveJoints* (page 126) or *MovePose* (page 138) commands.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveLin* (page 132) command is represented by `MotionCommandID` 3. See Section 5 for more details.

MoveLinRelTrf

This command has the same behavior as the *MoveLin* (page 132) command, but allows a desired pose to be specified relative to the current pose of the TRF. Thus, the arguments x, y, z, and ɣ represent the desired pose of the TRF with respect to the current pose of the TRF (i.e., the pose of the TRF just before executing the *MoveLinRelTrf* (page 134) command).

As with the *MoveLin* (page 132) command, if the complete motion cannot be performed, it will not even start and an error will be generated.

Syntax

MoveLinRelTrf(x,y,z,ɣ)

Arguments

- x, y, z: the position coordinates, in mm;
- ɣ: the orientation angle, in degrees.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveLinRelTrf* (page 134) command is represented by *MotionCommandID* 4. See Section 5 for more details.

MoveLinRelWrf

This command is similar to the *MoveLinRelTrf* (page 134) command, but instead of defining the desired pose with respect to the current pose of the TRF it is defined with respect to a reference frame that has the same orientation as the WRF but its origin is at the current position of the TCP.

Syntax

MoveLinRelWrf(x,y,z,γ)

Arguments

- x, y, z: the position coordinates, in mm;
- γ: the orientation angle, in degrees.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveLinRelWrf* (page 135) command is represented by *MotionCommandID* 5. See Section 5 for more details.

MoveLinVelTrf

This command moves the robot's TRF at the specified Cartesian velocity, defined relative to the TRF, or at a lower velocity if limited by joint velocity constraints (see *SetJointVelLimit* (page 152)). If needed, the joint velocities are proportionally reduced to ensure none exceed their specified limits.

The robot will decelerate to a complete stop after the duration specified by the command *SetVelTimeout* (page 167), unless a subsequent *MoveLinVelTrf* (page 136) or *MoveLinVelWrf* (page 137) command is issued, and. Additionally, the motion will stop if a *PauseMotion* (page 182) command is sent or if a motion limit is reached.

Note that this command, unlike position-mode motion commands, *does not generate motion errors when a joint limit (including the desired turn configuration) or an uncrossable singularity is encountered*. Instead, the robot simply stops before reaching the limit.

Syntax

MoveLinVelTrf($\dot{x}$,$\dot{y}$,$\dot{z}$,ω_z)

Arguments

- $\dot{x}$, $\dot{y}$, $\dot{z}$: the components of the linear velocity of the TCP expressed in the TRF, in mm/s;
- ω_z:

  the angular velocity of the TRF expressed in the TRF, in °/s.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveLinVelTrf* (page 136) command is represented by *MotionCommandID* 23. See Section 5 for more details.

MoveLinVelWrf

This command moves the robot's TRF at the specified Cartesian velocity, defined relative to the WRF, or at a lower velocity if limited by joint velocity constraints (see *SetJointVelLimit* (page 152)). If needed, the joint velocities are proportionally reduced to ensure none exceed their specified limits.

The robot will decelerate to a complete stop after the duration specified by the command *SetVelTimeout* (page 167), unless a subsequent *MoveLinVelTrf* (page 136) or *MoveLinVelWrf* (page 137) command is issued, and. Additionally, the motion will stop if a *PauseMotion* (page 182) command is sent or if a motion limit is reached.

Note that this command, unlike position-mode motion commands, *does not generate motion errors when a joint limit (including the desired turn configuration) or an uncrossable singularity is encountered*. Instead, the robot simply stops before reaching the limit.

Syntax

MoveLinVelWrf(ẋ,ẏ,ż,ω_z)

Arguments

- ẋ, ẏ, ż: the components of the linear velocity of the TCP with respect to the WRF, in mm/s;
- ω_z:

  the angular velocity of the TRF with respect to the WRF, in °/s.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MoveLinVelWrf* (page 137) command is represented by *MotionCommandID* 22. See Section 5 for more details.

MovePose

This command moves the robot's TRF to a specified pose relative to the WRF. The robot controller calculates all possible joint sets corresponding to the target pose, including those associated with singular robot postures. It then selects the target joint set based on the specified robot posture and turn configurations, if provided, or the one requiring the least time to reach.

The selected joint configuration is executed internally using a *MoveJoints* (page 126) command. As a result, all joint rotations start and stop simultaneously and move as quickly as possible, subject to the limits defined by the *SetJointVel* (page 151) and *SetJointVelLimit* (page 152) commands. The resulting motion is linear in joint space but nonlinear in Cartesian space, meaning the TCP's path to its final destination is not easily predictable.

Syntax

MovePose(x,y,z,γ)

Arguments

- x, y, z: the target position for the TRF with respect to the WRF, in mm;
- γ: the target orientation of the TRF about its z-axis

  relative to the WRF, in degrees.

Further details

With this command, the robot can transition through or begin/end at singular robot postures without any issues. However, as with the *MoveJoints* (page 126) command, if the complete motion cannot be executed due to joint limits, the motion will not start, and an error will be generated.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *MovePose* (page 138) command is represented by *MotionCommandID* 2. See Section 5 for more details.

SetAutoConf

This command enables or disables the automatic posture configuration selection, to be observed in the *MovePose* (page 138) and MoveLin* commands. This automatic selection, in conjunction with the turn configuration selection (see Section 3 and Section 3), allows the controller to choose the “closest” joint set corresponding to the target pose.

Syntax

SetAutoConf(e)

Arguments

- e: enable (1) or disable (0) automatic posture configuration selection.

Default values

The automatic posture configuration selection is enabled by default. If you disable it, the new desired posture configuration will be the one corresponding to the current robot position, i.e., the one after all preceding motion commands have been completed. Note, however, that if you disable the automatic posture configuration selection in a singular robot posture (i.e., when $\theta_3 = 0$), the new desired configuration will be {1}. Finally, the automatic robot configuration selection is also disabled as soon as the robot receives the command *SetConf* (page 147).

Usage restrictions

This command is added to the robot’s motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetAutoConf* (page 139) command is represented by *MotionCommandID* 16. See Section 5 for more details.

SetAutoConfTurn

This command enables/disables the automatic turn selection for the last joint of the robot (see Section 3 and Section 3). It affects the *MovePose* (page 138) command and all MoveLin* commands. When the automatic turn selection is enabled, and a *MovePose* (page 138) command is executed, the last joint will always take the shortest path, and rotate no more than 180°. In the case of a MoveLin* command, however, enabling the automatic turn selection simply allows the change of turn configuration along the linear move.

Syntax

SetAutoConfTurn(e)

Arguments

- e: enable (1) or disable (0) automatic turn configuration selection.

Default values

SetAutoConfTurn (page 140) is enabled by default. If you disable the automatic turn selection, the new desired turn configuration will be the one corresponding to the current robot position, i.e., the one after all preceding motion commands have been completed. Finally, the automatic turn configuration selection is also disabled as soon as the robot receives the command *SetConfTurn* (page 148).

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetAutoConfTurn* (page 140) command is represented by *MotionCommandID* 26. See Section 5 for more details.

SetBlending

This command enables/disables the robot's blending feature (Section 3). Note that there is blending only between consecutive movements with the position-mode joint-space commands *MoveJoints* (page 126), *MoveJointsRel* (page 128), *MovePose* (page 138) and *MoveJump* (page 130), or between consecutive movements with the position-mode Cartesian-space commands *MoveLin* (page 132), *MoveLinRelTrf* (page 134) and *MoveLinRelTrf* (page 134). For example, there will never be blending between the trajectories of a *MovePose* (page 138) command followed by a *MoveLin* (page 132) command.

Syntax

SetBlending(p)

Arguments

- p: percentage of blending, ranging from 0 (blending disabled) to 100.

Default values

Blending is enabled at 100% by default.

Further details

A blending of 100% corresponds to a blending that occurs 100% of the duration of the acceleration and deveLeration periods, controlled by *SetJointAcc* (page 150), *SetCartAcc* (page 142) and *SetJointVelLimit* (page 152).

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetBlending* (page 141) command is represented by *MotionCommandID* 7. See Section 5 for more details.

SetCartAcc

This command limits the Cartesian acceleration (both linear and angular) of the TRF relative to the WRF during movements resulting from Cartesian-space commands (see Figure 7). Note that using this command causes the robot to come to a complete stop, even if blending is enabled.

Syntax

SetCartAcc(p)

Arguments

- p: percentage of maximum acceleration of the TRF, ranging from 0.001 to 600.

Default values

The default end-effector acceleration limit is 50%.

Further details

When using large accelerations and a heavy payload, we recommend using the *SetPayload* (page 160) command. This allows the robot to predict the required torque with greater precision, improving path tracking accuracy. It also helps reduce the required margins when using torque limits (see the *SetTorqueLimitsCfg* (page 164)).

Note that the argument of this command is exceptionally limited to 600. This is because in firmware 8, a change was made to allow the robot to accelerate much faster. For backwards compatibility, however, 100% now corresponds to 100% in firmware 7 and before.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetCartAcc* (page 142) command is represented by *MotionCommandID* 12. See Section 5 for more details.

SetCartAngVel

This command sets the *desired and maximum* angular velocity of the robot TRF with respect to its WRF. It only affects the movements generated by the *MoveLin* (page 132), *MoveLinRelTrf* (page 134) and *MoveLinRelWrf* (page 135) commands. It has impact on these movement commands only if the move mode is velocity-based (see *SetMoveMode* (page 159)).

Syntax

SetCartAngAcc(ω)

Arguments

- ω: TRF angular velocity limit, in °/s, ranging from 0.001 to 5,000.

Default values

The default end-effector angular velocity limit is 45°/s.

Note

The actual angular velocity may be lower (but never higher) than requested at certain portions or throughout the linear path to ensure compliance with the joint velocity limits set by the *SetJointVelLimit* (page 152) command and the linear velocity limit set by the *SetCartLinVel* (page 144) command.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetCartAngVel* (page 143) command is represented by *MotionCommandID* 10. See Section 5 for more details.

SetCartLinVel

This command sets the *desired and maximum* linear velocity of the robot TRF with respect to its WRF. It only affects the movements generated by the *MoveLin* (page 132), *MoveLinRelTrf* (page 134) and *MoveLinRelWrf* (page 135) commands. It has impact on these movement commands only if the move mode is velocity-based (see *SetMoveMode* (page 159)).

Syntax

SetCartLinAcc(v)

Arguments

- v: TRF linear velocity limit, in mm/s, ranging from 0.001 to 5,000.

Default values

The default end-effector angular velocity limit is 150 mm/s.

Note

The actual TCP velocity may be lower (but never higher) than requested at certain portions or throughout the linear path to ensure compliance with the joint velocity limits set by the *SetJointVelLimit* (page 152) command and the linear velocity limit set by the *SetCartAngVel* (page 143) command.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetCartLinVel* (page 144) command is represented by *MotionCommandID* 11. See Section 5 for more details.

SetCheckpoint

This command defines a checkpoint in the motion queue. Thus, if you send a sequence of motion commands to the robot, then the command *SetCheckpoint* (page 145), then other motion commands, you will be able to know the exact moment when the motion command sent just before the *SetCheckpoint* (page 145) command was completed. At that precise moment, the robot will send back the response [3030][n] (on both ports), where n is a positive integer number defined by you. If blending was activated, the checkpoint response will be sent somewhere along the blending. If a checkpoint is the last queued command, in the absence of blending with another command, the checkpoint response will be sent once the robot has come to a stop (along with an EOB). Finally, note that you can use the same checkpoint number multiple times.

Syntax

SetCheckpoint(n)

Arguments

- n: an integer number, ranging from 1 to 8,000.

Responses

- [3030][n]
  - Sent when the checkpoint was reached.
- [3040][n]
  - Sent when the checkpoint was discarded and will never be reached (due to motion cleared, robot deactivated, error, safety stop. etc.)

Note

Using a checkpoint is the only reliable method to confirm whether a specific motion sequence has been completed. Do not rely on the EOM or EOB messages, as these may be received well before the motion or sequence is finished—or not received at all if these messages are not enabled.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetCheckpoint* (page 145) command is represented by *MotionCommandID* 17. See Section 5 for more details.

SetConf

This command sets the desired posture configuration to be observed in the *MovePose* (page 138) and MoveLin* commands (see Section 3 and Section 3). When a desired posture configuration is set, a *MovePose* (page 138) command will execute only if the final robot position can be in the desired posture configuration. In contrast, a MoveLin* command will execute only if the initial robot position already is in the desired posture configuration, and the final robot position is also in the desired posture configuration.

The posture configuration can be automatically selected, when executing a *MovePose* (page 138) or MoveLin* command, by using the *SetAutoConf* (page 139) command. Using *SetConf* (page 147) automatically disables the automatic posture configuration selection.

Syntax

SetConf(c_e)

Arguments

- c_e: elbow configuration parameter, either -1 or 1.

Default Values

Automatic posture configuration selection is enabled by default (see *SetAutoConf* (page 139)); when the robot starts, there is no default desired posture configuration. The desired posture configuration must be specified using the *SetConf* (page 147) command or the *SetAutoConf(0)* (page 139) command. The latter sets the desired posture configuration to the one of the current robot posture.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetConf* (page 147) command is represented by *MotionCommandID* 15. See Section 5 for more details.

SetConfTurn

This command sets the desired turn configuration for the last joint, c_t, to be observed in the *MovePose* (page 138) and MoveLin* commands (see Section 3 and Section 3). When c_t is set, a *MovePose* (page 138) command is executed only if the final robot position can be in the desired turn configuration. In contrast, when a c_t is set, a MoveLin* command will execute only if the final robot position can be — and the initial robot position already is — in the desired turn configuration.

The turn configuration can be automatically selected, when executing a *MovePose* (page 138) or MoveLin* command, by using the *SetAutoConf* (page 139) command. Using *SetConfTurn* (page 148) automatically disables the automatic turn configuration selection.

Syntax

SetConfTurn(c_t)

Arguments

- c_t: turn configuration, an integer between −10 and 10.

The turn configuration parameter defines the desired range for joint 4, according to the following inequality: $-180° + c_t360° < \theta_4 \leq 180° + c_t360°$.

Default values

There is no default desired turn configuration. The only way to set a desired turn configuration is to specify it with the command *SetConfTurn* (page 148) or to execute the command *SetAutoConfTurn(0)* (page 140). The latter sets the desired turn configuration to the one of the current position of the last joint.

Further details

This command is primarily useful if your end-effector is wired. In such as case, limit the can range of the last joint appropriately using the *SetJointLimits* (page 190) command. However, since the cabling will not be configured identically when the last joint is at a 5° versus 365°, for example, it is advisable to specify which of these two alternatives is preferred for a given pose. This can be achieved using the command *SetConfTurn* (page 148), with the appropriate turn configuration as argument.

If using a cable-less end-effector, then the automatic turn configuration should never be disabled.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetConfTurn* (page 148) command is represented by `MotionCommandID` 25. See Section 5 for more details.

SetJointAcc

This command limits the acceleration of the joints during movements resulting from joint-space commands (see Figure 7). Note that this command makes the robot come to stop, even if blending is enabled.

Syntax

SetJointAcc(p)

Arguments

- p: percentage of maximum acceleration of the joints, from 0.001 to 100.

Default values

The default joint acceleration limit is 100%.

Further details

When using large accelerations and a heavy payload, we recommend using the *SetPayload* (page 160) command. This allows the robot to predict the required torque with greater precision, improving path tracking accuracy. It also helps reduce the required margins when using torque limits (see the *SetTorqueLimitsCfg* (page 164)).

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetJointAcc* (page 150) command is represented by *MotionCommandID* 9. See Section 5 for more details.

SetJointVel

This command specifies the desired velocities of the robot joints during movements generated by the *MovePose* (page 138), *MoveJoints* (page 126), and *MoveJointsRel* (page 128) commands. It has impact on these movement commands only if if the move mode is velocity-based (see *SetMoveMode* (page 159)).

Syntax

SetJointVel(p)

Arguments

- p: percentage of the top rated joint velocities, ranging from 0.001 to 100.

Default values

By default, p = 25.

Further details

Note that the value of p is overridden by the argument of the command *SetJointVelLimit(p_o)* (page 152) if $p_o < p$. Also, it is not possible to limit the velocity of only one joint. With *SetJointVel* (page 151) and *SetJointVelLimit* (page 152), the maximum velocities of all joints are reduced proportionally.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetJointVel* (page 151) command is represented by *MotionCommandID* 8. See Section 5 for more details.

SetJointVelLimit

The *SetJointVelLimit* (page 152) overrides the default joint velocity limits. Unlike the *SetJointVel* (page 151) command, this command affects the movements generated by *all* Move* commands (even the MoveLinVel* ones).

Syntax

SetJointVelLimit(p_o)

Arguments

- p_o: percentage of the top rated joint velocities, ranging from 0.001 to 100.

Default values

By default, p_o = 100.

Further details

As of firmware 10.3, when the argument of *SetJointVelLimit* (page 152) is less than 100, the robot will optimize its joint accelerations in the case of slower movements.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetJointVelLimit* (page 152) command is represented by *MotionCommandID* 33. See Section 5 for more details.

SetMoveDuration

When the move mode has been set to time-based with the command *SetMoveMode(1)* (page 159), the motion-queue *SetMoveDuration* (page 153) sets the desired duration for the movement resulting from every subsequent position-mode command (except *MoveJump* (page 130)). The duration does not include the acceleration and deceleration phases.

Syntax

SetMoveDuration(t)

Arguments

- t: duration in seconds.

If the duration is 0, the robot will move as fast as possible, but only if the severity set with *SetMoveDurationCfg* (page 154) is 0 or 1.

Default values

By default, the duration is 3 seconds.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetMoveDuration* (page 153) command is represented by *MotionCommandID* 51. See Section 5 for more details.

SetMoveDurationCfg

This motion-queue command specifies what happens when a move command cannot meet the desired duration set by the *SetMoveDuration* (page 153) command, in time-based move mode.

For joint-space moves, this occurs when one or more joints would need to exceed their maximum velocity (*SetJointVelLimit* (page 152)). For linear moves, the robot may need to slow down in certain parts of the path due to joints reaching their velocity limits, such as near singularities.

Syntax

SetMoveDurationCfg(s)

Arguments

- s: severity, with
  - 0 for silent mode (no warning),
  - 1 for generating a warning message in the robot logs (also in MecaPortal), indicating the shortest possible duration for the movement command that failed to meet the desired duration,
  - 4 for generating an error with a code [3051], also indicating the shortest possible duration for the movement command that failed to meet the desired duration.

Default values

By default, s = 4.

Further details

Time scaling (*SetTimeScaling* (page 202)), recovery mode (Section 3) and manual mode may extend the move duration beyond the requested time when using time-based move mode. However, validation (if severity is set to 1 or 4 with *SetMoveDurationCfg* (page 154)) is performed regardless of whether time scaling, recovery mode, or manual mode has been applied. This ensures that programs do not need to be modified to run in reduced speed (time scaling), recovery, or manual mode.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetMoveDurationCfg* (page 154) command is represented by *MotionCommandID* 50. See Section 5 for more details.

SetMoveJumpApproachVel

This command is intended for reducing the speed during the initial and final moments of the *MoveJump* (page 130) motion (see Figure 10).

Syntax

SetMoveJumpApproachVel(v_{start}*,*p_{start}*,*v_{end}*,*p_{end}*)*

Arguments

- v_{start}: maximum allowed vertical speed near the start pose, in mm/s, from 0.001 to 700;
- p_{start}: initial portion of the retreat motion during which v_{start} is applied, in mm, from 0 to 102;
- v_{end}: maximum allowed vertical speed near the end pose, in mm/s, from 0.001 to 700;
- p_{end}: final portion of the approach motion during which v_{start} is applied, in mm, from 0 to 102.

Default values

By default, $v_{start} = v_{end} = 10$ and $p_{start} = p_{end} = 1$.

Further details

Note that if $p_{start} \geq |h_{start}|$, then the complete retract vertical motion will be limited in speed to v_{start}. Similarly, if $p_{end} \geq |h_{end}|$, then the complete approach vertical motion will be limited in speed to v_{end}. Also, if v_{start} or v_{end} is larger than the speed resulting from the *SetJointVel* (page 151) command, it will be ignored.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetMoveJumpApproachVel* (page 156) command is represented by *MotionCommandID* 47. See Section 5 for more details.

SetMoveJumpHeight

This command prescribes the exact distances the end-effector must move up or down, with a pure vertical translational motion, during the vertical portions of the *MoveJump* (page 130) movement. It also prescribes the minimum and maximum allowed heights for the lateral motion (see Figure 10).

Syntax

SetMoveJumpHeight(h_{start}*,*h_{end}*,*h_{min}*,*h_{max}*)*

Arguments

- h_{start}: height of the initial pure vertical translation, in mm, from −102 to 102;
- h_{end}: height of the final pure vertical translation, in mm, from −102 to 102;
- h_{min}: minimum height to reach while performing the lateral motion, with respect to the highest (if h_{start} and h_{end} are positive) or lowest (if h_{start} and h_{end} are negative) between the start and end poses, in mm, from −102 to 102;
- h_{max}: maximum height to reach while performing the lateral motion, with respect to the highest (if h_{start} and h_{end} are positive) or lowest (if h_{start} and h_{end} are negative) between the start and end poses, in mm, from −102 to 102.

Note

The direction of the heights (positive or negative) is with respect to the z-axis of the BRF. Use negative values if your robot is mounted upside-down (and your tool is installed at the flange farthest from the robot base).

Default values

By default, $h_{start} = h_{end} = 10$, $h_{min} = 0$, $h_{max} = 102$. The default values for h_{min} and h_{max} give full freedom to choose the optimal (quickest) path between the start and end poses. You may change h_{min} and h_{max} to avoid obstacles between the start and end poses, but be aware that this may result in slower (suboptimal) cycle times. Also, note that the highest point during the lateral motion can happen anywhere, not necessary in the middle. In addition, note that changing the joint velocities with the command *SetJointVel* (page 151) will also change the profile of the lateral motion.

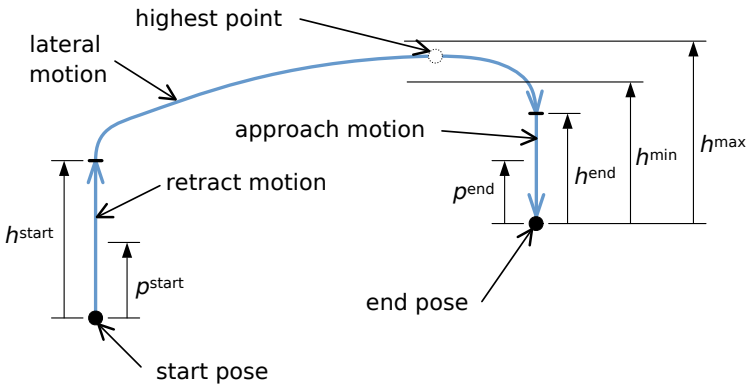


Figure 10: Settings for the MoveJump motion (projection on a vertical plane, the actual path is not in one plane)

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetMoveJumpHeight* (page 157) command is represented by *`MotionCommandID`* 46. See Section 5 for more details.

SetMoveMode

As discussed in Section 3, the timeline of a position-mode robot movement command (e.g., *MoveLin* (page 132), *MoveJoints* (page 126), and *MovePose* (page 138)) can be determined by specifying either the desired velocities or the desired duration. The choice between these two "submodes" is made using the motion-queue command *SetMoveMode* (page 159).

The command *SetMoveDurationCfg* (page 154) specifies what happens when a move command cannot meet the desired duration set by the *SetMoveDuration* (page 153) command, in time-based move mode.

Syntax

SetMoveMode(m)

Arguments

- m: submode, where
  - 0 selects the velocity-based submode, meaning the commands *SetJointVel* (page 151), *SetCartLinVel* (page 144), and *SetCartAngVel* (page 143) affect all subsequent position-mode movement commands.
  - 1 selects the time-based submode, meaning the command *SetMoveDuration* (page 153) affects all subsequent position-mode movement commands, except *MoveJump* (page 130).

Default

By default, m = 0.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetMoveMode* (page 159) command is represented by *MotionCommandID* 49. See Section 5 for more details.

SetPayload

This command sets the robot's payload mass and the center of mass relative to the robot's FRF.

It is inserted in the motion queue with other motion commands, allowing it to be executed, for example, after actions such as opening or closing the gripper.

Syntax

SetPayload(m,c_x,c_y,c_z)

Arguments

- m: the payload mass, in kilograms.
- c_x,c_y,c_z: the coordinates of the payload center of mass, relative to the robot's FRF, in millimeters.

Default values

By default, the payload mass is 0 kg.

Further details

The provided payload mass should include the weight of any components attached to the robot's flange, such as the end-effector and any workpieve being carried.

Although it is not mandatory to use this command, providing the payload data enables the robot to better estimate the required motor torques. This leads to several potential benefits, such as:

- Improved path tracking: The robot can move with greater accuracy and compensate for the additional load;
- Better torque limit management: Enhanced precision for the robot's torque limits option (see *SetTorqueLimitsCfg* (page 164)).

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetPayload* (page 160) command is represented by *MotionCommandID* 60. See Section 5 for more details.

SetTorqueLimits

This command sets thresholds for the torques applied to each motor, as percentages of the maximum allowable torques that can be applied at each motor. These thresholds can be interpreted in two different ways:

- When the second argument of *SetTorqueLimitsCfg* (page 164) is 0 or 1, the absolute values of the actual motor torques (*GetRtJointTorq* (page 264)), also reported as percentages of the maximum allowable torques, are compared to the respective thresholds.
- When the second argument of *SetTorqueLimitsCfg* (page 164) is 2, which is the default setting as of firmware 11.1, the absolute value of the difference between the actual (*GetRtJointTorq* (page 264)) and calculated motor torque (*GetRtTargetJointTorq* (page 271)) of each joint is compared with the respective threshold. Thus, in this case, the arguments of *SetTorqueLimits* (page 162) should be rather small, for example, about 10 (percent).

When a torque thresholds is exceeded, a customizable event is created. The event behavior can be set by the first argument of *SetTorqueLimitsCfg* (page 164).

This command is intended only to improve the chances of protecting your robot, its end-effector, and the surrounding equipment in the event of a collision. The actual torque in each motor (*GetRtJointTorq* (page 264)) is estimated by measuring the current in the corresponding drive. The calculated torque (*GetRtTargetJointTorq* (page 271)) is obtained from the dynamic model of the robot.

Syntax

SetTorqueLimits(τ_1,τ_2,τ_3,τ_4)

Arguments

- τ_i: torque threshold represented by a percentage of the maximum allowable torque that can be applied at motor i, where i = 1, 2, ..., 4 ranging from 0.001 to 100.

Default values

By default, all torque thresholds are set to 100%.

Further details

Unlike the *SetJointLimits* (page 190) commands, the *SetTorqueLimits* (page 162) command can only be applied after the robot has been activated. Note that high accelerations or large movements may also produce high torque peaks. Therefore, you should rely on this command only in the vicinity of obstacles, for example, while applying an adhesive. Remember that *SetTorqueLimits* (page 162) is a motion command and will therefore be inserted in the motion queue and not necessarily executed immediately.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetTorqueLimits* (page 162) command is represented by `MotionCommandID` 27. See Section 5 for more details.

SetTorqueLimitsCfg

This command sets the robot behavior when a joint torque exceeds the threshold set by the *SetTorqueLimits* (page 162) command. It also sends a torque limit status when the status changes (exceeded or not) for events severity greater than 0. For severity 4, a torque limit error is sent when torque exceeds the limit.

Syntax

SetTorqueLimitsCfg(s,m)

Arguments

- l: integer for the torque limit event severity
  - 0, no action;
  - 1, torque status event (message [3028]);
  - 2, pause motion and torque status event (message [3028]);
  - 4, torque status event (message [3028]) and torque limit error (message [3029]).
- m: integer defining the detection mode
  - 0 triggers a torque limit if the absolute value of any actual motor torque exceeds the respective torque limit set with *SetTorqueLimits* (page 162),
  - 1 is same as 0, but ignores joint acceleration/deceleration periods,
  - 2 triggers if any actual motor torque deviates from the corresponding calculated torque by more than the respective torque limit set with *SetTorqueLimits* (page 162).

With the option m = 0, you must use either very low accelerations (*SetJointAcc* (page 150)) or very high torque limits (*SetTorqueLimits* (page 162)).

The option m = 1 is mainly useful for joint-space movements, as revolute joints in Cartesian-space movements are generally always accelerating or decelerating.

Finally, with the option m = 2, the torque limits set by *SetTorqueLimits* (page 162) are *interpreted as maximum deviations rather than absolute limits*. This option allows for much finer control over torque limits and enables much quicker detection of collisions between the robot and its environment. To improve torque estimation accuracy, consider using the *SetPayload* (page 160) command.

Default values

By default, the event severity is set to 0, and the detection mode to 2.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetTorqueLimitsCfg* (page 164) command is represented by `MotionCommandID` 28. See Section 5 for more details.

SetTrf

This command defines the pose of the TRF with respect to the FRF. Note that this command makes the robot come to a complete stop, even if blending is enabled.

Syntax

SetTrf(x,y,z,γ)

Arguments

- x, y, z: the coordinates of the origin of the TRF with respect to the FRF, in mm;
- γ: the orientation of the TRF about its z-axis

  relative to the FRF, in degrees.

Default values

By default, the TRF coincides with the FRF.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetTrf* (page 166) command is represented by *MotionCommandID* 13. See Section 5 for more details.

SetVelTimeout

This command defines the timeout period following a velocity-mode motion command (*MoveJointsVel* (page 129), *MoveLinVelTrf* (page 136), or *MoveLinVelWrf* (page 137)). If no subsequent velocity-mode motion command is received within this period, all joint speeds will automatically be set to zero. The *SetVelTimeout* (page 167) command serves as a safety precaution and should be used accordingly. Note that the velocity-mode timeout is influenced by the *SetTimeScaling* (page 202) command.

Syntax

SetVelTimeout(t)

Arguments

t: desired timeout period, in seconds, ranging from 0.001 to 1.

Default values

By default, the velocity-mode timeout is 0.050 s.

Further details

The deceleration period begins after the velocity timeout. The deceleration time will depend on the current acceleration configured with *SetJointAcc* (page 150) or *SetCartAcc* (page 142) commands.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetVelTimeout* (page 167) command is represented by *MotionCommandID* 24. See Section 5 for more details.

SetWrf

This command defines the pose of the WRF with respect to the BRF. Note that this command makes the robot come to a complete stop, even if blending is enabled.

Syntax

SetWrf(x,y,z,γ)

Arguments

- x, y, z: the coordinates of the origin of the WRF with respect to the BRF, in mm;
- γ: the orientation of the WRF about its z-axis

  relative to the BRF, in degrees.

Default values

By default, the WRF coincides with the BRF.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetWrf* (page 168) command is represented by *MotionCommandID* 14. See Section 5 for more details.

Robot control commands

Contrary to motion commands, *robot control commands are executed immediately,* i.e., are instantaneous. The commands described in this section are used to control the status of the robot (e.g., activate the robot) and to configure the robot. These commands return a unique response, the generic response “[2085][Command successful: ‘...’.]” or some error message. For brevity, only the unique responses will be listed for each robot control command.

The robot control commands are listed below in several groups.

Motion-related commands

- *ClearMotion* (page 173)
- *PauseMotion* (page 182)
- *ResumeMotion* (page 185)
- *ResetError* (page 184)
- *SetCalibrationCfg* (page 186)
- *SetEob* (page 188)
- *SetEom* (page 189)
- *SetPStop2Cfg* (page 195)
- *SetTimeScaling* (page 202)

Robot status related commands

- *ActivateRobot* (page 171)
- *DeactivateRobot* (page 175)
- *RebootRobot* (page 183)
- *SetRecoveryMode* (page 198)

Simulation commands

- *ActivateSim* (page 172)
- *DeactivateSim* (page 176)

Utilities commands

- *SetRealTimeMonitoring* (page 196)
- *SetRobotName* (page 199)
- *SetRtc* (page 200)

- *SetMonitoringInterval* (page 192)
- *SyncCmdQueue* (page 208)
- *LogTrace* (page 180)
- *LogUserCommands* (page 181)
- *TcpDump* (page 209)
- *TcpDumpStop* (page 210)

Program execution commands

- *StartProgram* (page 203)
- *StartSaving* (page 204)
- *StopSaving* (page 206)
- *SetOfflineProgramLoop* (page 194)

Network commands

- *ConnectionWatchdog* (page 174)
- *EnableEtherNetIp* (page 177)
- *EnableProfinet* (page 178)
- *SwitchToEtherCat* (page 207)
- *SetNetworkOptions* (page 193)
- *SetCtrlPortMonitoring* (page 187)

Joint limits commands

- *SetJointLimits* (page 190)
- *SetJointLimitsCfg* (page 191)

ActivateRobot

This command activates all motors and disables the brakes of the joints.

Syntax

ActivateRobot(e)

Arguments

- e: the argument is optional; if the argument is used and is 1, the command forces a re-initialization of the drives.

Responses

- [2000][Motors activated.]

Usage restrictions

This command can be executed in any robot state.

If the robot is already activated, the response is returned and the robot does nothing.

Cyclic protocols

In cyclic protocols, the *ActivateRobot* (page 171) command is mapped to the *ActivateRobot* bit in the *RobotControl* data. See Table 4 for more details.

ActivateSim

Our robots support a simulation mode in which all of the robot's hardware including our EOAT are simulated and nothing moves. This mode allows you to test programs with the robot's hardware (i.e., hardware-in-the-loop simulation), without the risk of damaging the robot or its surroundings. Simulation mode can be activated and deactivated with the *ActivateSim* (page 172) and *DeactivateSim* (page 176) commands.

As of firmware 11.1, a new fast simulation mode is available, enabling commands to execute as quickly as possible. This significantly speeds up the testing of commands and programs.

Syntax

ActivateSim(m)

Arguments

- none: enable using the default simulation mode type (see *SetSimModeCfg* (page 201));
- m: integer specifying the simulation mode type as
  - 0, disabled (equivalent to using the command *DeactivateSim* (page 176)),
  - 1, normal (real-time) simulation mode,
  - 2, fast simulation mode.

Responses

- [2045][The simulation mode is enabled.]
- [2046][The simulation mode is disabled.]
- [1027][Simulation mode can only be enabled/disabled while the robot is deactivated.]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *ActivateSim* (page 172) command performed by setting the *ActivateSim* bit in the *RobotControl* data. See Table 4 for more details.

ClearMotion

This command stops the robot movement in the same fashion as the *PauseMotion* (page 182) command (i.e., by decelerating). The rest of the trajectory is deleted. The command *ResumeMotion* (page 185) must be sent to make the robot ready to execute new motion commands.

Syntax

ClearMotion()

Responses

- [2044][The motion was cleared.]

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the *ClearMotion* (page 173) command is mapped to the *ClearMotion* bit in the *MotionControl* data. See Table 5 for more details.

ConnectionWatchdog

For safety reasons, your application may start a communication watchdog with a timeout. The application must send another *ConnectionWatchdog* (page 174) command before the defined timeout otherwise the robot will automatically stop moving and report a safety stop with the message [3086][1]. The goal is to make sure that the robot quickly stops moving if communication with the TCP application is interrupted for any reason (including network failure or bug/freeze/dead-lock of the controlling application).

Syntax

ConnectionWatchdog(t)

Arguments

- t: desired timeout period, in seconds, ranging from 0.001 to $(2^{32} - 2)/1000$. If the argument is zero, the connection watchdog is canceled.

Default values

By default, the robot will supervise the TCP connection but only when the robot is moving, and as soon as it detects a connection loss, it will stop moving and return the message [3086][1]. However, the delay between the connection loss and the detection may vary from a few milliseconds to several seconds, depending on your network activity.

Responses

- [2177][1]
- [2177][0]

The first response is sent when the connection watchdog is activated for the first time. The second response is sent when the connection watchdog is deactivated with *ConnectionWatchdog(0)* (page 174).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols. However, each of these protocols has its own mechanism for managing communication timeouts. For example, in EtherCAT, the master can detect a communication issue if a slave fails to respond within the expected cycle time, triggering a watchdog timeout or setting an error status in the process image.

DeactivateRobot

This command disables all motors and engages the brakes on the robot joints. You must deactivate the robot in order to use certain commands (e.g., *SetJointLimits* (page 190), *SetNetworkOptions* (page 193)).

Syntax

DeactivateRobot()

Responses

- [2004][Motors deactivated.]

Further details

Your robot does not need homing and even if it is deactivated, the optional vacuum and I/O module can still function with the appropriate *_Immediate command.

Note

By deactivating the robot, you will lose all settings (parameters) that are not persistent, such as the definitions of the TRF and the WRF, the desired turn of the last joint, etc.

Usage restrictions

This command can only be executed when the robot is activated.

Cyclic protocols

In cyclic protocols, the *DeactivateRobot* (page 175) command is mapped to the *DeactivateRobot* bit in the *RobotControl* data. See Table 4 for more details.

DeactivateSim

This command deactivates simulation mode.

Syntax

DeactivateSim()

Responses

- [2046][The simulation mode is disabled.]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *DeactivateSim* (page 176) command is performed by clearing the *ActivateSim* bit in the *RobotControl* data. See Table 4 for more details.

EnableEtherNetIp

This command enables or disables EtherNet/IP slave stack, allowing the robot to be controlled or monitored by a EtherNetIP controller.

Syntax

EnableEtherNetIp(e)

Arguments

- e: EtherNet/IP mode setting. The possible values are:
  - 0: Disable EtherNet/IP;
  - 1: Enable EtherNet/IP;
  - 2: Enable EtherNet/IP in monitoring mode only.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 0 (EtherNet/IP is disabled).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

EnableProfinet

This command enables or disables the PROFINET slave stack, allowing the robot to be controlled or monitored by a PROFINET controller. When enabled, it also forwards LLDP packets between the robot's two Ethernet ports.

Syntax

EnableProfinet(e)

Arguments

- e: PROFINET mode setting. The possible values are:
  - 0: Disable PROFINET;
  - 1: Enable PROFINET;
  - 2: Enable PROFINET in monitoring mode only.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 0 (PROFINET is disabled).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

Home

This command has effect only on our Meca500 robot.

LogTrace

This command inserts a comment into the user and robot logs (see Section 9). It is useful for debugging, allowing you to show our support team where exactly a certain event occurs.

Syntax

LogTrace(s)

Arguments

- s: a text string (the comment).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

LogUserCommands

This command enables/disables the logging of commands received by the robot and the responses sent by the robot, as well as the logging of the beginning of execution of motion commands. This command has effect only on the user and robot logs (see Section 9).

Syntax

LogUserCommands(e_1,e_2)

Arguments

- e_1: enable (1) or disable (0) logging of received commands and sent responses;
- e_1: enable (1) or disable (0) logging of beginning of execution of motion commands.

Default values

Both logging states are disabled by default.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

PauseMotion

This command stops the robot's movement. It is executed immediately upon receipt (within 5 ms of being sent, depending on your network configuration). The robot decelerates to a stop rather than engaging the brakes. For instance, if a *MoveLin* (page 132) command is in progress when the *PauseMotion* (page 182) command is received, the robot's TCP will stop somewhere along the linear path. To determine the exact stop position, you can use the *GetRtCartPos* (page 259) or *GetRtJointPos* (page 263) commands.

The *PauseMotion* (page 182) command pauses the robot's motion without deleting the remaining trajectory, allowing it to be resumed with the *ResumeMotion* (page 185) command. This feature is particularly useful for custom HMIs that require a pause button or for situations where an unexpected issue arises (e.g., if the robot is applying adhesive and the reservoir runs empty).

Syntax

PauseMotion()

Responses

- [2042][Motion paused.]
- [3004][End of movement.]

The first response (2042) is always sent, whereas the second (3004) is sent only if the robot was moving when the command was received.

Additional details

If a motion error occurs while the robot is paused (e.g., if another moving object collides with the robot), the motion is cleared, and the trajectory can no longer be resumed.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the *PauseMotion* (page 182) command is mapped to the *PauseMotion* bit in the *MotionControl* data. See Table 5 for more details.

RebootRobot

This command reboots the robot. While similar, rebooting differs from power cycling.

Syntax

RebootRobot()

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *RebootRobot* (page 183) command is represented by *MotionCommandID* 200. See Section 5 for more details.

ResetError

This command resets the robot error status.

Syntax

ResetError()

Responses

- [2005][The error was reset.]
- [2006][There was no error to reset.]

The first response (2005) is generated if the robot was in error mode, whereas the second response (2006) is sent if the robot was not in error mode.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341).

Note that when an error occurs while the robot is deactivated, it is reported using the appropriate status message, but the robot does not enter an error state and does not require to *ResetError* (page 184).

Cyclic protocols

In cyclic protocols, the *ResetError* (page 184) command is mapped to the *ResetError* bit in the *RobotControl* data. See Table 4 for more details.

ResumeMotion

This command resumes the robot's movement if it was previously paused under one of the following conditions:

- By the *PauseMotion* (page 182) command.
- Due to a torque overload configured in pause motion mode (see *SetTorqueLimitsCfg* (page 164)).
- By the external signal P-Stop 2 , which is no longer present.

The robot resumes the remaining trajectory from the position where it came to a stop (after deceleration), unless an error occurred after the *PauseMotion* (page 182) or the robot was deactivated and then reactivated.

The *ResumeMotion* (page 185) command must also be sent after the *ClearMotion* (page 173) command. However, the robot will remain stationary until another motion command is received or retrieved from the motion queue. Additionally, the *ResumeMotion* (page 185) command must be sent after the *ResetError* (page 184) command.

Syntax

ResumeMotion()

Responses

- [2043][Motion resumed.]

Additional details

It is not possible to pause the motion along a trajectory, move the end-effector away, and then resume the trajectory from where it left off. Any motion commands sent while the robot is paused will be added to the end of the motion queue.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the *ResumeMotion* (page 185) command is mapped to the *ResumeMotion* bit in the *MotionControl* data. See Table 5 for more details.

SetCalibrationCfg

If your robot has undergone our optional calibration service, you can use this command to disable the calibration and revert to the robot's nominal parameters (such as link lengths and joint offsets). Calibration can be re-enabled at any time.

Use the *GetRobotCalibrated* (page 244) command to check whether your robot has been calibrated.

Syntax

SetCalibrationCfg(e)

Arguments

- e: enable (1) or disable (0) the calibration.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 1 (calibration is enabled), even if the robot has not been calibrated.

Responses

- [2170][e]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *SetCalibrationCfg* (page 186) command is represented by *MotionCommandID* 156. See Section 5 for more details.

SetCtrlPortMonitoring

Although data is sent synchronously over the control and monitoring ports, socket delays can cause desynchronization at the reception. If perfect synchronization is necessary, you must request a copy of the monitoring port data send to the control port by using the *SetCtrlPortMonitoring* (page 187) command.

Syntax

SetCtrlPortMonitoring(e)

Arguments

- e: enable (1) or disable (0) monitoring data over the control port.

Default values

By default, the monitoring on the control port is disabled.

Responses

- [2096][Monitoring on control port enabled]
- [2096][Monitoring on control port disabled]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

SetEob

When the robot completes a motion command or a block of motion commands, it can send the "[3012] [End of block.]" message. This means that there are no more motion commands in the queue and the robot velocity is zero. This message can be enabled/disabled using the *SetEob* (page 188) command.

Syntax

SetEob(e)

Arguments

- e: enable (1) or disable (0) the end-of-block message.

Default values

By default, the end-of-block message is enabled.

Responses

- [2054][End of block is enabled.]
- [2055][End of block is disabled.]

Note

We do not recommend using the "End of block" message to detect the completion of a program's execution. Instead, use the *SetCheckpoint* (page 145) command.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

SetEom

The robot can also send the "[3004][End of movement.]" message as soon as the robot stops moving. This can happen after the commands *MoveJoints* (page 126), *MoveJointsRel* (page 128), *MovePose* (page 138), *MoveJump* (page 130), *MoveLin* (page 132), *MoveLinRelTrf* (page 134), *MoveLinRelWrf* (page 135), *PauseMotion* (page 182) and *ClearMotion* (page 173) commands, as well as after the *SetCartAcc* (page 142) and *SetJointAcc* (page 150) commands. If blending is enabled (even only partially), then there would be no end-of-movement message between two consecutive Cartesian-space commands (*MoveLin* (page 132), *MoveLinRelTrf* (page 134), *MoveLinRelWrf* (page 135)) or two consecutive joint-space commands (*MoveJoints* (page 126), *MovePose* (page 138), *MoveJump* (page 130)).

Syntax

SetEom(e)

Arguments

- e: enable (1) or disable (0) the end-of-movement message.

Default values

By default, the end-of-movement message is disabled.

Responses

- [2052][End of movement is enabled.]
- [2053][End of movement is disabled.]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

SetJointLimits

This command redefines the lower and upper limits of a robot joint. To apply these user-defined joint limits, execute the command *SetJointLimitsCfg(1)* (page 191). The new joint limits must remain within the default limits.

Syntax

SetJointLimits(n,$q_{n,min}$,$q_{n,max}$)

Arguments

- n: joint number, an integer;
- $q_{n,min}$: lower joint limit, in degrees (for joints 1, 2, and 4) or in mm (for joint 3);
- $q_{n,max}$: upper joint limit, in degrees (for joints 1, 2, and 4) or in mm (for joint 3).

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default joint limits are specified in the technical specifications of the robot's user manual. Use *SetJointLimits(n,0,0)* (page 190) to reset the joint limits of joint n to its factory default values or simply disable the user-defined joint limits with the command *SetJointLimitsCfg(0)* (page 191).

Responses

- [2092][n]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *SetJointLimits* (page 190) command is represented by *MotionCommandID* 150. See Section 5 for more details.

SetJointLimitsCfg

This command enables or disables the user-defined limits set by the *SetJointLimits* (page 190) command. If the user-defined limits are disabled, the default joint limits become active. However, user-defined limits remain in memory, and can be re-enabled, even after a power down.

Syntax

`SetJointLimitsCfg(e)`

Arguments

- e: enable (1) or disable (0) the user-defined joint limits.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 0.

Responses

- [2093][User-defined joint limits enabled.]
- [2093][User-defined joint limits disabled.]

Note

If any robot joints are inadvertently moved outside the defined limits, the robot will not activate. To resolve this, enable recovery mode (see Section 3), which allows movement of the joints even when they are outside the configured limits.

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *SetJointLimitsCfg* (page 191) command is represented by `MotionCommandID` 151. See Section 5 for more details.

SetMonitoringInterval

This command is used to set the time interval at which real-time feedback from the robot is sent from the robot over TCP port 10001 (see the description for *SetRealTimeMonitoring* (page 196) and Table 3 for more details).

Syntax

SetMonitoringInterval(t)

Arguments

- t: desired time interval, in seconds, ranging from 0.001 to 1.

Default values

By default, the monitoring time interval is 0.015 s.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols. In cyclic protocols, the *cycle time* is configured on the PLC (i.e., on the master).

SetNetworkOptions

This command is used to set persistent parameters affecting the network connection. *The new parameter values will take effect only after a robot reboot.*

Syntax

SetNetworkOptions(n)

Arguments

- n: number of successive keep-alive TCP packets that can be lost before the TCP connection is closed, where n is an integer number ranging from 0 to 60.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 3.

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

This command is not available in cyclic protocols.

SetOfflineProgramLoop

This command is used only for the Meca500.

SetPStop2Cfg

This command is used to set the behavior of the robot when the P-Stop 2 signal is activated during automatic mode.

Syntax

SetPStop2Cfg(l)

Arguments

- l: severity
  - 2, for PauseMotion. Robot motion is paused but commands in the motion queue remain queued. New commands can be queued.
  - 3, for ClearMotion. Robot motion is paused and all commands in the motion queue are cleared. The robot will refuse to add any new commands in the motion queue until the P-Stop 2 condition is reset using *ResumeMotion* (page 185).

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 3.

Responses

- [2178][PStop2 configuration set successfully]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

This command is not available in cyclic protocols.

SetRealTimeMonitoring

TCP port 10001 (i.e., the monitoring port) transmits the robot's joint set and TRF pose, as well as some other data (see Section 4), at the rate specified by the *SetMonitoringInterval* (page 192) command. The *SetRealTimeMonitoring* (page 196) command enables the transmission of various additional real-time data over the monitoring port. Each set of data is preceded by a monotonic timestamp in microseconds, with respect to an internal clock. Essentially, you get the same responses as with the GetRt* and GetRtTarget* commands, but on the monitoring port, instead of on the control port, and at every monitoring interval, rather than only when requested.

You can send the *SetRealTimeMonitoring* (page 196) command even if the robot is not activated.

Syntax

SetRealTimeMonitoring(n_1,n_2,...)

Arguments

- n_1,n_2: a list of number codes or names, as follows
  - 2200 or TargetJointPos, for the response of the *GetRtTargetJointPos* (page 270) command;
  - 2201 or TargetCartPos, for the response of the *GetRtTargetCartPos* (page 266) command;
  - 2202 or TargetJointVel, for the response of the *GetRtTargetJointVel* (page 272) command;
  - 2203 or TargetJointTorq, for the response of the *GetRtTargetJointTorq* (page 271) command;
  - 2204 or TargetCartVel, for the response of the *GetRtTargetCartVel* (page 267) command;
  - 2210 or JointPos, for the response of the *GetRtJointPos* (page 263) command;
  - 2211 or CartPos, for the response of the *GetRtCartPos* (page 259) command;
  - 2212 or JointVel, for the response of the *GetRtJointVel* (page 265) command;
  - 2213 or JointTorq, for the response of the *GetRtJointTorq* (page 264) command;
  - 2214 or CartVel, for the response of the *GetRtCartVel* (page 260) command;
  - 2218 or Conf, for the response of the *GetRtConf* (page 261) command (sent only when changed);
  - 2219 or ConfTurn, for the response of the *GetRtConfTurn* (page 262) command (sent only when changed);

- 2220 or Accel, for the response of the *GetRtAccelerometer* (page 258) command;
- 2321 or GripperForce, for the response of the *GetRtGripperForce* (page 298) command;
- 2322 or GripperPos, for the response of the *GetRtGripperPos* (page 299) command;
- 2343 or VacuumPressure, for the response of the *GetRtVacuumPressure* (page 304) command;
- All, to enable all of the above responses.

Default values

After a power up, none of the above messages are enabled.

Responses

- [2117][n_1, n_2 ...]
  - n_1, n_2 ...: a list of response codes.

Additional details

The *SetRealTimeMonitoring* (page 196) command does not have a cumulative effect; if you execute the command *SetRealTimeMonitoring(All)* (page 196) and then the command *SetRealTimeMonitoring(TargetCartPos)* (page 196) or the command *SetRealTimeMonitoring(2201)* (page 196), you will only enable message 2201. Further details about the monitoring port are presented in Section 4.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

SetRecoveryMode

Moving the robot when its joints are outside the user-defined limits, or when there is a torque overload, a self-collision, or a work-zone limit, is impossible. For these situations, it is useful to enable the recovery mode (Section 3) with the *SetRecoveryMode* (page 198) command.

When the recovery mode is enabled, and the robot is activated, virtually all motion commands are accepted, but joint and Cartesian velocities and accelerations are significantly limited, for safety reasons.

Syntax

SetRecoveryMode(e)

Arguments

- e: enable (1) or disable (0) the recovery mode.

Default values

By default the recovery mode is deactivated.

Responses

- [2049][Recovery mode enabled]
- [2050][Recovery mode disabled]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *SetRecoveryMode* (page 198) command is mapped to the *EnableRecoveryMode* bit in the *RobotControl* data. See Table 4 for more details.

SetRobotName

This command allows you to change the robot's name. The command is useful when multiple robots are connected on the same network. The *SetRobotName* (page 199) command also changes the hostname of the robot in the case of a DHCP connection. The robot's name is displayed in the upper right corner of the MecaPortal, as well as in the browser tab hosting the web interface. You can also retrieve the robot's name with the command *GetRobotName* (page 245).

Syntax

SetRobotName(s)

Arguments

- s: string containing the robot's name. It should contain a maximum of 63 characters, alphanumeric or hyphens, but should not start with a hyphen.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is mcs500.

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

This command is not available in cyclic protocols.

SetRtc

When the robot is powered on, its internal clock starts at the date at which the robot image was built. Each time you connect to the robot via the MecaPortal, the internal clock of the robot is automatically adjusted to UTC. Other than connecting to the robot using the MecaPortal, another solution is to send the *SetRtc* (page 200) command to the robot (from the PLC or any application controlling the robot), if you want all timestamps in the robot's log files to be with respect to UTC. Note, however, that this command does not affect the timestamps of the data sent over the monitoring and control ports, which are with respect to an internal monotonic microseconds timer that cannot be reset.

Syntax

SetRtc(t)

Arguments

- t: Epoch time as defined in Unix (i.e., number of seconds since 00:00:00 UTC January 1, 1970).

Default values

By default the recovery mode is deactivated.

Responses

- [2049][Recovery mode enabled]
- [2050][Recovery mode disabled]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *SetRtc* (page 200) command is represented by the *HostTime* data. See Section 5 for more details.

SetSimModeCfg

Our robots support a simulation mode in which all of the robot's hardware including our EOAT are simulated and nothing moves. Simulation mode can be activated and deactivated with the *ActivateSim* (page 172) and *DeactivateSim* (page 176) commands (these commands can only be executed when the robot is deactivated).

The *SetSimModeCfg* (page 201) command configures the default simulation mode type (fast or normal) enabled when *ActivateSim* (page 172) is executed without an argument, when the Activate Sim button in the MecaPortal is pressed, or when simulation mode is enabled using a cyclic protocol.

Syntax

SetSimModeCfg(m)

Arguments

- m: integer specifying the default simulation mode type
  - 1, normal (real-time) simulation mode,
  - 2, fast simulation mode.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 1.

Responses

- [2188][Simulation mode configuration set successfully.]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

While simulation can be activated in cyclic protocols, the default simulation mode type (i.e., fast or normal) cannot be selected in cyclic protocols. You must configure the default simulation mode type using the TCP command *SetSimModeCfg* (page 201).

SetTimeScaling

This command sets the time scaling of the trajectory generator. By calling this command with an argument p of less than 100, all robot motions remain exactly the same (i.e., the path remains the same), but everything will be (100 − p) percent slower, including time delays (e.g., the pause set by the command *Delay* (page 125)). In other words, this command is more than a simple velocity override.

When using the MecaPortal, you can change the time scaling in real time with the “Time Scaling” slider at the bottom of the program panel.

Syntax

SetTimeScaling(p)

Arguments

- p: time scaling percentage, from 0.001 to 100.

Default values

By default, p = 100.

Responses

- [2015] [p]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *SetTimeScaling* (page 202) command is represented by *MotionCommandID* 48. See Section 5 for more details.

StartProgram

This command starts a program that has been previously saved in the robot's memory. Executing this command will launch the specified program only once.

Syntax

StartProgram(s)

Arguments

- s: string containing the program name. It should contain a maximum of 63 characters among the 62 alphanumericals (A..Z, a..z, 0..9), the underscore and the hyphen.

Responses

- [2063][Offline program s started.]
- [3017][No offline program saved.]

Note

The MecaPortal allows saving of programs using sting-based name rather than numbers, unlike the command StartSaving. However, if you wish to start these programs through a cyclic protocol, you should only use integer numbers as program names.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the *StartProgram* (page 203) command is represented by *MotionCommandID* 100. See Section 5 for more details.

StartSaving

This command is used to save commands in the robot's internal memory. These are referred to as *offline programs* (page 340) that can later be played using the *StartProgram* (page 203) command.

The saved program will remain in the robot internal memory even after disconnecting the power. Saving a new program with the same argument overwrites the existing program.

The robot records all commands sent between the *StartSaving* (page 204) and *StopSaving* (page 206) commands.

Note

We recommend using the MecaPortal code editor to edit, save, and delete offline programs, as it is easier and more flexible than using *StartSaving* (page 204) / *StopSaving* (page 206).

Note

The robot will execute but not record request commands (Get*). If the robot receives a change of state command (*Home* (page 179), *PauseMotion* (page 182), *SetEom* (page 189), etc.) while recording, it will abort saving the program.

Syntax

StartSaving(n)

Arguments

- n: program number, where n ≤ 500 (maximum number of programs that can be stored).

Responses

- [2060][Start saving program.]

Usage restrictions

This command can be executed in any robot state.

If the robot is deactivated, the program is saved without executing received commands.

Cyclic protocols

This command is not available in cyclic protocols.

StopSaving

This command will make the controller save the program and stop saving.

> **Note**
>
> We recommend using the MecaPortal code editor to edit, save, and delete offline programs, as it is easier and more flexible than using *StartSaving* (page 204) / *StopSaving* (page 206).

Syntax

StopSaving()

Responses

- [2061][n commands saved.]
- [2064][Offline program looping is enabled.]
- [2065][Offline program looping is disabled.]
- [1022][Robot was not saving the program.]

Two responses will be generated: the first (2061) and the second (2064) or third (2065). If you send this command while the robot is not saving a program, the fourth response (1022) will be returned.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

SwitchToEtherCat

This command will enable EtherCAT and disable the other three protocols (EtherCAT is an exclusive protocol that cannot be used at the same time as other Ethernet-based protocols).

Note

Enabling EtherCAT will disable all other communication protocols (TCP/IP, EtherNet/IP, PROFINET). The MecaPortal is NOT accessible while in EtherCAT mode.

There are two ways to disable EtherCAT (and thus re-enable another communication protocols):

1. Reset the DisableEtherCAT subindex of the *Robot control* (page 44) object.
2. Perform a network configuration reset (see the robot's user manual for the procedure).

Syntax

SwitchToEtherCat()

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, set the *DisableEtherCAT* bit in the *Robot Control* data to 1 to disable the EtherCAT protocol and switch back to TCP/IP protocol. See Table 4 for more details.

SyncCmdQueue

This command is used for associating an ID number with any non-motion command, thus providing means to identify the command that sent a specific response. It it is executed immediately.

Syntax

SyncCmdQueue()

Arguments

- n : a non-negative integer number, ranging from 0 to $2^{32} - 1$.

Responses

- [2097][n]

Additional details

For example, sending SyncCmdQueue(123) just before the *GetStatusRobot* (page 277) command allows the application to know if a received robot status (code 2007) is the response of the *GetStatusRobot* (page 277) request (i.e., preceded by [2097][123]) or of an older status request.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

TcpDump

This command starts an Ethernet capture (PCAP file format) on the robot, for the specified duration. The Ethernet capture will be part of the logs archive, which can be retrieved from the MecaPortal.

Syntax

TcpDump()

Arguments

- n: duration in seconds.

Responses

- [3035][TCP dump capture started for n seconds.]
- [3036][TCP dump capture stopped.]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

TcpDumpStop

This command is needed if you want to stop the TCP dump started with the *TcpDump(n)* (page 209) commands, before the timeout period of n seconds.

Syntax

TcpDumpStop()

Responses

- [3036][TCP dump capture stopped.]

Usage restrictions

This command can be executed in any robot state.

It does nothing if no TCP dump capture was started.

Cyclic protocols

This command is not available in cyclic protocols.

Data request commands

Most request commands return values for parameters that were either previously configured using the corresponding Set* command (e.g. *SetJointVel* (page 151) and *GetJointVel* (page 229)) or are simply the default values of those parameters (e.g. 50 in the case of the joint velocity). A few of the request commands return values for parameters that have been automatically assigned (e.g., at the factory as in the case of *GetProductType* (page 241), or after a firmware upgrade, as in the case of *GetFwVersion* (page 225)).

Contrary to motion commands, *robot control commands are executed immediately*, i.e., are instantaneous. Therefore, if you send a *SetTrf* (page 166) command, then a *MovePose* (page 138) command, then another *SetTrf* (page 166) command, and immediately after that a *GetTrf* (page 251) command, you will get the arguments of the first *SetTrf* (page 166) command.

The following is the alphabetically ordered list of data request commands that have a corresponding Set* command:

- *GetAutoConf* (page 213)
- *GetAutoConfTurn* (page 214)
- *GetBlending* (page 215)
- *GetCalibrationCfg* (page 216)
- *GetCartAcc* (page 217)
- *GetCartAngVel* (page 218)
- *GetCartLinVel* (page 219)
- *GetCheckpoint* (page 220)
- *GetCheckpointDiscarded* (page 221)
- *GetConf* (page 222)
- *GetConfTurn* (page 223)
- *GetEthernetIpEnabled* (page 224)
- *GetJointAcc* (page 226)
- *GetJointLimits* (page 227)
- *GetJointLimitsCfg* (page 228)
- *GetJointVel* (page 229)
- *GetJointVelLimit* (page 230)
- *GetMonitoringInterval* (page 232)
- *GetMoveDuration* (page 233)

- *GetMoveDurationCfg* (page 234)
- *GetMoveJumpApproachVel* (page 235)
- *GetMoveJumpHeight* (page 236)
- *GetMoveMode* (page 237)
- *GetNetworkOptions* (page 238)
- *GetPayload* (page 240)
- *GetProfinetEnabled* (page 242)
- *GetPStop2Cfg* (page 239)
- *GetRealTimeMonitoring* (page 243)
- *GetRobotName* (page 245)
- *GetSimModeCfg* (page 247)
- *GetTimeScaling* (page 248)
- *GetTorqueLimits* (page 249)
- *GetTorqueLimitsCfg* (page 250)
- *GetTrf* (page 251)
- *GetVelTimeout* (page 252)
- *GetWrf* (page 253)

The following is the list of data request commands that return read-only data, which cannot be modified by the user:

- *GetFwVersion* (page 225)
- *GetModelJointLimits* (page 231)
- *GetProductType* (page 241)
- *GetRobotCalibrated* (page 244)
- *GetRobotSerial* (page 246)

A few other data request commands exist, but these are presented in the sections *Work zone supervision and collision prevention commands* (page 279) (e.g., *GetToolSphere* (page 283)), *Commands for optional accessories* (page 291) and *Commands for managing variables (beta)* (page 328) (e.g., *GetVariable* (page 336)).

GetAutoConf

This command returns the state of the automatic posture configuration selection, which can be influenced by the *SetAutoConf* (page 139) and *SetConf* (page 147) commands.

Syntax

GetAutoConf()

Responses

- [2028][e]
  - e: enabled (1) or disabled (0).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetAutoConf* (page 213) is represented by *DynamicDataTypeID* 20. See Section 5 for more details.

GetAutoConfTurn

This command returns the state of the automatic turn configuration selection, which can be influenced by the *SetAutoConfTurn* (page 140) and *SetConfTurn* (page 148) commands.

Syntax

GetAutoConfTurn()

Responses

- [2031][e]
  - e: enabled (1) or disabled (0).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetAutoConfTurn* (page 214) is represented by *DynamicDataTypeID* 20. See Section 5 for more details.

GetBlending

This command returns the blending percentage, which is set using the *SetBlending* (page 141) command.

Syntax

GetBlending()

Responses

- [2150][p]
  - p: blending percentage, ranging from 0 (blending disabled) to 100.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetBlending* (page 215) is represented by *DynamicDataTypeID* 21. See Section 5 for more details.

GetCalibrationCfg

This command returns the state of the optional robot calibration, configured using the *SetCalibrationCfg* (page 186) command.

Syntax

GetCalibrationCfg()

Responses

- [2171][e]
  - e: enabled (1) or disabled (0).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetCalibrationCfg* (page 216) is represented by *DynamicDataTypeID* 30. See Section 5 for more details.

GetCartAcc

This command returns the desired limit for the acceleration of the Tool Reference Frame (TRF) relative to the World Reference Frame (WRF), set using the *SetCartAcc* (page 142) command.

Syntax

GetCartAcc()

Responses

- [2156][p]
  - p: percentage of the maximum acceleration of the TRF.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetCartAcc* (page 217) is represented by *DynamicDataTypeID* 22. See Section 5 for more details.

GetCartAngVel

This command returns the desired limit for the angular velocity of the Tool Reference Frame (TRF) relative to the World Reference Frame (WRF), set using the *SetCartAngVel* (page 143) command.

Syntax

GetCartAngVel()

Responses

- [2155][ω]
  - ω: TRF angular velocity limit, in degrees per second (°/s).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetCartAngVel* (page 218) is represented by *DynamicDataTypeID* 22. See Section 5 for more details.

GetCartLinVel

This command returns the desired Tool Center Point (TCP) velocity limit, configured using the *SetCartLinVel* (page 144) command.

Syntax

GetCartLinVel()

Responses

- [2154][v]
  - v: TCP velocity limit, in millimeters per second (mm/s).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetCartLinVel* (page 219) is represented by *DynamicDataTypeID* 22. See Section 5 for more details.

GetCheckpoint

This command returns the argument of the last executed *SetCheckpoint* (page 145).

Syntax

GetCheckpoint()

Responses

- [2157][n]
  - n: checkpoint number.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetCheckpoint* (page 220) is represented by the *CheckpointReached* field of the *MotionStatus* section (see Section 5).

GetCheckpointDiscarded

Returns the id of the most recently discarded checkpoint (as posted with *SetCheckpoint* (page 145)).

Checkpoint can be discarded by *ClearMotion* (page 173), *DeactivateRobot* (page 175), by robot entering error state or safety stop state.

Syntax

GetCheckpointDiscarded()

Responses

- [2149][n]
  - n: most recently discarded checkpoint number.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetCheckpointDiscarded* (page 221) is represented by the *CheckpointDiscarded* field of the *MotionStatus* section (see Section 5).

GetConf

This command returns the desired posture configuration (see Section 3), or more precisely, the posture configuration that will be applied to the next *MovePose* (page 138) or MoveLin* command in the motion queue. This configuration is either explicitly specified using the *SetConf* (page 147) command or automatically assigned when the *SetAutoConf(0)* (page 139) command is executed.

Syntax

GetConf()

Responses

- [2029][c_e]
  - c_e: elbow configuration parameter, either -1 or 1†.

† If automatic posture configuration selection is enabled, the value is an asterisk, i.e., the response is [2029][*].

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetConf* (page 222) is represented by *DynamicDataTypeID* 20. See Section 5 for more details.

GetConfTurn

This command returns the desired turn configuration for the last joint (see Section 3), i.e., the turn configuration that will be applied to the next *MovePose* (page 138) or MoveLin* command in the motion queue. This is either the turn configuration explicitly specified using the *SetConfTurn* (page 148) command or the one automatically assigned when the *SetAutoConfTurn(0)* (page 140) command was executed.

Syntax

GetConfTurn()

Responses

- [2036][c_t]
  - c_t: turn configuration parameter, an integer or an asterisk†.

† If automatic turn configuration selection is enabled, the response is [2036][*].

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetConfTurn* (page 223) is represented by *DynamicDataTypeID* 20. See Section 5 for more details.

GetEthernetIpEnabled

This command returns the state of the Ethernet/IP protocol. See the description of the *EnableEtherNetIp* (page 177) command for more details.

Syntax

GetEthernetIpEnabled()

Responses

- [2073][e]
  - e: 0, 1 or 2 as defined in the description of the *EnableEtherNetIp* (page 177) command.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetFwVersion

This command returns the version of the firmware installed on the robot.

Syntax

GetFwVersion()

Responses

- [2081][vx.x.x]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetFwVersion* (page 225) is represented by *DynamicDataTypeID* 1. See Section 5 for more details.

GetJointAcc

This command returns the desired joint accelerations reduction factor, set using the *SetJointAcc* (page 150) command.

Syntax

GetJointAcc()

Responses

- [2153][p]
  - p: percentage of maximum joint accelerations.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetJointAcc* (page 226) is represented by *DynamicDataTypeID* 22. See Section 5 for more details.

GetJointLimits

This command returns the current effective joint limits, i.e., the default joint limits or the user-defined limits if applied using the *SetJointLimits* (page 190) command and enabled using the *SetJointLimitsCfg* (page 191) command.

Syntax

GetJointLimits(n)

Arguments

- n: joint number, an integer.

Responses

- [2090][n, $q_{n,min}$, $q_{n,max}$]
  - n: joint number, an integer;
  - $q_{n,min}$: lower joint limit, in degrees (for joints 1, 2, and 4) or in millimeters (for joint 3);
  - $q_{n,max}$: upper joint limit, in degrees (for joints 1, 2, and 4) or in millimeters (for joint 3).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetJointLimits* (page 227) is represented by *DynamicDataTypeID* 14 and 15. See Section 5 for more details.

GetJointLimitsCfg

This command returns the status of the user-enabled joint limits, defined by the *SetJointLimitsCfg* (page 191).

Syntax

GetJointLimitsCfg()

Responses

- [2094][e]
  - e: status, 1 for enabled, 0 for disabled.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetJointLimitsCfg* (page 228) is represented by *DynamicDataTypeID* 11. See Section 5 for more details.

GetJointVel

This command returns the desired joint velocities reduction factor, set using the *SetJointVel* (page 151) command.

Syntax

GetJointVel()

Responses

- [2152][p]
  - p: percentage of maximum joint velocities.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetJointVel* (page 229) is represented by *DynamicDataTypeID* 22. See Section 5 for more details.

GetJointVelLimit

This command returns the desired joint velocities override, set using the *SetJointVelLimit* (page 152) command.

Syntax

GetJointVelLimit()

Responses

- [2169][p]
  - p: percentage of maximum joint velocities override.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetJointVelLimit* (page 230) is represented by *DynamicDataTypeID* 22. See Section 5 for more details.

GetModelJointLimits

This command returns the factory default joint limits.

Syntax

GetModelJointLimits(n)

Arguments

- n: joint number, an integer.

Responses

- [2113][n, $q_{n,min}$, $q_{n,max}$]
  - n: joint number, an integer number between 1 and 4;
  - $q_{n,min}$: lower joint limit, in (for joints 1, 2, and 4) or in mm (for joint 3);
  - $q_{n,max}$: upper joint limit, in (for joints 1, 2, and 4) or in mm (for joint 3).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetModelJointLimits* (page 231) is represented by *DynamicDataTypeID* 12 and 13. See Section 5 for more details.

GetMonitoringInterval

This command returns the time interval at which real-time feedback from the robot is sent over TCP port 10001.

Syntax

GetMonitoringInterval()

Responses

- [2116][t]
  - t: time interval, in seconds.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetMoveDuration

This command returns the default duration set by the *SetMoveDuration* (page 153) command.

Syntax

GetMoveDuration()

Responses

- [2191][t]
  - t: duration for time-based moves.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetMoveDuration* (page 233) is represented by *DynamicDataTypeID* 29. See Section 5 for more details.

GetMoveDurationCfg

This command returns the severity of the response when a move command cannot meet the desired duration set by the *SetMoveDuration* (page 153) command, in time-based move mode.

Syntax

GetMoveDurationCfg()

Responses

- [2190][s]
  - s: 0 for silent mode, 1 for generating a warning message, 4 for generating an error.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetMoveDurationCfg* (page 234) is represented by *DynamicDataTypeID* 29. See Section 5 for more details.

GetMoveJumpApproachVel

This command returns the desired initial and final velocity parameters for the *MoveJump* (page 130) command, set using the *SetMoveJumpApproachVel* (page 156) command.

Syntax

GetMoveJumpApproachVel()

Responses

- [2175][v_{start}, p_{start}, v_{end}, p_{end}]
  - v_{start}: maximum allowed vertical speed near the start pose, in mm/s, from 0.001 to 700;
  - p_{start}: initial portion of the retreat motion during which v_{start} is applied, in mm, from 0 to 102;
  - v_{end}: maximum allowed vertical speed near the end pose, in mm/s, from 0.001 to 700;
  - p_{end}: final portion of the approach motion during which v_{start} is applied, in mm, from 0 to 102.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the commands *GetMoveJumpApproachVel* (page 235) is represented by *DynamicDataTypeID* 28. See Section 5 for more details.

GetMoveJumpHeight

This command returns the different height parameters for the *MoveJump* (page 130) command, set with the *SetMoveJumpHeight* (page 157) command.

Syntax

GetMoveJumpHeight()

Responses

- [2174][h_{start}, h_{end}, h_{min}, h_{max}]
  - h_{start}: height from start pose to reach using a pure vertical translation, before the lateral motion begins, in mm;
  - h_{end}: height from end pose from where to begin the final pure vertical translation, in mm;
  - h_{min}: minimum height to reach while performing the lateral motion, with respect to the highest (if h_{start} and h_{end} are positive) or lowest (if h_{start} and h_{end} are negative) between the start and end poses, in mm;
  - h_{max}: maximum height allowed to reach while performing the lateral motion, with respect to the highest (if h_{start} and h_{end} are positive) or lowest (if h_{start} and h_{end} are negative) between the start and end poses, in mm.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the commands *GetMoveJumpHeight* (page 236) is represented by *DynamicDataTypeID* 27. See Section 5 for more details.

GetMoveMode

This command returns the default move mode set by the *SetMoveMode* (page 159) command.

Syntax

GetMoveMode()

Responses

- [2189][m]
  - m: 0 for velocity-based and 1 for time-based move mode.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetMoveMode* (page 237) is represented by *DynamicDataTypeID* 29. See Section 5 for more details.

GetNetworkOptions

This command returns the parameters affecting the network connection.

Syntax

GetNetworkOptions()

Responses

- [2119][n_1, n_2, n_3, n_4, n_5, n_6]
  - n_1: number of successive keep-alive TCP packets that can be lost before the TCP connection is closed, where n_1 is an integer number ranging from 0 to 60;
  - n_2, n_3, n_4, n_5, n_6: currently not used.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetPStop2Cfg

This command returns the severity level set with the *SetPStop2Cfg* (page 195) command.

Syntax

GetPStop2Cfg(l)

Responses

- [2178][l]
  - 2, for PauseMotion;
  - 3, for ClearMotion.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetPayload

This command returns the configured payload mass and center of mass, as set using the *SetPayload* (page 160) command.

Syntax

GetPayload()

Responses

- [2192][m, c_x,c_y,c_z]
  - m: The payload mass (in kilograms).
  - c_x,c_y,c_z: The coordinates of the payload center of mass, relative to the robot's *FRF* (page 339), in millimeters.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the commands *GetPayload* (page 240) is represented by *DynamicDataTypeID* 31. See Section 5 for more details.

GetProductType

This command returns the type (model) of the product.

Syntax

GetProductType()

Responses

- [2084][MCS500]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the commands *GetProductType* (page 241) is represented by *DynamicDataTypeID* 2. See Section 5 for more details.

GetProfinetEnabled

This command returns the state of the PROFINET protocol. See the description of the *EnableProfinet* (page 178) command for more details.

Syntax

GetProfinetEnabled()

Responses

- [2077][e]
  - e: 0, 1 or 2 as defined in the description of the *EnableProfinet* (page 178) command.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetRealTimeMonitoring

This command returns the numerical codes of the responses that have been enabled using the *SetRealTimeMonitoring* (page 196) command.

Syntax

GetRealTimeMonitoring()

Responses

- [2117][n_1, n_2, ...]

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetRobotCalibrated

This command returns a response of 1 if the robot has undergone our optional calibration service, or 0 if it has not.

Syntax

GetRobotCalibrated()

Responses

- [2122][s]
  - s: status (1 if the robot has been calibrated, 0 if it has not).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRobotCalibrated* (page 244) is represented by *DynamicDataTypeID* 30. See Section 5 for more details.

GetRobotName

This command returns the robot's name, set with the command *SetRobotName* (page 199). Note that the robot name is used as a host name when the robot's network configuration uses DHCP.

Syntax

GetRobotName()

Responses

- [2095][s]
  - s: string containing the robot's name.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetRobotSerial

This command returns the serial number of the robot, except for robots manufactured before 2021. The serial number of all robots can also be found on the back of the robot's base.

Syntax

GetRobotSerial()

Responses

- [2083][s]
  - s: string containing the robot's serial number.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the commands *GetRobotSerial* (page 246) is represented by *DynamicDataTypeID* 3. See Section 5 for more details.

GetSimModeCfg

This command returns the default simulation mode set using the *SetSimModeCfg* (page 201) command.

Syntax

GetSimModeCfg()

Responses

- [2187][m]
  - m: 1 for normal (real-time) and 2 for fast simulation mode.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetTimeScaling

This command returns the time scaling percentage set using the *SetTimeScaling* (page 202) command.

Syntax

GetTimeScaling()

Responses

- [2015][p]
  - p: current time scaling percentage.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the commands *GetTimeScaling* (page 248) is represented by *DynamicDataTypeID* 54. See Section 5 for more details.

GetTorqueLimits

Returns the current joint torque thresholds, as configured in the motion queue by the command *SetTorqueLimits* (page 162).

Syntax

GetTorqueLimits()

Responses

- [2161][τ_1, τ_2, τ_3, τ_4]
  - τ_i: percentage of the maximum allowable torque that can be applied at motor i, where i = 1, 2, ..., 4 ranging from 0.001 to 100.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the commands *GetTorqueLimits* (page 249) is represented by *DynamicDataTypeID* 25. See Section 5 for more details.

GetTorqueLimitsCfg

This command returns the desired behavior of the robot when a joint torque exceeds the thresholds set by the *SetTorqueLimits* (page 162). This desired behavior is configured using the *SetTorqueLimitsCfg* (page 164) command.

Syntax

GetTorqueLimitsCfg()

Responses

- [2160][l, m]
  - l: an integer defining the torque limit event severity (see *SetTorqueLimitsCfg* (page 164));
  - m: an integer defining the detection mode (see *SetTorqueLimitsCfg* (page 164)).

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the commands *GetTorqueLimitsCfg* (page 250) is represented by *DynamicDataTypeID* 24. See Section 5 for more details.

GetTrf

This command returns the current definition of the TRF with respect to the FRF, set using the *SetTrf* (page 166) command.

Syntax

GetTrf()

Responses

- [2014][x, y, z, γ]
  - x, y, z: the coordinates of the origin of the TRF with respect to the FRF, in mm;
  - γ: orientation of the TRF about its z-axis

    with respect to the FRF, in degrees.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetTrf* (page 251) is represented by the *TargetTrf* section (see Section 5).

GetVelTimeout

This command returns the timeout for velocity-mode motion commands, set using the *SetVelTimeout* (page 167) command.

Syntax

GetVelTimeout()

Responses

- [2151][t]
  - t: timeout, in seconds.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetVelTimeout* (page 252) is represented by *DynamicDataTypeID* 21. See Section 5 for more details.

GetWrf

This command returns the current definition of the WRF with respect to the BRF, set using the *SetWrf* (page 168) command.

Syntax

GetWrf()

Responses

- [2014][x, y, z, γ]
  - x, y, z: the coordinates of the origin of the WRF with respect to the BRF, in mm;
  - γ: orientation of the WRF about its z-axis

    with respect to the BRF, in degrees.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

In cyclic protocols, the command *GetWrf* (page 253) is represented by the *TargetWrf* section (see Section 5).

Real-time data request commands

The commands in this section provide real-time data about the robot's current status, and are instantaneous (i.e., executed immediately). Additionally, there are real-time data commands for the robot's external tool accessories, which will be covered later.

Examples of robot data include the current joint set, the length of the motion queue, and the status of torque limits. There are two categories of real-time data commands for robot positioning:

- **Real-time sensor data** These commands return data based on live measurements from the robot's sensors. Examples include *GetRtJointTorq* (page 264), *GetRtJointPos* (page 263), *GetRtCartPos* (page 259), etc.
- **Real-time target data** These commands return data based on targets calculated by the trajectory planner. Examples include *GetRtTargetJointPos* (page 270), *GetRtTargetCartPos* (page 266), etc.

For instance, if the robot is active and stationary, the *GetRtTargetJointPos* (page 270) command will consistently return the same joint set. However, the robot is never perfectly still since the motors are continuously controlled by the drives. As a result, the revolute joints, for example, may oscillate by ±0.001° around the desired angles. If you execute the *GetRtJointPos* (page 263) command twice in quick succession while the robot is stationary, you may notice slight differences in the responses.

In more dynamic situations, such as when a high force is applied or during rapid movements, the differences between the actual joint positions (*GetRtJointPos* (page 263)) and the target positions (*GetRtTargetJointPos* (page 270)) can be more significant. These differences increase further during rapid motions with high payloads or in the event of a collision.

Each GetRt* command response begins with a timestamp, measured in microseconds.

The following is the list of real-time data request commands, in alphabetical order:

- *GetRtTargetCartPos* (page 266)
- *GetRtTargetCartVel* (page 267)
- *GetRtTargetConf* (page 268)
- *GetRtTargetConfTurn* (page 269)
- *GetRtTargetJointPos* (page 270)
- *GetRtTargetJointTorq* (page 271)
- *GetRtTargetJointVel* (page 272)
- *GetRtTrf* (page 273)
- *GetRtWrf* (page 274)
- *GetRtc* (page 275)
- *GetSafetyStopStatus* (page 276)
- *GetStatusRobot* (page 277)
- *GetTorqueLimitsStatus* (page 278)

A few other real-time data request commands exist, but these are presented in the sections *Work zone supervision and collision prevention commands* (page 279) (*GetCollisionStatus* (page 282), *GetWorkZoneStatus* (page 286)) and *Commands for optional accessories* (page 291).

GetCmdPendingCount

This command returns the number of motion commands that are currently in the motion queue.

Syntax

GetCmdPendingCount()

Responses

- [2080][n]
  - n: number of motion commands in the queue.

Note that the robot will compile several (~25) commands in advance. These compiled commands are not included in this count, though they may not yet have started executing.

Usage restrictions

This command can only be executed when the *robot is ready for motion* (page 341). Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]

Cyclic protocols

This command is not available in cyclic protocols.

GetOperationMode

This command returns the operation mode selected by the key switch on the power suppl of the robot.

Syntax

GetOperationMode()

Responses

- [2076][m]
  - m: 0 for locked mode, 1 for automatic mode, and 2 for manual mode.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetOperationMode* (page 257) is represented by the *OperationMode* field of the *SafetyStatus* section (see Section 5).

GetRtAccelerometer

This command is currently available only on the Meca500.

GetRtCartPos

This command returns the pose of the TRF with respect to the WRF, as calculated from the current joint set read by the joint encoders. It also returns a timestamp.

Syntax

GetRtCartPos()

Responses

- [2211][x, y, z, γ]
  - t: timestamp in microseconds;
  - x, y, z: the coordinates of the origin of the TRF with respect to the WRF, in mm;
  - γ: orientation of the TRF about its z-axis

    with respect to the WRF, in degrees.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtCartPos* (page 259) is represented by *DynamicDataTypeID* 41. See Section 5 for more details.

GetRtCartVel

This command returns the current Cartesian velocity vector of the TRF with respect to the WRF, as calculated from the real-time data coming from the joint encoders.

Syntax

GetRtCartVel()

Responses

- [2214][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]
  - t: timestamp in microseconds;
  - $\dot{x}$, $\dot{y}$, $\dot{z}$: components of the linear velocity vector of the TCP with respect to the WRF, in mm/s;
  - ω_z: angular velocity of the TRF with respect to the WRF, in °/s.

The current TCP speed with respect to the WRF is therefore $(\dot{x}^2 + \dot{y}^2 + \dot{z}^2)^{1/2}$.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtCartVel* (page 260) is represented by *DynamicDataTypeID* 44. See Section 5 for more details.

GetRtConf

Unlike the *GetConf* (page 222) command, which returns the desired posture configuration parameters, the *GetRtConf* (page 261) command returns the current posture configuration parameters, calculated from real-time data provided by the joint encoders. Additionally, the *GetRtConf* (page 261) command includes a timestamp in its response.

Syntax

GetRtConf()

Responses

- [2218][c_e]
  - c_e: elbow configuration parameter, -1, 1, or 0[†].

[†] At the corresponding singularity, we return 0, but display the text “n/a” in the web interface.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtConf* (page 261) is represented by *DynamicDataTypeID* 45. See Section 5 for more details.

GetRtConfTurn

Contrary to *GetConfTurn* (page 223), which returns the desired turn configuration parameter, *GetRtConfTurn* (page 262) returns the current turn configuration parameter, as calculated from the real-time data coming from the encoder of the last joint. In addition, the *GetRtConfTurn* (page 262) command returns a timestamp.

Syntax

GetRtConfTurn()

Responses

- [2219][t, c_t]
  - t: timestamp in microseconds;
  - c_t: turn configuration parameter, an integer number.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtConfTurn* (page 262) is represented by *DynamicDataTypeID* 45. See Section 5 for more details.

GetRtJointPos

This command returns the current joint set read by the joint encoders. It also returns a timestamp.

Syntax

GetRtJointPos()

Responses

- [2210][t, θ_1, θ_2, d_3, θ_4]
  - t: timestamp in microseconds;
  - θ_1, θ_2, θ_4: the angles of joint 1, 2, and 4, in degrees;
  - d_3: the position of joint 3, in mm.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtJointPos* (page 263) is represented by *DynamicDataTypeID* 40. See Section 5 for more details.

GetRtJointTorq

This command returns the current joint torques, or more specifically, the current motor torques.

Syntax

GetRtJointTorq()

Responses

- [2213][t, τ_1, τ_2, τ_3, τ_4]
  - t: timestamp in microseconds;
  - τ_i: the torque of motor i as a signed percentage of the maximum allowable torque (i = 1, 2, 3, 4).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtJointTorq* (page 264) is represented by *DynamicDataTypeID* 43. See Section 5 for more details.

GetRtJointVel

This command returns the current joint velocities, calculated by differentiating the joint encoders data.

Syntax

GetRtJointVel()

Responses

- [2212][t, ω_1, ω_2, v_3, ω_4]
  - t: timestamp in microseconds;
  - ω_1, ω_2, ω_4: the rates of change of joints 1, 2, and 4;
  - v_3: the rate of change of joint 3, in mm/s.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtJointVel* (page 265) is represented by *DynamicDataTypeID* 42. See Section 5 for more details.

GetRtTargetCartPos

This command returns the current target pose of the TRF relative to the WRF, rather than the pose derived from real-time data provided by the joint encoders.

Syntax

GetRtTargetCartPos()

Responses

- [2201][x, y, z, γ]
  - t: timestamp in microseconds;
  - x, y, z: the coordinates of the origin of the TRF with respect to the WRF, in mm;
  - γ: orientation of the TRF about its z-axis

    with respect to the WRF, in degrees.

Note

The deprecated GetPose command, which is still supported, returns the same data, except for the timestamp. Additionally, the message ID differs and is 2027.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetCartPos* (page 266) is represented by the *TargetEndEffectorPose* section (see Section 5).

GetRtTargetCartVel

This command returns the current target Cartesian velocity vector of the TRF with respect to the WRF.

Syntax

GetRtTargetCartVel()

Responses

- [2204][t, $\dot{x}$, $\dot{y}$, $\dot{z}$, ω_z]
  - t: timestamp in microseconds;
  - $\dot{x}$, $\dot{y}$, $\dot{z}$: components of the linear velocity vector of the TCP with respect to the WRF, in mm/s;
  - ω_z: angular velocity of the TRF with respect to the WRF, in °/s.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetCartVel* (page 267) is represented by *DynamicDataTypeID* 34. See Section 5 for more details.

GetRtTargetConf

This command returns the posture configuration parameters calculated from the current target joint set.

Syntax

GetRtTargetConf()

Responses

- [2208][c_s, c_e, c_w]
  - c_s: shoulder configuration parameter, -1, 1, or 0†;
  - c_e: elbow configuration parameter, -1, 1, or 0†;
  - c_w: wrist configuration parameter, -1, 1, or 0†.

† At the corresponding singularity, we return 0, but display the text “n/a” in the web interface.

- [2208][c_e]
  - c_e: elbow configuration parameter, -1, 1, or 0†.

† At the corresponding singularity, we return 0, but display the text “n/a” in the web interface.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetConf* (page 268) is represented by the *TargetConfiguration* section (see Section 5).

GetRtTargetConfTurn

This command returns the turn configuration parameters calculated from the current target joint value for the last joint.

Syntax

GetRtTargetConfTurn()

Responses

- [2209][t, c_t]
  - t: timestamp in microseconds;
  - c_t: turn configuration parameter, an integer number.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetConfTurn* (page 269) is represented by the *TargetConfiguration* section (see Section 5).

GetRtTargetJointPos

This command returns the current target joint set.

Syntax

GetRtTargetJointPos()

Responses

- [2200][t, θ_1, θ_2, d_3, θ_4]
  - t: timestamp in microseconds;
  - θ_1, θ_2, θ_4: the angles of joint 1, 2, and 4, in degrees;
  - d_3: the position of joint 3, in mm.

Note

The deprecated GetJoints command, which remains supported, returns the same data, except for the timestamp. The message ID is also different, being 2026.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetJointPos* (page 270) is represented by the *TargetJointSet* section (see Section 5).

GetRtTargetJointTorq

This command returns the current target (calculated) motor torques.

Syntax

GetRtTargetJointTorq()

Responses

- [2203][t, τ_1, τ_2, τ_3, τ_4]
  - t: timestamp in microseconds;
  - τ_i: the torque of motor i as a signed percentage of the maximum allowable torque (i = 1, 2, 3, 4).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetJointTorq* (page 271) is represented by *DynamicDataTypeID* 33. See Section 5 for more details.

GetRtTargetJointVel

This command returns the current target joint velocities.

Syntax

GetRtTargetJointVel()

Responses

- [2202][t, ω_1, ω_2, v_3, ω_4]
  - t: timestamp in microseconds;
  - ω_1, ω_2, ω_4: the rates of change of joints 1, 2, and 4;
  - v_3: the rate of change of joint 3, in mm/s.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTargetJointVel* (page 272) is represented by *DynamicDataTypeID* 32. See Section 5 for more details.

GetRtTrf

This command returns the current definition of the TRF with respect to the FRF, set by the *SetTrf* (page 166) command. It returns exactly the same pose as the *GetTrf* (page 251) command, but the response code is different, and a timestamp precedes the pose data.

Syntax

GetRtTrf()

Responses

- [2229][x, y, z, γ]
  - t: timestamp in microseconds;
  - x, y, z: the coordinates of the origin of the TRF with respect to the FRF, in mm;
  - γ: orientation of the TRF about its z-axis

    with respect to the FRF, in degrees.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtTrf* (page 273) is represented by the *TargetTrf* section (see Section 5).

GetRtWrf

This command returns the current definition of the WRF with respect to the BRF, set by the *SetWrf* (page 168) command. It returns exactly the same pose as the *GetWrf* (page 253) command, but the response code is different, and a timestamp precedes the pose data.

Syntax

GetRtWrf()

Responses

- [2228][x, y, z, γ]
  - t: timestamp in microseconds;
  - x, y, z: the coordinates of the origin of the WRF with respect to the BRF, in mm;
  - γ: orientation of the WRF about its z-axis

    with respect to the BRF, in degrees.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtWrf* (page 274) is represented by the *TargetWrf* section (see Section 5).

GetRtc

This command returns the current Epoch Time in seconds, set using the *SetRtc* (page 200), after every reboot of the robot. Note that this is different from the timestamp returned by all GetRt* commands, which is in microseconds. Furthermore, these two time measurements have different zero references.

Syntax

GetRtc()

Responses

- [2140][t]
  - t: Epoch time as defined in Unix (i.e., number of seconds since 00:00:00~UTC January~1, 1970).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtc* (page 275) is represented by the *RobotTimestamp* section (see Section 5).

GetSafetyStopStatus

This command returns the status of specific safety stop signals.

Syntax

GetSafetyStopStatus(n)

Arguments

- n: any of the following four-digit codes:
  - 3032, for the state of the P-Stop 2 safety stop signal;
  - 3069, for the state of the P-Stop 1 safety stop signal;
  - 3070, for the state of the E-Stop safety stop signal;
  - 3080, for the state of the operation mode safety stop signal;
  - 3081, for the state of the enabling device released safety stop signal;
  - 3082, for the state of the voltage fluctuation safety stop signal;
  - 3083, for the state of the safety stop signal associated with robot reboot or reset signal;
  - 3084, for the state of the safety stop signal associated with a safety signal mismatch;
  - 3085, for the state of the safety stop signal associated with a standstill fault;
  - 3086, for the state of the safety stop signal associated with a connection drop;
  - 3087, for the state of the safety stop signal associated with a minor error.

Responses

- [3032][n], etc.
  - n: 0, 1 or 2, as described in Section 4.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, a broad spectrum of safety-related information is reported in the *SafetyStatus* section (see Section 5).

GetStatusRobot

This command returns the status of the robot.

Syntax

GetStatusRobot()

Responses

- [2007][as, hs, sm, es, pm, eob, eom]
  - as: activation state (1 if robot is activated, 0 otherwise);
  - hs: homing state (1 if homing already performed, 0 otherwise);
  - sm: simulation mode (0 if simulation is disabled, 1 if real-time simulation is enabled, 2 if fast simulation is enabled);
  - es: error status (1 for robot in error mode, 0 otherwise);
  - pm: pause motion status (1 if robot is in pause motion, 0 otherwise);
  - eob: end of block status (1 if robot is not moving and motion queue is empty, 0 otherwise);
  - eom: end of movement status (1 if robot is not moving, 0 if robot is moving).

Note that pm = 1 if a *PauseMotion* (page 182) or a *ClearMotion* (page 173) was sent, or if the robot is in error mode.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetStatusRobot* (page 277) is represented by the *RobotStatus* section (see Section 5).

GetTorqueLimitsStatus

This command returns the status of the torque limits (whether a torque limit is currently exceeded).

Syntax

GetTorqueLimitsStatus()

Responses

- [3028][s]
  - s: status (0 if no detection, 1 if a torque limit was exceeded).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetTorqueLimitsStatus* (page 278) is represented by the *ExcessiveTorque* section (see Section 5).

Work zone supervision and collision prevention commands

In addition to using the *SetJointLimits* (page 190) and *SetJointLimitsCfg* (page 191) commands to further constrain the robot's joint limits, you can also define a work zone with the *SetWorkZoneLimits* (page 290) command (see Figure 11). This command sets a bounding box in the base reference frame (BRF). Similarly, you can define a "tool sphere" in the flange reference frame (FRF) using the *SetToolSphere* (page 288) command.

You can then use the *SetWorkZoneCfg* (page 289) command to configure the robot to monitor whether its links, tool sphere (including optional tooling), or flange center point (FCP) remain within the work zone. Additionally, the *SetCollisionCfg* (page 287) command enables the robot to prevent collisions between its links, tool sphere, and optional tooling.

For both configurations, you can choose to have the robot either generate a warning (supervision only) or create a motion error (preventing a work zone breach or collision). Typically, you will want to prevent collisions, which is why the term "collision prevention" is used. Conversely, you may only wish to detect work zone breaches without preventing them, hence the term "work zone supervision."

Note that you can use the MecaPortal to define these settings. For example, you can enable the display of the work zone and tool sphere through the settings menu in the 3D view panel of the MecaPortal. When collisions occur, the colors of the colliding bodies will change to red.

Figure 11 illustrates the objects currently supervised. The base STL model also includes part of the cables coming from the base (not shown).

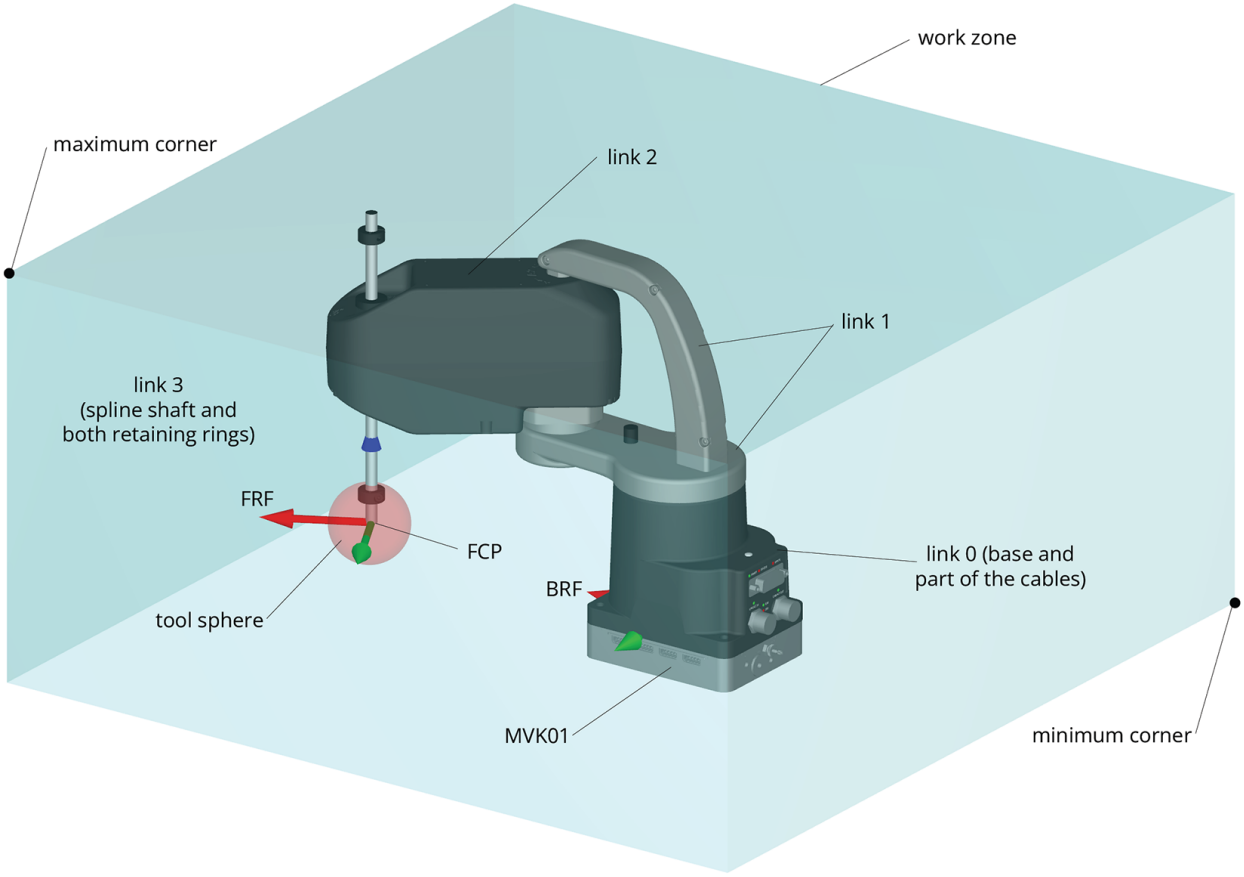


Figure 11: Objects tested in the work zone supervision and collision prevention feature

Danger

The work zone supervision and collision prevention feature is not safety rated. Additionally, when the robot handles heavy or large objects at high speeds and with significant blending, there is a possibility that work zone breaches or collisions may be detected a few milliseconds too late.

The following is the complete list of work zone supervision and collision prevention commands, in alphabetical order:

- *GetCollisionStatus* (page 282)
- *GetWorkZoneStatus* (page 286)
- *SetCollisionCfg* (page 287) / *GetCollisionCfg* (page 281)
- *SetToolSphere* (page 288) / *GetToolSphere* (page 283)
- *SetWorkZoneCfg* (page 289) / *GetWorkZoneCfg* (page 284)
- *SetWorkZoneLimits* (page 290) / *GetWorkZoneLimits* (page 285)

GetCollisionCfg

This command returns the severity level set with the *SetCollisionCfg* (page 287) command.

Syntax

GetCollisionCfg()

Responses

- [2181][l]
  - l: severity level.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *GetCollisionCfg* (page 281) command is represented by *DynamicDataTypeID* 36. See Section 5 for more details.

GetCollisionStatus

This command returns the current collision status (refer to Figure 11).

Syntax

GetCollisionStatus

Responses

- [2182][v, g_1, $o_{id,1}$, g_2, $o_{id,2}$]
  - v: collision state (1 or 0†),
  - g_1, g_2: group identifier of first and second colliding objects:
    - 0 for links
    - 1 for FCP
    - 2 for tool
  - $o_{id,1}$, $o_{id,2}$: object ID of first and second in collision, depending on group identifier, as follows:
    - If g = 0 (links): 0 for robot base, 1 for link 1, 2 for link 2, etc.
    - If g = 1 (FCP): 0 for FCP (flange center point).
    - If g = 2 (tool): 0 for tool sphere, 20,000 for MVK01,.

† If v = 0, $g_1 = g_2 = o_{id,1} = o_{id,2} = 0$.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *GetCollisionStatus* (page 282) command is represented by *DynamicDataTypeID* 37. See Section 5 for more details.

GetToolSphere

This command returns the current definition of the tool sphere, set with the *SetToolSphere* (page 288) command.

Syntax

GetToolSphere()

Responses

- [2167][x, y, z, r]
  - x, y, z: the coordinates of the center of the tool sphere with respect to the FRF, in mm;
  - r: the radius of the tool sphere, in mm.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *GetToolSphere* (page 283) command is represented by *DynamicDataTypeID* 19. See Section 5 for more details.

GetWorkZoneCfg

This command returns the current work zone configuration, set with the *SetWorkZoneCfg* (page 289) command.

Syntax

GetWorkZoneCfg()

Responses

- [2163][l, m]
  - l: event severity;
  - m: supervision mode.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *GetWorkZoneCfg* (page 284) command is represented by *DynamicDataTypeID* 17. See Section 5 for more details.

GetWorkZoneLimits

This command returns the current definition of the bounding box with respect to the BRF, set with the *SetWorkZoneLimits* (page 290) command.

Syntax

GetWorkZoneLimits()

Responses

- [2165][x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]
  - x_{min}, y_{min}, z_{min}: the coordinates of the minimum corner of the cuboid in the BRF, in mm;
  - x_{max}, y_{max}, z_{max}: the coordinates of the maximum corner of the cuboid in the BRF, in mm.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *GetWorkZoneLimits* (page 285) command is represented by *DynamicDataTypeID* 18. See Section 5 for more details.

GetWorkZoneStatus

This command returns the current work zone violation status (refer to Figure 11).

Syntax

GetWorkZoneStatus()

Responses

- [2183][v, g, o_{id}]
  - v: work zone violation state (1 or 0†),
  - g: group identifier of object in breach: - 0 for links - 1 for FCP - and 2 for tool
  - o_{id}: object ID, depending on group identifier number, as follows:
    - If g = 0 (links): 0 for robot base, 1 for link 1, 2 for link 2, etc.
    - If g = 1 (FCP): 0 for FCP (flange center point).
    - If g = 2 (tool): 0 for tool sphere, 20,000 for MVK01,.

† If v = 0, g = o_{id} = 0.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *GetWorkZoneStatus* (page 286) command is represented by *DynamicDataTypeID* 38. See Section 5 for more details.

SetCollisionCfg

This command specifies the event severity for the collision supervision (robot links, tool sphere, andMPM500 module).

Syntax

SetCollisionCfg(l)

Arguments

- l: integer defining the collision detection event severity as
  - 0, silent (i.e., collisions are verified but no action is taken, other than to log them internally);
  - 1, generate a warning (message [2182]) every time a new imminent collision is detected;
  - 4, generate a warning (message [2182]) and a motion error (message [3041]) every time a new imminent collision is detected.

Default values

This setting is persistent and retains its value even after power cycling the robot. The factory default is 4.

Responses

- [2180] [Collision configuration set successfully.]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *SetCollisionCfg* (page 287) command is represented by *MotionCommandID* 154. See Section 5 for more details.

SetToolSphere

This command defines a sphere fixed in the flange reference frame (FRF). Interferences between that sphere and the robot links as well as the outside of a bounding box set with the *SetWorkZoneLimits* (page 290) command can then be supervised, as defined by the *SetWorkZoneCfg* (page 289) and *SetCollisionCfg* (page 287) commands.

Syntax

SetToolSphere(x,y,z,r)

Arguments

- x, y, z: the coordinates of the center of the tool sphere in the FRF, in mm;
- r: the radius of the tool sphere, in mm.

Default values

This setting is persistent and retains its values even after power cycling the robot. The factory default is x = y = z = 0 and r = 0. Note that setting all four arguments to zero is equivalent to disabling the tool sphere. However, if r = 0, but one of the coordinates is not zero, the tool sphere will be a point.

Responses

- [2168] [Tool sphere set successfully.]

Note

The MCS500 does not verify interferences between the tool sphere and the spline shaft.

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *SetToolSphere* (page 288) command is represented by *MotionCommandID* 155. See Section 5 for more details.

SetWorkZoneCfg

This command specifies the “event severity” for the work zone limits supervision and the robot parts that need to be verified.

Syntax

SetWorkZoneCfg(l,m)

Arguments

- l: integer defining the work zone breach detection event severity as
  - 0: silent (i.e., work zone breach is verified but no action is taken, other than to log them internally);
  - 1: generate a warning (message [2183]) every time a new imminent work zone breach is detected;
  - 4: generate a warning (message [2183]) and a motion error (message [3049]) every time a new imminent work zone breach is detected.
- m: integer defining the work zone breach verification mode as
  - 1: verify whether the FCP (flange center point) is inside the work zone;
  - 2: verify whether the tool is completely inside the work zone (tool is the tool sphere defined with the *SetToolSphere* (page 288) command);
  - 3: verify whether the tool AND all robot links are completely inside the work zone.

Default values

This setting is persistent and retains its values even after power cycling the robot. The factory default is l = 4 and m = 1.

Responses

- [2164] [Work zone configuration set successfully.]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *SetWorkZoneCfg* (page 289) command is represented by *MotionCommandID* 152. See Section 5 for more details.

SetWorkZoneLimits

This MotionCommandID defines a bounding box (a cuboid), the sides of which are parallel to the axes of the base reference frame (BRF). The arguments of the command are the coordinates of two diagonally opposite corners, referred to as “minimum” and “maximum” corners, such that each coordinate of the minimum corner is smaller that the corresponding coordinate of the maximum corner.

Syntax

SetWorkZoneLimits(x_{min}*,*y_{min}*,*z_{min}*,*x_{max}*,*y_{max}*,*z_{max}*)*

Arguments

- x_{min}, y_{min}, z_{min}: the coordinates of the minimum corner of the cuboid in the BRF, in mm;
- x_{max}, y_{max}, z_{max}: the coordinates of the maximum corner of the cuboid in the BRF, in mm.

Default values

This setting is persistent and retains its values even after power cycling the robot. The factory default is $x_{min} = y_{min} = z_{min} = -10{,}000$ and $x_{max} = y_{max} = z_{max} = 10{,}000$. To reset the arguments to their factory default values, deactivate the robot and send the command *SetWorkZoneLimits(0,0,0,0,0,0)* (page 290).

Responses

- [2166] [Workspace limits set successfully.]

Usage restrictions

This command can only be executed when the robot is deactivated.

Cyclic protocols

In cyclic protocols, the *SetWorkZoneLimits* (page 290) command is represented by *MotionCommandID* 153. See Section 5 for more details.

Commands for optional accessories

This section regroups all commands that are used to control or request data from the optional accessory for your robot: vacuum and I/O module (MVK01). Some of the commands in this section are queued, others are instantaneous (Get*, *SetExtToolSim* (page 312), and *_Immediate).

The following is the complete list of commands used for the vacuum and I/O module:

- *GetIoSim* (page 296)
- *GetRtInputState* (page 301)
- *GetRtIoStatus* (page 302)
- *GetRtOutputState* (page 303)
- *GetRtVacuumPressure* (page 304)
- *GetRtVacuumState* (page 305)
- *GetVacuumPurgeDuration* (page 307)
- *GetVacuumThreshold* (page 308)
- *SetIoSim* (page 316)
- *SetOutputState* (page 317)
- *SetOutputState_Immediate* (page 318)
- *SetVacuumPurgeDuration* (page 319)
- *SetVacuumPurgeDuration_Immediate* (page 320)
- *SetVacuumThreshold* (page 321)
- *SetVacuumThreshold_Immediate* (page 322)
- *VacuumGrip* (page 324)
- *VacuumGrip_Immediate* (page 325)
- *VacuumRelease* (page 326)
- *VacuumRelease_Immediate* (page 327)

GetExtToolFwVersion

This command is available only on the Meca500 robot.

GetGripperForce

This command is available only on the Meca500 robot.

GetGripperRange

This command is available only on the Meca500 robot.

GetGripperVel

This command is available only on the Meca500 robot.

GetIoSim

This instantaneous command returns the state of the simulation of the MVK01 vacuum and I/O module set by the command *SetIoSim* (page 316) (or the default one).

Syntax

GetIoSim(b_{id})

Arguments

- b_{id}: I/O bank ID, should be 1.

Responses

- [2056][b_{id}, e]
  - b_{id}: I/O bank ID, should be 1;
  - e: status of the simulation mode (1 if enabled, 0 if disabled).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols.

GetRtExtToolStatus

This command is available only on the Meca500 robot.

GetRtGripperForce

This command is available only on the Meca500 robot.

GetRtGripperPos

This command is available only on the Meca500 robot.

GetRtGripperState

This command is available only on the Meca500 robot.

GetRtInputState

This instantaneous command returns the state of the eight digital inputs of the MVK01 module.

Syntax

GetRtInputState(b_{id})

Arguments

- b_{id}: I/O bank ID, currently 1

Responses

- [2341][t, b_{id},p_1,p_2,p_3,p_4, p_5,p_6,p_7,p_8]†
  - t: timestamp in microseconds;
  - b_{id}: I/O bank ID, currently 1;
  - p_i: state of input pin i (i = 1, 2, …, 8), 1 for high and 0 for low.

† If an MVK01 module is not present, no input values will be included in the response and there will be no error.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtInputState* (page 301) is represented by *DynamicDataTypeID* 72. See Section 5 for more details.

GetRtIoStatus

This instantaneous command returns the status of the MVK01 vacuum and I/O module.

Syntax

GetRtIoStatus(b_{id})

Arguments

- b_{id}: I/O bank ID, currently 1

Responses

- [2330][t, b_{id}, present, simMode, errorCode]
  - t: timestamp in microseconds;
  - b_{id}: I/O bank ID, currently 1 (for MVK01);
  - present: 1 if the MVK01 module has been detected, 0 if otherwise;
  - simMode: state (1 for enabled, 0 for disabled) of the MVK01 simulation mode (see *SetIoSim* (page 316));
  - errorCode: error code (0 if no error).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtIoStatus* (page 302) is represented by *DynamicDataTypeID* 72. See Section 5 for more details.

GetRtOutputState

This instantaneous command returns the state of the eight digital outputs of the MVK01 module.

Syntax

GetRtOutputState(b_{id})

Arguments

- b_{id}: I/O bank ID, currently 1

Responses

- [2340][t, b_{id},p_1,p_2,p_3,p_4, p_5,p_6,p_7,p_8]†
  - t: timestamp in microseconds;
  - b_{id}: I/O bank ID, currently 1;
  - p_i: state of output pin i (i = 1, 2, …, 8), 1 for high and 0 for low.

† If an MVK01 module is not present, no output values will be included in the response and there will be no error.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtOutputState* (page 303) is represented by *DynamicDataTypeID* 72. See Section 5 for more details.

GetRtVacuumPressure

This instantaneous command returns the current pressure in the vacuum chamber of the MVK01 vacuum and I/O module.

Syntax

GetRtVacuumPressure()

Responses

- [2343][t, p]
  - t: timestamp in microseconds;
  - p: pressure of the vacuum chamber, in kPa (usually non-positive, except during purging).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtVacuumPressure* (page 304) is represented by *DynamicDataTypeID* 73. See Section 5 for more details.

GetRtVacuumState

This instantaneous command returns the current state of the pneumatic part of the MVK01 vacuum and I/O module.

Syntax

GetRtVacuumState()

Responses

- [2342][t, v, h, p]
  - t: timestamp in microseconds;
  - v: state of vacuum generation (1 if vacuum is being generated, 0 if not);
  - p: state of air purge (1 if air is being purged in order to quickly release a part, 0 if not);
  - h: state of holding part (1 if vacuum is being generated and a part is being held, 0 if not).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetRtVacuumState* (page 305) is represented by *DynamicDataTypeID* 73. See Section 5 for more details.

GetRtValveState

This command is available only on the Meca500 robot.

GetVacuumPurgeDuration

This instantaneous command returns the duration of the air purge on the MVK01 vacuum and I/O module set by the command *SetVacuumPurgeDuration* (page 319) (or the default one).

Syntax

GetVacuumPurgeDuration()

Responses

- [2173][t_p]
  - t_p: duration of air purge in seconds.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the command *GetVacuumPurgeDuration* (page 307) is represented by *DynamicDataTypeID* 26. See Section 5 for more details.

GetVacuumThreshold

This instantaneous command returns the current pressure thresholds (negative values) for the MVK01 vacuum and I/O module set by the command *SetVacuumThreshold* (page 321) (or the default ones).

Syntax

GetVacuumThreshold()

Responses

- [2172][p_h, p_r]
  - p_h: threshold pressure below which the robot considers that a part is held, in kPa;
  - p_r: threshold pressure above which the robot considers that a part is no longer held, in kPa.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the commands *GetVacuumThreshold* (page 308) is represented by *DynamicDataTypeID* 26. See Section 5 for more details.

GripperClose

This command is available only on the Meca500 robot.

GripperOpen

This command is available only on the Meca500 robot.

MoveGripper

This command is available only on the Meca500 robot.

SetExtToolSim

This command is available only on the Meca500 robot.

SetGripperForce

This command is available only on the Meca500 robot.

SetGripperRange

This command is available only on the Meca500 robot.

SetGripperVel

This command is available only on the Meca500 robot.

SetIoSim

This instantaneous command toggles the simulation mode for the MVK01 vacuum and I/O module. With the simulation enabled, you can use all the MVK01 commands (e.g., *VacuumGrip* (page 324), *SetOutputState* (page 317)) and if the MVK01 is physically present, these commands will not have any physical impact (i.e., they will only be simulated).

Syntax

SetIoSim(b_{id},e)

Arguments

- b_{id}: I/O bank ID, must be 1 (for MVK01);
- e: state of simulation mode (1 to enable, 0 to disable).

Default values

By default, the simulation mode of the MVK01 is disabled.

Responses

- [2056][b_{id}, e]
  - b_{id}: I/O bank ID, must be 1 (for MVK01);
  - e: state of simulation mode (1 to enable, 0 to disable).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *SetIoSim* (page 316) command is represented by *MotionCommandID* 36. See Section 5 for more details.

SetOutputState

This queued command is used to control the digital outputs of the MVK01 vacuum and I/O module.

Syntax

SetOutputState(b_{id},p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8)

Arguments

- b_{id}: I/O bank ID, currently 1 (for MVK01);
- p_i: state of output pin i (i = 1, 2, …, 8), 1 to set, 0 to reset, and −1 or * to keep unchanged.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetOutputState* (page 317) command is represented by *MotionCommandID* 34. See Section 5 for more details.

SetOutputState_Immediate

This is the same command as *SetOutputState* (page 317), but it is instantaneous rather than queued and can be executed even when the robot is deactivated.

Syntax

SetOutputState_Immediate(b_{id},p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8)

Arguments

- b_{id}: I/O bank ID, currently 1;
- p_i: state of output pin i (i = 1, 2, …, 8), 1 to set, 0 to reset, and −1 or * to keep unchanged.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *SetOutputState_Immediate* (page 318) command is represented by *MotionCommandID* 35. See Section 5 for more details.

SetVacuumPurgeDuration

This queued command sets the duration of the air purge for ejecting a part when using the commands VacuumRelease*.

Syntax

SetVacuumPurgeDuration(t_p)

Arguments

- t_p: duration in seconds.

Default values

By default, t_p = 0.1.

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetVacuumPurgeDuration* (page 319) command is represented by *MotionCommandID* 43. See Section 5 for more details.

SetVacuumPurgeDuration_Immediate

This is the same command as *SetVacuumPurgeDuration* (page 319), but it is instantaneous, rather than queued and can be executed even when the robot is deactivated.

Syntax

SetVacuumPurgeDuration_Immediate(t_p)

Arguments

- t_p: duration in seconds.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *SetVacuumPurgeDuration_Immediate* (page 320) command is represented by *MotionCommandID* 44. See Section 5 for more details.

SetVacuumThreshold

This queued command sets the thresholds for the vacuum sensor (which measures only negative pressure) in the MVK01 vacuum and I/O module that will be used for reporting whether a part is being held or not.

Syntax

SetVacuumThreshold(p_h,p_r)

Arguments

- p_h: when the negative pressure sensed is smaller than p_h, the robot reports that a part is being held. The value for this argument ranges from −100 kPa to −5 kPa.
- p_r: when the value of the negative pressure sensed is larger than p_r, the robot reports no part is being held. The value for this argument ranges from −95 kPa to 0 kPa. The value of p_r must be larger than the < value of p_h by at least 5 kPa.

Default values

By default, p_h = −40 kPa, and p_r = −30 kPa. To reset to the default values, reactivate the robot or send the command *SetVacuumThreshold(0,0)* (page 321).

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *SetVacuumThreshold* (page 321) command is represented by *MotionCommandID* 41. See Section 5 for more details.

SetVacuumThreshold_Immediate

This is the same command as *SetVacuumThreshold* (page 321), but it is instantaneous, rather than queued and can be executed even when the robot is deactivated.

Syntax

SetVacuumThreshold_Immediate(p_h,p_r)

Arguments

- p_h: when the negative pressure sensed is smaller than p_h, the robot reports that a part is being held. The value for this argument ranges from −100 kPa to −5 kPa.
- p_r: when the value of the negative pressure sensed is larger than p_r, the robot reports no part is being held. The value for this argument ranges from −95 kPa to 0 kPa. The value of p_r must be larger than the < value of p_h by at least 5 kPa.

Default values

By default, p_h = −40 kPa, and p_r = −30 kPa. To reset to the default values, reactivate the robot or send the command *SetVacuumThreshold(0,0)* (page 321).

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *SetVacuumThreshold_Immediate* (page 322) command is represented by *MotionCommandID* 42. See Section 5 for more details.

SetValveState

This command is available only on the Meca500 robot.

VacuumGrip

This queued command activates the suction in the MVK01 vacuum and I/O module.

Syntax

VacuumGrip()

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *VacuumGrip* (page 324) command is represented by *MotionCommandID* 37. See Section 5 for more details.

VacuumGrip_Immediate

This instantaneous command activates the suction in the MVK01 vacuum and I/O module. Unlike its queued equivalent (*VacuumGrip* (page 324)), it can be executed even when the robot is deactivated.

Syntax

VacuumGrip_Immediate()

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *VacuumGrip_Immediate* (page 325) command is represented by *MotionCommandID* 38. See Section 5 for more details.

VacuumRelease

This queued command deactivates the suction in the MVK01 vacuum and I/O module.

Syntax

VacuumRelease()

Usage restrictions

This command is added to the robot's motion queue and can only be executed when the *robot is ready for motion* (page 341), not in an error state and when no safety signal conditions are present. Otherwise, the robot will report one of the following:

- [1005][The robot is not activated.]
- [1011][The robot is already in error.]

Cyclic protocols

In cyclic protocols, the *VacuumRelease* (page 326) command is represented by *MotionCommandID* 39. See Section 5 for more details.

VacuumRelease

This instanteneous command deactivates the suction in the MVK01 vacuum and I/O module. Unlike its queued equivalent (*VacuumRelease* (page 326)), it can be executed even when the robot is deactivated.

Syntax

VacuumRelease_Immediate()

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

In cyclic protocols, the *VacuumRelease_Immediate* (page 327) command is represented by *MotionCommandID* 40. See Section 5 for more details.

Commands for managing variables (beta)

> **Warning**
>
> Please note that this feature is in beta and may undergo changes, including potential API changes.

We also provide an API to manage persistent variables. These variables persist after the robot is rebooted. They can be used in programs that control the robot (via the TCP API or cyclic protocols).

Variables are ideal for saving information that can vary between different robots running the same program, such as reference positions, desired velocities, or delays. This allows the program to automatically adapt to each robot it is executed on.

Variables can be referenced by the programs saved in the robot (by passing variables instead of numeric arguments to robot commands). They can also be referenced by a PLC (by using variables as arguments for motion commands or by setting variables, then starting a program that uses them).

The complete list of variables management commands is:

- *CreateVariable* (page 334)
- *DeleteVariable* (page 335)
- *GetVariable* (page 336)
- *ListVariables* (page 337)
- *SetVariable* (page 337)

The following provides a summary of important details regarding variables in the robot:

- **Persistence** Variables are saved on the robot and persist after rebooting. However, a factory reset removes all variables from the robot's persistent storage.
- **Access and modification** Variables can be accessed, created, modified, or deleted in any robot state (robot activated, robot deactivated, robot in recovery mode, etc.).
- **Data types** Variables can store a variety of JSON-supported values, including booleans, numbers, strings, and arrays.

The variable type is deduced by the value provided to *CreateVariable* (page 334). Command *SetVariable* (page 337) will be refused if the provided value is of a different type. No automatic type conversion is performed by the robot.

Managing variables in the MecaPortal

The robot's MecaPortal web interface has a configuration panel to view, edit, create and delete variables. For more information on the MecaPortal, please refer to *mecaportal_config_menu* of the MecaPortal operating manual.

Managing variables in robot programs (TCP API)

In robot programs, you can use *CreateVariable* (page 334), *DeleteVariable* (page 335) or *SetVariable* (page 337) commands to manage your variables.

We suggest to create a program responsible for variables creation (using *CreateVariable* (page 334)) that is called once, then refer to (or modify) these variables in your robot programs.

You can refer to these variables by using *vars.myGroup.myVar* when calling robot API functions, where *myGroup* is the case-sensitive name of the group (you can have subgroups as well) and *myVar* is the case-sensitive name of the variable. There are two ways to use variables in robot TCP API commands:

Single-value variables

- A single-value variable holds a single value and has the prefix *vars*.
- Example: *SetPayload(vars.myGroup.m, vars.myGroup.cx, vars.myGroup.cy, vars.myGroup.cz)* (page 160)

Unrolling array variables

- An unrolling array variable holds an array and has the prefix **vars*. The asterisk (*) unrolls the array to pass individual elements to the function.
- Example: MoveJoints(*vars.myGroup.myJointPos)

Managing variables with Mecademicpy (Python API)

The Python API provides a simplified API for managing variables. All robot variables are synchronized and stored in the robot class as attributes of *robot.vars*.

Creating or deleting a robot variable

The *robot* Python class provides the same functions to manage variables:

- *robot.CreateVariable*
- *robot.SetVariable*
- *robot.DeleteVariable*
- *robot.GetVariable*
- *robot.ListVariables*

Please refer to Mecademicpy's documentation for details on these function calls.

Directly accessing robot variables through Python attributes

While a Python script can access a variable using *robot.GetVariable* and modify it with *robot.SetVariable* or *robot.CreateVariable*, our Python API offers a more convenient approach: variables are available as attributes of *robot.vars*.

To access a variable, use: *robot.vars.myGroup.myVar*

To modify a variable, assign a new value: *robot.vars.myGroup.myVar = [1, 2, 3, 4, 5, 6]*

Warning

Setting a variable value is a blocking operation, as it involves sending a TCP request to the robot and waiting for confirmation. Additionally, the robot must write the new value to persistent storage. For optimal performance, Python scripts should use local variables (and not robot variables) for values that change frequently during runtime.

Managing variables with cyclic protocols

In cyclic protocols (see Section 5), variables are accessed by their cyclic ID, which is defined when the variable is created.

The cyclic ID of a variable must be in the range [10000,19999].

By referring to this cyclic ID, cyclic protocols can modify variables and use to their values as arguments for motion commands.

Note

Cyclic protocols do not support creating, deleting, getting, or listing variables.

Setting a variable

A variable is modified by using its cyclic ID as the command ID in a sent cyclic command. For more information on how to send a command using cyclic protocols, see Section 5.

The six floating-point values of the motion command are used to set the new value of the variable as follows:

- Cyclic protocols support only setting variables of type number or array of numbers;
- For a variable of type number, the first argument of the cyclic command is used as the new value;
- For a variable of type array of numbers, the corresponding number of cyclic command arguments are used to update the array. The size of the array remains unchanged and cannot be modified through cyclic protocols;

Note

Remember that the variable type is defined when the variable is created (*CreateVariable* (page 334)) and cannot be changed afterward. The robot also does not perform any automatic type conversion. Therefore, boolean or string variables cannot be assigned through cyclic protocols.

Referencing a variable

To use a variable (or multiple variables) as arguments for cyclic protocol motion commands, proceed as follows:

- Set the desired command ID (example: 2 for *MovePose* (page 138));
- Set the `UseVariables` bit in cyclic motion control data (see Section 5);
- Set the variable ID(s) to use as the float arguments of the motion parameters structure (see Section 5).

As a result:

- The chosen variable value(s) will be used in place of the inline motion command arguments;
- The values of the selected variables will be concatenated and used as (up to six) arguments for the motion command. This allows you to combine the values from multiple variables for a single motion command.

Example

In this example, we create a variable that is defined as an array of 6 float values, then use it to call the *MovePose* (page 138) command.

> **Note**
>
> It is also possible to pass multiple variables to the function, for example, one array of three floats for [x, y, z] and another for [alpha, beta, gamma], or six separate variables each holding a single float. However, for simplicity, the following example uses a single variable containing all six float values.

Assuming we have previously created the following variable (using the MecaPortal or the TCP API):

- *myCartPos*
  - Array of six floating-point values, representing [x, y, z, alpha, beta, gamma]
  - Cyclic ID: 10000

The PLC can modify the variable *myCartPos* as shown below:

- Send motion command with ID 10000 (referring to *myCartPos*), using motion command arguments *[190.0, 0.0, 308.9, -1, 75, -2]* (see Section 5);

The PLC then selects the variable to be used in a *MovePose* (page 138) command as follows:

- Set motion command with ID 2 (*MovePose* (page 138))
- With the *UseVariables* bit set (see Section 5);
- And as command arguments, the cyclic ID of the variable to use: *[10000,0, 0, 0, 0, 0]*;
- Note that here up to 6 variables could be used for calling the command, in this example we refer to a single variable that contains an array of 6 float.

As a result:

- The *MovePose* (page 138) command will be executed using the six values from the *myCartPos* variable.
- The result is: *MovePose(190.0, 0.0, 308.9, -1, 75, -2)* (page 138)

CreateVariable (beta feature)

This command creates a variable that is saved on the robot and persists even after a reboot. A variable is defined by its case-sensitive name, a value (supporting various types) and an optional cyclic ID.

For more information, see *Commands for managing variables (beta)* (page 328).

Syntax

CreateVariable(name, value, cyclicId, override)

Arguments

- name:
  - A unique name for this variable (e.g., “myVar”);
  - *Variable names are case sensitive*;
  - May include several case-sensitive prefixes (e.g., “myGroup.mySubgroup.mainWrf”);
  - If the name already exists, the behavior of *CreateVariable* (page 334) depends on the override argument.
- value:
  - The value to assign to the variable;
  - The value can be any basic JSON type: boolean, number, string, or array (but not a JSON object). Remember that in JSON syntax, boolean values must be lowercase (i.e., true or false).
  - Examples:
    - a boolean: *CreateVariable(myBoolVar, true)*
    - a number: *CreateVariable(myIntVar, -0.153)*
    - a string: *CreateVariable(myStringVar, "Hello world!")*
    - an array: *CreateVariable(myArrayVar, [190.0, 0.0, 308.9, 0, 90, 0])*
- cyclicId (optional, 0 by default):
  - The unique ID used to refer to this variable in cyclic protocols in the range [10000,19999] or 0;
  - When 0 (or omitted), no cyclic ID is associated with the variable;
  - If the provided cyclic ID is already in use, *CreateVariable* (page 334) will fail with error [1552];
  - If a non-zero cyclic ID is used, the value must be a number or an array of numbers, otherwise *CreateVariable* (page 334) will fail with error [1552].
- override (optional, 0 by default):

  - Specifies how *CreateVariable* (page 334) behaves when a variable with the same name already exists:
    * 1 to update the existing variable's value and cyclic ID with the new ones;
    * 0 to return the error [1552] if the existing variable has a different type or cyclic ID; otherwise do nothing and leave the variable unchanged.

Responses

- [2552] [name, value, cyclicId, override]
- [1552] [errorMsg]
  - errorMsg:
    * An error message explaining why variable creation failed;
    * e.g., "Cyclic ID 100001 is already used by variable myOtherVar".

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols. See *Managing variables with cyclic protocols* (page 332).

DeleteVariable (beta feature)

This command deletes a variable from the robot.

For more information, see *Commands for managing variables (beta)* (page 328).

Syntax

DeleteVariable(name)

Arguments

- name: name of the variable to delete (e.g., "myVar").

Responses

- [2553] [name]
- [1553] [errorMsg]
  - errorMsg:
    * An error message explaining why variable deletion failed;

    * e.g., “Cannot delete variable myVar (not found)”.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols. See *Managing variables with cyclic protocols* (page 332).

GetVariable (beta feature)

This command returns the value of a robot variable.

For more information, see *Commands for managing variables (beta)* (page 328).

Syntax

GetVariable(name)

Arguments

- name: name of the variable to get (e.g., “myVar”).

Responses

- [2551] [name, value, cyclicId]
- [1551] [errorMsg]
  - errorMsg:
    * An error message explaining why the variable could not be retrieved;
    * e.g., “Variable ‘myVar’ does not exist”.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols. See *Managing variables with cyclic protocols* (page 332).

ListVariables (beta feature)

This command returns the list of all variable names that exist on the robot.

For more information, see *Commands for managing variables (beta)* (page 328).

Syntax

ListVariables()

Responses

- [2550] [var1, var2, ...]
- [1550] [errorMsg]
  - errorMsg:
    - An error message explaining why the variable could not be listed.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

This command is not available in cyclic protocols. See *Managing variables with cyclic protocols* (page 332).

SetVariable (beta feature)

This command modifies a robot variable. The modification persists even after a reboot.

A variable's value can only be changed to a value of the same type. For example, a variable holding a number cannot be assigned a string or an array.

No automatic conversion is performed between supported types. For instance assigning an integer to a boolean value, or vice versa, is not supported.

To change a variable's type, use the 'override' option in the *CreateVariable* (page 334) command.

For more information, see *Commands for managing variables (beta)* (page 328).

Syntax

SetVariable(name, value)

Arguments

- name:
  - The name of the variable to modify (e.g., “myVar”).
- value:
  - The new value to assign to the variable;
  - The value can be any basic JSON type: boolean, number, string, or array (but not a JSON object). Remember that in JSON syntax, boolean values must be lowercase (i.e., true or false);
  - *SetVariable* (page 337) will fail with error [1554] if you try to assign a value of a different type or array length.
  - Examples:
    - a boolean: `SetVariable(myBoolVar, true)`
    - a number: `SetVariable(myIntVar, -0.153)`
    - a string: `SetVariable(myStringVar, "Hello world!")`
    - an array: `SetVariable(myArrayVar, [190.0, 0.0, 308.9, 0, 90, 0])`

Responses

- [2554] [name, value]
- [1554] [errorMsg]
  - errorMsg:
    - An error message explaining why the variable modification failed;
    - e.g., “Cannot set variable my_var (not found)”.

Usage restrictions

This command can be executed in any robot state.

Cyclic protocols

See *Managing variables with cyclic protocols* (page 332).

Terminology

Below is the list of terms used by us in our technical documentation.

active line: The line in the MecaPortal where the cursor is currently positioned.

BRF: Base Reference Frame.

Cartesian space: The four-dimensional space defined by the position (x, y, z) and orientation (γ) of the TRF with respect to the WRF.

control port: The TCP port 10000, over which commands to the robot and messages from the robot are sent.

data request commands: Commands used to request some data regarding the robot (e.g., *GetTrf* (page 251), *GetBlending* (page 215), *GetJointVel* (page 229)). These commands are executed immediately and generally return values for parameters that have already been configured (sent and executed) with a Set* command (or the default values).

default value: There are different settings in the robot controller that can be configured using Set* commands (e.g., *SetCartAcc* (page 142)). Many of these settings have default values. Every time the robot is powered up, these settings are initialized to their default values. In the case of motion commands settings, their values are also initialized to their default values every time the robot is deactivated. In contrast, some settings are persistent and their values are stored on an SD drive.

detailed event log: This file mirrors the content of the event log panel in the MecaPortal when in detailed mode. It can be downloaded from the MecaPortal (see Section 9 of the Programming Manual).

distal link: In the MCS500, this is the black-anodized body that holds the spline shaft.

EOAT: End-of-arm tooling.

EOB: End-of-block message, [3012][], sent by default every time the robot has stopped moving AND its motion queue is empty. You can disable this message with the command *SetEob* (page 188).

EOM: End-of-motion message, [3004][], sent by the robot whenever it has stopped moving for at least 1 ms, if this option is activated with *SetEom* (page 189).

error mode: The robot goes into error mode when it encounters an error while executing a command or a hardware problem (see Table 1).

FCP: Flange Center Point. The origin of the FRF.

FRF: Flange Reference Frame.

instantaneous commands: These are commands that are executed immediately, as soon as received by the robot. All data request commands (Get*), all robot control commands, all work zone supervision and collision prevention commands and some optional accessories commands (*_Immediate) are instantaneous.

inverse kinematics: The problem of obtaining the robot joint sets that correspond to a desired end-effector pose. See Section 3 of the Programming manual for more details.

joint position: The joint angle associated with a rotary joint or the position of joint 3.

joint set: The set of all joint positions.

joint space: The four-dimensional space defined by the positions of the robot joints.

monitoring port: The TCP port 10001, over which data is sent periodically from the robot.

motion commands: Commands used to construct the robot trajectory (e.g., *Delay* (page 125), *MoveJoints* (page 126), *SetTRF* (page 166), *SetBlending* (page 141)). When a Mecademic robot receives a motion command, it places it in a motion queue. The command will be run once all preceding motion commands have been executed.

motion queue: The buffer where motion commands that were sent to the robot are stored and executed on a FIFO basis by the robot.

offline program: A sequence of commands saved in the internal memory of the robot. The term *offline* is often omitted and will eventually be removed altogether.

online mode programming: Programming the robot in online mode involves moving it directly to each desired robot position, typically using jogging controls.

PDO (**Process Data Object**): In EtherCAT, a Process Data Object (PDO) is a data structure used for exchanging real-time cyclic data between an EtherCAT master and its slave devices. PDOs can contain individual bits, bytes, or words.

persistent settings: Some settings in the robot controller have default values (e.g., the robot name set by the command *SetRobotName* (page 199)), but when changed, their new values are written on an SD drive and persist even if the robot is powered off.

pose: The position and orientation of one reference frame with respect to another.

position mode: One of the two control modes, in which the robot's motion is generated by requesting a target end-effector pose or joint set (see Section 3 of the Programming Manual).

robot posture configuration: The two-value (-1 or 1) parameter c_e that normally defines each of the two possible robot postures for a given pose of the robot's end-effector.

proximal link: This is the clear-anodized body between the base and the distal link.

queued commands: Commands that are placed in the motion queue, rather than executed immediately. All motion commands are queued commands, as well as some external-tool commands.

reach: The maximum distance between the axis of joint 1 and the axis of joint 4.

real-time data request commands: Commands used to request some real-time data regarding the current status of robot (e.g., *GetRtTrf* (page 273), *GetRtCartPos* (page 259), *GetStatusRobot* (page 277)).

retaining ring: In the MCS500, this is each of the two circular clamps on both ends of the spline shaft.

robot control commands: Commands used to immediately control the robot, (e.g., *ActivateRobot* (page 171), *PauseMotion* (page 182), *SetNetworkOptions* (page 193)). These commands are executed immediately, i.e., are instantaneous.

robot is ready for motion: The robot is considered *ready* to receive motion commands, i.e. when it is activated.

Note that if the robot is in error or if a safety stop condition is present, it will refuse motion commands, but it will still be considered *ready* since its motion queue remains initialized and retains the latest received settings (e.q., velocity, acceleration, blending, WRF, TRF, etc.).

robot log: This file is a more detailed version of the user log, intended primarily for our support team. It can be downloaded from the MecaPortal (see Section 9 of the Programming Manual).

robot position: A robot position is equivalent to either a joint set or the pose of the TRF relative to the WRF, along with the definitions of both reference frames, and the robot posture and last joint turn configuration parameters.

robot posture: The arrangement of the robot links. Equivalent to a joint set in which all joint angles are normalized, i.e. have been converted to the range $(-180°, 180°]$.

SDO (**Service Data Object**): In EtherCAT, a Service Data Object (SDO) is a data structure used for non-real-time communication between an EtherCAT master and its slave devices. SDOs are typically used to configure device parameters and access diagnostic information through the object dictionary. Unlike PDOs, SDOs exchange structured data rather than individual bits or bytes.

singularities: A robot posture where the robot's end-effector is blocked in some directions even if no joint is at a limit (see Section 3 of the Programming Manual).

spline shaft: This is the the groved reciprocating shaft.

TCP: Tool Center Point. The origin of the TRF. Not to be confused with Transmission Control Protocol.

TRF: Tool reference frame.

turn configuration parameter: Since the last joint of the robot can rotate multiple revolutions, the turn configuration parameter defines the revolution number.

user log: This file is a simplified log containing user-friendly traces of major events (e.g., robot activation, movement, E-Stop activation). It can be downloaded from the MecaPortal (see Section 9 of the Programming Manual).

velocity mode: One of the two control modes, in which the robot's motion is generated by requesting a target joint velocity vector or end-effector Cartesian velocity vector (see Section 3 of the Programming Manual).

workspace: The Cartesian workspace of a robot is the set of all feasible poses of its TRF with respect to its WRF. Note that many of these poses can be attained with more than one set of configuration parameters.

WRF: World reference frame.